AFRL-IF-RS-TR-2004-144
Final Technical Report
June 2004

DIVA (DATA INTENSIVE ARCHITECTURE)

USC Information Sciences Institute

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. G215, J099

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-144 has been reviewed and is approved for publication

APPROVED: /s/
CHRISTOPHER J. FLYNN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** JUNE 2004	**3. REPORT TYPE AND DATES COVERED** **FINAL** **Mar 98 – Dec 02**

4. TITLE AND SUBTITLE DIVA (DATA INTENSIVE ARCHITECTURE)	**5. FUNDING NUMBERS** G - F30602-98-2-0180 PE - 62110E, 62301E PR - G215 TA - 00 WU - 01
6. AUTHOR(S) John J. Granacki, Mary Hall, Jeffrey Draper, Jeff LaCoss, Jacqueline Chame, Tim Barrett, Alvin Despain, Jean-Luc Gaudiot	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) USC Information Sciences Institute 4676 Admiralty Way Marina Del Rey CA 90292-6695	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington VA 22203-1714 AFRL/IFTC 525 Brooks Road Rome NY 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRLIF-RS-TR-2004-144

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Christopher J. Flynn/IFTC/(315) 330-3249 Christopher.Flynn@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT *APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 Words)*

The design, development and implementation of a prototype system of a novel computer architecture based on PIM (Processing-In-Memory) technology are presented. The simulator and emulator that were used to develop and evaluate the overall concepts and system are also described. The DIVA system uses PIM-based “smart memories” to improve the effective processor-memory bandwidth. The DIVA “smart memory” significantly improves the performance of data-intensive applications over their performance on conventional processor memory systems that suffer from the traditional performance bottleneck caused by the speed gap between high performance microprocessors and DRAMs (Dynamic Random Access Memory) used in main memory. The chips developed for DIVA represent the first smart-memory devices supporting virtual addressing and capable of executing multiple threads of control. DIVA achieves enhanced performance through multiple mechanisms including both coarse-grain and fine-grain parallelism. The simulation results for a broad class of applications run on the DIVA simulator and reported in the literature show significant speedups over conventional processor architectures. Running some of these applications on the prototype hardware has validated these speedups.

14. SUBJECT TERMS Computer Architecture, Data Intensive, Processing-In-Memory, PIM, Memory Bandwidth, VLSI, Compiler, Memory			**15. NUMBER OF PAGES** 404 **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

Table of Contents

List of Figures

List of Tables

1. Executive Summary

The DIVA (Data IntensiVe Architecture) Project has developed a prototype workstation class system using VLSI PIM (Processor-In-Memory) chips as smart-memory coprocessors to a conventional microprocessor. These chips represent the first smart memory devices to support virtual addressing and be capable of executing multiple threads of control. The DIVA PIM VLSI is fabricated in TSMC 0.18-micron technology. The chip measures 9.8 mm on a side and contains 55 million transistors.

The successful demonstration of the DIVA prototype system incorporating this chip involved research in several areas including: System Architecture, Software System Architecture, PIM Architecture, VLSI Architecture/Implementation, Emulator Architecture/Design and the actual development of the prototype system hardware and software. These areas involved teams made up of staff from USC/Information Sciences Institute, the University of Notre Dame, Caltech, the University of Delaware and AlphaTech, Inc.

The goals of the DIVA Project were to demonstrate the capabilities of PIM technology as smart memory in a system:

- Exploit the inherent memory bandwidth
    - embedded DRAM technology
- Cover a broad range of applications:
    - irregular memory accesses (sparse-matrices & pointers)
    - image processing and multimedia (streaming computations)
- Evolutionary application migration path
    - PIMs also support standard memory accesses
    - familiar parallel programming paradigm
- Prototype a workstation-class system
    - VLSI PIM chips in standard memory modules

All these goals were met. The projected peak performance on a DIVA system with 32 PIMs is 40 GOPS, with an aggregate memory bandwidth of 160 Gbytes/second. This is more than two orders of magnitude bandwidth increase over conventional systems meeting the DIS (Data Intensive Systems) goal. A 35-x speedup on the “cornerturn” benchmark, a matrix transpose kernel function found in many data intensive DoD applications, was also demonstrated.

The DIVA VLSI PIM developed under the DARPA Data Intensive Systems (DIS) Program is proving to be effective in ameliorating the processor-memory bottleneck present in most of today's computing systems. In addition, DIVA PIM technology has been incorporated into the MONARCH (MOrphable Networked ARCHitecture) Project under the DARPA PCA (Polymorphous Computer Architecture) Program and Godiva in partnership with Hewlett Packard on the HPCS (High Productivity Computing System) Program.

2. Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 50-60% per year while memory access times improve at merely 5-7%. Further, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [Burger96]. Recent VLSI technology trends offer a promising solution to bridging the processor-memory gap: embedded-DRAM technology integrates logic with high-density memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (with hundreds of gigabit/second aggregate bandwidth available on a chip --- up to 2 orders of magnitude over conventional DRAM). Latency to on-chip logic is also reduced, down to as little as one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The system described in this report, DIVA (Data IntensiVe Architecture), leverages embedded-DRAM technology to replace or augment the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. System bandwidth limitations are thus overcome in three ways: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processor memory "nodes" per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus. The DIVA system architecture is focused on achieving the following four goals: (1) developing PIMs that can serve as the only memory in the system, assuming the dual roles of "smart memories" and conventional memory; (2) supporting a wide range of familiar programming paradigms, closely related to parallel computing; (3) targeting applications that are severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets; and, (4) developing a VLSI device to exploit memory and communications bandwidth in PIM-based systems while making efficient use of on-chip resources for target applications. These four goals distinguish DIVA from other PIM-based architectures.

The integration into a conventional system affords the simultaneous benefits of PIM technology and a high-performance microprocessor host, yielding high performance for mixed workloads. Since PIM processors are usually not as sophisticated as state-of-the-art microprocessors due to on-chip space constraints, systems using PIMs alone in a multiprocessor may sacrifice performance on uniprocessor computations [Saulsbury96][Kogge94], while SoC (System-on-a-Chip) solutions (e.g., the IRAM [Patterson97] and the Mitsubishi M32R/D [Mitsubishi99]) limit the application domain. DIVA's support for a broad range of familiar parallel programming paradigms, including task parallelism for irregular computations, distinguishes it from systems with restricted applicability (such as to SIMD parallelism [Elliot99][Gokhale95][Patterson97]), as well as systems requiring a novel programming methodology or compiler technology to configure logic [Babb99], or to manage a complex memory, computation and communication hierarchy [Kang99]. DIVA's PIM-to-PIM interconnect improves upon approaches that serialize communication through the host, which decreases bandwidth by introducing added traffic on the processor memory bus [Oskin98][Gokhale95].

A major challenge in meeting the above four goals is the integrated system design, which implements the system architecture and spans the applications, systems software, host-to-memory interface, memory-to-memory interconnect, PIM software and embedded DRAM VLSI devices.

The remainder of this report is organized as follows. The next section summarizes the DIVA system architecture, to set the context for the PIM microarchitecture and other sections that follow. Section 4 describes the VLSI architecture and implementation in detail. Section 5 presents the compiler optimization, implementation and performance results. Section 6 describes the DIVA system simulator that supported the applications and architectural development throughout the DIVA Project. Section 7 sets out the details of how the DIS benchmarks and stressmarks as well as other application code were used with the simulator to evaluate DIVA's performance. Section 8 summarizes our approach to an FPGA (Field Programmable Gate Array) based emulator and the lessons that we learned in this endeavor. Section 9 presents the system integration that was required to produce a successful system prototype demonstration at DARPA Tech 2002. In the remaining sections, we summarize our results, technology transfer, publications and conclusions.

3. System Architecture

A driving principle of the DIVA system architecture is to efficiently utilize PIM technology in a way that requires only "evolutionary" software support. This principle demands an approach that enables integration of PIM features into conventional systems as seamlessly as possible. Therefore, DIVA chips will be packaged as conventional memory modules. Inserted onto a conventional microprocessor motherboard, the memory on the DIVA chips is accessed by the host microprocessor as if it were conventional memory.

In Figure 1, we show a small set of PIMs connected to a single external host processor through a host-memory interface. The PIM chips communicate through separate PIM-to-PIM channels.

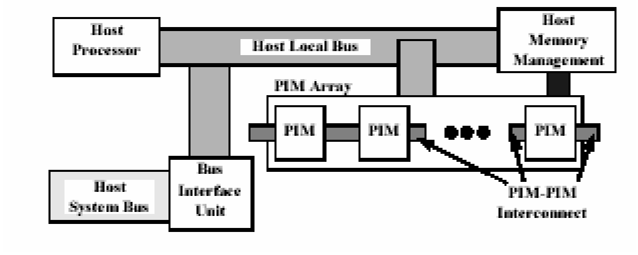


Figure 1. DIVA system architecture

This separate memory-to-memory interconnect enables communication between memories without involving the host processor.

Spawning computation, gathering results, synchronizing activity, or simply accessing non-local data is accomplished via parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received [vonEicken92]. Parcels are distinguished from active messages in that the destination of a parcel is an object in memory, not a specific processor.

Parcels are transmitted through a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic. This interconnect must be amenable to the dense packing requirement of memory devices and allow the addition or removal of devices from the system. For system sizes of the scale expected for DIVA (on the order of 32 PIM chips), this combination of requirements favors a one dimensional network [Kang00]. Future generations of DIVA-like systems that contain large numbers of PIM chips will require a more complex interconnection network and are the topic of future research.

Parcels, application code, and data contain virtual addresses. To translate these addresses without the overhead of maintaining conventional page tables at each node, we classify DIVA memory according to usage [Hall99]: (1) *global memory* visible to the host and PIM nodes; (2) *dumb memory* allocated as conventional pages in a host application's virtual space and untouched by PIM node processing; and, (3) *local memory* used exclusively by PIM node routines. To condense translation information, rather than page tables, we use segments, each of which is defined by segment registers which are used by the node address translation unit as discussed below.

The primary functions of the node address translation unit are to translate virtual addresses to physical addresses for those accesses, which are locally resident, and to provide access protection. The types of accesses generated by a DIVA PIM processor that require translation include instruction fetches and data accesses to memory or memory-mapped devices such as parcel buffers, generated by load or store instructions.

Given the simplicity of the address translation scheme, very little hardware support is needed to effect efficient translation. A segment base address register and limit register is needed for each of the eight local segments. Also, one virtual base, limit, and physical base register are needed for each resident global segment. The initial DIVA architecture provides four sets of global segment registers, although alternative architectures could provide more. The address translation unit contains no direct support for home node translation, although the preferred system programming is such that the global segments resident on a node form the portion of global memory for which that node is the home node. If this is not the case, address faults invoke system software, which performs the home node translation.

In addition to local segments, a node maintains translation information for its portion of global memory. Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation [Saulsbury95]. Thus, each node's portion of global memory includes objects for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

Memory management functionality is distributed among the host's standard operating system, augmented with support for PIMs, and run-time kernels on each PIM processor. Unlike standard multiprocessor systems, the host, which has a system-level view, remains a central figure in system-level scheduling, disk I/O operations, and memory management. The PIM run-time kernel must collaborate with the host on system-level operations, such as loading PIM programs and data, memory management of PIM-visible segments, and PIM context switches between different user programs. The challenge in this collaboration is that there are really two views of memory that must be maintained. For dumb pages and for disk I/O of PIM-visible segments, the host sees memory as standard 4Kbyte pages; the PIM run-time kernel instead views PIM-visible memory as variable-sized segments [Hall00].

4. VLSI Architecture and Implementation

The goal of the VLSI development on the DIVA project was to produce a prototype chip that demonstrated the enormous bandwidth available between memory blocks and processing subcomponents on a processing-in-memory (PIM) device. As the following sections discuss, the DIVA project was very successful with its VLSI demonstrations and was the first effort under the Data-Intensive Systems (DIS) program to deliver working silicon. The bulk of this effort can be categorized into chip-level architecture research and VLSI implementation.

4.1 PIM Chip Architecture

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and networking/communication hardware. Although a PIM may consist of multiple nodes, each of which are primarily comprised of a few megabytes of memory and a node processor, Figure 2 shows a PIM with a single node, which reflects the focus of the research that was conducted on the DIVA project. Nodes on a PIM chip share a single PIM Routing Component (PiRC) and a host interface. The PiRC is responsible for routing parcels on and off chip. The host interface implements the JEDEC standard SDRAM (Synchronous Dynamic Random Access Memory) protocol so that memory accesses as well as parcel activity initiated by the host appear as conventional memory accesses from the host perspective. More details of the PiRC can be found in [Kang00] and more information on the host interface is given in [Draper02a].

Figure 2 also shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. Within the host interface, a parcel buffer (PBUF) is a buffer that is memory-mapped into the host processor's address space, permitting application-level communication through parcels. Each PIM node also has a PBUF, memory-mapped into the node's local address space. More information on the PBUF design is found in Appendix A2: DIVA Node Architecture manual.

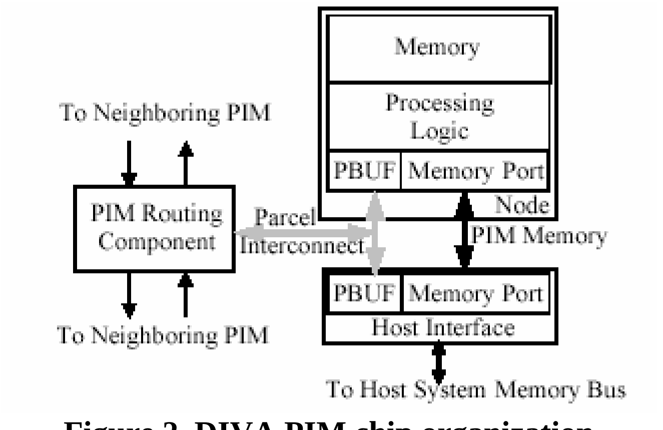


Figure 2. DIVA PIM chip organization

Figure 3 shows the major control and data connections within a node, with the 256-bit memory data bus as the centerpiece. The DIVA PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a scalar datapath that performs sequential operations on 32-bit operands, and a WideWord datapath that performs fine-grain parallel operations on 256-bit operands. Both datapaths execute from a single instruction stream under the control of a single 5-stage DLX (Deluxe)-like pipeline. The instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap.

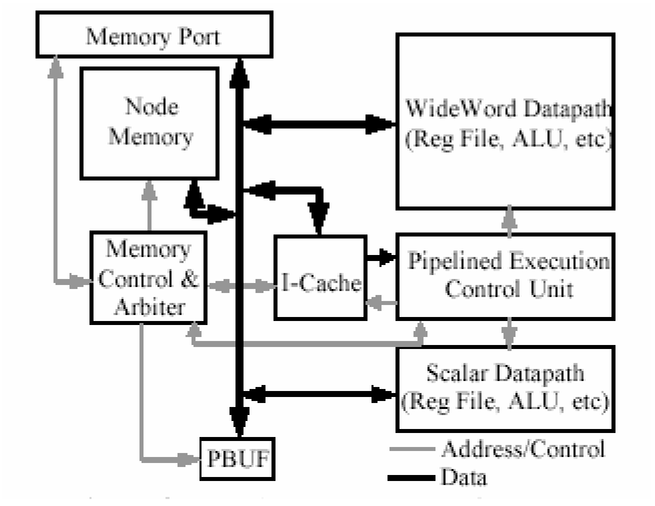

Memory Port
Node Memory
WideWord Datapath (Reg File, ALU, etc)
Memory Control & Arbiter
I-Cache
Pipelined Execution Control Unit
Scalar Datapath (Reg File, ALU, etc)
PBUF
Address/Control
Data

Figure 3. DIVA PIM node architecture

Each datapath has its own independent general-purpose register file, 32 32-bit registers for the scalar datapath and 32 256-bit registers for the WideWord datapath, but special instructions permit direct transfers between datapaths without going through memory. Although not supported in the initial DIVA prototype, floating-point extensions to the WideWord datapath will be provided in future implementations. The memory arbiter/controller is responsible for generating proper control signals to the memory macro. Its functions include initiating refresh cycles as needed and arbitrating between the host memory port and the execution control unit for access to the memory macro. Furthermore, it tracks and maintains an open row in the DRAM macro to enable page-mode accesses as often as possible. Another key component of each PIM node is an instruction cache, which was included in the DIVA design to keep instruction accesses to the memory macro from interfering with data accesses as much as possible. Each node also contains a parcel buffer (PBUF), as described earlier. The following sections briefly discuss the scalar and WideWord subcomponents, highlighting some of the more notable features. More detail on these microarchitectures as well as those of other subcomponents of the DIVA PIM chip can be found in the Appendices.

4.1.1 Microarchitecture: The Scalar Processor

As noted earlier, the combination of the execution control unit and scalar datapath is a standard RISC processor and serves as the DIVA scalar processor, or microcontroller. It coordinates all activity within a DIVA PIM node. This section details the microarchitecture of this component by first presenting an overview of the instruction set architecture, followed by a description of the pipeline and discussion of special features. More detail of the instruction set can be found in Appendix A1: DIVA Instruction Set Manual.

Instruction set architecture overview

Much like the Hennessy and Patterson DLX architecture, most DIVA scalar instructions use a three-operand format to specify two source-registers and a destination register, as shown in Figure 4. For these types of instructions, the opcode generally denotes a class of operations, such as arithmetic, and the function denotes a specific operation, such as add. The **C** bit indicates whether the operation performed by the instruction execution updates condition codes. In lieu of a second source register, a 16-bit immediate value may be specified, as shown in Figure 5. The scalar instruction set includes the typical arithmetic functions add, subtract, multiply, and divide; logical functions AND, OR, NOT, and XOR; and logical/arithmetic shift operations. In addition, there are a number of special instructions, described in Special Features section below. Load/store instructions adhere to the immediate format, where the address for the memory operation is formed by the addition of an immediate value to the contents of **rA**, which serves as a base address. The DIVA scalar processor does not support a base-plus-register addressing mode because such a mode requires an extra read port on the register file for store operations.

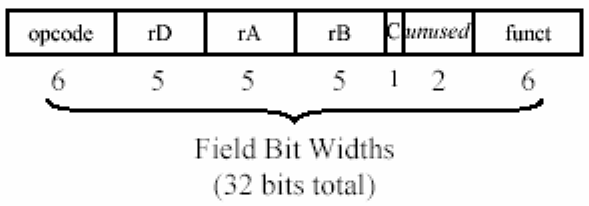


Figure 4. Scalar register instruction format

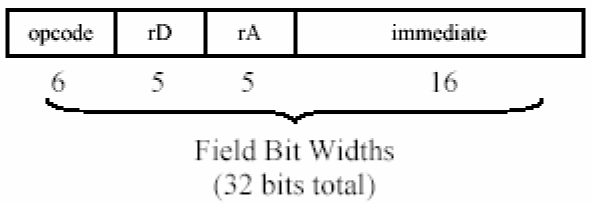


Figure 5. Scalar immediate instruction format

Branch instructions use a different format. The branch target address may be PCrelative, useful for relocatable code, or calculated using a base register combined with an offset, useful with table-based branch targets. In both formats, the offset is in units of instruction words, or 4 bytes. By specifying the offset in instruction words, rather than bytes, a larger branch window results. To support function calls, the branch instruction format also includes a bit for specifying linkage, that is, whether a return instruction address should be saved in R31. The branch format also includes a 3-bit condition field to specify one of eight branch conditions: always, equal, not equal, less than, less than or equal, greater than, greater than or equal, or overflow.

Pipeline description and associated hazards

A high-level schematic of the pipeline execution control unit and scalar datapath is shown in Figure 6. The pipeline is a standard DLX-like 5-stage pipeline, with the following stages: (1) instruction fetch; (2) decode and register read; (3) execute; (4) memory; and, (5) write-back. Figure 6 indicates these five stages with respect to the data-path registers and also indicates the write-back and bypass datapaths. The pipeline controller contains the necessary logic to handle data, control, and structural hazards. Data hazards occur when there are read-after-write register dependences between instructions that co-exist in the pipeline. The controller and datapath contain the necessary forwarding, or bypass, logic to allow pipeline execution to proceed without stalling in most data dependence cases. The only exception to this generality involves the load instruction, where a "bubble" must be inserted between the load instruction and an immediately following instruction that uses the load target register as one of its source operands.

Control hazards occur for branch instructions. Unlike the DLX architecture, which uses explicit comparison instructions and testing of a general-purpose register value for branching decisions, the DIVA design incorporates condition codes that may be updated by most arithmetic/logical instructions. The condition codes used for branching decisions are:

- EQ - set if the result is zero
- LT - set if the result is negative
- GT - set if the result is positive
- OV - set if the operation overflows

The DIVA pipeline design imposes a 1-delay slot branch, so that the instruction following a branch instruction is always executed. Since branches are always resolved within the second stage of the pipeline, no stalls or bubbles are associated with branch instructions.

Since the general-purpose register file contains 2 read ports and 1 write port, it may sustain two operand reads and 1 result write every clock cycle; thus, the register file design introduces no structural hazards. The only structural hazard that impacts the pipeline operation is the node memory. Pipeline stalls may occur in the instruction fetch stage if an instruction cache miss occurs. The pipeline will resume once the cache fill memory request has been satisfied. Likewise, stalls occur any time a load/store instruction reaches the memory stage of the pipeline until the memory operation is completed.

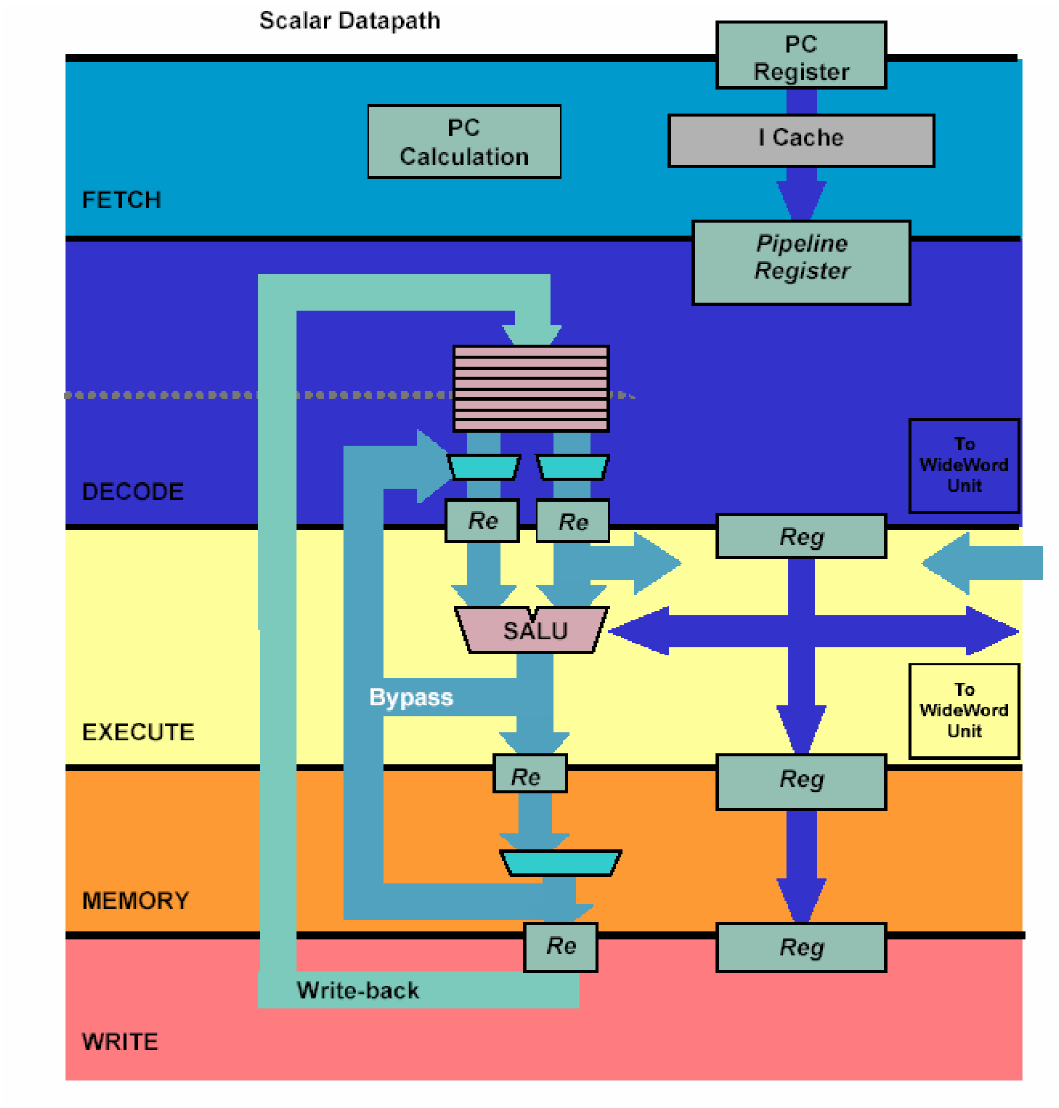


Figure 6. Scalar datapath and pipeline stages

Special features
The novelty of the DIVA scalar processor lies in the special features that support DIVA-specific functions. Although by no means exhaustive, this section highlights some of the more notable capabilities.

Run-time Kernel Support
The execution control unit supports supervisor and user modes of processing and also maintains a number of special-purpose and protected registers for support of exception handling, address translation, and general OS (Operating System) services. Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external component like the PBUF, are handled by a common mechanism.

The exception-handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, to maintain a flexible implementation platform. It provides an integrated mechanism for handling hardware and software exception sources and a flexible priority assignment scheme that minimizes the amount of time that exception recognition is disabled. While the hardware design allows traditional stack-based exception handlers, it also supports a non-recursive dispatching scheme that uses DIVA hardware features to allow preemption of lower priority exception handlers.

The impact of run-time kernel support on the scalar processor design is the addition of a modest number of special-purpose and protected (or supervisor-level) registers and a non-negligible amount of complexity added to the pipeline control for entering/exiting exception handling modes cleanly. When the scalar processor control unit detects an exception, the logic performs a number of tasks within a single clock cycle to prepare the processor for entering an exception handler in the next clock cycle.

Those tasks include:

- determining which exception to handle by prioritizing among simultaneously occurring exceptions,
- setting up shadow registers to capture critical state information, such as the processor status word register, the instruction address of the faulting instruction, the memory address if the exception is an address fault, etc,
- configuring the program counter logic to load an exception handler address on the next clock cycle, and
- setting up the processor status word register to enter supervisor mode with exception handling temporarily disabled.

Once invoked, the exception handler first stores other pieces of user state and interrogates various pieces of state hardware to determine how to proceed. Once the exception handler routine has completed, it restores user state and then executes a return-from-exception instruction, which copies the shadow register contents back into various state registers to resume processing at the point before the exception was encountered. If it is impossible to resume previous processing due to a fatal exception, the run-time kernel exception handler may choose to terminate the offending process.

Interaction with the WideWord Datapath

There are a number of features in the scalar processor design involving communication with the WideWord datapath that greatly enhance performance. The path to/from the WideWord datapath in the execute stage of the pipeline facilitates the exchange of data between the scalar and WideWord datapaths without going through memory. This capability distinguishes DIVA from other architectures containing vector units, such as AltiVec. This path also allows scalar register values to be used to specify WideWord functions, such as indices for selecting subfields within WideWords and indices into permutation look-up tables. Instead of requiring an immediate value within a WideWord instruction for specifying such indices, this register-based indexing capability enables more intelligent, efficient code design.

There are also a couple of instructions that are especially useful for enabling efficient data mining operations. ELO, encode leftmost one, and CLO, clear leftmost one, are instructions that generate a 5-bit index corresponding to the bit position of the leftmost one in a 32-bit value and clear the leftmost one in a 32-bit value, respectively. These instructions are especially useful for examining the 32-bit WideWord condition code register values, which may be transferred to scalar general-purpose registers to perform such tests. For instance, with this capability, finding and processing data items that match a specified key are accomplished in much fewer instructions than a sequence of bit masking and shifting involved in 32 bit tests, which is required with conventional processor architectures.

There are some variations of the branch/call instructions that also interact with the WideWord datapath. The **BA** (branch on all) instruction specifies that a branch is to be taken if the status of condition codes within every subfield of the WideWord datapath matches the condition specified in the BA instruction. The **BN** (branch on none) instruction specifies that a branch is to be taken if the status of condition codes within no subfield of the WideWord datapath matches the condition specified in the BN instruction. With proper code structuring around these instructions, inverse forms of these branches, such as branch on any or branch on not all, can also be affected.

Miscellaneous Instructions
There are also several other miscellaneous instructions that add some complexity to the processor design. The probe instruction allows a user to interrogate the address translation logic to see if a global address is locally mapped. This capability allows users who wish to optimize code for performance to avoid slow, overhead-laden address translation exceptions. Also, an instruction cache invalidate instruction allows the supervisor kernel to evict user code from the cache without invalidating the entire cache and is useful in process termination cleanup procedures. Lastly, there are versions of load/store instructions that “lock” memory operations, which are useful for implementing synchronization functions, such as semaphores or barriers.

4.1.2 Microarchitecture: The WideWord Processor
The combination of the execution control unit and WideWord datapath is regarded as the WideWord Processor. This component enables superword-level parallelism on wide words of 256 bits, similar to multimedia extensions such as MMX and AltiVec. This fine-grain parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. Selective execution, direct transfers to/from other register files, integration with communication, as well as the ability to access main memory at very low latency, distinguish the DIVA WideWord capabilities from MMX and AltiVec. This section details the microarchitecture of this component by first presenting an overview of the instruction set architecture, followed by a brief description of the pipeline. More detail can be found in [Draper02a].

WideWord Instruction set architecture

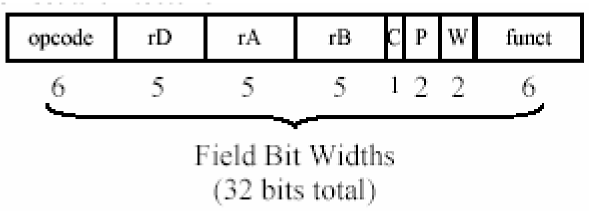


Figure 7. WideWord instruction format

As shown in Figure 7, most DIVA WideWord instructions use a three-operand format to specify two 256-bit source registers and a 256-bit destination register. The opcode generally denotes a class of operations, such as arithmetic, and the function denotes a specific operation, such as add or subtract. The **C** bit indicates whether the operation performed by the instruction execution updates condition codes. The **W** field indicates the operand width, allowing WideWord data to be treated as a packed array of objects of eight, sixteen, or thirty-two bits in size. This characteristic means the WideWord ALU (Arithmetic Logic Unit) can be represented as a number of variable-width parallel ALUs. The **P** field indicates the participation mode, a form of selective subfield execution that depends on the state of local and neighboring condition codes. Under selective execution, only the results corresponding to the subfields that participate in the computation are written back, or committed, to the instruction's destination register. The subfields that participate in the conditional execution of a given instruction are derived from the condition codes or a mask register, plus the instruction's 2-bit participation field.

The WideWord instruction set consists of roughly 30 instructions implementing typical arithmetic instructions like add, subtract, and multiply; logical functions like AND, OR, NOT, XOR; and logical/arithmetic shift operations. In addition, there are load/store and transfer instructions that provide for rich interactions between the scalar and WideWord datapaths.

Some special instructions include permutation, merge, and pack/unpack. The WideWord permutation network supports fast alignment and reorganization of data in wide registers. The permutation network enables any 8-bit data field of the source register to be moved into any 8-bit data field of the destination register. A permutation is specified by a permutation vector, which contains 32 indices corresponding to the 32 8-bit subfields of a WideWord destination register. A WideWord permutation instruction selects a permutation vector by either specifying an index into a small set of hard-wired commonly used permutations or a WideWord register whose contents are the desired permutation vector. The merge instruction allows a WideWord destination to be constructed from the intermixing of subfields from two source operands, where the source for each destination subfield is selected by a condition specified in the instruction. This merge instruction effects efficient sorting. The pack/unpack instructions allow the truncation/elevation of data types and are especially useful in pixel processing.

Pipeline description

Identical to and tightly integrated with the scalar pipeline, the pipeline of the WideWord datapath is a standard DLX-like 5-stage pipeline, with the following stages: (1) instruction fetch; (2) decode and register read; (3) execute; (4) memory; and, (5) writeback. Data hazards occur when there are read-after-write register dependences between instructions that co-exist in the pipeline. The controller and datapath contain the necessary forwarding, or bypass, logic to allow pipeline execution to proceed without stalling in most data dependence cases. Register forwarding is complicated somewhat by the participation capability. Participation status must be forwarded along with each subfield to effect correct forwarding.

4.2 VLSI Development

From a host of potential foundries for fabrication, the selections were quickly narrowed down to two possible embedded DRAM candidates early in the DIVA project: IBM and TSMC. IBM clearly had more experience in the embedded DRAM arena, so early efforts in the DIVA VLSI development task targeted the IBM CMOS7LD 0.25∝m embedded DRAM process, and a scalar processor test chip was fabricated in HP CMOS14 0.5∝m technology through MOSIS. (The HP process was used for early prototyping because its logic speed matched that of the IBM process, and prototypes could be built very cheaply through this route.) A test vehicle on the TSMC 0.25∝m process was also fabricated to gain familiarity with that technology. Although the DIVA team entered into a research collaboration contract with the Blue Gene team at IBM Watson, the DIVA project was not granted access to IBM fabrication capability in a timely manner. Therefore, in the final half of the project, the VLSI development for the integrated PIM prototype targeted the TSMC 0.18∝m process. This process was introduced with an embedded DRAM capability, but that capability was later phased out, so the DIVA prototype PIM was fabricated with SRAM (Synchronous Random Access Memory) as a placeholder for embedded DRAM.

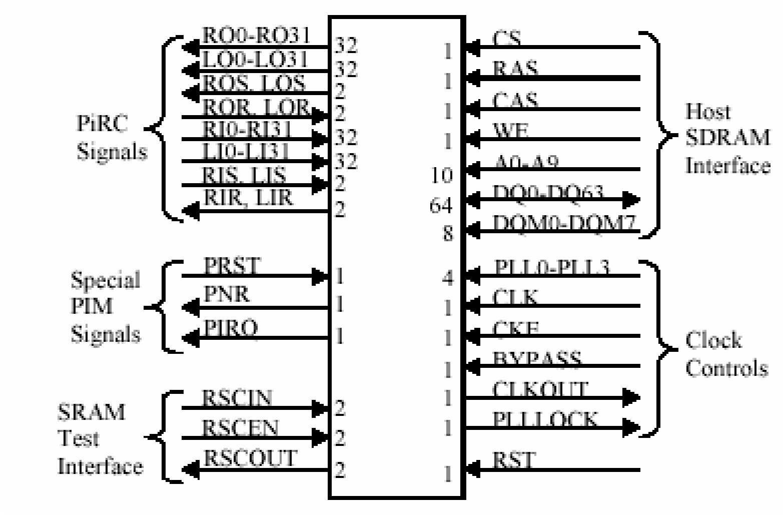


Figure 8. Prototype PIM signal summary

As part of the core VLSI development task, a new CAD tool flow was installed. To accommodate rapid design of the PIM chip, we relied heavily on the ability to specify the chip design with RTL-level VHDL and synthesize this description into a gate-level netlist of standard cells. The VHDL was optimized and synthesized using Synopsys Design Analyzer, targeting the Artisan standard cell library for TSMC 0.18∝m technology. The entire chip was placed and routed, including clock tree routing, with Cadence Silicon Ensemble. Physical verification, including DRC, LVS, and antennae checking, was performed with Mentor Calibre. Back-annotated simulation to verify correct operation and timing of the design was performed within the Cadence Verilog environment.

A description of the external signals of the first prototype PIM chip is shown in Figure 8. There are primarily two external interfaces: a host interface for implementing the JEDEC SDRAM standard and the PiRC signals for inter-PIM communication. Additionally, there are signals for configuring and monitoring the PLL (Phase Locked Loop) clock multiplier, testing the node SRAMs, and reset and interrupt capabilities.

This prototype chip implements one PIM node (consisting of a 32-bit scalar processor, 256-bit WideWord Unit, 4Kbyte instruction cache, 8Mbit node SRAM, and node parcel buffer), PIM routing component (PiRC), and host interface (containing an external SDRAM interface and host parcel buffer). The design was submitted on August 23, 2001 for fabrication on a TSMC 0.18∝m generic process offered through MOSIS. The intellectual property used in the chip design is from three different vendors:

- Artisan
  - standard cells for synthesized logic
  - pads
  - 32-word x 32-bit scalar register file
  - 32-word x 256-bit WideWord register file (implemented as two x128 banks)
  - 4kbyte SRAM for instruction cache core (implemented as two banks of
  - 128 word x 128-bit SRAMs)
  - 128 word x 20-bit SRAM for instruction cache tags
- Virage Logic
  - 8 Mbit SRAM (with redundancy to allow repair) (implemented as two banks of 32768 words x 128 bits)
  - fuse boxes for the configuration of the SRAM
- NurLogic
  - PLL for clock multiplication and deskewing

The resulting chip is 9.8mm on a side and contains approximately 200,000 placeable objects, where a placeable object is anything from a 2-transistor inverter to a 4 Mbit SRAM macro. The chip contains approximately 55 million transistors, with 2 million in the logic and smaller SRAMs and 53 million in the 8 Mbits of node SRAM. The chip contains 352 pads: 240 signal I/O, 56 grounds, 28 pad Vdd (3.3V), and 28 core Vdd (1.8V).

The silicon die were received near the end of October 2001, and packaged chips were received near the end of November 2001. Photos of the die and package are shown in Figure 9. Due to delays in procurement of test fixtures, full-scale testing did not commence until February 2002.

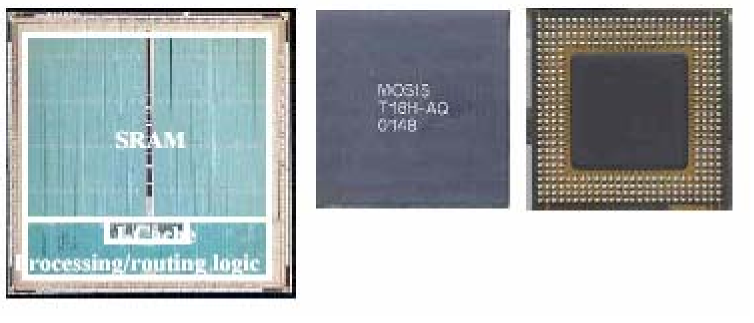


Figure 9. DIVA PIM prototype chip

The preliminary testing was conducted with the use of a custom-built PCB in an incremental fashion. First, with all functional units in reset, we applied power and an input clock signal to test the PLL clock multiplier, IP purchased from NurLogic. The PLL was functional over a wide range of frequencies, voltages, and all possible configurations of input settings. This verification proved that we had successfully integrated IP from a 3rd-party vendor into our design flow. We then proceeded to functional testing with the use of an HP 16702A logic analysis system. Pattern generator modules were utilized to apply test vectors to the inputs of the chip, and timing/state capture modules were used to sense the outputs of the chip. A photo of the lab test setup is shown in Figure 10. The chip was tested for functionality at a testbench speed of 80MHz.

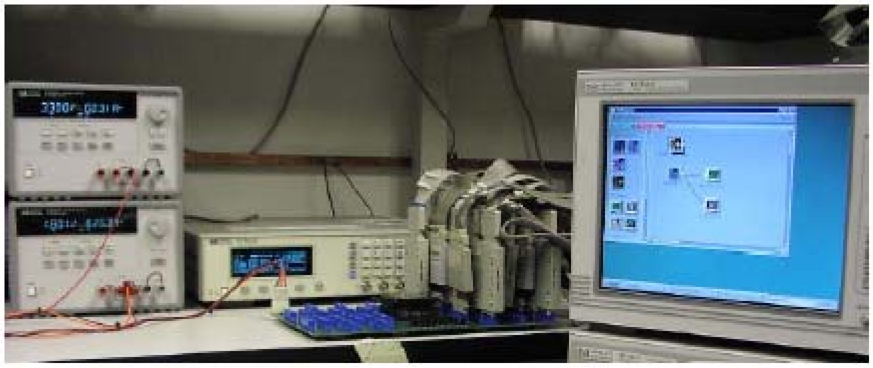


Figure 10. PIM testbench setup

We first verified the operation of the memory access capability of the PIM chip by performing writes/reads to the internal memory through the host memory interface of the PIM chip. After verifying normal memory operation for the lowest 64KB region of memory, we proceeded to PIM processor checkout. The procedure consisted of downloading code through the host memory interface, releasing the PIM processor from reset to execute the code, and then verifying correct operation by reading back results through the host memory interface. After confirming the validity of this debugging approach through a small arbitrary code example, we proceeded to test the execution of the Cornerturn core loop, which had been coded to exploit novel features of the DIVA PIM WideWord Unit. Reading the memory locations that contained the output matrix and verifying that the input matrix had indeed been transposed confirmed successful execution of the code. (The logic analyzer display showing the start of the transposed matrix is shown in Figure 11). We then began some speed testing to determine the clock frequency operating range of the PIM chip. We were able to execute the Cornerturn application at 160MHz while dissipating only 800mW. Even in this limited test setup, the chip achieved a peak 1.28GOPS (32-bit ops) and 5.12 GB/s memory bandwidth. After passing these initial tests, the chip was released to the system integration team where many more results were achieved (refer to the system integration section for details).

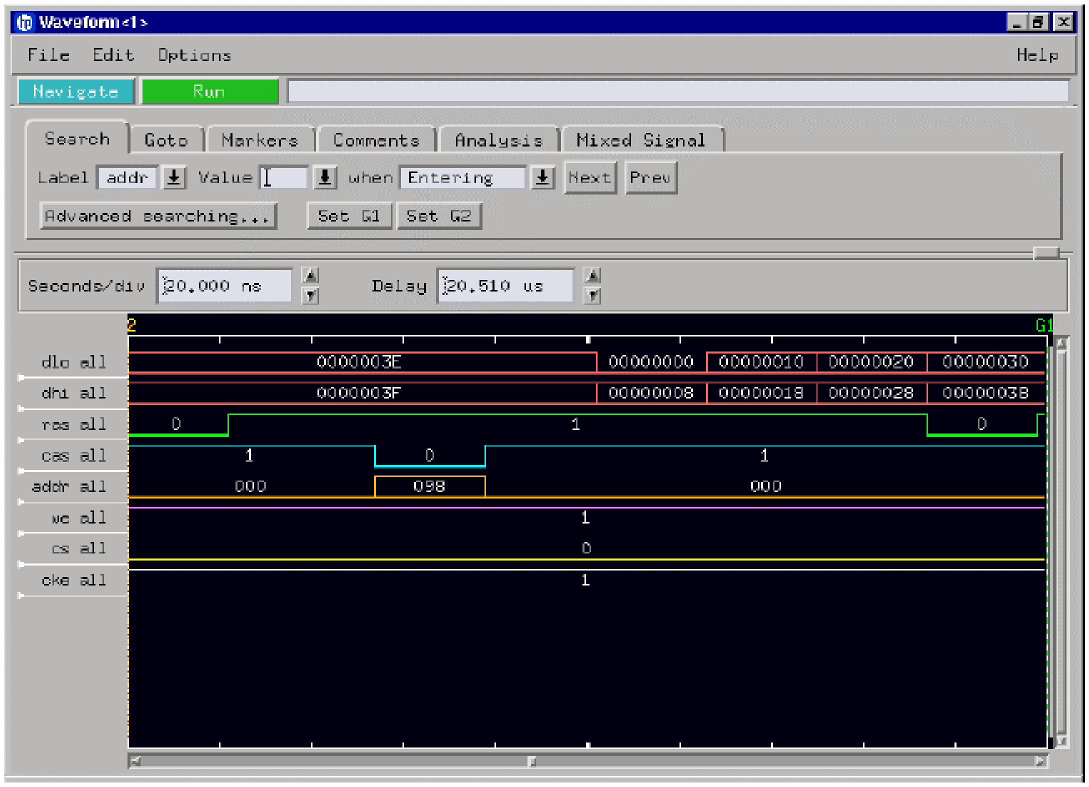

Waveform<1>
File Edit Options Help
Navigate Run
Search Goto Markers Comments Analysis Mixed Signal
Label addr Value when Entering Next Prev
Advanced searching... Set G1 Set G2
Seconds/div 20.000 ns
Delay 20.510 us
dlo all 0000003E 00000000 00000010 00000020 00000030
dhi all 0000003F 00000008 00000018 00000028 00000038
ras all 0 1 0
cas all 1 0 1
addr all 000 098 000
we all 1
cs all 0
cke all 1

Figure 11. Display of read operation "cornerturn" output matrix

4.3 Ongoing and Future Work

While finishing preparations for testing the first chip, we were also working on the designs of the address translation unit and floating point capability for the second turn of the chip. The address translation was completed and integrated into the existing design and validated through simulation, including exception handling related to address faults within a few months. After performing some initial sizing estimates, we realized that we would not be able to fit 4 parallel double-precision floating-point units in our WideWord area budget, so we targeted 8 single-precision units. As technology continues to scale, future PIMs may revisit the possibility of WideWord double-precision capability. Each single-precision unit implements the basic floating-point functions: add, subtract, multiply divide. We used the MIT RAW design as a guideline, but due to DIVA pipeline constraints were not able to use the RAW design as is. We spent most of our time on the design of the divider and then optimizing to merge the subcomponents to share resources that all subcomponents need, such as operand formatting, rounding, and normalization. We selected a divider design based on a Taylor series expansion approach developed by Liddicoat at Stanford [Liddicoat02]. This design achieved a fairly high-performance divide capability while minimizing silicon area. We synthesized the entire FPU (Floating Point Unit) design, and the resulting post-synthesis area projections indicated an area of 0.32 mm^2 for each single-precision floating-point unit, or a total of approximately 2.5 mm^2 for eight such units in the WideWord datapath.

We re-architected the exception-handling unit to accommodate integration of the exceptions from the WideWord floating-point units. Each of the eight single-precision floating-point units of the WideWord datapath reports five types of exceptions: divide by zero, inexact, invalid, overflow, and underflow. The only inconsistency with the IEEE-754 standard is the underflow exception, which we use in place of supporting denormalized numbers and arithmetic. We have combined the overflow and underflow status outputs into one value called precision status so that the resulting 4 exception types of all 8 single-precision FPUs can be contained in one 32-bit register. We have defined a new special-purpose register (SPR) in our architecture to capture this information.

Work is now continuing under separate funding to implement the exception integration and thereby complete the integration of floating-point capability into the DIVA design. Under the HPCS-funded Godiva project, a DDR SDRAM interface is also being added to the rev 2 PIM chip for its insertion into an Itanium2-based HP Long's peak server.

5. Compiler

We have developed a compiler for the DIVA PIM processor that generates optimized code in the DIVA ISA. As will be discussed in the context of system integration, the DIVA compiler backend is based on the Gnu GCC compiler, ported from the PowerPC toolset. GCC is a commonly used optimizing compiler, but it targets conventional scalar instruction sets. To support optimizations targeting the unique bandwidth-exploiting features of the DIVA ISA, we developed front-end compiler technology that performs DIVA-specific optimizations, as captured in Figure 12.

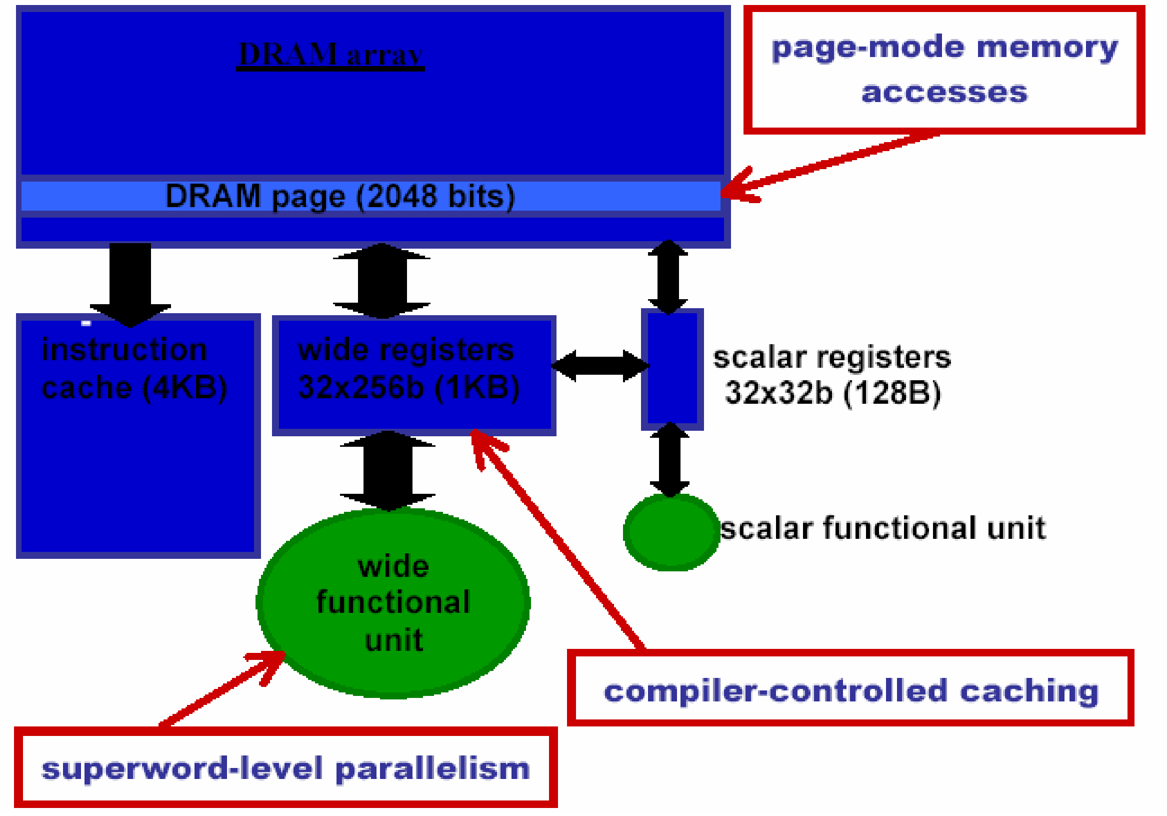

DRAM array
page-mode memory accesses
DRAM page (2048 bits)
instruction cache (4KB)
wide registers 32x256b (1KB)
scalar registers 32x32b (128B)
scalar functional unit
wide functional unit
compiler-controlled caching
superword-level parallelism

Figure 12. DIVA-specific compiler optimizations

In Figure 12, the ovals represent the functional units of the DIVA PIM chip. As has been previously described in the architecture discussion, there are both a 32-bit scalar functional unit and a separate 256-bit wide functional unit. The shaded rectangles in the figure represent on-chip storage. There is the DRAM array, which in today's technology could have up to 32Mbytes, although in our prototype it is a 1Mbyte SRAM array, as previously described. A 4Kbyte I-cache holds the instruction stream, so that memory accesses are predominantly focused on the program data. In addition, there are separate register files associated with each functional unit, a 32-element, 32-bit scalar register file, and a 32-element, 256-bit wide register file.

The unshaded rectangles in the figure point to our compiler's targets of optimization. DIVA's Wide functional unit has operations similar to a multimedia extension architecture such as the PowerPC AltiVec, where the data type is larger than a machine word, and can be configured to perform SIMD parallel operations on different field widths, 8-bit, 16-bit and 32-bit. This type of fine-grain parallelism is referred to as *superword-level parallelism* (*SLP*). Optimizations targeting SLP are the first priority of our compiler. The second priority relates to the WideWord register file, which is 1Kbyte of storage very close to the processor, and the fact that our architecture does not have a data cache. Our target applications that can exploit the bandwidth of the WideWord datapath could also benefit from the increased bandwidth and lower latency of a data cache, as compared to accessing from the DRAM array. For this same class of applications, however, compiler technology can also derive the data access patterns and manage storage explicitly. For this purpose, we have developed new optimizations in the DIVA compiler to support *compiler-controlled caching* in the WideWord register file. Further optimization benefits are obtained from exploiting spatial locality in the DRAM array. When the application accesses memory, the latency of a memory access varies depending upon whether the access is nearby the previous access. The DRAM first selects a *page* or row (assumed to be 2048 bits) and then a 256-bit or 32-bit column within that row. Accesses to the same row as the previous access are referred to as *pagemode accesses*, and have a 3x lower latency than other accesses, which are said to be in *random mode*. Our compiler performs optimizations to maximize the number of memory accesses that are in page mode.

Figure 13 illustrates the components of the DIVA compiler. The DIVA front-end compiler is based on SUIF, a research compiler infrastructure developed at Stanford University. The SUIF-based DIVA front end takes as input a C or Fortran program and generates optimized code in *MrC*, a C-like language with extensions for superword-level parallelism developed for the PowerPC AltiVec. The optimized *MrC* code is the input to the DIVA compiler backend, as shown in Figure 13.

The DIVA compiler backend is based on a superword-extended AltiVec GCC backend available from Motorola. The AltiVec GCC backend takes *MrC* code and generates AltiVec vector instructions similar to DIVA WideWord instructions. To generate DIVA PIM code, we integrated the DIVA GCC backend that previously generated DIVA scalar code only with the AltiVec GCC backend. The final DIVA GCC backend generates code that uses both PIM scalar and WideWord instructions.

Figure 13 shows the DIVA GCC backend and the AltiVec GCC backend for illustration purposes, as both take optimized code from the SUIF-based front-end compiler. The AltiVec backend was a useful tool for testing and tuning optimizations performed by the SUIF-based front-end compiler during the time the DIVA PIM chip was not yet available for software experiments.

The remainder of this section describes the optimizations performed by our frontend compiler, the implementation, and performance results.

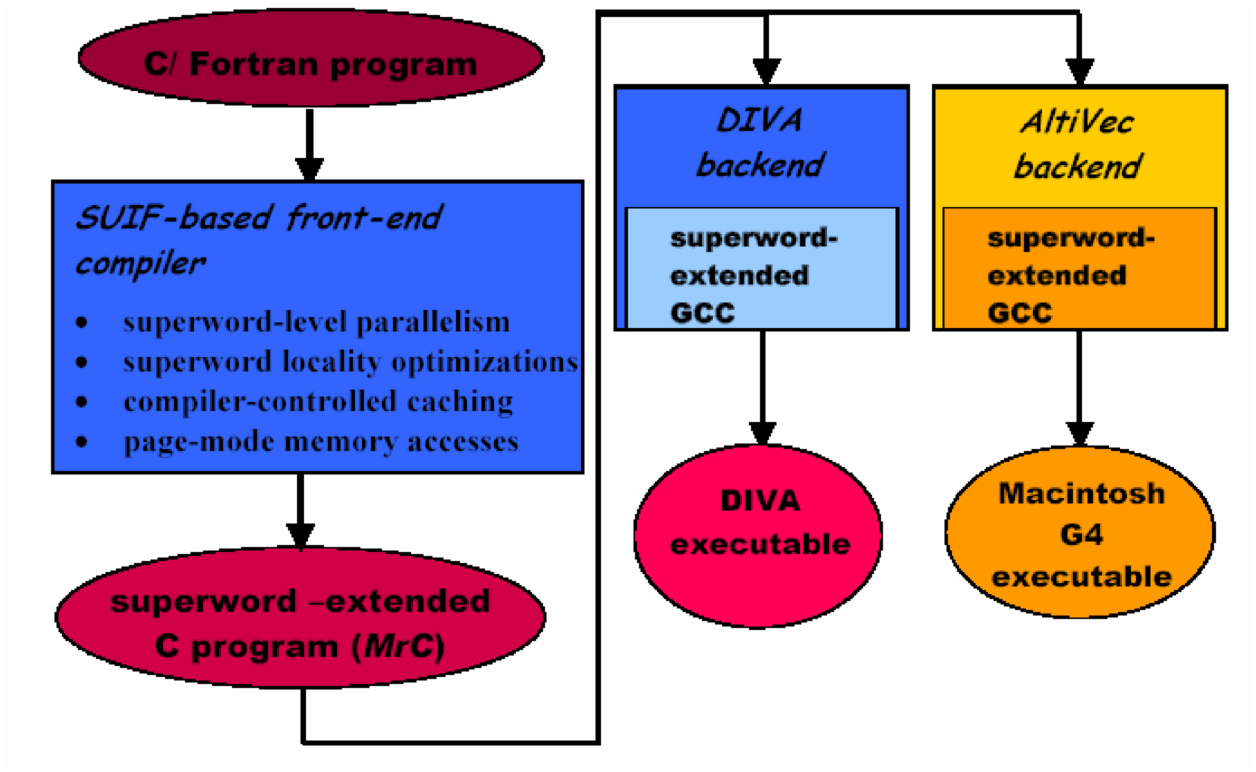

C/ Fortran program
SUIF-based front-end compiler
superword-level parallelism
superword locality optimizations
compiler-controlled caching
page-mode memory accesses
superword –extended C program (MrC)
DIVA backend
superword-extended GCC
AltiVec backend
superword-extended GCC
DIVA executable
Macintosh G4 executable

Figure 13. DIVA PIM Compiler Technology

5.1 DIVA PIM front-end compiler

To develop a DIVA PIM compiler that automatically generates optimized code targeting superword-level parallelism, we have collaborated with Saman Amarasinghe and Samuel Larsen at MIT. The initial MIT SUIF-based compiler automatically recognizes SLP and generates optimized code targeting the PowerPC AltiVec multimedia instructions. The DIVA compiler is built upon the MIT-SLP implementation and generates code targeting DIVA's WideWord instructions.

In addition to superword-level parallelism, the DIVA SUIF-based compiler performs optimizations for compiler-controlled caching in the wide register file. We developed and implemented new analyses for identifying temporal and spatial reuse of data in loop nest computations. Our compiler performs a new optimization called *superword replacement*, whereby accesses to superwords in memory are replaced by accesses to temporary registers, so that the DIVA backend register allocator tries to keep these temporaries in wide registers. This approach adapts related techniques for exploiting temporal reuse in scalar registers, but must also account for parallelism and spatial reuse.

The DIVA SUIF-based front-end compiler automatically generates optimized *MrC* code for six scientific/multimedia benchmarks: TOMCATV and SWIM from the SPEC'95 benchmark suite, and the media kernels VMM (vector-matrix multiply), MMM (matrix-matrix multiply), FIR (Finite Impulse Response Filter) and YUV (RGB to YUV conversion).

We also completed an implementation and experiment in our DIVA compiler to automatically reorder memory accesses to achieve page-mode memory accesses, rather than random-mode memory accesses, and thus greatly reduce memory latency. The compiler unrolls inner loops and reorders memory accesses when there are no data dependencies that prevent doing so, such that accesses within the same page are performed consecutively. On four of the above benchmarks, VMM, MMM, YUV and FIR, we observed speedups ranging from 1.25 to 2.19X on the DIVA simulator, as compared to not performing the reordering of memory accesses. This work has been reported in two publications [Chame00][Shin02b].

Under DIVA funding, we also began an evaluation of requirements to extend MIT-SLP so that it can parallelize more programs of interest, such as the DIS Transitive Closure stressmark and NAS CG. We have identified the need to extend MIT-SLP to support parallelization of constructs containing conditionals for Transitive Closure, and to optimize movement of data between scalar and wide register files, since movement between register files is not supported in the AltiVec.

5.2 DIVA PIM backend compiler

As the AltiVec GCC backend was an experimental and unsupported system, we encountered a number of challenges in merging the DIVA GCC backend with the AltiVec component. Determining which GCC patches to integrate and which to omit required a lot of information gathering and trial-and-error. We successfully completed the integration, and began porting the AltiVec GCC backend to generate DIVA WideWord code. Under DIVA funding, we implemented a subset of DIVA WideWord instructions and the GCC backend generated WideWord code for VMM, a kernel that performs a vector-matrix multiply. The AltiVec version of the compiler has generated code for many more applications, as discussed in more detail below.

We have performed extensive experiments with the optimized code generated by our compiler, for both DIVA and AltiVec. The experiments were performed both in an instruction simulator of the DIVA ISA and in the PowerPC G4 (with an AltiVec). The optimizations for data reuse in WideWord registers result in a reduction in scalar memory accesses of over 90% for the four kernels and over 35% in SWIM and TOMCATV. In addition, we observe a reduction of WideWord memory accesses of over 50% for three of the four kernels, and over 85% in SWIM and TOMCATV. These reductions indicate that even more improvement can be expected on DIVA, where there is no data cache. On the AltiVec, overall we are showing speedups ranging from 1.7X to 12.3X over scalar execution, with an average of 4.2X. Speedups due to our compiler optimizations for compiler-controlled caching go from 1.3 to 2.8, with an average of = 2.2, over the MIT-SLP compiler upon which we base our implementation. This work has been reported in three publications [Chame00][Shin02a] [Shin03].

5.3 Additional Compiler Research

Beyond the node compiler implementation, we planned a long-term strategy for system-level compilation (*i.e., host and multiple PIMs)* that is being pursued under separate funding. As was discussed in the context of the DIVA system architecture, we designed DIVA such that it could be programmed using conventional solutions from parallel computing, rather than requiring a programming paradigm specific to DIVA or to PIMs. As a system-level programming strategy, we have adopted Unified Parallel C (UPC), a relatively new parallel programming language. UPC was developed as a unification of the best ideas among several research C compilers that support a global address space, and allow high-level specification of data distribution in an SPMD (Single Program Multiple Data) abstraction for highend shared-memory, distributed-shared-memory and even distributed-memory parallel systems. The development of the UPC language and its implementations has been motivated by DoD interest and support. There are several commercial UPC compilers, and there are a number of defense applications already written in UPC. We chose UPC for all these reasons, as well as the fact that we can develop DIVA target applications that are pointer-based in a C-based language, but cannot in other parallel programming languages such as, for example, CoArray Fortran.

As part of future work, we are collaborating with Lawrence Berkeley Laboratories and UC Berkeley to develop a UPC compiler for the DIVA prototype. They have an ongoing UPC compiler effort, to develop a portable UPC compiler.

6. System Simulator

We developed a simulator of the DIVA system architecture that was used throughout the duration of the project for several application and architectural studies. Among these studies were the investigation of performance of data-intensive applications on DIVA, the analysis of architectural design trade-offs and bottlenecks and studies that evaluated and provided feedback to the design of the DIVA Instruction Set Architecture (ISA).

The DIVA system simulator (DSIM) uses RSIM (http://rsim.cs.uiuc.edu/rsim) as a framework, with significant extensions. RSIM is an event-driven simulator that models shared-memory multiprocessors built with state-of-the-art multiple-issue, out-of-order superscalar processors. DSIM extensions include a simpler PIM processor with a WideWord unit, the DIVA memory system, the parcel communication mechanism and the PIM-to-PIM interconnect. DSIM supports the DIVA PIM ISA.

The DSIM host processor is taken directly from RSIM, as well as the host first and second-level caches. The host processor architecture is based on the MIPS R10000, which is configured as a four-issue processor with two integer arithmetic units, two floating-point units and one address unit. Loads are non-blocking. It has a 32Kbyte L1 and a 1Mbyte L2 cache, both two-way associative, with access times of 1 and 10 cycles, respectively. Both L1 and L2 caches are pipelined and support multiple outstanding requests to distinct cache lines.

The host is connected to the DIVA memory system via a split-transaction, 64-bit bus. The memory system consists of the aggregation of all PIM memories, where each local memory is visible from both host and local PIM processor. DSIM maintains the current open row of each memory bank to determine the memory access type (page or random mode) and simulates arbitration between host and PIM accesses. The memory latencies seen by the host are 52 cycles for page-mode accesses and 60 cycles for random mode, and include the bus transfer delay, the memory arbitration time and the DRAM access time (4 and 12 cycles for page and random mode, respectively). The memory latencies seen by the local PIM processor, including arbitration and DRAM access times, are 6 and 14 cycles for page- and random-mode accesses, respectively.

DSIM also models the parcel mechanism and the PIM-to-PIM interconnection in detail. Applications executing on DSIM have direct access to the parcel buffers via parcel handling functions that perform the writing/reading to/from the memory mapped parcel buffers. These parcel handling functions are part of DSIM's application library, and support the full set of parcel buffer status reads, triggering/non-triggering writes to the send parcel buffers and destructive/nondestructive reads from the receive parcel buffers.

The application library also supports a cache-line-flush function to enforce coherence between the host caches and PIM memory, and synchronization functions. The functions in the application library are linked with the application code, and their execution is simulated by DSIM as part of the application.

The simulator parameters used in our application studies were based on the conservative assumption that the PIM processor runs at half the speed of the host processor. Although the inherent speed of the logic is no slower, we make this assumption because the WideWord register accesses could impact the clock speed.

7. Application Studies

We performed several application studies, using the DIS Stressmark Suite as well as other data-intensive or high-performance-computing benchmarks, including NAS CG and the template-matching (TM) component of the Sandia ATR benchmark. We first describe the DIVA implementations of the DIS stressmarks, then we present experimental results on the stressmarks and other benchmarks, and later we discuss our earlier application studies.

7.1 DIS Stressmarks

This section contains a description of our implementation of the Cornerturn, Pointer, Transitive Closure and Neighborhood stressmarks. For each of these stressmarks, we describe how the stressmark is mapped to DIVA, including computation and data partitioning, host-and-PIM and PIM-to-PIM communication and synchronization. We also describe how the WideWord unit is used, when applicable (Pointer and Neighborhood do not use the PIM WideWord unit).

Cornerturn.

The DIVA implementation of *Cornerturn* performs a hierarchical matrix transpose, where the matrix is partitioned into blocks and each block is assigned to a PIM node. The transpose of each block is computed by partitioning the block into sub-blocks, which are then transposed in WideWord registers using permutation operations. We present below a simplified implementation, which is valid for square matrices only.

The host performs the initial block partitioning, keeping a table with the assignment of blocks to PIMs, and coordinates synchronization between host and PIMs. In the first phase of the computation, each PIM computes the transpose of its local block. After that each pair of PIMs owning blocks that need to be swapped to form the transposed matrix communicate using the PIM-to-PIM network.

The local block transpose is performed as a set of transposes of 8x8 sub-blocks (except for block sizes that are not multiple of the number of matrix elements that fit in a WideWord register). For the out-of-place transpose, each 8x8 sub-block is loaded into the WideWord register file (an 8x8 matrix with 32-bit elements requiring 8 WideWord registers), and transposed via a sequence of permutation operations. The transposed sub-block is then stored back in memory at the target location. In the in-place transpose (of square blocks) two subblocks of size 8x8 are loaded in WideWord registers, each sub-block is transposed in registers, and then the transposed sub-blocks are stored back in memory, swapping locations to form the transposed block. This implementation takes advantage of the large capacity of the WideWord register file, avoiding loads and stores to memory during the transpose of each 8x8 sub-block.

After computing its local transposed block, each PIM exchanges its transposed block with the PIM that owns the location of the block in the transposed matrix. For example, for a square matrix divided into four blocks where block-00 is assigned to PIM-0, block-01 to PIM-1, block-10 to PIM-2 and block-11 to PIM-3, PIM-1 exchanges its transposed block with PIM-2. PIM-0 and PIM-3 keep their transposed blocks since they should remain in the same location in the transposed matrix.

The communication phase is performed in 2 steps: in the first step PIMs owning blocks in the upper triangular sub-matrix send their blocks to PIMs owning blocks in the lower triangular sub-matrix; the second step completes the exchange of blocks with PIMs in the lower triangular sub-matrix sending blocks to PIMs in the upper triangular sub-matrix.

Finally, this implementation of Cornerturn avoids contention on the PIM-to-PIM network by assigning each pair of blocks that will exchange locations in the transposed matrix to neighbor PIMs. This assignment is based on the fact that communication occurs between fixed pairs of PIMs, and that when assigning a block to a PIM it is possible to determine the location of its transposed block in the transposed matrix, and then assign the block corresponding to this location to the nearest PIM available.

Our HOST version of Cornerturn shows high memory stall times for input sizes that do not fit in the host L2 cache. This application has very little temporal reuse, since each matrix element is accessed a few times only during each matrix transpose. Thus primarily spatial reuse is exploited in cache, and each new cache line is only reused a few times. In the PIM version, the WideWord datapaths also exploit the available spatial reuse. Furthermore, the WideWord loads/stores and operations on eight matrix elements at a time also reduce the number of accesses to memory. Finally, the latency seen by the PIM processor is lower than that suffered by the host for large input sizes. For example, a 1024x1024 matrix is four times larger than the host L2 cache, resulting in memory stall times corresponding to 98% of the host execution time. On the other hand, the 1-PIM version spends 40% of the execution time stalled for memory, due to the lower on-chip latencies and a reduction on the number of memory accesses (the average latency seen by the PIM is 11.6 cycles, since most of the accesses are in random mode).

Transitive Closure
The implementation of Transitive Closure for DIVA is based on the DIS sample code, and uses a dense matrix to represent the distance graph. It exploits both fine-grain parallelism, by performing WideWord arithmetic operations on eight 32-bit elements of the matrix in parallel, and coarse-grain parallelism, by partitioning the data and computation among PIM nodes.

The host processor computes the matrix partition and coordinates synchronization. Matrices *din* and *dout* are partitioned by rows and a set of consecutive rows is assigned to each PIM node. For the main loop nest of Transitive Closure, for each iteration of the outer loop *k*, each PIM node performs the inner-loop computation (loops *i* and *j*) on its local set of rows, using a copy of row *k* previously sent by the PIM that owns row *k*. Therefore, for each iteration of loop *k*, the PIM node that owns row *k* sends a copy of this row to all other PIMs. All PIM nodes synchronize on each iteration of loop *k*, after the communication phase.

The multicast of a matrix row from one PIM to all other PIMs is performed using the multicast mode supported by the DIVA parcel buffer mechanism. The sender processor writes a parcel payload to the parcel buffer, and then writes a parcel header for each destination PIM. The write to the parcel header triggers the sending of the parcel to the specified destination. This multicast mode allows the sender processor to write the parcel payload only once, reducing the cost of assembling parcels in the parcel buffer.

The local computation on each PIM node takes advantage of the WideWord unit in the computation of the minimum value of each pair of elements from two matrix rows. Selective execution using a WideWord operation (*wmrgcc*) merges the contents of two WideWord registers according to condition-code bits, allowing an efficient computation of the minimum value of each pair of elements of two WideWord operands.

Finally, for both the HOST and PIM versions, the inner loops (loops *i* and *j*) of the main loop nest were interchanged, so that the HOST can benefit from spatial locality at the caches, and PIMs can exploit spatial reuse in WideWord registers.

Our PIM implementation benefits from fine-grain and coarse-grain parallelism, and also from the higher bandwidths available on chip. For example, the HOST version for input tc05.in spends 65.2% of its execution time stalled due to cache misses, with 11.3% of the misses satisfied at the L1 and 58.4% satisfied at the L2, resulting in an average memory latency of 6.7 cycles. The 1-PIM version shows a higher average memory latency (9.5 cycles), but it issues less memory accesses, since the WideWord unit is used to transfer data to/from memory and perform the computation. Therefore the 1-PIM memory stall time is smaller than that of the HOST version. The use of the WideWord unit also results in exploiting spatial reuse, since the matrix is accessed with stride one in the row dimension.

Pointer
Our implementation of Pointer is based on the sample code provided by Atlantic Aerospace. We mapped Pointer to DIVA by partitioning both threads and the field array among PIM nodes. To reduce communication costs, PIM nodes are partitioned into groups so that each group has a copy of the array; the size of each group is the minimum number of PIM nodes required to keep

one copy of the array. For example, for a 4 MByte array and 16 PIM nodes, and assuming that each PIM node can keep 2 MBytes of data, the PIMs would be partitioned into 8 groups of 2 PIMs, each group keeping a copy of the array.

Each PIM node is initially assigned a set of threads. Each PIM node starts a thread (from its own set) and proceeds as follows:

1. When a ``hop" is to a location mapped to the PIM, it computes the median and next hop as in the original sample code.
2. When a ``hop" is to a location mapped to a remote PIM node, it sends the ``hop"(in a parcel) to the remote node, which will then continue hoping on this thread.
3. After sending a remote hop out, the PIM checks if it has received any parcels containing ``hops" to be executed locally. If there is a parcel, it goes to step 1.
4. When a thread is completed, the PIM node that executed the last hop marks the thread ``done" and sends a parcel to the PIM that owns that thread signaling that the thread is done.

Finally, the host processor checks for threads that are done and signals the PIMs when all threads are done.

In our experiments, the HOST version performs better than the 1-PIM version when the input size fits in the host L1 or L2 caches (as in p05.in and p20.in). The PIM version performs better than the host version when the input data set fits in one PIM node and does not fit in the host cache (such data is not reported since none of the DIS input sizes satisfies this condition). Our PIM version of Pointer does not speedup when the array must be partitioned among PIMs. The main reason our Pointer does not scale well is that the rate of communication per hops is very small, and the local computation (an average of a couple of hops) is not enough to amortize the cost of PIM-to-PIM communication.

Neighborhood

The Neighborhood implementation on DIVA exploits coarse-grain parallelism by partitioning the computation among PIM nodes. Each PIM computes a partial histogram locally, and at the end of the computation phase, the PIM nodes perform a parallel reduction to compute the final histogram. The parallel reduction takes n-1 steps, where n is the number of PIM nodes. The communication is scheduled to take advantage of the PIM-to-PIM interconnection topology (bi-directional ring), avoiding contention in the network.

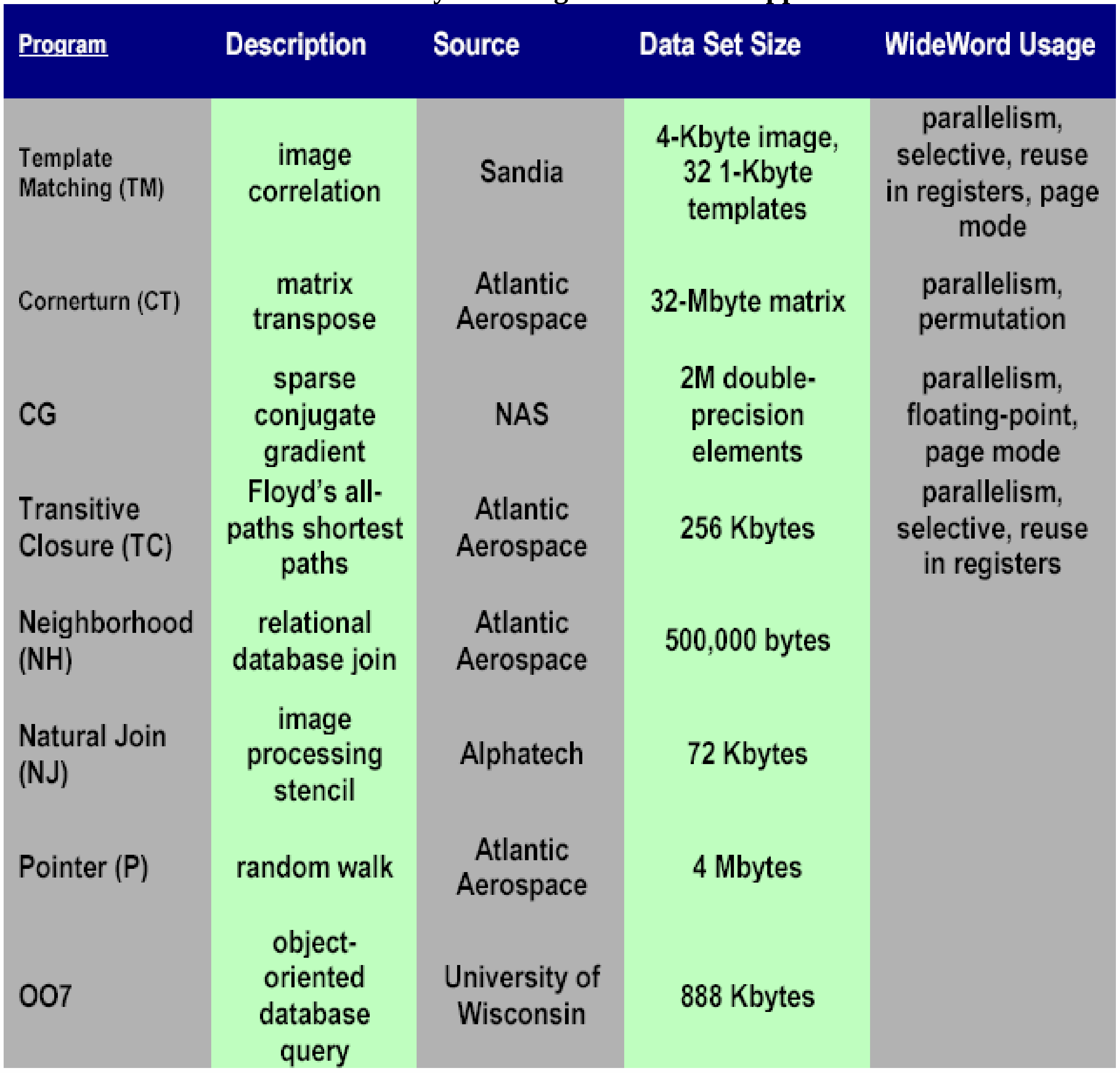
The 1-PIM version of Neighborhood performs worse than the host version when the image fits in the host L2 cache, for several reasons: the memory latencies seen by the PIM are larger than the L2 access time; the PIM nodes operate at half the speed of the host; and our implementation of Neighborhood does not take advantage of the WideWord unit. When coarse-grain parallelism is exploited by partitioning the computation among several PIM nodes, the PIM version speeds up considerably with respect to the host.

7.2 Experimental evaluation

1-PIM performance

To measure the performance potential of the DIVA architecture, we examine in detail eight benchmark applications, summarized in the Table 1.

Table 1. Summary of the eight benchmark applications



Program	Description	Source	Data Set Size	WideWord Usage
Template Matching (TM)	image correlation	Sandia	4-Kbyte image, 32 1-Kbyte templates	parallelism, selective, reuse in registers, page mode
Cornerturn (CT)	matrix transpose	Atlantic Aerospace	32-Mbyte matrix	parallelism, permutation
CG	sparse conjugate gradient	NAS	2M double-precision elements	parallelism, floating-point, page mode
Transitive Closure (TC)	Floyd's all-paths shortest paths	Atlantic Aerospace	256 Kbytes	parallelism, selective, reuse in registers
Neighborhood (NH)	relational database join	Atlantic Aerospace	500,000 bytes	
Natural Join (NJ)	image processing stencil	Alphatech	72 Kbytes	
Pointer (P)	random walk	Atlantic Aerospace	4 Mbytes	
OO7	object-oriented database query	University of Wisconsin	888 Kbytes	

These applications span a broad range of domains including scientific computing, databases and image processing. They exhibit both coarse grain parallelism (which allows computation to be spread across PIMs) and, in some cases, fine grain parallelism (which can be exploited through execution in the WideWord unit). CG, Neighborhood, Pointer, OO7 and Natural Join exhibit irregular or mixed (regular and irregular) data access patterns, resulting in high memory access overheads on conventional architectures. Cornerturn, Transitive Closure and Template Matching are dense matrix computations with regular access patterns, although memory bandwidth becomes a limiting factor in exploiting the significant available parallelism. These three and CG rely on the WideWord unit to exploit parallelism and PIM bandwidths. Hereon, we use abbreviations for each of the program names, with a suffix -H for host and -P for PIM.

The graph in Figure 14 summarizes 1-PIM performance as compared to execution on the conventional host processor. Five of the eight programs speed up significantly compared against host execution, two remain about the same, and one program is slowed down. (All programs speed up when multiple PIMs are used.) Overall, the average speedup is 3.39X.

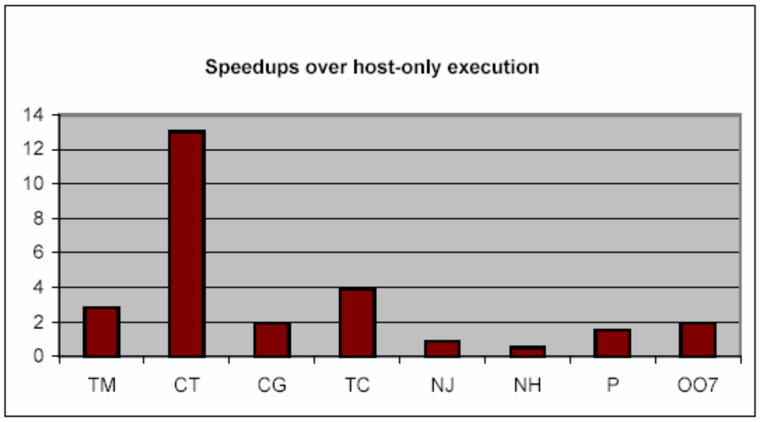


Figure 14. Summary of 1-PIM performance relative to host

Several factors contribute to these speedups, including the lower memory stall times on the PIM nodes and the benefits of the WideWord unit in exploiting fine-grain parallelism and taking advantage of page-mode memory. These factors are discussed in detail in the subsections that follow.

Reduction in Memory Stall Time

To illustrate the impact of memory latencies on the applications' total execution times, Figure 15 shows the busy and memory stall components of host only execution. We see from the figure that five of the eight programs spend more than 40% of their time stalled in memory accesses.

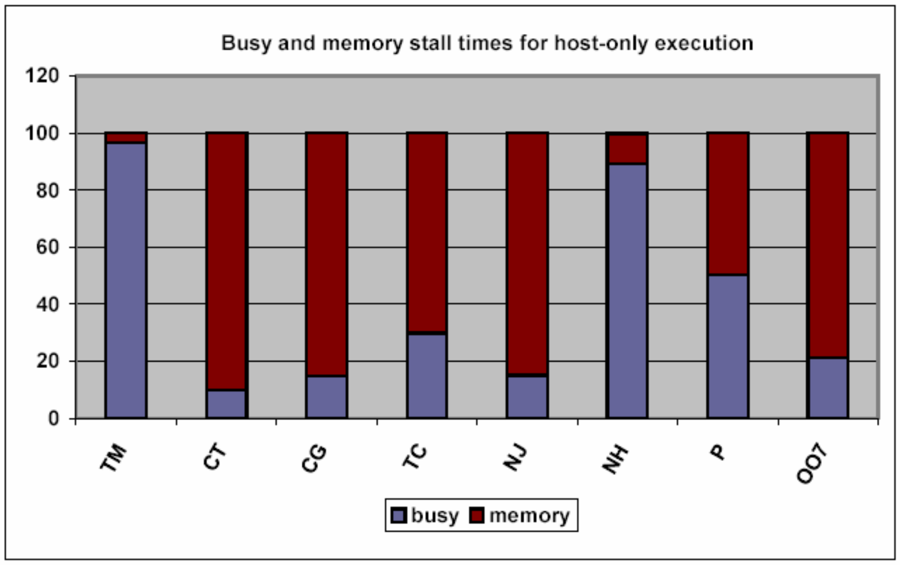


Figure 15. Host-only busy and memory stall times for the eight programs

PIMs reduce memory stall time in two ways: (1) lower latency to memory; and, (2) higher bandwidth to memory through wide loads and stores. (A third reduction occurs as a result of coarse-grain parallelism across the PIMs.) DIVA achieves a reduction in memory stall time for these five programs ranging from 13.89% for Natural Join to 95% for Cornerturn, as shown in Figure 16.

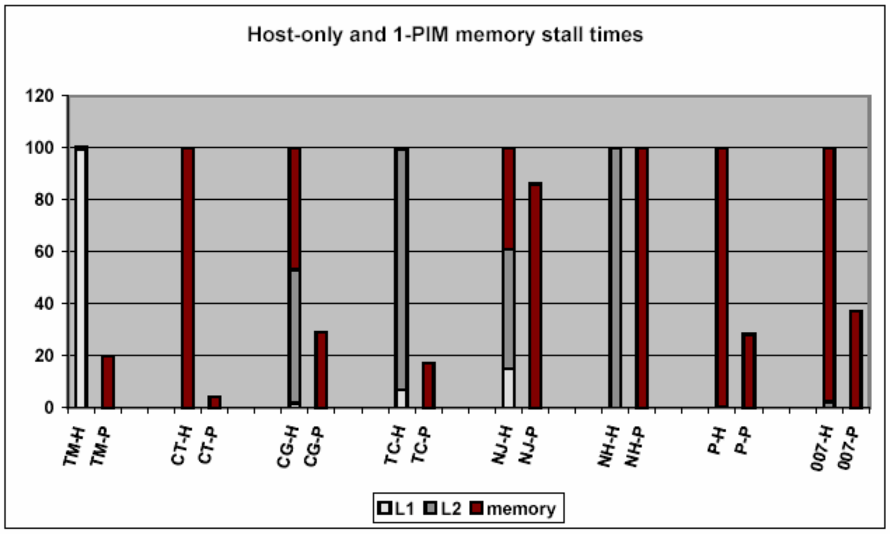


Figure 16. Memory stall times of host-only and 1-PIM execution

The host version of Template Matching (TM-H) has a memory stall time of only 3% of its total execution time. The reason is that the data set size fits in the L2 host cache and the working set of each loop fits in the L1 cache, and therefore the data reuse exhibited by TM is effectively exploited. Even though TM-H does not suffer from large memory stall times, the 1-PIM version (TM-P) has even smaller stall times due to the high data bandwidth at the PIM node. The use of the WideWord unit for loading/storing and operating on 256-bit objects, plus the reuse of data in WideWord registers reduces the memory stall time to 20% of that of TM-H.

Cornerturn has a memory stall time of 90.17% when running on the host. This application has very little temporal reuse, since each matrix element is accessed only twice (one read and one write) during the matrix transpose. Thus primarily spatial reuse is exploited in cache, and each new cache line is only reused a few times (1 load and 1 store per element, and 8 elements per cache line) once loaded, and then never used again. In the PIM version, the WideWord datapaths also exploit the available spatial reuse. Furthermore, the WideWord loads/stores and operations on 8 matrix elements at a time also reduce the number of accesses to memory.

Finally, the latency seen by the PIM processor (average of 11.57 cycles, since most of the accesses are in random mode) is much lower than that suffered by the host. The combination of these factors reduces the CT-P memory stall time to 4.32% of that of CT-H.

CG also benefits from the lower memory latencies on the PIM node. Since the data set size does not fit in the host caches and the irregular access patterns cause conflict misses, CG-H spends 85.21% of its execution time stalled due to cache misses. Although most of the misses are satisfied at the L2 cache (51.32%), 46% of the stall time is due to accesses to the DRAM. On the

PIM, 78% of the memory accesses are page-mode accesses, and the average latency seen by the processor is only 5.91 cycles.

TC-P benefits from both fine-grain parallelism and the higher bandwidths available on chip. TC-H spends 70% of its execution time stalled due to cache misses, with 47.14% of the misses satisfied at the L1 and 52.81% satisfied at the L2, resulting in an average miss latency of 6.23 cycles. On the PIM version, the average memory latency is of 5.57 cycles, due to 67% of page-mode accesses. In addition to lower memory latencies, TC-P also has a smaller number of memory accesses since the WideWord unit is used to transfer the data to/from memory and perform the computation. Therefore the memory stall time of TC-P is smaller than that of the host version. The use of the WideWord unit also results in the added benefit of exploiting spatial reuse; since the matrix is accessed with stride one in the row dimension.

Neighborhood shows an increase in memory stall time because the data fits in cache, and thus the memory latency at the PIM is larger than that of the host. This increase in memory stall time and the fact that the PIM processor runs at half the speed of the host results in a slowdown with respect to host-only execution.

Pointer has no spatial reuse and little temporal reuse, and since the data set size is larger than the L2 cache, P-H stalls for memory for 49.8% of its execution time, with most misses satisfied at the DRAM. P-P has roughly the same number of loads and stores, but the average latency seen by the PIM is much smaller than the memory latency suffered by the host, even though most of the PIM accesses are random-mode accesses.

Natural Join has little temporal reuse and high cache miss rates, even though the data set size fits in the L2 cache. NJ-P shows a reduction of 13.8% in memory stall times due to the lower average latency seen by the PIM processor. OO7 also has almost no temporal reuse and OO7-H suffers from a large amount of cache misses. On the PIM version the memory stall time is reduced by 62.8%, again as a result of the smaller on-chip latency.

Benefits from WideWord Unit and Page Mode Memory Accesses.
To isolate the benefits of the WideWord unit, we compare scalar versions against versions tuned to take advantage of the WideWord unit and page-mode memory accesses for the four programs that utilize the wide datapaths. These results are shown in Figure 17. Speedups are significant, ranging from1.19X for CG up to 17.96X for TM, with an average improvement of 9.93X.

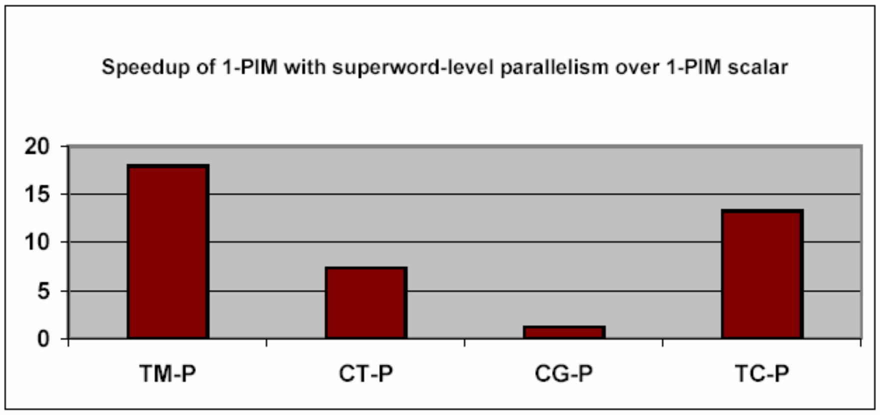


Figure 17. Benefits of WideWord instructions and page-mode memory accesses

CG's key computation is a sparse matrix-vector multiply. Due to the mixed regular/irregular nature of data accesses, we only exploit fine-grain parallelism in the WideWord unit for the regular portions of the computation. The dense vector accesses are loaded into WideWord registers, and the dense vector multiplies are performed in the WideWord floating-point unit. The accumulates into the sparse matrix are performed sequentially. Selective execution is used to select the field of the WideWord operand that participates in the operation. Further performance improvements are obtained by reordering memory accesses, grouping streaming accesses to the dense arrays to achieve page mode memory access latencies.

The CT implementation performs a hierarchical in-place matrix transpose where the smallest submatrices, of size 8x8, are transposed in WideWord registers. Each 8x8 submatrix is loaded into the WideWord register file (an 8x8 matrix with 32-bit elements requiring 8 WideWord registers), and transposed via a sequence of permutation operations. The transposed submatrix is then stored back in memory. This implementation takes advantage of the large capacity of the WideWord register file, avoiding loads and stores to memory during the transpose of each 8x8 submatrix.

TM computes three correlation values between an image and each of 32 templates, each correlation corresponding to a loop nest. The DIVA implementation, which is described in detail in [chame00], takes advantage of the inherent fine-grain parallelism by operating on 32 8-bit image pixels and 32 8-bit template elements at a time. Since a template is represented as a 32-by-32 matrix of 8-bit elements, an entire template row fits into one WideWord register. Also, since the innermost loop of each loop nest traverses one template row, the entire inner loop computation is transformed into a sequence of WideWord operations on one template row and 32 pixels of an image row, therefore eliminating the innermost loop. The accumulation of the pixel values is achieved by a parallel reduction sum, and the result of the reduction sum is added to the correlation value using selective execution. To exploit temporal reuse in WideWord registers, we applied common loop transformations, particularly unroll-and-jam. In addition, we exploited spatial reuse by shifting an image subrow held in a WideWord register by one pixel, to move the window of the image to be compared against the template. As in CG, we also reordered memory accesses to achieve page mode latencies.

TC uses a dense matrix to represent the distance graph. It exploits fine-grain parallelism by performing WideWord arithmetic operations on eight 32-bit elements of the matrix that are held in WideWord registers. Selective execution using a WideWord operation (*wmrgcc*) merges the contents of two WideWord registers according to condition-code bits, allowing an efficient computation of the minimum value of each pair of elements of two WideWord operands. Similar to TM, we use unroll-and-jam to obtain temporal reuse in the WideWord register file.

Overall Speedups

In Figure 18, we present speedups for four benchmarks, using the DIVA system over executing the applications on the host processor. Our experiments show significant improvements over the host-only execution for the three DIS stressmarks (Transitive Closure, Cornerturn and Neighborhood) and NAS CG, with speedups ranging from 19.4X to 39.5X on a 64-node system. These high speedups are in spite of the fact that the PIM processors are running at half the speed of the host, and are in-order, single-issue, vs. out-of-order, 4-issue for the host.

Our CG implementation performs a parallel reduction to accumulate partial results computed locally by each PIM processor. During this parallel reduction phase, a PIM node sends its local copy of the result array to another PIM node. This transfer of a large amount of data to a same destination processor is well suited for the streaming mode supported by our parcel mechanism. In Transitive, there is a communication phase on each iteration of the outermost loop of a 3-deep loop nest. During this phase, one PIM processor sends its local copy of a matrix row to all other processors executing the parallel application. This communication pattern can take advantage of the multicast mechanism supported in DIVA. Similarly, Neighborhood exhibits communication patterns that can take advantage of the streaming parcel mode.

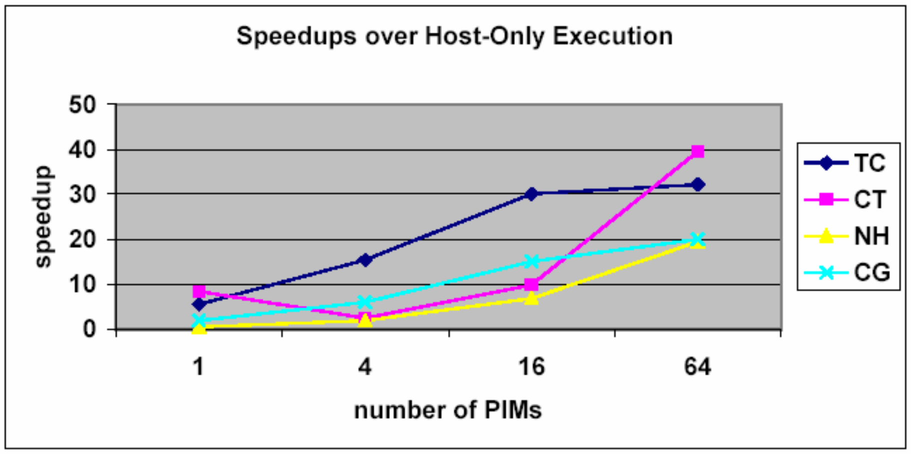


Figure 18. Speedup on four benchmarks as a function of the number of PIMs

7.3 Earlier Application Studies

At the initial phase of the project, we derived a set of benchmarks that could be used for evaluation purposes throughout the project. This initial set consisted of six benchmarks selected from well-known scientific benchmark suites (NAS, Splash-2), pointer-based and database benchmarks (Sparse from McGill and OO7 from University of Wisconsin), as well as the template-matching component of Sandia's ATR application, and the Munkres benchmark provided by Alphatech.

To evaluate the design of the DIVA ISA, we performed experiments using this set of benchmarks. One of the goals of the experiments was to identify useful permutation patterns for rearranging data in the PIM wide registers, using the wide unit permutation network. The DIVA PIM ISA supports efficient permutation operations for a set of frequently used permutation patterns; this application study identified frequently used permutation patterns, such as data shifting, reductions, sorting, gather and scatter, which were integrated into the DIVA PIM ISA.

In another experiment, we performed simulations on the template-matching component of Sandia’s ATR to evaluate the benefits and trade-offs of the WideWord datapaths. Using WideWord operations for exploiting fine-grain parallelism and data reuse in the WideWord registers, we obtained a 13x reduction in the number of dynamic instructions and a 300x reduction in the number of dynamic memory accesses. These improvements led to an overall speedup of 38.3 on a system with 32 PIMs.

We demonstrated a speedup of 20.6x on the NAS CG benchmark, over execution on a high-end workstation based on the MIPS R1000. Several architecture features of DIVA contributed to these speedups: the lower memory latencies on PIM chips, the PIMs wide datapaths for parallel memory operations and efficient communication, and a WideWord floating-point unit that allows four double floating-point operations to be performed in parallel. For these experiments, we modeled in the simulator a WideWord floating-point unit capable of performing four double precision floating-point operations (our second DIVA chip supports eight single precision floating-point operations performed in parallel).

We performed an initial mapping of three of the DIS benchmarks (Image Understanding, Ray Tracing and Method of Moments) to the DIVA architecture, including data and computation partitioning between host and PIM processors, parallelization (coarse- or fine-grain), and data locality optimizations. We did not complete our studies of the DIS benchmarks, since soon after performing the mappings, the DIS stressmarks were introduced and became the benchmark suite used by all the DIS projects. We subsequently concentrated our resources on experimenting with the DIS stressmarks. As a result, we did not produce performance results for the benchmarks. Nevertheless, for archival purposes, we include the most interesting aspects of the mappings here. We spent the most time on Image Understanding, which has three core computations: a Morphological Filter that compares a kernel to an image, Region Selection based on results of filtering, and Feature Extraction that identifies features within the regions. The first of these was handcoded to use DIVA's WideWord unit. The second, which accounted for only a small amount of the sequential computation, was performed on the host processor. The third part is executed in the DIVA PIMs. For Ray Tracing, we obtained good parallel speedups by replicating a small object database on each PIM and performing the screen pixel computation in a cyclic fashion. If instead the object database is large and replication is not feasible, the costs of frequent irregular communication would dominate performance.

8. Emulator

8.1 Hardware

As part of the DIVA architecture development, an FPGA-based emulator was constructed to provide an early platform for software development and demonstrations. This effort produced two versions of hardware in response to track developments and requirements emerging from the primary architecture effort.

The DIVA emulator is a single-board peripheral device designed to plug into a commercial Linux PC system. It is based on commercial Xilinx Field- Programmable Gate Arrays (FPGAs) and may be configured to support a wide variety of applications beyond the emulation of DIVA processors. The emulator is designed to support rapid configuration as a DIVA PIM processor for executing DIVA programs, however, it is also a general-purpose FPGA engine capable of supporting a wide range of hardware modeling applications. Table 2 summarizes the hardware features of the emulator.

Table 2. Emulator Hardware Features

FEATURE	DESCRIPTION	COMMENTS
FPGAs	Xilinx Virtex XCV2000E	100% available for user configuration
Logic capacity	10 million gate equivalents	Subject to logic usage & routing complexity
Memory – DRAM	4 megabytes per FPGA	EDO DRAM – 70 ns access
Memory – SRAM	256 kilobytes per FPGA	15 ns access
Memory – FPGA	192 kilobytes per FPGA	Internal – subject to logic utilization
Power management		Per FPGA/DRAM/SRAM
Bus interface	PCI v2.1 compliant	AMCC 5920
Expansion	192 channels	Inter-board ribbon cable
FPGA configuration	From host computer	Requires Linux driver

As is shown in Figure 19 of the first version of the DIVA emulator, the emulator circuit is constructed on two printed circuit boards stacked to form a thin sandwich. The emulator meets PCI physical size restrictions, even with components mounted on both sides of the two boards.

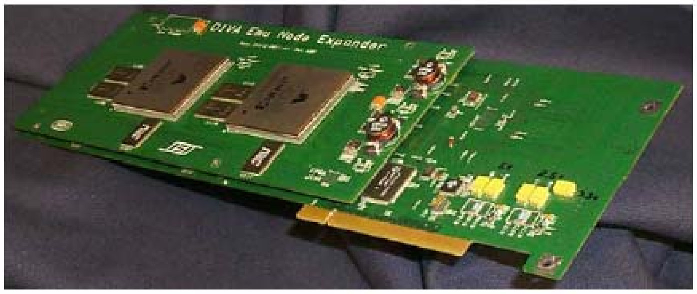


Figure 19. Photograph of emulator board

In addition to the FPGAs, DRAM and SRAM memories, and PCI bus interface ASIC (Application Specific Integrated Circuit), the main emulator board also contains a small Atmel microcontroller used for power control and FPGA thermal monitoring, and voltage regulation circuits to supply the FPGAs with power. The Atmel microcontroller can communicate with the host system via a “mailbox” in the PCI interface ASIC, enabling the host to issue commands for power control and clock rate generation. Figure 20 depicts how the emulator board components are interconnected, and can be used as a guide for partitioning new logic designs so they can best fit the available resources.

The on-board power regulation circuit delivers 1.8 VDC and 2.5 VDC to the FPGAs and other on-board devices. The 1.8 V level is used for powering the FPGA internal circuits, while the 2.5 V rail is used to supply power to the input/output pins of the FPGAs, memories, and PCI interface ASIC.

8.2 Software

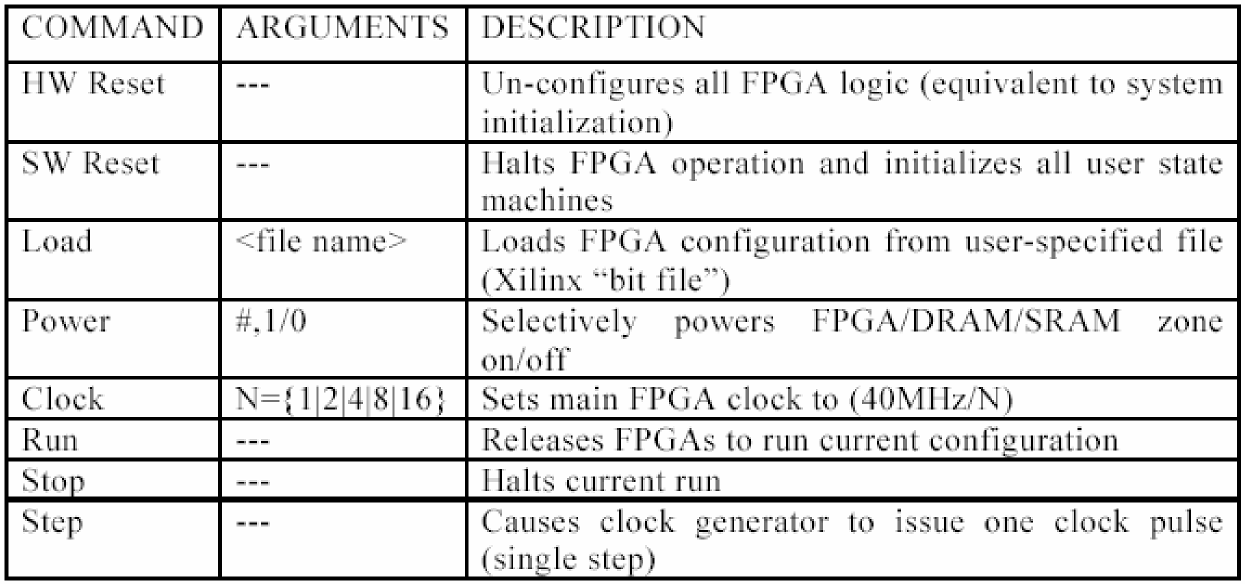
8.2.1 Linux Driver

The Linux driver for the emulator is written to be compatible with RedHat Linux v7. The driver provides interrupt-handling code (not used in DIVA emulations) plus basic services – device open, read, write, etc. – Required by applications programs such as the user command program.

8.2.2 User Command Program

The emulator user control program is a simple application that provides the user with a simple set of commands to control the emulator board. Table 3 is a short description of the commands available to users.

Table 3. User control commands



COMMAND	ARGUMENTS	DESCRIPTION
HW Reset	---	Un-configures all FPGA logic (equivalent to system initialization)
SW Reset	---	Halts FPGA operation and initializes all user state machines
Load	<file name>	Loads FPGA configuration from user-specified file (Xilinx “bit file”)
Power	#,1/0	Selectively powers FPGA/DRAM/SRAM zone on/off
Clock	N={1|2|4|8|16}	Sets main FPGA clock to (40MHz/N)
Run	---	Releases FPGAs to run current configuration
Stop	---	Halts current run
Step	---	Causes clock generator to issue one clock pulse (single step)

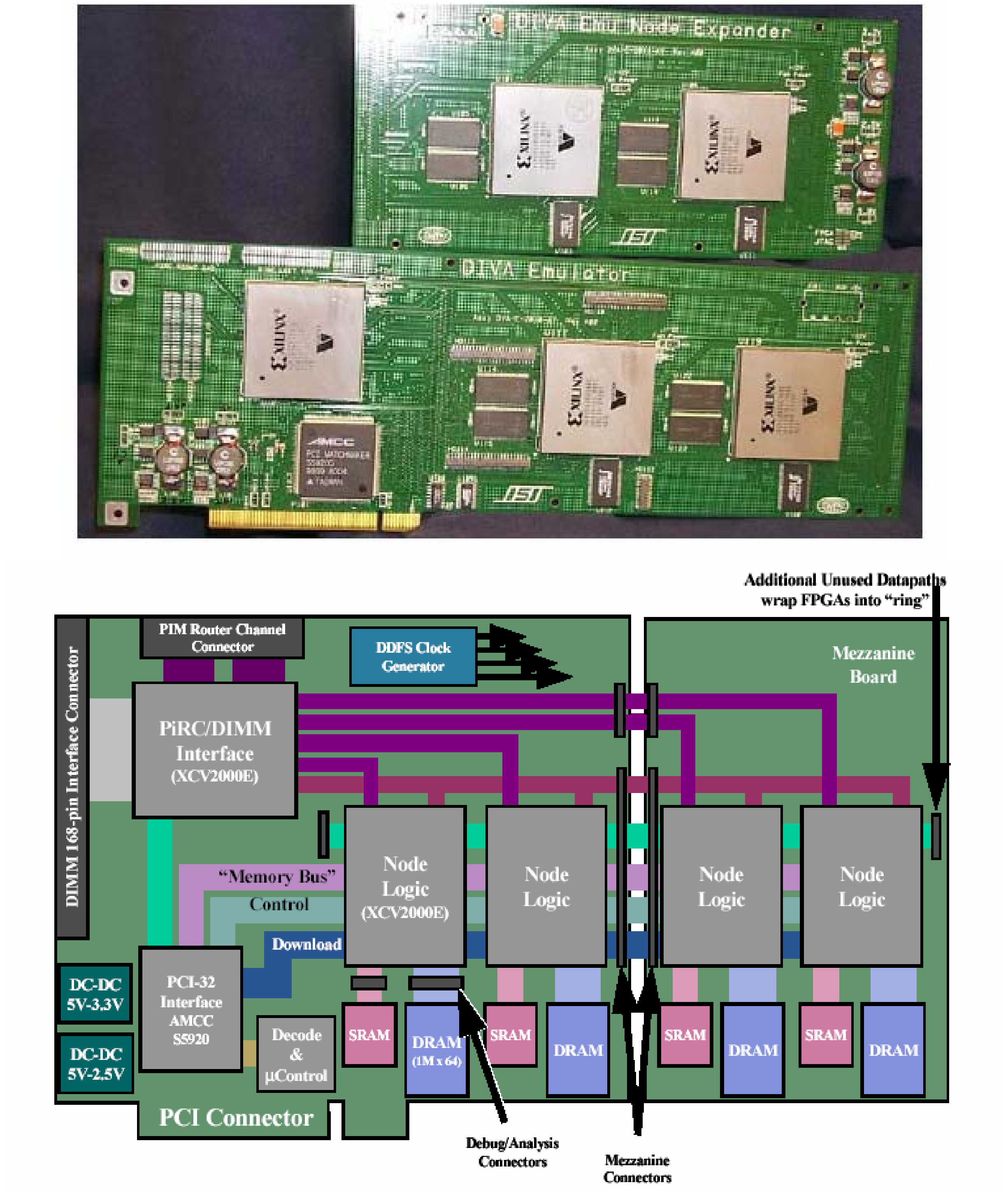
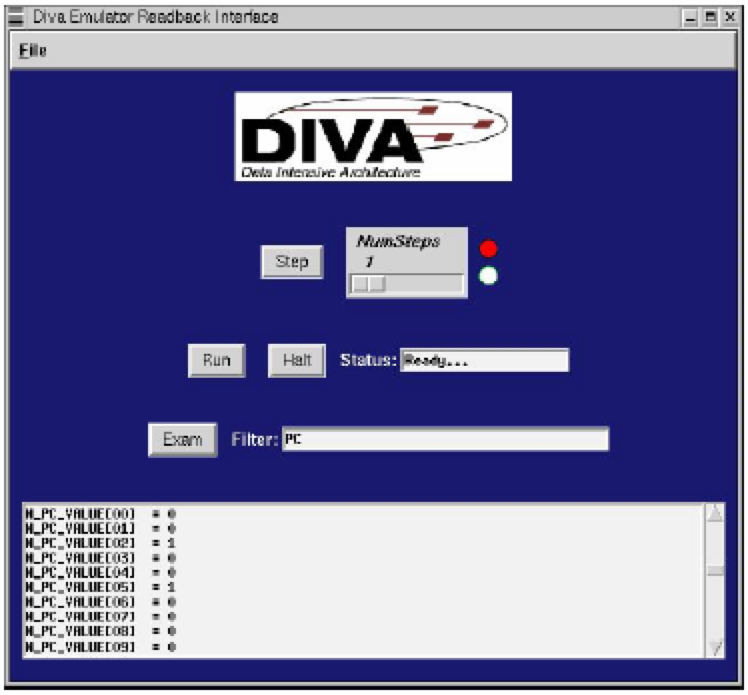


Figure 20. Schematic and photograph of emulator board interconnect details

8.2.3 Graphical User Interface

Figure 21 shows the user command program display panel. The underlying text-only command interface has been overlaid by a simple graphical interface that allows the user to control the operation of the emulator, including single- or multiple-clock execution stepping, and a display panel to report the contents of registers and memory locations within the emulated processor.



Status: Ready...

Filter: PC

```
N_PC_VALUE[00]  = 0
N_PC_VALUE[01]  = 0
N_PC_VALUE[02]  = 1
N_PC_VALUE[03]  = 0
N_PC_VALUE[04]  = 0
N_PC_VALUE[05]  = 1
N_PC_VALUE[06]  = 0
N_PC_VALUE[07]  = 0
N_PC_VALUE[08]  = 0
N_PC_VALUE[09]  = 0
```

Figure 21. Emulator GUI

8.3 Edge Detect Demonstration

The emulator was used to demonstrate execution of a simple DIVA program for edgedetection (Sobel filtering) in a small (256x256 pixel) image. While simple in construction, this program requires the execution of over two million DIVA instructions to complete. The photographs in Figure 22 are typical of images used in the demonstration, and the corresponding results of edge detection. Changing the threshold value used to determine the presence of an edge, or light/dark transition can reduce the amount of "clutter" visible in the result.

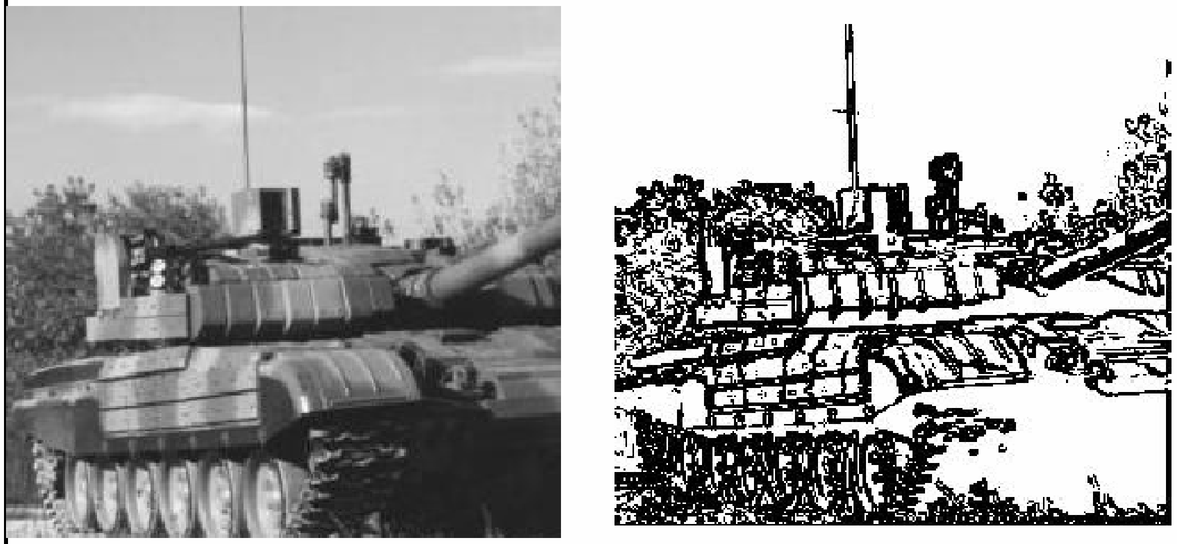


Figure 22. Test image input (left) & Sobel-filtered output image (right)

In this demonstration, the host system loaded the DIVA PIM program into memory– SRAM – on the emulator card. The input image was loaded into PIM storage –DRAM – by the host system. The emulator was directed to run the program, which used the original in PIM storage to generate results that were placed in another region of DRAM. When execution completed, the host could read the results directly from PIM storage and display it in a window for viewing. The edge detect program required approximately one second to execute.

8.4 Lessons Learned

Several valuable lessons were learned during the development of the emulator.

8.4.1 Nominal Clock Rate Isn't

According to Xilinx, the DIVA emulator was the first design to use the XCV1000 devices. It soon became apparent that the FPGAs would not support the initial target of 40-megahertz clock speed – the FPGA wiring resources would not consistently propagate signals. In fact, Xilinx provided special wiring paths to propagate critical signals over long distances within the FPGA. Unfortunately, these wiring paths constituted less than ten percent of the available wiring resources, requiring that every new FPGA design be hand placed and routed for efficiency. As a result, the nominal clock rate of the emulator was reduced to ten megahertz.

8.4.2 Partitioning Across FPGAs Is A Hard Problem

As the architecture evolved, it became apparent that a PIM processor with a full WideWord datapath would not fit in a single XCV1000. This forced a large amount of effort to be expended in partitioning the node across two FPGAs: one for the scalar (32-bit) datapath and the instruction pipeline, one for the WideWord datapath.

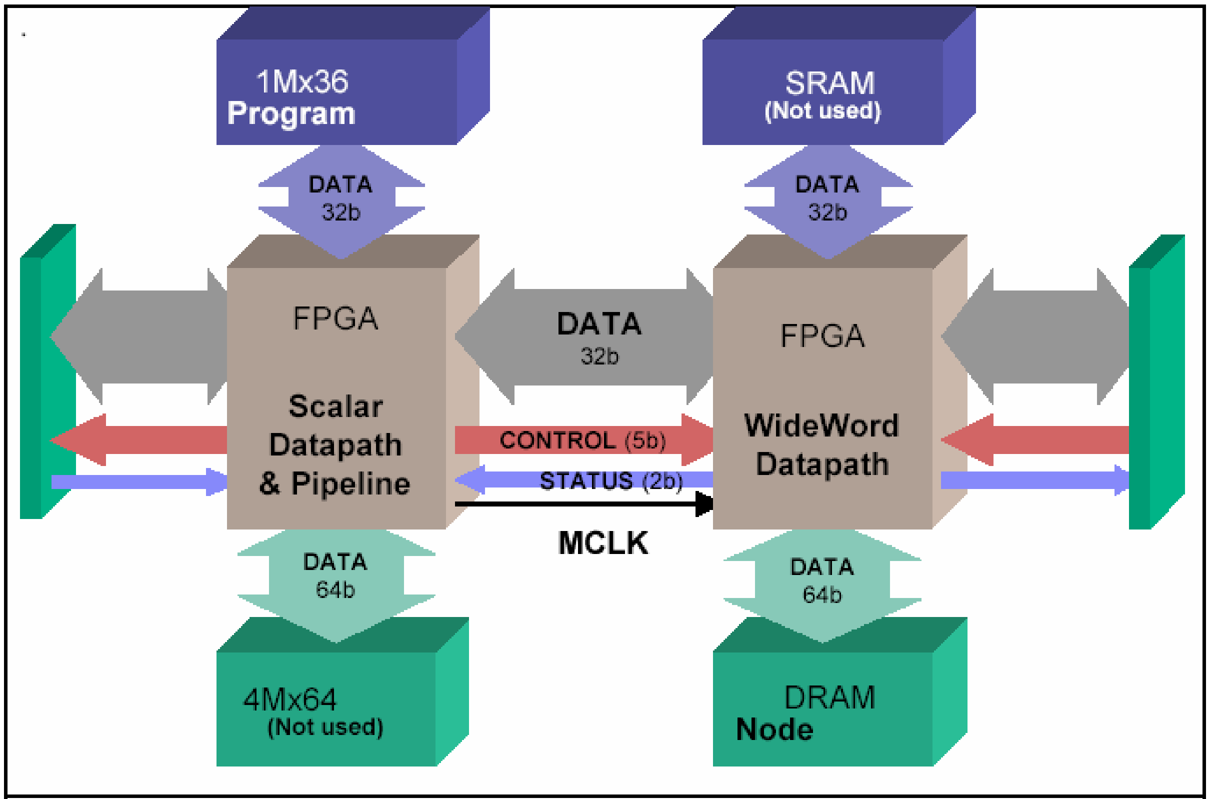


Figure 23. PIM node architecture partitioning across two Xlinx FPGAs

Figure 23 shows how the PIM processor was partitioned across the emulator FPGAs and other board-level resources. First, while the emulator effort as started at the beginning of the DIVA effort, the evolving nature of the architecture made it very difficult to anticipate the eventual logic requirements of the ASIC. The first version of the emulator was built with Virtex XCV1000 devices, which claimed to deliver a capacity of one million logic gate equivalents. As DIVA was originally conceived, this would have been more than adequate to configure a full DIVA PIM processor – indeed; this was the reason four copies of the FPGA/DRAM/SRAM cluster were implemented on a single board

8.4.3 FPGA Tools Are Not Robust (WideWord Impact)

After the scalar 32-bit processor was demonstrated with the edge-detect program, the WideWord (256-bit) datapath design was begun. This design was simplified by the fact that the scalar datapath could be replicated and modified to implement the variable word width features of the WideWord instructions. This modified datapath was then copied eight times to produce the WideWord logic. At this point in the design the design tools distributed by the FPGA manufacturer, Xilinx, broke, and did so in unpredictable ways. Compilation runs would freeze, abort at random points in the process, or would refuse to begin. Incomplete runs would not produce any output data, so it was essentially impossible to determine what aspect of the design was causing the failure.

Although Xilinx responded to some of these errors with additional releases of software, we did not receive the level of support required to work through these problems. It was decided that the design of the WideWord unit would have to be further partitioned to get any design to complete.

8.4.4 Cycle Accuracy Requires More Clock Cycles

The basic operating requirement for the emulator was to provide cycle-accurate results. That is, at the end of every clock cycle, every register should contain correct results. This requirement enabled the emulator design to be further partitioned so that the WideWord could be represented by four 64-bit datapaths, each executing the current instruction in one quarter of the pipeline clock. This partitioning drove the final execution speed of the emulator to 2.5 MHz, which is still very acceptable when compared to software simulations. Figure 24 depicts the basic pipeline clock partitioned into eight microcycles.

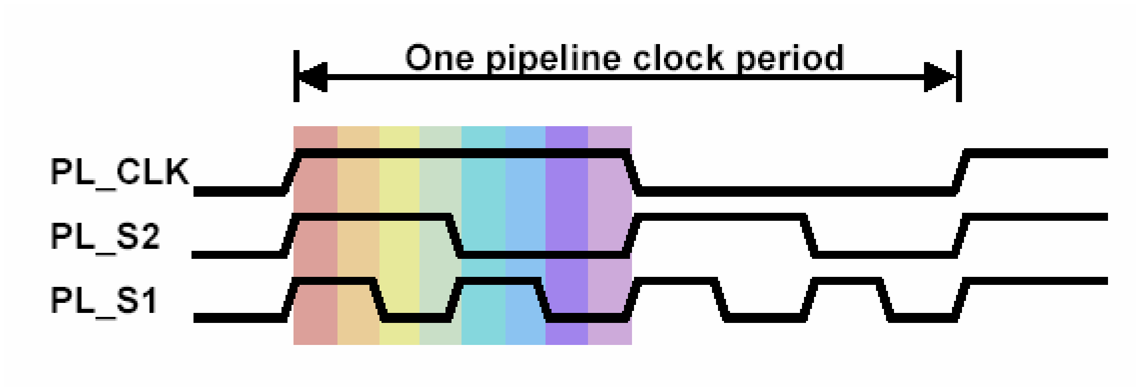


Figure 24. Partitioning of clock cycles into microcycles

The colored bands illustrate how one pipeline clock can be divided into sixteen microcycles should the need arise. The emulated DIVA PIM hardware executes WideWord instructions using eight micro-cycles – four are used for each of the 64-bit operations, the remaining four are used to guarantee safe data storage in the WideWord register file and to avoid bus conflicts when making a selection among one of the four 64-bit data fields.

9. Prototype System Integration

The goal of the prototype system was to produce a stable, high bandwidth demonstration platform for DIVA PIMs. In addition it was to provide an environment in which to debug and performance monitor the first PIM chips.

The demonstration platform required several areas of effort including:

- Host Node Board
- Host Peripheral IO
- Host Operating System Code
- PIM-ulator
- Assembler & Linker
- PIM-Specific Code
- PIM SO-DIMM

9.1 Host Node Board

A custom PPC 603e based node board was used from funding under the ASNT project. It contains an MPC106 combination memory controller and host bridge for PCI. Designed at ISI this allows straightforward modifications to both hardware and firmware for PIM operation.

9.2 Host Peripheral I/O

The host node PCI port provides a method for off-the-shelf subsystems to be used for standard I/O functions. An expansion CPCI (Compact Peripheral Component Interconnect) chassis and ethernet, video, scsi, and serial io cards were purchased and checked out with the PPC 603e-based PCs on hand from the ASNT project.

9.3 Host Operating System Code

Though the host node has only skeleton firmware, it was thought that Linux would be able to boot when provided with a device tree. That was necessary but not sufficient. Each peripheral may contain its own custom firmware that must be executed in a delicate interplay with the host node firmware (either Open Firmware or BIOS compliant) in order to be LINUX (or any other OS for that matter) bootable. Progress has been made toward hand-executing this interplay, but in the end the pace was insufficient for the project needs. Per the PIM Specific Code section below, a small OS called RTEMS was to be used for the PIM and was also pressed into service for the host node. A port of RTEMS was made to the host node and its skeleton boot firmware that allowed TTY console communication in a matter of weeks. The port accomplished three things: provided experience with RTEMS in an easily debugged environment (the host node), made the host node capable of controlling and performance monitoring the PIM, and finally provided a reasonable operating system for the development of PIM memory management code. It was used to great effect in the DARPA Tech 2002 demonstration of the host node and PIM noted in the summary below.

9.4 PIM-ulator

Concern over both the schedule and functionality of the first PIM chip coupled with the existence of unique hardware led to the creation of the PIM-ulator. The ASNT Bridge node hardware contained six powerful FPGA devices that allowed one host node to communicate to another via external L2 cache cycles. In that way one host node could simulate the PIM processor and memory while the other acted as a normal host node. This configuration allowed a path for OS and memory management software and operational interaction between host and a pseudo-PIM without the real PIM chip.

9.5 Assembler and Linker

Open source tools from the gnu project have been on plan from the project outset. The first Assembler for DIVA was a port pulled from the MIPs branch of the gnu assembler tree due to similarities in the Instruction Set Architecture. It was used for the Emulator area of the project described elsewhere in this document. The port required some 660 unique versions of 94 DIVA instructions. The assembler and linker saw standalone use in the Emulator and then more extensive and integrated use as the chip was brought-up and tested.

High-level compiler support was desired for the wide-word chip functionality. The front-end of the compiler (gcc) was pulled from the PPC branch of the gnu tree due to the availability of PPC Altivec extensions. From this branch, the backend of the compiler was modified to produce DIVA assembly mnemonics as input to the assembler. The two worlds of MIPs and PPC collided as the gcc tool chain was used as a whole. The PPC-based backend was sufficiently incompatible with the MIPs based assembler to require a port of the MIPs rewrite to the PPC assembler base. The compiler was then able to work from the DIVA-modified Altivec extension front end, through the DIVA-modified backend and finally out the DIVAmodified PPC assembler and linker. This combination has seen much use in conjunction with the PIM hardware and the host node system.

9.6 PIM Specific Code

The PIM is to have multiple threads operational on the chip under control of the Run- Time Kernel (RTK). Initially it was to be a custom in-house design, but as the intricacies of coherent management of memory from the host node side and PIM node side became apparent it was decided to concentrate on those intricacies and use something off-the-shelf for the bulk of the less novel details. RTEMS, real-time operating system initially designed for mission critical guidance and control systems was chosen for its capabilities, small footprint and open-source status. It was ported and built for the PPC-based host node as mentioned above under Host Operating System Code.

The memory management code was the target of much effort leading to a paper published in the Proceedings of the Workshop on Intelligent Memory Systems, held in conjunction with Architectural Support for Programming Languages and Operating Systems in November 2000. The code development of this PIM-specific code was implemented and simulated in a LINUX environment and is to be ported to RTEMS with only a moderated amount of expected effort.

9.7 PIM SO-DIMM

After the PIM chip passed initial functional test in a test board connected to a logic analyzer, the design of a system memory board was finished and fabricated. It consisted of two PIM chips on an SDRAM SO-DIMM form factor memory board. The two chips may be interconnected to each other or to other PIMs on other memory boards. Logically this interconnection is accomplished with the Parcel buffer; physically it is with ribbon cables. These memory cards were tested out in the host node first as common SDRAM memory, addressed with two different chip-selects from the memory controller. With reliable operation of the memory subsystem the focus turned to running the Cornerturn stressmark kernel on the chip.

9.8 DARPATech Demonstration

In the spring of 2002 ISI was invited to present a demonstration of DIVA PIM technology at the DARPATech Symposium at the end of July. It provided an additional goal and focus during those months. Ten packaged PIM chips were assembled onto five SO-DIMM memory module boards, one shown in Figure 25.

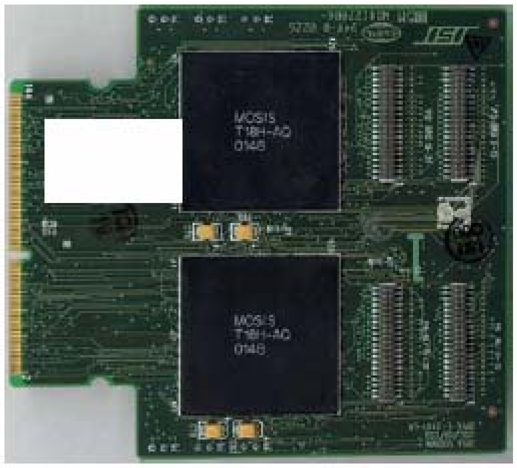


Figure 25. SO-DIMM memory module board.

Within a week the memory interface to both chips was proven operational. The next business day the Cornerturn stressmark code that was verified on the PIM test board was running at speed in the PIM on the SO-DIMM inserted into the host node demonstration system, shown in Figure 26.

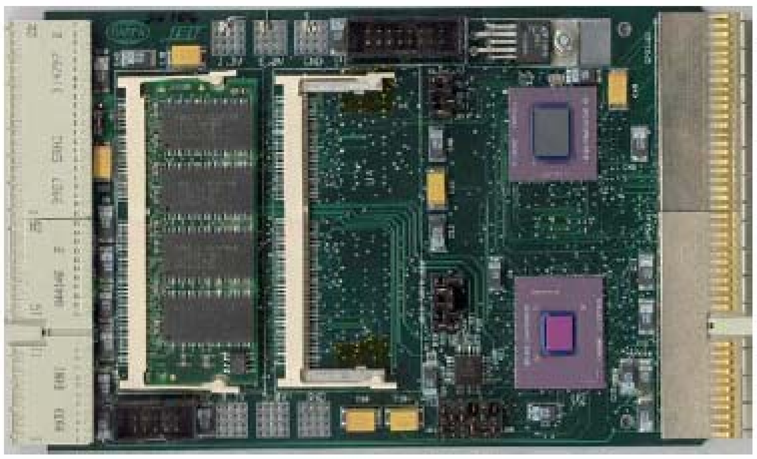


Figure 26. Host node demonstration system

Many different aspects of the host node and PIM required attention and could have jeopardized the demonstration. Memory tests that logged number of and location of last error were written for the PIM memory to ensure enough good memory space for the code and data. The host node memory controller required parameters for the new memory since there is no host node Open Firmware (BIOS). The host node SO-DIMM sockets were replaced with 22.5 degree sockets to accommodate the oversized wing of the PCB that holds the PIM chips. Small clock and reset modules were made to provide these functions to the host node when standing alone in a CPCI cage. A chip reset line which enables operation from a reset vector was also wired to a pin set aside for such on the memory socket, while the host node CPLD (Complex Programmable Logic Device) was enhanced with register support of an I/O line wired to that pin for reset control.

The software provided another set of constraints. Our goal became to put the PowerPC host node into a race with the PIM. RTEMS was used to manage this race on the host and at the same time use task priorities to ensure full processing time was given to the host. The PIM Cornerturn application was hand written and hand assembled, while the host Cornerturn was written in C and automatically compiled and assembled with gcc and gas for the PowerPC with no optimizations. The resultant assembly code was compared for similarity to the PIM code and was within a few percent of the same cycle count.

The code and data were loaded through an emulator to both the PowerPC and the PIM memories. The data size was 32k bytes, 8k 32-bit integer values. This data size was deliberately larger than the PPC603e 16k byte data cache. Then under control of RTEMS via the PCI serial card the demonstration was started. The host counted off 1000 iterations of Cornerturn. The PIM was let run during that time. The PIM performed over 35,000 iterations yielding a 35x speedup. The clock speed of the 603e was 166 MHz while the PIM was 133 MHz. The numbers illustrate both the large penalty for cache miss behavior on the host (~13 bus cycles @ 66MHz for 205ns) and the large benefit of very low-latency (~3 cycles @ 133MHz for 23ns) access to main memory for the PIM processor.

9.9 Stressmark-on-Chip Verification

Continuing forward, we realized that many parts of the system required verification at once: the chip, the system interface, the assembler, the compiler backend as well as the compiler. With a small team and a plan for a second release of the chip with more features, we have adopted a test strategy of using the DIS Stressmark suite with known inputs and outputs to give maximum functional coverage with minimum effort. To that end, we have taken the C versions of Cornerturn and Transitive Closure through the DIVA compiler and assembler. The kernel of the stressmark is then extracted, setup code and the known input data is appended and the code is run on the chip. The outputs are then checked against known good output from gcc builds and runs on a Sparc workstation.

This method has turned up a handful of bugs in several different areas and is proving to be a viable approach under the limited time constraints.

Recent verification work has shown successful execution of bi-directional message passing, along with transitive closure, pointer, and 2-pim transitive with integral chip-to-chip communications.

9.10 Future Work

The integration effort as a whole is still paying dividends. The HPCS project is using the current system to measure DIVA's performance on the StreamAdd benchmark and project expected performance for the HPCS-sponsored Godiva system. The next chip turn incorporates a DDR interface, mounted on full size DIMM memory cards plugged into a commodity Itanium-based workstation as a test bench for the larger system concepts.

10. Publications

[Hall99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, J. Shin, J. Park, “Mapping Irregular Computations to DIVA, a Data-Intensive Architecture,” In *Proceedings of SC99*, Nov. 1999.

[Chame00] J. Chame, M.W. Hall, and J. Shin, “Compiler Transformations for Exploiting Bandwidth in PIM-Based Systems,” In *Proceedings of Solving the Memory Wall Workshop*, held in conjunction with the International Symposium on Computer Architecture, June 2000.

[Kang00] Chang Woo Kang, Jeff Draper, “A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System,” *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, August 2000.

[Hall00] M.W. Hall and C. Steele, “Memory Management in PIM-Based Systems,’’ In *Proceedings of the Workshop on Intelligent Memory Systems*, held in conjunction with Architectural Support for Programming Languages and Operating Systems, Boston, MA, Nov. 2000.

[Draper02a] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, “The Architecture of the DIVA Processing-In-Memory Chip,” In *Proceedings of the International Conference on Supercomputing*, June, 2002.

[Draper02b] Jeffrey Draper, Jeff Sondeen, Sumit Mediratta, Ihn Kim, “Implementation of a 32-bit RISC Processor for the Data-Intensive Architecture Processing-In-Memory Chip,” *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, July 2002.

[Chiueh02] Herming Chiueh, Jeffrey Draper, Sumit Mediratta, Jeff Sondeen, The Address Translation Unit of the Data-Intensive Architecture (DIVA) System, *Proceedings of the 28th European Solid-State Circuit Conference*, September 2002.

[Draper02c] Jeffrey Draper, Jeff Sondeen, Chang Woo Kang, “Implementation of a 256-bit WideWord Processor for the Data-Intensive Architecture (DIVA) Processing-In-Memory (PIM) Chip,” *Proceedings of the 28th European Solid-State Circuit Conference*, September 2002.

[Shin02a] J. Shin, J. Chame and M. W. Hall, “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures.” In *Proceedings of the Parallel Architectures and Compilation Techniques Conference*, Sept. 2002, *Selected as distinguished paper*.

[Shin02b] J. Shin, J. Chame and M. W. Hall, “A Compiler Algorithm for Exploiting Page-Mode Memory Accesses in Embedded-DRAM Devices," In *Proceedings of the Fourth Workshop on Media and Stream Processors Workshop*, held in conjunction with MICRO '02, November, 2002. *Selected as best student paper*.

[Shin03] J. Shin, J. Chame and M. W. Hall, “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures,” Distinguished paper selected from PACT '02, to appear in *Journal of Instruction-Level Parallelism*.

11. Professional Personnel

11.1 Research Area Leaders:

- Dr. John J. Granacki, Principal Investigator
- Dr. Mary Hall, Co Principal Investigator
- Dr. Jeffrey Draper, VLSI Team Leader
- Dr. Jacqueline Chame, Simulation and Applications Team Leader
- Mr. Jeffrey LaCoss, Emulator Team Leader
- Mr. Tim Barrett, System Integration Team Leader

11.2 Doctoral students

- Dr. Louis Luh, PhD, May 2000, Thesis Title: High-Speed CMOS Continuous-Time Switched-Current Sigma-Delta Modulators
- Dr. Herming Chiueh, PhD, Aug 2002, Thesis Title: A Thermal Management Design for System-on-Chip Circuits and Advanced Computer Systems
- Dr. Yuyu Chang, PhD. September 2002, Thesis Title: CMOS Giga-Hertz Band Filters with Automatic Tuning Circuitry for Communication Applications
- Joong-Seok Moon, PhD expected Aug 2003
- Jaewook Shin (PIM-specific optimizations, integration of DIVA scalar GCC and AltiVec-extended GCC, integration of DIVA compiler with MIT-SLP, DIVA implementations of CornerTurn, Field and NAS CG, DIVA simulator library implementation components), Phd expected 2004
- Chun Chen (DIVA implementations of Neighborhood stressmark, Image Understanding and Ray Tracing benchmarks, port of simulator to Condor)
- Hang Shi (DIVA implementations of Transitive stressmark, Method of Moments benchmark)
- Ruoming Pang (DIVA implementations of Natural Join and OO7 benchmarks)
- Chang Woo Kang, PhD TBD
- Ihn Kim, PhD TBD
- Taek-Jun Kwon, PhD TBD
- Sumit Mediratta, PhD TBD

11.3 Masters students

- Somphol Boonjing (Application Binary Interface for node compiler), MS December 2000
- Sachit Chandra (VLSI) MS expected Aug 2003
- Gokhan Daglikoca (VLSI)
- Prashant Desai (design for assembler, backend compiler, integration with WideWord instructions), MS December 2000
- Rommel Dongre (GCC scalar backend implementation), MS December 2001

- Yamini Kaur (VLSI)
- Junaid Qazi (VLSI)
- Shyam Sethuram (DIVA simulator implementation components), MS May 2002
- Vijay Srinivasan MS December 2003

11.4 Other Collaborators

- USC/ISI: Mr. Dale Chase, Mr. Jeff Sondeen, Dr. Bill Athas, Dr. Jeff Koller, Dr. Craig Steele, Mr. Mike Gorman, Dr. Apporv Srivastava, Ms. Diane Delute, Mr. Bert White, Dr. Pedro Diniz. Mr. Pablo Moissett
- Caltech: Dr. Thomas Sterling, Mr. Daniel Savarese
- University of Notre Dame: Dr. Peter Kogge, Dr. Jay Brockman, Dr. Vincent Freeh, Mr. Bedros Hanouik, Mr. Richard Murphy, Mr. Rich Kendall, Mr. Alexi Koundraiov, Ms. Shannon Kuntz, Mr. Jason Zawodny, Mr. Arun Rodrigues, Mr. Edward Kang
- University of Delaware: Dr. Guang Gao, Dr. Kevin Theobald, Mr. Tom Geiger
- AlphaTech: Dr. Mark Luettgen, Dr. Bob Tenney

12. Results, Conclusions & Technology Transfer

The single most important result produced by the DIVA Project is ***a complete working system that demonstrates the advantages of PIM technology used as "smart memories"***. This is the proof of concept that "smart memory" can help ameliorate the "memory wall" that limits the performance of present day memory systems.

This achievement paves the way for further research on systems with heterogeneous memory systems, that is, PIM and conventional DRAM used together; PIM-based memory hierarchies, for example, PIM caches; studying and evaluating larger applications problems; namely, those that cannot be run on a simulator or emulator and combining this technology in new ways or incorporating it with other technology into new architectures.

Two follow-on research projects have already started to build on the DIVA technology MONARCH under the DARPA-sponsored Polymorphous Computer Architecture Program and Godiva under the High Productivity Computing System Program. Perhaps of even greater significance these new projects are expanding the research and extending the technology in partnership with large industrial partners. MONARCH is a joint project with the Raytheon Corporation, a leading defense contractor and Mercury Computing, the largest supplier of embedded computers to the military. Godiva is a joint project with Hewlett Packard, a major U.S. computer vendor. Both of these projects represent significant IP transfer from DIVA but also represent a high possibility for insertion of DIVA technology into real military and commercial systems.

A "second turn" of the DIVA VLSI funded under the MONARCH Project will also incorporate floating-point unit into the WideWord unit greatly enhancing DIVA's applicability to a broader class of scientific problems.

The DIVA team has briefed many of the research leaders of major U.S. companies like IBM, Intel, Hewlett Packard and Sun, as well as several venture capitalists that have expressed an interest in DIVA technology. We have also briefed the Deputy Under Secretary of Defense for

Science and Technology, NSA's Director of Computing and the DOE's ASCI Program Manger. We will continue to inform the decision makers about this technology.

The main issue with the acceptance of PIM and Embedded-DRAM technology is the cost-performance, that is, does the added cost of combining DRAM on the same chip with the logic processing warrant the added expense of manufacturing these die. This is a complex question and depends on the specific application and also the semiconductor technology. At this time, there is definitely a premium to be paid for the added performance offered by systems that use PIM technology.

13. Inventions, or patent disclosures

No inventions were disclosed or patents submitted by the USC DIVA research team.

14. References

[Babb99] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua and S. Amarasinghe, "Parallelizing Applications Into Silicon," In Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, April, 1999.

[Burger96] D. Burger, J. Goodman and A. Kagi. "Memory Bandwidth Limitations of Future Microprocessors," In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA),* May, 1996.

[Chame00] J. Chame, M.W. Hall, and J. Shin, "Compiler Transformations for Exploiting Bandwidth in PIM-Based Systems," In *Proceedings of Solving the Memory Wall Workshop*, held in conjunction with the International Symposium on Computer Architecture, June 2000.

[Chiueh02] Herming Chiueh, Jeffrey Draper, Sumit Mediratta, Jeff Sondeen, The Address Translation Unit of the Data-Intensive Architecture (DIVA) System, *Proceedings of the 28th European Solid-State Circuit Conference*, September 2002.

[Draper02a] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, C. W. Kang, I. Kim, G. Daglikoca, "The Architecture of the DIVA Processing-In-Memory Chip," In *Proceedings of the International Conference on Supercomputing*, June, 2002.

[Draper02b] Jeffrey Draper, Jeff Sondeen, Sumit Mediratta, Ihn Kim, "Implementation of a 32-bit RISC Processor for the Data-Intensive Architecture Processing-In-Memory Chip," *Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, July 2002.

[Draper02c] Jeffrey Draper, Jeff Sondeen, Chang Woo Kang, "Implementation of a 256- bit WideWord Processor for the Data-Intensive Architecture (DIVA) Processing-In- Memory (PIM) Chip," *Proceedings of the 28th European Solid-State Circuit Conference*, September 2002.

[vonEicken92] T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", In *Proc. Of the 19th International Symposium on Computer Architecture*, May 1992.

[Elliot99] D. Elliot, M. Stumm, W. Snelgrove, C. Cojocaru, R. McKenzie, "Computational RAM: Implementing Processors in Memory," *IEEE Design and Test of Computers*, January-March, 1999, pp. 32-41.

[Gokhale95] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*, April 1995, pp. 23-31.

[Hall99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, J. Shin, J. Park, "Mapping Irregular Computations to DIVA, a Data-Intensive Architecture," In *Proceedings of SC99*, Nov. 1999.

[Hall00] M.W. Hall and C. Steele, "Memory Management in PIM-Based Systems," In *Proceedings of the Workshop on Intelligent Memory Systems*, held in conjunction with Architectural Support for Programming Languages and Operating Systems, Boston, MA, Nov. 2000.

[IBM] IBM Microelectronics, "Embedded DRAM",
http://www.chips.ibm.com/products/asics/products/edram.

[Kang99] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," In Proceedings of the IEEE International Conference on Computer Design, Oct. 1999.

[Kang00] Chang Woo Kang, Jeff Draper, "A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System," *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, August 2000.

[Kogge94] P. Kogge. "The EXECUBE Approach to Massively Parallel Processing," 1994 Int. Conf. on Parallel Processing, Chicago, IL, August, 1994.

[Liddicoat02] "High-Performance Arithmetic for Division and the Elementary Functions.", Ph.D. Dissertation, Stanford University, January 2002.

[Mitsubishi99] Mitsubishi, ":M32R/D Series: 32-bit RISC Processor, On-chip DRAM," www.mitsubishi-chips.com/data/ datasheets/mcus/m32rdgrp.html, May 6, 1999.

[Oskin98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. "Active Pages: A Model of Computation for Intelligent Memory". In *Proc. of the 25th International Symposium on Computer Architecture (ISCA)*, June, 1998.

[Patterson97] D. Patterson et al., "A Case for Intelligent DRAM: IRAM," *IEEE Micro*, April 1997.

[RSIM] http://www-ece.rice.edu/~RSIM

[RTEMS] www.rtems.com

[Saulsbury95] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin, “An Argument for Simple COMA”, In *Proc. of the Symposium on High-Performance Computer Architecture*, 1995.

[Shin02a] J. Shin, J. Chame and M. W. Hall, “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures.” In *Proceedings of the Parallel Architectures and Compilation Techniques Conference*, Sept. 2002,

[Shin02b] J. Shin, J. Chame and M. W. Hall, “A Compiler Algorithm for Exploiting Page-Mode Memory Accesses in Embedded-DRAM Devices," In *Proceedings of the Fourth Workshop on Media and Stream Processors Workshop*, held in conjunction with MICRO '02, November, 2002.

[Shin03] J. Shin, J. Chame and M. W. Hall, “Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures,” Distinguished paper selected from PACT '02, to appear in *Journal of Instruction-Level Parallelism*.

Appendix A: DIVA PIM Processor ISA

DIVA PIM Processor Instruction Set Manual

University of Southern California
Information Sciences Institute

DIVA
Data Intensive Architecture

Chapter 1 - DIVA Instruction Set Overview

Scalar Instruction Formats

As shown in Figure 1, the DIVA scalar instruction uses a three-operand format to specify two 32-bit source registers and a 32-bit target register. For arithmetic/logical instructions using this format, there is also a **C** bit to indicate whether the current instruction updates condition codes. However, the **C** bit indicates signed/unsigned arithmetic for multiply/divide instructions, since these instructions never update condition codes by definition. In lieu of a second source register, a 16-bit immediate value may be specified, as shown in Figure 2.

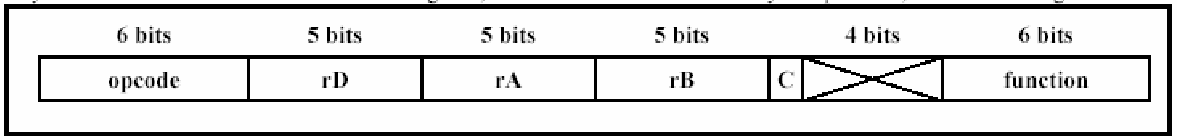


Figure 1 Format R for Scalar Register Operations

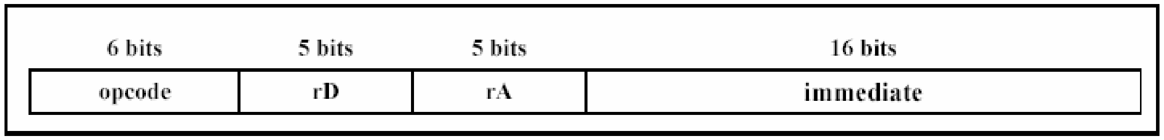


Figure 2 Format I for Scalar Immediate Operations

The branch instruction formats are shown in Figure 3. The branch target address may be PC-relative or calculated using a base register ORed with an offset. In both formats, the offset is in units of words, or 4 bytes, since instructions must be on a 4-byte boundary. Furthermore, the **L** bit specifies linkage, that is, whether a return instruction address should be saved in R31, referred to as a call instruction. Also, the **CCC** field specifies one of eight branch conditions: always, equal, not equal, less than, less than or equal, greater than, greater than or equal, or overflow. See the branch and call instruction descriptions for details.

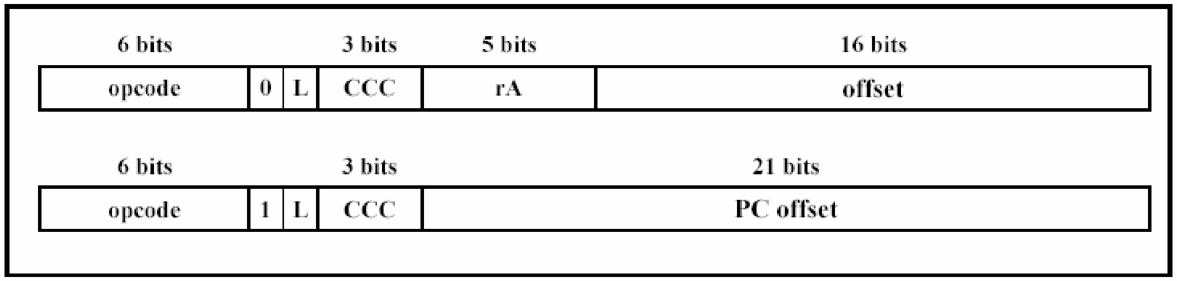


Figure 3 Format B for Branches

WideWord Instruction Formats

As shown in Figure 4, "WideWord Arithmetic/Logical Format," WideWord instructions follow the general form of scalar instructions. Additional control information is included to manage the data fields of the WideWord, and to modify the execution of the instruction. Figure 5 shows the format for transfers within the WideWord register file and across the scalar and WideWord register files.

6 bits	5 bits	5 bits	5 bits		2 bits	2 bits	6 bits
opcode	wrD	wrA	wrB	C	PP	WW	function

Figure 4 Format W for WideWord Arithmetic/Logical Operations

6 bits	5 bits	5 bits	5 bits		2 bits	2 bits	6 bits
opcode	rD	rA	$I_{A/D}$	T	PP	WW	function

Figure 5 Format T for Wide-Word and Inter-Register File Transfers

The control fields are defined as follows:

WW (width)

The ***WW*** field sets the width of the WideWord operands to eight, sixteen, or thirty-two bits, which primarily affects the shift operations and the configuration of the carry chain for additions and subtractions. For the merge instruction, these bits specify the condition on which the merge is based. The encoding of these bits is listed in the following table:

WW Value	Operand Width	Assembler Mnemonic
00	8 bits	b
01	16 bits	h
10	32 bits	w
11	Reserved	NA

C (condition code enable)

The ***C*** bit indicates whether condition codes will be updated as a result of the current instruction's execution. However, the ***C*** bit indicates signed/unsigned arithmetic for multiply, pack, and unpack instructions.

PP (participation)

The ***PP*** field interacts with condition codes to control whether a computation is performed on a given data field. The participation field can specify that a data field participate always, only if a condition local to its own data field is true, only if the data field is the leftmost field with a condition that is true, or only if the data field is the rightmost field with a condition that is true. The condition that is inspected for participation depends on the value of the ***PM*** (participation mode) register. Refer to the architecture document for more details. The encoding of the ***PP*** bits is listed in the following table:

PP Value	Participation Definition	Assembler Mnemonic
00	Always participate	a
01	Specified by local condition	o
10	Leftmost participation	l
11	Rightmost participation	r

T (type)

The ***T*** bit governs whether the current instruction operates on a vector or scalar. Depending on the function, ***rD*** or ***rA*** may specify a WideWord register. In this case, the ***T*** bit specifies whether the current transfer instruction refers to the WideWord register as a whole vector or instead uses $I_{A/D}$ to index a sub-field of the WideWord register.

$I_{A/D}$

Value to be used as an index when a sub-field of a WideWord is involved in a transfer. Depending on the function, this index field may be an immediate or a scalar GPR specifier. Also, $I_{A/D}$ may be coupled with either ***rD*** or ***rA*** depending on the direction of the transfer as specified by the function.

Condition Codes

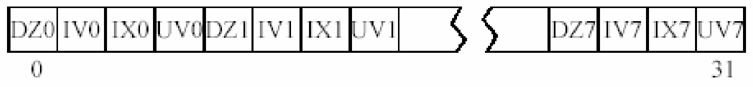
The scalar condition code register, ***CC***, consists of 5 bits. The first three bits of ***CC*** are set by an algebraic comparison of the result to zero; the other two bits have slightly more peculiar semantics. The condition codes have the ***CC*** bit labels and semantics as indicated below. Note that LT, GT, EQ, and CA condition codes are updated only if the current instruction has its condition code enable bit set. The OV condition

Condition Code	CC bit	Description
LT	0	This bit is set when the result is negative.
GT	1	This bit is set when the result is positive and non-zero.
EQ	2	This bit is set when the result is zero.
OV	3	This bit is set to indicate overflow has occurred during execution of an add or subtract instruction. This bit is not altered by any other instructions. In practice, the OV bit is set if the carry out of bit 0 is not equal to the carry out of bit 1 (assuming big Endian bit labeling).
CA	4	In general, the carry bit (CA) is set to indicate that a carry out of bit 0 occurred during execution of an add or subtract instruction. This bit is not altered by any other instructions.

code is updated for any scalar add or subtract operation, regardless of the condition code enable bit setting, and is sticky; that is, it is only cleared when the condition code register is read.

The 32-bit LT, GT, EQ, OV, and CA registers of the WideWord datapath have analogous semantics to the corresponding condition code of the scalar datapath. For instance, each bit of the WideWord LT register is set if the result of its corresponding 8-bit datapath is negative. However, there are subtleties due to the configurability of the operand sizes. For example, if a WideWord instruction specifies that operands are to be treated as 32-bit values, the condition codes are grouped into eight groups of 4, where each bit of a group is updated with the same value to reflect a condition for the group's corresponding 32-bit result.

Similar to condition codes, the WideWord floating-point status register (FPSR - special-purpose register 15) may be updated to reflect exception conditions for floating-point operations. This register is a 32-bit register arranged in groups of 4 status conditions for each of the eight 32-bit floating-point units in the WideWord datapath. The 4 status conditions are: divide by zero (DZ), invalid (IV), inexact (IX), and unsupported value (UV). DZ, IV, and IX are typical IEEE-754 floating-point exceptions. Refer to the IEEE-754 standard for details. UV indicates that either overflow or underflow occurred at some point during the program. All bits of FPSR are sticky; once set, they remain set until FPSR is read via an mfspr instruction. The bit arrangement for FPSR is shown below.



FPSR Bit Arrangement

Concise List

TABLE 1. DIVA Instruction Set

FUNC	DESCRIPTION	FUNC	DESCRIPTION	FUNC	DESCRIPTION
SYS	System Call	**MTSPR**	Move to special-purpose reg	**B*x***	Branch on scalar condition
ICLI	Instruction Cache Line Invalidate	**MFSPR**	Move from special-purpose reg	**BA*x***	Branch on all WideWord conditions
RFE	Return from Exception	**MTPR**	Move to protected reg	**BN*x***	Branch on no WideWord condition
		MFPR	Move from protected reg	**CALL*x***	Call on scalar condition
	Scalar Instructions	**MTATR**	Move to address translation reg	**CALLA*x***	Call on all WideWord conditions
ADD	Add	**MFATR**	Move from address translation reg	**CALLN*x***	Call on no WideWord condition
ADDE	Add extended				
ADDI	Add immediate		**WideWord Instructions**		
ADDIC	Add immediate w/ condition codes	**WADD**	Add		
SUB	Subtract	**WADDE**	Add extended		
SUBE	Subtract extended	**WSUB**	Subtract		
SUBU	Subtract unsigned	**WSUBE**	Subtract extended		**Special WideWord Instructions**
MUL	Multiply	**WSUBU**	Subtract unsigned	**WPRM**	Permute
MULU	Multiply unsigned	**WMULES**	Multiply even signed	**WPRMI**	Permute immediate
DIV	Divide	**WMULEU**	Multiply even unsigned	**WMRG**	Merge based on condition codes
DIVU	Divide unsigned	**WMULOS**	Multiply odd signed	**WPKS**	Pack using signed arithmetic
AND	And	**WMULOU**	Multiply odd unsigned	**WPKU**	Pack using unsigned arithmetic
ANDI	And immediate	**WAND**	And	**WUPKH**	Unpack high-order byte/halfword
ANDIC	And immediate w/ condition codes	**WNOT**	Bitwise inversion	**WUPKL**	Unpack low-order byte/halfword
NOT	Bitwise inversion	**WOR**	Or		
OR	Or	**WXOR**	Xor		**Transfer Instructions**
ORI	Or immediate	**WSLL**	Shift left logical	**MVSW**	Move scalar to WW
ORIC	Or immediate w/ condition codes	**WSLLI**	Shift left logical immediate	**MVSWI**	Move scalar to WW, indirect
ORIS	Or immediate shifted	**WSRA**	Shift right arithmetic	**MVWS**	Move WW to scalar
XOR	Xor	**WSRAI**	Shift right arithmetic immediate	**MVWSI**	Move WW to scalar, indirect
XORI	Xor immediate	**WSRL**	Shift right logical	**MVWW**	Move WW to WW
XORIC	Xor immediate w/ condition codes	**WSRLI**	Shift right logical immediate	**MVWWI**	Move WW to WW, indirect
SLL	Shift left logical	**WLD**	Load Reg from Mem		
SLLI	Shift left logical immediate	**WST**	Store Reg to Mem		
SRA	Shift right arithmetic	**WFABS**	Floating-point absolute value		
SRAI	Shift right arithmetic immediate	**WFADD**	Floating-point add		**Miscellaneous Instructions**
SRL	Shift right logical	**WFDIV**	Floating-point divide	**LOKL**	Lock Load
SRLI	Shift right logical immediate	**WFMUL**	Floating-point multiply	**LOKS**	Lock Store
LD	Load Reg from Mem	**WFNEG**	Floating-point negate	**PROBE**	Probe address to determine
ST	Store Reg to Mem	**WFSUB**	Floating-point subtract		locality
		WFTI	Floating-point to integer conversion		
ELO	Encode leftmost one	**WITF**	Integer to floating-point conversion		
CLO	Clear leftmost one				

Alphabetical list of instructions

TABLE 2. Preliminary Encoding of DIVA Instruction Set

Instruction	Format	Encoding					
		6 bits	**5 bits**	**5 bits**	**5 bits**	**5 bits**	**6 bits**
ADD	R	000011	rD	rA	rB	0XXXX	100000
ADDC	R	000011	rD	rA	rB	1XXXX	100000
ADDE	R	000011	rD	rA	rB	0XXXX	100001
ADDEC	R	000011	rD	rA	rB	1XXXX	100001
ADDI	I	100000	rD	rA	immediate		
ADDIC	I	100001	rD	rA	immediate		
AND	R	000011	rD	rA	rB	0XXXX	101000
ANDC	R	000011	rD	rA	rB	1XXXX	101000
ANDI	I	101000	rD	rA	immediate		
ANDIC	I	101001	rD	rA	immediate		
B*x*	B	111111	00CCC	rA	offset		
B*x*	B	111111	10CCC	PC-relative offset			
BA*x*	B	111100	00CCC	rA	offset		
BA*x*	B	111100	10CCC	PC-relative offset			
BN*x*	B	111101	00CCC	rA	offset		
BN*x*	B	111101	10CCC	PC-relative offset			
CALL*x*	B	111111	01CCC	rA	offset		
CALL*x*	B	111111	11CCC	PC-relative offset			
CALLA*x*	B	111100	01CCC	rA	offset		
CALLA*x*	B	111100	11CCC	PC-relative offset			
CALLN*x*	B	111101	01CCC	rA	offset		
CALLN*x*	B	111101	11CCC	PC-relative offset			
CLO	R	000011	rD	rA	00000	0XXXX	001001
DIV	R	000011	00000	rA	rB	0XXXX	100111
DIVU	R	000011	00000	rA	rB	1XXXX	100111
ELO	R	000011	rD	rA	00000	0XXXX	001000
ICLI	I	110011	00000	rA	offset		
LD	I	110000	rD	rA	offset		
LOKL	I	110110	rD	rA	offset		
LOKS	I	110111	rD	rA	offset		
MFATR	R	000000	rD	atrA	00000	XXXXX	000010
MFPR	R	000000	rD	prA	00000	XXXXX	000000
MFSPR	R	000001	rD	sprA	00000	XXXXX	000100
MTATR	R	000000	atrD	rA	00000	XXXXX	000011
MTPR	R	000000	prD	rA	00000	XXXXX	000001
MTSPR	R	000001	sprD	rA	00000	XXXXX	000101
MUL	R	000011	00000	rA	rB	0XXXX	100110
MULU	R	000011	00000	rA	rB	1XXXX	100110

TABLE 2. Preliminary Encoding of DIVA Instruction Set

Instruction	Format	Encoding					
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
MVSW	T	000100	wrD	rA	I_D	TPPWW	000100
MVSWI	T	000100	wrD	rA	R_{ID}	000WW	100100
MVWS	T	000100	rD	wrA	I_A	000WW	000001
MVWSI	T	000100	rD	wrA	R_{IA}	000WW	100001
MVWW	T	000100	wrD	wrA	I_A	TPPWW	000000
MVWWI	T	000100	wrD	wrA	R_{IA}	1PPWW	100000
NOT	R	000011	rD	rA	00000	0XXXX	101110
NOTC	R	000011	rD	rA	00000	1XXXX	101110
OR	R	000011	rD	rA	rB	0XXXX	101100
ORC	R	000011	rD	rA	rB	1XXXX	101100
ORI	I	101100	rD	rA	immediate		
ORIC	I	101101	rD	rA	immediate		
ORIS	I	101110	rD	rA	immediate		
PROBE	I	110010	rD	rA	offset		
RFE	R	000000	XXXXX	XXXXX	XXXXX	XXXX	111111
SLL	R	000011	rD	rA	rB	0XXXX	000000
SLLC	R	000011	rD	rA	rB	1XXXX	000000
SLLI	R	000011	rD	rA	shift_amount	0XXXX	000010
SLLIC	R	000011	rD	rA	shift_amount	1XXXX	000010
SRA	R	000011	rD	rA	rB	0XXXX	000101
SRAC	R	000011	rD	rA	rB	1XXXX	000101
SRAI	R	000011	rD	rA	shift_amount	0XXXX	000111
SRAIC	R	000011	rD	rA	shift_amount	1XXXX	000111
SRL	R	000011	rD	rA	rB	0XXXX	000001
SRLC	R	000011	rD	rA	rB	1XXXX	000001
SRLI	R	000011	rD	rA	shift_amount	0XXXX	000011
SRLIC	R	000011	rD	rA	shift_amount	1XXXX	000011
ST	I	110001	rD	rA	offset		
SUB	R	000011	rD	rA	rB	0XXXX	100010
SUBC	R	000011	rD	rA	rB	1XXXX	100010
SUBE	R	000011	rD	rA	rB	0XXXX	100011
SUBEC	R	000011	rD	rA	rB	1XXXX	100011
SUBU	R	000011	rD	rA	rB	1XXXX	100100
SYS	R	000001	code				000000
WADD	W	000010	wrD	wrA	wrB	0PPWW	100000
WADDC	W	000010	wrD	wrA	wrB	1PPWW	100000
WADDE	W	000010	wrD	wrA	wrB	0PPWW	100001
WADDEC	W	000010	wrD	wrA	wrB	1PPWW	100001
WAND	W	000010	wrD	wrA	wrB	0PPWW	101000
WANDC	W	000010	wrD	wrA	wrB	1PPWW	101000

TABLE 2. Preliminary Encoding of DIVA Instruction Set

		Encoding					
Instruction	Format	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
WFABS	W	011101	wrD	wrA	00000	0PP10	000101
WFABSC	W	011101	wrD	wrA	00000	1PP10	000101
WFADD	W	011101	wrD	wrA	wrB	0PP10	000000
WFADDC	W	011101	wrD	wrA	wrB	1PP10	000000
WFDIV	W	011101	wrD	wrA	wrB	0PP10	000111
WFDIVC	W	011101	wrD	wrA	wrB	1PP10	000111
WFMUL	W	011101	wrD	wrA	wrB	0PP10	000110
WFMULC	W	011101	wrD	wrA	wrB	1PP10	000110
WFNEG	W	011101	wrD	wrA	00000	0PP10	000100
WFNEGC	W	011101	wrD	wrA	00000	1PP10	000100
WFSUB	W	011101	wrD	wrA	wrB	0PP10	000001
WFSUBC	W	011101	wrD	wrA	wrB	1PP10	000001
WFTI	W	011101	wrD	wrA	00000	0PP10	000010
WFTIC	W	011101	wrD	wrA	00000	1PP10	000010
WITF	W	011101	wrD	wrA	00000	0PP10	000011
WITFC	W	011101	wrD	wrA	00000	1PP10	000011
WLD	I	110100	wrD	rA	offset		
WMRG	W	000010	wrD	wrA	wrB	CPPWW	101111
WMULES	W	000010	wrD	wrA	wrB	0PPWW	100110
WMULEU	W	000010	wrD	wrA	wrB	1PPWW	100110
WMULOS	W	000010	wrD	wrA	wrB	0PPWW	100111
WMULOU	W	000010	wrD	wrA	wrB	1PPWW	100111
WNOT	W	000010	wrD	wrA	00000	0PPWW	101110
WNOTC	W	000010	wrD	wrA	00000	1PPWW	101110
WOR	W	000010	wrD	wrA	wrB	0PPWW	101100
WORC	W	000010	wrD	wrA	wrB	1PPWW	101100
WPRM	W	000010	wrD	wrA	wrB	0PP00	001000
WPRMI	W	000010	wrD	wrA	rB	0PP00	001001
WPKS	W	000010	wrD	wrA	wrB	000WW	001110
WPKU	W	000010	wrD	wrA	wrB	100WW	001110
WSLL	W	000010	wrD	wrA	wrB	0PPWW	000000
WSLLC	W	000010	wrD	wrA	wrB	1PPWW	000000
WSLLI	W	000010	wrD	wrA	shift_amount	0PPWW	000010
WSLLIC	W	000010	wrD	wrA	shift_amount	1PPWW	000010
WSRA	W	000010	wrD	wrA	wrB	0PPWW	000101
WSRAC	W	000010	wrD	wrA	wrB	1PPWW	000101
WSRAI	W	000010	wrD	wrA	shift_amount	0PPWW	000111
WSRAIC	W	000010	wrD	wrA	shift_amount	1PPWW	000111
WSRL	W	000010	wrD	wrA	wrB	0PPWW	000001
WSRLC	W	000010	wrD	wrA	wrB	1PPWW	000001

TABLE 2. Preliminary Encoding of DIVA Instruction Set

Instruction	Format	Encoding					
		6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
WSRLI	W	000010	wrD	wrA	shift_amount	0PPWW	000011
WSRLIC	W	000010	wrD	wrA	shift_amount	1PPWW	000011
WST	I	110101	wrD	rA	offset		
WSUB	W	000010	wrD	wrA	wrB	0PPWW	100010
WSUBC	W	000010	wrD	wrA	wrB	1PPWW	100010
WSUBE	W	000010	wrD	wrA	wrB	0PPWW	100011
WSUBEC	W	000010	wrD	wrA	wrB	1PPWW	100011
WSUBU	W	000010	wrD	wrA	wrB	1XXXX	100100
WUPKH	W	000010	wrD	wrA	00000	C00WW	001101
WUPKL	W	000010	wrD	wrA	00000	C00WW	001100
WXOR	W	000010	wrD	wrA	wrB	0PPWW	101010
WXORC	W	000010	wrD	wrA	wrB	1PPWW	101010
XOR	R	000011	rD	rA	rB	0XXXX	101010
XORC	R	000011	rD	rA	rB	1XXXX	101010
XORI	I	101010	rD	rA	immediate		
XORIC	I	101011	rD	rA	immediate		

TABLE 3. Special-Purpose Registers

NAME	SPR Number	DESCRIPTION
CC	0	LT, GT, EQ, OV, and CA bits of scalar processor
HI	1	most significant 32 bits of multiplication result, quotient of division
LO	2	least significant 32 bits of multiplication result, remainder of division
LT	8	32-bit Less Than register of WideWord Unit
GT	9	32-bit Greater Than register of WideWord Unit
EQ	10	32-bit Equal register of WideWord Unit
CA	11	32-bit Carry register of WideWord Unit
OV	12	32-bit Overflow register of WideWord Unit
M	13	32-bit WideWord Mask register used in conditional execution
PM	14	5-bit WideWord Participation Mode register used in conditional execution
FPSR	15	32-bit WideWord Floating-Point status register

TABLE 4. Protected Registers

NAME	PR Number	DESCRIPTION
PSW	0	32-bit processor status word
SSW	1	Stored value of PSW, used in exception handling
EID	2	16-bit environment identifier register
FADR	3	32-bit address of faulting instruction (stored value of PC)
SCR_0 - SCR_3	4 - 7	32-bit supervisor scratch registers
ESW	8	32-bit exception source word
EMR	9	32-bit exception mask register
ESR	10	32-bit exception set register
ERR	11	32-bit exception reset register
MADR	12	32-bit faulting memory address
TIMER	13	32-bit programmable delay timer
RCL	14	Low order 32 bits of real-time clock
RCH	15	High order 32 bits of real-time clock

TABLE 5. Address Translation Registers

NAME	ATR Number	DESCRIPTION
SB_0 - SB_7	0 - 7	32-bit local segment base registers
SL_0 - SL_7	8 - 15	32-bit local segment limit registers
GVB_0 - GVB_3	16 - 19	32-bit global segment virtual base registers
GL_0 - GL_3	20 - 23	32-bit global segment limit registers
GPB_0 - GPB_3	24 - 27	32-bit global segment physical base registers

Chapter 2 - Instruction Descriptions

Notation

This chapter gives detailed individual instruction descriptions. We use Big-Endian byte and bit labeling, meaning that bit/byte 0 is the most significant. Other conventions are listed in the table below.

TABLE 6. Instruction Glossary

Symbol	Meaning	Symbol	Meaning
$A \leftarrow B$	Assignment	MEM[EA]	Memory contents at effective address EA
$A \parallel B$	Bit string concatenation	0x*value*	Hexadecimal value
x^y	x replicated y times	0b*value*	Binary value
$x_{y,z}$	Selection of bits y through z from x	frX	Floating-point register X
$x \wedge y$	x bitwise ANDed with y	(rX)	Contents of general-purpose register X
$x \vee y$	x bitwise ORed with y	PC	Program counter
$x \oplus y$	x bitwise exclusive ORed with y	IADR	Instruction address
$\neg x$	bitwise inversion of x		

Note that the IADR of an instruction is equivalent to the PC value while the instruction is in the fetch stage of the pipeline.

Precedence

The following table gives the rules of precedence and associativity for the pseudocode operators. All operators on the same line have equal precedence, and all operators on a given line have higher precedence than those on the lines below them.

TABLE 7. Precedence of Pseudocode Operators

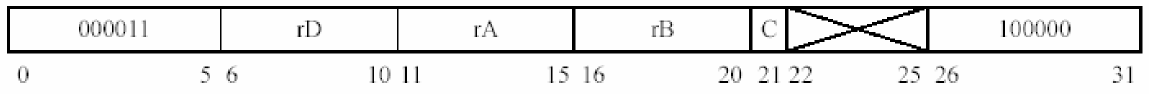
Operator	Associativity
x[n]	left to right
$x_{y,z}$	left to right
x^y	left to right
$\neg$	right to left
$\times$, $\div$	left to right
+, -	left to right
$\parallel$	left to right
=, !=, <, <=, >, >=	left to right
$\oplus$, $\wedge$	left to right
$\vee$	left to right
$\leftarrow$	none

add*x* - Add

Scalar Unit

add rD, rA, rB (C = 0)

addc rD, rA, rB (C = 1)



$$rD \leftarrow (rA) + (rB)$$

The sum (rA) + (rB) is placed into rD.

Other registers altered:

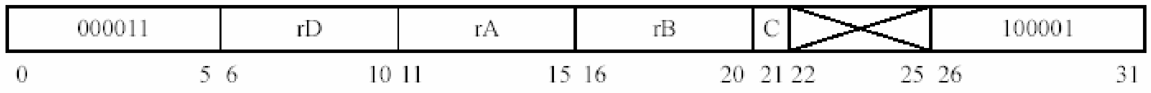
- If C =1, scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

adde*x* - Add Extended

Scalar Unit

adde rD, rA, rB (C = 0)

addec rD, rA, rB (C = 1)



$$rD \leftarrow (rA) + (rB) + CA$$

The sum (rA) + (rB), using the carry bit CA as the carry in, is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

addi - Add Immediate

Scalar Unit

addi rD, rA, IMM

100000	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) + ((IMM_0)^{16} \parallel IMM)$$

The sum (rA) + IMM (sign-extended to form a 32-bit value) is placed into rD.

Other registers altered:

- Scalar condition code OV is set if the operation causes overflow.

addic - Add Immediate Recording Condition Code

Scalar Unit

addic rD, rA, IMM

100001	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) + ((IMM_0)^{16} \parallel IMM)$$

The sum (rA) + IMM (sign-extended to form a 32-bit value) is placed into rD.

Other registers altered:

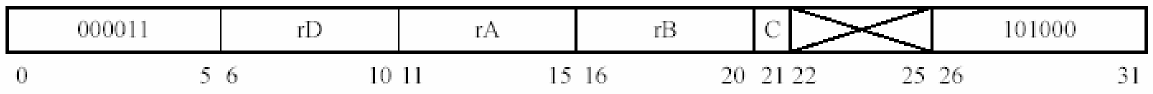
- Scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

and*x* - AND

Scalar Unit

and rD, rA, rB (C = 0)

andc rD, rA, rB (C = 1)



$rD \leftarrow (rA) \wedge (rB)$

The contents of rA are ANDed with rB, and the result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

andi - AND Immediate

Scalar Unit

andi rD, rA, IMM

101000	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) \wedge (0^{16} \,\|\, IMM)$$

The contents of rA are ANDed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- None

andic - AND Immediate Recording Condition Codes

Scalar Unit

andic rD, rA, IMM

101001	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) \wedge (0^{16} \,\|\, IMM)$$

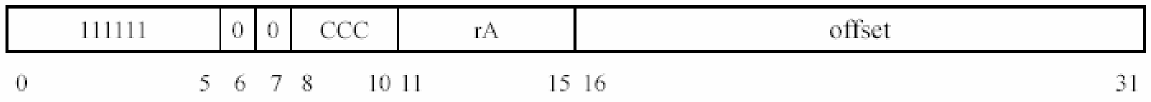
The contents of rA are ANDed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

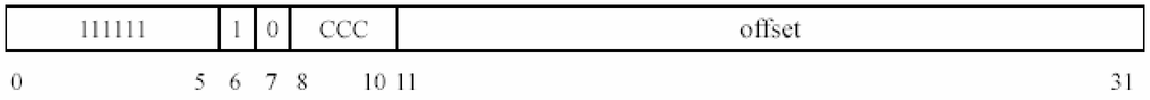
- Scalar condition code registers: LT, GT, EQ

b*x*- Branch

b*x* rA, offset (register-relative format)



b*x* offset (PC-relative format)



```
if scalar condition indicated by CCC
    if PC-relative format
        $PC \leftarrow IADR + ((offset_0)^9 \| offset \| 00)$
    else
        $PC \leftarrow ((rA) \wedge 0xFFFFFFFC) \vee ((offset_0)^{14} \| offset \| 00)$
```

This branch instruction is conditional upon the scalar condition specified by CCC. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot).

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	b rA, offset	b offset
001	beq rA, offset	beq offset
010	bne rA, offset	bne offset
011	blt rA, offset	blt offset
100	ble rA, offset	ble offset

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
101	bgt rA, offset	bgt offset
110	bge rA, offset	bge offset
111	bov rA, offset	bov offset

Other registers altered:

- None

The **ret** instruction is a simplified mnemonic for **b r31, 0**.

ba*x*- Branch on All

ba*x* rA, offset (register-relative format)

0–5	6	7	8–10	11–15	16–31
111100	0	0	CCC	rA	offset

ba*x* offset (PC-relative format)

0–5	6	7	8–10	11–31
111100	1	0	CCC	offset

```
if condition indicated by CCC is true for all WideWord datapaths
    if PC-relative format
        $PC \leftarrow IADR + ((offset_0)^9 \| offset \| 00)$
    else
        $PC \leftarrow ((rA) \wedge 0xFFFFFFFC) \vee ((offset_0)^{14} \| offset \| 00)$
```

This conditional branch instruction succeeds if the condition specified by CCC is true for all WideWord datapaths. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot).

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	b rA, offset	b offset
001	baeq rA, offset	baeq offset
010	bane rA, offset	bane offset
011	balt rA, offset	balt offset
100	bale rA, offset	bale offset

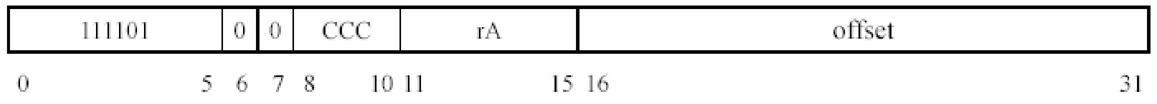
CCC	**Register-Relative Mnemonic**	**PC-Relative Mnemonic**
101	bagt rA, offset	bagt offset
110	bage rA, offset	bage offset
111	baov rA, offset	baov offset

Other registers altered:

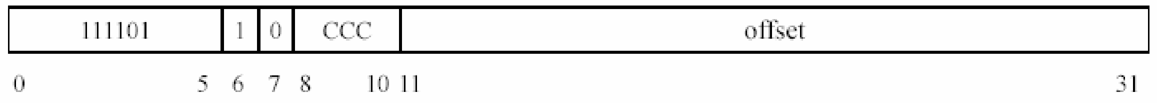
- None

bn*x*- Branch on None

bn*x* rA, offset (register-relative format)



bn*x* offset (PC-relative format)



if condition indicated by CCC is false for all WideWord datapaths

if PC-relative format

$$PC \leftarrow IADR + ((offset_0)^9 \| offset \| 00)$$

else

$$PC \leftarrow ((rA) \wedge 0x\text{FFFFFFFC}) \vee ((offset_0)^{14} \| offset \| 00)$$

This conditional branch instruction succeeds if the condition specified by CCC is false for all WideWord datapaths. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot).

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	b rA, offset	b offset
001	bneq rA, offset	bneq offset
010	bnne rA, offset	bnne offset
011	bnlt rA, offset	bnlt offset
100	bnle rA, offset	bnle offset

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
101	bngt rA, offset	bngt offset
110	bnge rA, offset	bnge offset
111	bnov rA, offset	bnov offset

Other registers altered:

- None

call*x*- Call

call*x* rA, offset (register-relative format)

111111	0	1	CCC	rA	offset
0 5	6	7	8 10	11 15	16 31

call*x* offset (PC-relative format)

111111	1	1	CCC	offset
0 5	6	7	8 10	11 31

```
if scalar condition indicated by CCC
    $r31 \leftarrow IADR + 8$
    if PC-relative format
        $PC \leftarrow IADR + ((offset_0)^9 \| offset \| 00)$
    else
        $PC \leftarrow ((rA) \wedge 0x\mathrm{FFFFFFFC}) \vee ((offset_0)^{14} \| offset \| 00)$
```

This call instruction is conditional upon the scalar condition specified by CCC. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot). The effective address of the instruction following the delay slot is placed into r31.

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	call rA, offset	call offset
001	calleq rA, offset	calleq offset
010	callne rA, offset	callne offset

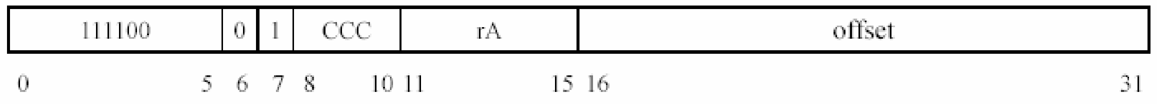
CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
011	calllt rA, offset	calllt offset
100	callle rA, offset	callle offset
101	callgt rA, offset	callgt offset
110	callge rA, offset	callge offset
111	callov rA, offset	callov offset

Other registers altered:

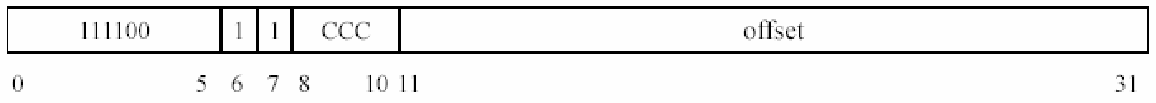
- None

call*ax*- Call on All

call*ax* rA, offset (register-relative format)



call*ax* offset (PC-relative format)



if condition indicated by CCC is true for all WideWord datapaths

$r31 \leftarrow IADR + 8$

if PC-relative format

$$PC \leftarrow IADR + ((offset_0)^9 \parallel offset \parallel 00)$$

else

$$PC \leftarrow ((rA) \wedge 0x\mathrm{FFFFFFFC}) \vee ((offset_0)^{14} \parallel offset \parallel 00)$$

This conditional call instruction succeeds if the condition specified by CCC is true for all WideWord datapaths. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot). The effective address of the instruction following the delay slot is placed into r31.

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	call rA, offset	call offset
001	callaeq rA, offset	callaeq offset
010	callane rA, offset	callane offset

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
011	callalt rA, offset	callalt offset
100	callale rA, offset	callale offset
101	callagt rA, offset	callagt offset
110	callage rA, offset	callage offset
111	callaov rA, offset	callaov offset

Other registers altered:

None

callnx- Call on None

callnx rA, offset (register-relative format)

111101	0	1	CCC	rA	offset
0–5	6	7	8–10	11–15	16–31

callnx offset (PC-relative format)

111101	1	1	CCC	offset
0–5	6	7	8–10	11–31

```
if condition indicated by CCC is false for all WideWord datapaths
    $r31 \leftarrow IADR + 8$
    if PC-relative format
        $PC \leftarrow IADR + ((offset_0)^9 \| offset \| 00)$
    else
        $PC \leftarrow ((rA) \wedge 0x\mathrm{FFFFFFFC}) \vee ((offset_0)^{14} \| offset \| 00)$
```

This conditional call instruction succeeds if the condition specified by CCC is false for all WideWord datapaths. For the register-relative format, the target address is formed by ORing the offset with the contents of rA. For the PC-relative format, the target address is formed by adding the offset to the instruction address. In both cases, the offset is considered to be a signed instruction count, so it is shifted left two bits and sign-extended. Furthermore, the least two significant bits of the contents of rA are ignored in the register-relative format so that a proper instruction-aligned address results. The next instruction is always executed (one delay slot). The effective address of the instruction following the delay slot is placed into r31.

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
000	call rA, offset	call offset
001	callneq rA, offset	callneq offset
010	callnne rA, offset	callnne offset

CCC	Register-Relative Mnemonic	PC-Relative Mnemonic
011	callnlt rA, offset	callnlt offset
100	callnle rA, offset	callnle offset
101	callngt rA, offset	callngt offset
110	callnge rA, offset	callnge offset
111	callnov rA, offset	callnov offset

Other registers altered:

None

clo*x* - Clear Leftmost One

Scalar Unit

clo rD, rA (C = 0)

cloc rD, rA (C = 1)

000011	rD	rA	00000	C		001001
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 31 to 0
    if (rA)_i
        $tmp \leftarrow i$
$rD \leftarrow (rA) \wedge (1^{tmp} \| 0 \| 1^{31-tmp})$
```

The contents of rA are searched to find the leftmost bit that is a one. The resulting value of clearing this bit but retaining the other bits is then stored in rD.

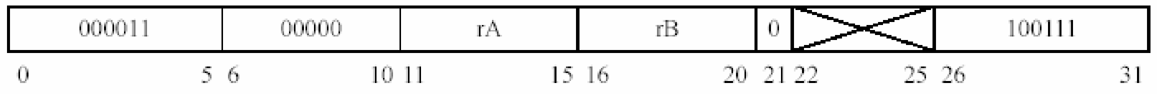
Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

div - Divide

Scalar Unit

div rA, rB



$HI \leftarrow (rA) \div (rB)$

$LO \leftarrow (rA)\mathrm{mod}(rB)$

The contents of rA are divided by the contents of rB, treating both operands as signed values. No condition codes are updated as a result of this operation. When the operation completes, the quotient word is loaded into special register HI, and the remainder word is loaded into special register LO. This operation requires 38 clock cycles in the worst case and thus requires some amount of scheduling.

Other registers altered:

- None

divu - Divide Unsigned

Scalar Unit

divu rA, rB

000011	00000	rA	rB	1		100111
0 5	6 10	11 15	16 20	21	22 25	26 31

$HI \leftarrow (rA) \div (rB)$

$LO \leftarrow (rA) \bmod (rB)$

The contents of rA are divided by the contents of rB, treating both operands as unsigned values. No condition codes are updated as a result of this operation. When the operation completes, the quotient word is loaded into special register HI, and the remainder word is loaded into special register LO. This operation requires 38 clock cycles in the worst case and thus requires some amount of scheduling.

Other registers altered:

- None

elo - Encode Leftmost One

Scalar Unit

elo rD, rA

000011	rD	rA	00000	0		001000
0 5	6 10	11 15	16 20	21	22 25	26 31

```
$tmp \leftarrow$ 0xFFFFFFFF
for i = 31 to 0
    if $(rA)_i$
        $tmp \leftarrow i$
$rD \leftarrow tmp$
```

The contents of rA are searched to find the leftmost bit that is a one. The index of this bit is then stored in rD. If no bit of the contents of rA is a one, the value 0xFFFFFFFF is stored in rD.

Other registers altered:

- None

icli - Instruction Cache Line Invalidate

icli rA, offset

110011	00000	rA	offset
0 5	6 10	11 15	16 31

$EA \leftarrow (rA) + ((offset_0)^{16} \| offset)$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. If the EA is contained in the instruction cache, the cache line containing that address is invalidated. This instruction may be executed only in supervisor mode.

Other registers altered:

- None

ld - Load General-Purpose Register

Scalar Unit

ld rD, rA, offset

110000	rD	rA	offset
0 5	6 10	11 15	16 31

$EA \leftarrow 0x\mathrm{FFFFFFFC} \wedge ((rA) + ((offset_0)^{16} \| offset))$

$rD \leftarrow \mathrm{MEM[EA]}$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 32-bit word at the memory location specified by EA (ignoring the least two significant bits to ensure a 32-bit aligned address) is then loaded into rD.

Other registers altered:

- None

lokl - Lock Load

Scalar Unit

lokl rD, rA, offset

110110	rD	rA	offset
0 5	6 10	11 15	16 31

$EA \leftarrow 0x\text{FFFFFFFC} \wedge ((rA) + ((offset_0)^{16} \parallel offset))$

$rD \leftarrow \text{MEM[EA]}$

$LOCK \leftarrow 1$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 32-bit word at the memory location specified by EA (ignoring the least two significant bits to ensure a 32-bit aligned address) is then loaded into rD. The hardware lock bit is also set and remains set until a **loks** instruction is executed or an exception occurs.

Other registers altered:

- None

loks - Lock Store

Scalar Unit

loks rD, rA, offset

110111	rD	rA	offset
0 5	6 10	11 15	16 31

```
$EA \leftarrow 0xFFFFFFFC \wedge ((rA) + ((offset_0)^{16} \| offset))$
if (LOCK = 1)
    MEM[EA] $\leftarrow rD$
$rD \leftarrow LOCK^{32}$
$LOCK \leftarrow 0$
```

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 32-bit word contents of rD are conditionally stored at the memory location specified by EA (ignoring the least two significant bits to ensure a 32-bit aligned address). The success or failure of the store operation is indicated by the contents of rD after execution of the instruction. If an exception occurs between the last **lokl** and this **loks** instruction, the store is inhibited from taking place and the **loks** fails. The operation of **loks** is undefined when the address is different from the address used in the last **lokl**.

Other registers altered:

- None

mfatr - Move from Address Translation Register

Scalar Unit

mfatr rD, atrA

000000	rD	atrA	00000	0		000010
0 5	6 10	11 15	16 20	21	22 25	26 31

$rD \leftarrow (atrA)$

The contents of address translation register atrA are stored in rD. A list of the address translation registers and their encoding is found in Table 5. This instruction may be executed only in supervisor mode.

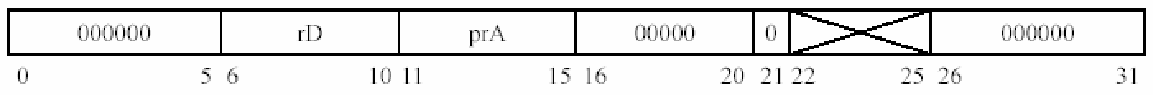
Other registers altered:

- None

mfpr - Move from Protected Register

Scalar Unit

mfpr rD, prA



$rD \leftarrow (prA)$

The contents of protected register prA are stored in rD. A list of the protected registers and their encoding is found in Table 4. This instruction may be executed only in supervisor mode.

Other registers altered:

- None

mfspr - Move from Special-Purpose Register

Scalar Unit

mfspr rD, sprA

000001	rD	sprA	00000	0		000100
0 5	6 10	11 15	16 20	21	22 25	26 31

rD ← (*sprA*)

The contents of special-purpose register sprA are stored in rD. A list of the special-purpose registers and their encoding is found in Table 3.

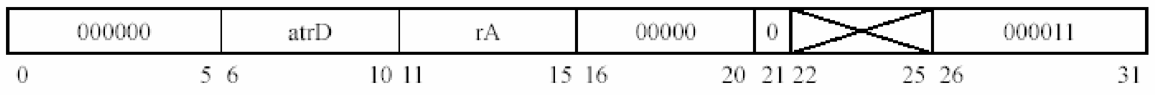
Other registers altered:

- None

mtatr - Move to Address Translation Register

Scalar Unit

mtatr   atrD, rA



$atrD \leftarrow (rA)$

The contents of general-purpose register rA are stored in address translation register atrD. A list of the address translation registers and their encoding is found in Table 5. This instruction may be executed only in supervisor mode.

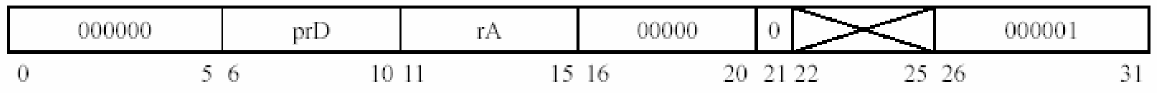
Other registers altered:

- None

mtpr - Move to Protected Register

Scalar Unit

mtpr prD, rA



prD ← (*rA*)

The contents of general-purpose register rA are stored in protected register prD. A list of the protected registers and their encoding is found in Table 4. This instruction may be executed only in supervisor mode.

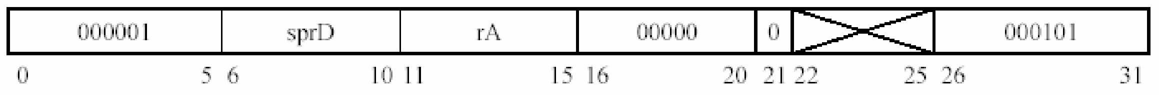
Other registers altered:

- None

mtspr - Move to Special-Purpose Register

Scalar Unit

mtspr sprD, rA



sprD ← (*rA*)

The contents of general-purpose register rA are stored in special-purpose register sprD. A list of the special-purpose registers and their encoding is found in Table 3.

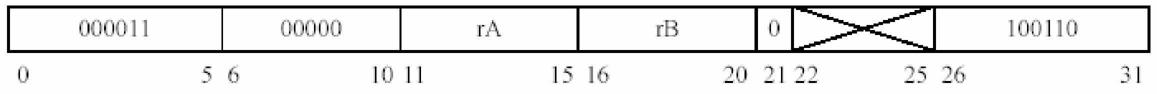
Other registers altered:

- None

mul - Multiply

Scalar Unit

mul rA, rB



$LO \leftarrow ((rA) \times (rB))_{32,\,63}$

$HI \leftarrow ((rA) \times (rB))_{0,\,31}$

The contents of rA are multiplied by the contents of rB, treating both operands as signed values. No condition codes are updated as a result of this operation. When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word is loaded into special register HI. This operation requires 4 clock cycles and thus requires some amount of scheduling.

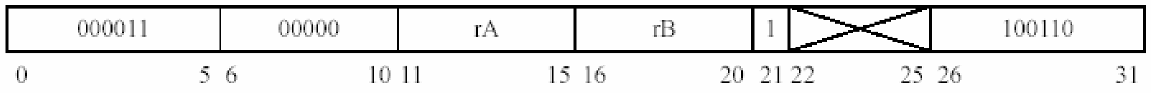
Other registers altered:

- None

mulu - Multiply Unsigned

Scalar Unit

mulu rA, rB



$LO \leftarrow ((rA) \times (rB))_{32,\,63}$

$HI \leftarrow ((rA) \times (rB))_{0,\,31}$

The contents of rA are multiplied by the contents of rB, treating both operands as unsigned values. No condition codes are updated as a result of this operation. When the operation completes, the low-order word of the double result is loaded into special register LO, and the high-order word is loaded into special register HI. This operation requires 4 clock cycles and thus requires some amount of scheduling.

Other registers altered:

- None

mvsw*x* - Move from Scalar to WideWord

mvsw*w* **wrD, rA, index** **(T = 0)**

mvswr*pw* **wrD, rA** **(T = 1)**

000100	wrD	rA	index	T	PPWW	000100
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

```
$base \leftarrow index \wedge mask$
if (T = 0)
    $wrD_{base \times 8, (base \times 8) + (size - 1)} \leftarrow (rA)_{(32 - size), 31}$
else
    for i = 0 to (256 - size) by size
        $wrD_{i, i + (size - 1)} \leftarrow (rA)_{(32 - size), 31}$
```

If T=0, some portion or all of the contents of rA are transferred to a subfield of wrD, starting at the byte specified by the byte index. Depending on the size of the data to be transferred, the least significant bits of the index may be ignored to ensure proper alignment. If T=1, the contents of rA are replicated to form a 256-bit value which is transferred to wrD, subject to the participation mode specified by PP.

Other registers altered:

- None

mvswi - Move from Scalar to WideWord Indirect

mvswi*w*   wrD, rA, rB

000100	wrD	rA	rB	0	00WW	100100
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

$base \leftarrow (rB)_{27,31} \wedge mask$

$wrD_{base \times 8, (base \times 8) + (size - 1)} \leftarrow (rA)_{(32 - size), 31}$

Some portion or all of the contents of rA are transferred to a subfield of wrD, starting at the byte specified by the low-order bit contents of rB. Depending on the size of the data to be transferred, the least significant bits of the contents of rB may be ignored to ensure proper alignment.

Other registers altered:

- None

mvws - Move from WideWord to Scalar

mvws*w* rD, wrA, index

000100	rD	wrA	index	0	00WW	000001
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

$base \leftarrow index \wedge mask$

$rD_{(32-size),\,31} \leftarrow (wrA)_{base \times 8,\,(base \times 8) + (size - 1)}$

if (size != 32)

$\quad rD_{0,\,(32 - size - 1)} \leftarrow 0^{(32 - size)}$

A subfield of the contents of wrA starting at the byte specified by the byte index are transferred to rD. Depending on the size of the data to be transferred, the least significant bits of the index may be ignored to ensure proper alignment. For data sizes less than 32 bits, the high-order bits of rD are cleared.

Other registers altered:

- None

mvwsi - Move from WideWord to Scalar Indirect

mvwsi*w*   rD, wrA, rB

000100	rD	wrA	rB	0	00WW	100001
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

$base \leftarrow (rB)_{27,\,31} \wedge mask$

$rD_{(32-size),\,31} \leftarrow (wrA)_{base \times 8,\,(base \times 8)+(size-1)}$

if (size != 32)

$\quad rD_{0,\,(32-size-1)} \leftarrow 0^{(32-size)}$

A subfield of the contents of wrA starting at the byte specified by the low-order bits of the contents of rB are transferred to rD. Depending on the size of the data to be transferred, the least significant bits of the contents of rB may be ignored to ensure proper alignment. For data sizes less than 32 bits, the high-order bits of rD are cleared.

Other registers altered:

- None

mvww*x* - Move from WideWord to WideWord

mvww*p*   wrD, wrA   (T = 0)

mvwwr*pw*   wrD, wrA, index   (T = 1)

000100	wrD	wrA	index	T	PPWW	000000
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

```
$base \leftarrow index \wedge mask$
if (T = 0)
    $wrD \leftarrow (wrA)$
else
    for i = 0 to (256 - size) by size
        $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{base \times 8,\, (base \times 8)+(size-1)}$
```

If T=0, the entire 256-bit contents of wrA are transferred to wrD, subject to the participation mode specified by PP. If T=1, the subfield of wrA starting at the byte specified by the byte index and of the size indicated by the WW bits is replicated to form a 256-bit value which is transferred to wrD, subject to the participation mode specified by PP. Depending on the size of the data to be transferred, the least significant bits of the index may be ignored to ensure proper alignment.

Other registers altered:

- None

mvwwir - Move from WideWord to WideWord Indirect Replicating

mvwwir*pw* wrD, wrA, rB

000100	wrD	wrA	rB	1	PPWW	100000
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	mask
00	8	0b11111
01	16	0b11110
10	32	0b11100

$base \leftarrow (rB)_{27,31} \wedge mask$

```
for i = 0 to (256 - size) by size
    $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{base \times 8,\, (base \times 8)+(size-1)}$
```

The subfield of wrA starting at the byte specified by the low-order bits of the contents of rB and of the size indicated by the WW bits is replicated to form a 256-bit value which is transferred to wrD, subject to the participation mode specified by PP. Depending on the size of the data to be transferred, the least significant bits of the contents of rB may be ignored to ensure proper alignment.

Other registers altered:

- None

not*x* - NOT

Scalar Unit

not rD, rA (C = 0)

notc rD, rA (C = 1)

000011	rD	rA	00000	C	╳	101110
0 5	6 10	11 15	16 20	21	22 25	26 31

$$rD \leftarrow \neg(rA)$$

The bitwise inversion of the contents of rA is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

or*x* - OR

Scalar Unit

or rD, rA, rB (C = 0)

orc rD, rA, rB (C = 1)

000011	rD	rA	rB	C		101100
0 5	6 10	11 15	16 20	21	22 25	26 31

$$rD \leftarrow (rA) \vee (rB)$$

The contents of rA are ORed with rB, and the result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

ori - OR Immediate

Scalar Unit

ori rD, rA, IMM

101100	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) \vee (0^{16} \,||\, IMM)$$

The contents of rA are ORed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- None

oric - OR Immediate Recording Condition Codes

Scalar Unit

oric rD, rA, IMM

101101	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) \vee (0^{16} \,\|\, IMM)$$

The contents of rA are ORed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- Scalar condition code registers: LT, GT, EQ

oris - OR Immediate Shifted

Scalar Unit

oris rD, rA, IMM

101110	rD	rA	IMM
0 5	6 10	11 15	16 31

$$rD \leftarrow (rA) \vee (IMM \,\|\, 0^{16})$$

The contents of rA are ORed with IMM (appended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- None

probe - Probe Address

Scalar Unit

probe rD, rA, offset

110010	rD	rA	offset
0 5	6 10	11 15	16 31

```
$EA \leftarrow (rA) + ((offset_0)^{16} \| offset)$
if EA is locally mapped
    $rD \leftarrow$ 0xFFFFFFFF
else
    $rD \leftarrow$ 0x00000000
```

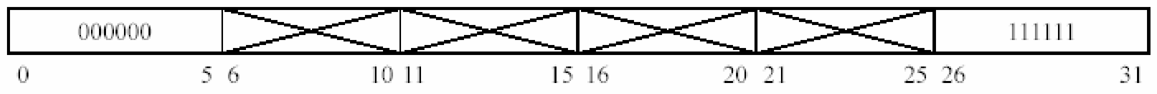
The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The effective address is then forwarded to the address translation hardware to determine if the address is a valid local address. The success or failure of the operation is indicated by the contents of rD after execution of the instruction.

Other registers altered:

- None

rfe - Return from Exception

rfe



$PC \leftarrow (FADR)$

$PSW \leftarrow (SSW)$

The program counter, PC, is loaded with the contents of the protected register FADR. Similarly, the PSW is loaded with the contents of SSW. The next instruction is always executed (one delay slot).

Other registers altered:

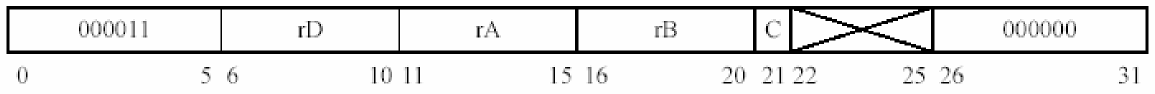
- None

sll*x* - Shift Left Logical

Scalar Unit

sll rD, rA, rB (C = 0)

sllc rD, rA, rB (C = 1)



$s \leftarrow (rB)_{27,\,31}$

$rD \leftarrow (rA)_{s,\,31} \,\|\, 0^s$

The contents of rA are shifted left by the number of bits specified by the low order five bits contained as contents of rB, inserting zeros into the low order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

slli*x* - Shift Left Logical Immediate

Scalar Unit

slli rD, rA, shift_amount (C = 0)

sllic rD, rA, shift_amount (C = 1)

000011	rD	rA	shift_amount	C	⊠	000010
0 5	6 10	11 15	16 20	21	22 25	26 31

$s \leftarrow$ shift_amount

$rD \leftarrow (rA)_{s,\,31} \,\|\, 0^{s}$

The contents of rA are shifted left by *shift_amount* bits, inserting zeros into the low-order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

sra*x* - Shift Right Arithmetic

Scalar Unit

sra rD, rA, rB (C = 0)

srac rD, rA, rB (C = 1)

000011	rD	rA	rB	C		000101
0 5	6 10	11 15	16 20	21	22 25	26 31

$s \leftarrow (rB)_{27,\,31}$

$rD \leftarrow ((rA)_0)^s \,\|\, (rA)_{0,\,(31-s)}$

The contents of rA are shifted right by the number of bits specified by the low order five bits contained as contents of rB, sign-extending the high-order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

sra*ix* - Shift Right Arithmetic Immediate

Scalar Unit

srai rD, rA, shift_amount (C = 0)

sraic rD, rA, shift_amount (C = 1)

000011	rD	rA	shift_amount	C		000111
0 5	6 10	11 15	16 20	21	22 25	26 31

$s \leftarrow$ shift_amount

$rD \leftarrow ((rA)_0)^s \,\|\, (rA)_{0,\,(31-s)}$

The contents of rA are shifted right by *shift_amount* bits, sign-extending the high-order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

srl*x* - Shift Right Logical

Scalar Unit

srl rD, rA, rB (C = 0)

srlc rD, rA, rB (C = 1)

000011	rD	rA	rB	C	╳	000001
0 5	6 10	11 15	16 20	21	22 25	26 31

$s \leftarrow (rB)_{27,\,31}$

$rD \leftarrow 0^{s} \,\|\, (rA)_{0,\,(31\text{-}s)}$

The contents of rA are shifted right by the number of bits specified by the low order five bits contained as contents of rB, inserting zeros into the high-order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

srli*x* - Shift Right Logical Immediate

Scalar Unit

srli **rD, rA, shift_amount (C = 0)**

srlic **rD, rA, shift_amount (C = 1)**

000011	rD	rA	shift_amount	C		000011
0 5	6 10	11 15	16 20	21	22 25	26 31

$s \leftarrow$ shift_amount

$$rD \leftarrow 0^s \,\|\, (rA)_{0,\,(31\text{-}s)}$$

The contents of rA are shifted right by *shift_amount* bits, inserting zeros into the high-order bits of the result. The result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

st - Store General-Purpose Register

Scalar Unit

st rD, rA, offset

110001	rD	rA	offset
0 5	6 10	11 15	16 31

$EA \leftarrow 0x\mathrm{FFFFFFFC} \wedge ((rA) + (offset_0)^{16} \,\|\, offset)$

$\mathrm{MEM[EA]} \leftarrow rD$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 32-bit word contents of rD are stored at the memory location specified by EA (ignoring the least two significant bits to ensure a 32-bit aligned address).

Other registers altered:

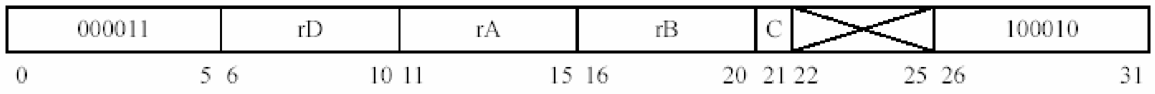
- None

sub*x* - Subtract

Scalar Unit

sub rD, rA, rB (C = 0)

subc rD, rA, rB (C = 1)



$rD \leftarrow (rA) + \neg(rB) + 1$

The contents of rB are subtracted from the contents of rA, and the result is placed into rD.

Other registers altered:

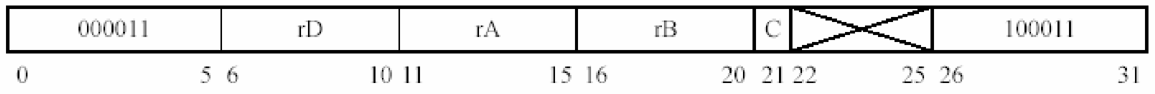
- If C =1, scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

sube*x* - Subtract Extended

Scalar Unit

sube rD, rA, rB (C = 0)

subec rD, rA, rB (C = 1)



$rD \leftarrow (rA) + \neg(rB) + CA$

The contents of rB are subtracted from the contents of rA, using the carry bit CA as the carry in, and the result is placed into rD.

Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

subu - Subtract

Scalar Unit

subu rD, rA, rB

000011	rD	rA	rB	1		100100
0 5	6 10	11 15	16 20	21	22 25	26 31

$rD \leftarrow (rA) + \neg(rB) + 1$

The contents of rB are subtracted from the contents of rA, and the result is placed into rD. This instruction is identical to **sub** except that the OV condition code is updated to reflect unsigned arithmetic.

Other registers altered:

- Scalar condition code registers: LT, GT, EQ, CA
- Scalar condition code OV is set if the operation causes overflow.

sys - System Call

sys

000001	code	000000
0 5	6 25	26 31

A system call is made by setting bit 19 of the ESW (Exception Source Word) register which in turn triggers an exception. Refer to Chapter 9 of the DIVA PIM Node Architecture manual for details regarding exceptions.

Other registers altered:

- None

wadd*x* - WideWord Add

WideWord Unit

waddpw *(pw italic)* **wrD, wrA, wrB (C = 0)**

waddcpw **wrD, wrA, wrB (C = 1)**

000010	wrD	wrA	wrB	C	PPWW	100000
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	**size**
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+(size-1)} \leftarrow (wrA)_{i,\,i+(size-1)} + (wrB)_{i,\,i+(size-1)}$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate sums of the aligned data fields of wrA and wrB are placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ, CA
- A WideWord OV condition code bit is set if the operation in its corresponding datapath causes overflow.

waddex - WideWord Add Extended

WideWord Unit

wadde*pw* wrD, wrA, wrB (C = 0)

waddec*pw* wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	100001
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{i,\, i+(size-1)} + (wrB)_{i,\, i+(size-1)} + CA_{i/8}$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate sums of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Each data field uses the associated bit of the WideWord Carry register as a carry in for the operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ, CA
- A WideWord OV condition code bit is set if the operation in its corresponding datapath causes overflow.

wand*x* - WideWord AND

WideWord Unit

wand*pw*   wrD, wrA, wrB (C = 0)

wandc*pw*   wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	101000
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	**size**
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{i,\, i+(size-1)} \wedge (wrB)_{i,\, i+(size-1)}$
```

The 256-bit contents of wrA are ANDed with the 256-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies and how condition codes are updated for this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wfabs*x* - WideWord Floating-Point Absolute Value

WideWord Unit

wfabs*p* wrD, wrA (C = 0)

wfabsc*p* wrD, wrA (C = 1)

011101	wrD	wrA	00000	C	PP10	000101
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+31} \leftarrow |(wrA)_{i,\,i+31}|$ (using floating-point arithmetic)
```

The 256-bit contents of wrA are treated as 8 single-precision floating-point operands. The aggregate absolute values of the floating-point operands of wrA are placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfadd*x* - WideWord Floating-Point Add

WideWord Unit

wfadd*p* wrD, wrA, wrB (C = 0)

wfaddc*p* wrD, wrA, wrB (C = 1)

011101	wrD	wrA	wrB	C	PP10	000000
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+31} \leftarrow (wrA)_{i,\,i+31} + (wrB)_{i,\,i+31}$ (using floating-point arithmetic)
```

The 256-bit contents of wrA and wrB are treated as 8 single-precision floating-point operands. The aggregate floating-point sums of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfdiv*x* - WideWord Floating-Point Divide

WideWord Unit

wfdiv*p* **wrD, wrA, wrB (C = 0)**

wfdivc*p* **wrD, wrA, wrB (C = 1)**

011101	wrD	wrA	wrB	C	PP10	000111
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+31} \leftarrow (wrA)_{i,\,i+31} \div (wrB)_{i,\,i+31}$ (using floating-point arithmetic)
```

The 256-bit contents of wrA and wrB are treated as 8 single-precision floating-point operands. The aggregate floating-point quotients of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfmul*x* - WideWord Floating-Point Multiply

WideWord Unit

wfmul*p* wrD, wrA, wrB (C = 0)

wfmulc*p* wrD, wrA, wrB (C = 1)

011101	wrD	wrA	wrB	C	PP10	000110
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,i+31} \leftarrow (wrA)_{i,i+31} \times (wrB)_{i,i+31}$ (using floating-point arithmetic)
```

The 256-bit contents of wrA and wrB are treated as 8 single-precision floating-point operands. The aggregate floating-point products of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfneg*x* - WideWord Floating-Point Negate

WideWord Unit

wfneg*p*   wrD, wrA   (C = 0)

wfnegc*p*   wrD, wrA   (C = 1)

011101	wrD	wrA	00000	C	PP10	000100
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+31} \leftarrow -((wrA)_{i,\,i+31})$ (using floating-point arithmetic)
```

The 256-bit contents of wrA are treated as 8 single-precision floating-point operands. The aggregate negations of the floating-point operands of wrA are placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfsub*x* - WideWord Floating-Point Subtract

WideWord Unit

wfsub*p*   wrD, wrA, wrB (C = 0)

wfsubc*p*   wrD, wrA, wrB (C = 1)

011101	wrD	wrA	wrB	C	PP10	000001
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i, i+31} \leftarrow (wrA)_{i, i+31} - (wrB)_{i, i+31}$ (using floating-point arithmetic)
```

The 256-bit contents of wrA and wrB are treated as 8 single-precision floating-point operands. The aggregate floating-point differences of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wfti*x* - WideWord Floating-Point to Integer

WideWord Unit

wfti*p* **wrD, wrA** **(C = 0)**

wftic*p* **wrD, wrA** **(C = 1)**

011101	wrD	wrA	00000	C	PP10	000010
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+31} \leftarrow \mathrm{int}((wrA)_{i,\,i+31})$ (assuming floating-point input operand)
```

The 256-bit contents of wrA are treated as 8 single-precision floating-point operands. Each single-precision floating-point operand is converted to a 32-bit integer, and the aggregation of these 8 integers are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

witf*x* - WideWord Integer to Floating-Point

WideWord Unit

witf*p* **wrD, wrA** **(C = 0)**

witfc*p* **wrD, wrA** **(C = 1)**

011101	wrD	wrA	00000	C	PP10	000011
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 224 by 32
    if PP bits and conditions are set accordingly
        $wrD_{i,i+31} \leftarrow \mathrm{fp}((wrA)_{i,i+31})$  (assuming integer input operand)
```

The 256-bit contents of wrA are treated as eight 32-bit integer operands. Each integer operand is converted to a singe-precision floating-point number, and the aggregation of these 8 single-precision floating-point numbers are placed into wrD, subject to participation. Floating-point exceptions may be triggered by this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ
- FPSR may also be updated if any floating-point exceptions occur.

wld - Load WideWord Register

WideWord Unit

wld wrD, rA, offset

110100	wrD	rA	offset
0 5	6 10	11 15	16 31

$EA \leftarrow 0x\text{FFFFFFE0} \wedge ((rA) + ((offset_0)^{16} \| offset))$

$wrD \leftarrow \text{MEM[EA]}$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 256-bit value at the memory location specified by EA (ignoring the least five significant bits to ensure a 256-bit aligned address) is then loaded into wrD.

Other registers altered:

- None

wmrg*x* - WideWord Merge

WideWord Unit

wmrg*cp*   wrD, wrA, wrB (C = 0)

wmrgc*cp*   wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	101111
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	CC	Mnemonic (*c*)
00	EQ	eq
01	LT	lt
10	GT	gt
11	M	m

```
for i = 0 to 248 by 8
    if PP bits and conditions are set accordingly
        if $CC_{i/8} = 1$
            $wrD_{i,i+7} \leftarrow (wrA)_{i,i+7}$
        else
            $wrD_{i,i+7} \leftarrow (wrB)_{i,i+7}$
```

Each bit of the WideWord condition code register specified by the WW bits of the instruction serves as a selector. If the bit is 1, the corresponding byte contents of wrA are placed into the corresponding byte lane of wrD, subject to participation. If the bit is 0, the corresponding byte contents of wrB are placed into the corresponding byte lane of wrD, subject to participation.

Other registers altered:

If C =1, WideWord condition code registers: LT, GT, EQ

wmules - WideWord Multiply Even Signed

WideWord Unit

wmules*pw* wrD, wrA, wrB

000010	wrD	wrA	wrB	0	PPWW	100110
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
01	8
10	16

```
for i = 0 to (256 - $2 \times$ size ) by $2 \times$ size
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(2 \times size - 1)} \leftarrow (wrA)_{i,\, i+(size-1)} \times (wrB)_{i,\, i+(size-1)}$
```

Each even-numbered signed-integer byte or half-word of wrA is multiplied by the corresponding signed-integer byte or half-word of wrB, where the WW field determines if the 256-bit contents of wrA and wrB are treated as bytes or half-words. The resulting signed halfword or word products are placed, in the same order, into wrD, subject to participation. No condition codes are updated as a result of this operation.

Other registers altered:

- None

wmuleu - WideWord Multiply Even Unsigned

WideWord Unit

wmuleu*pw* wrD, wrA, wrB

000010	wrD	wrA	wrB	1	PPWW	100110
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
01	8
10	16

```
for i = 0 to (256 - $2 \times$ size ) by $2 \times$ size
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+(2\times size-1)} \leftarrow (wrA)_{i,\,i+(size-1)} \times (wrB)_{i,\,i+(size-1)}$
```

Each even-numbered unsigned-integer byte or half-word of wrA is multiplied by the corresponding unsigned-integer byte or half-word of wrB, where the WW field determines if the 256-bit contents of wrA and wrB are treated as bytes or half-words. The resulting unsigned half-word or word products are placed, in the same order, into wrD, subject to participation. No condition codes are updated as a result of this operation.

Other registers altered:

- None

wmulos - WideWord Multiply Odd Signed

WideWord Unit

wmulos*pw* wrD, wrA, wrB

000010	wrD	wrA	wrB	0	PPWW	100111
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
01	8
10	16

```
for i = 0 to (256 - $2 \times size$ ) by $2 \times size$
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(2\times size-1)} \leftarrow (wrA)_{i+size,\, i+(2\times size-1)} \times (wrB)_{i+size,\, i+(2\times size-1)}$
```

Each odd-numbered signed-integer byte or half-word of wrA is multiplied by the corresponding signed-integer byte or half-word of wrB, where the WW field determines if the 256-bit contents of wrA and wrB are treated as bytes or half-words. The resulting signed halfword or word products are placed, in the same order, into wrD, subject to participation. No condition codes are updated as a result of this operation.

Other registers altered:

- None

wmulou - WideWord Multiply Odd Unsigned

WideWord Unit

wmulou*pw* wrD, wrA, wrB

000010	wrD	wrA	wrB	1	PPWW	100111
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
01	8
10	16

```
for i = 0 to (256 - $2 \times size$ ) by $2 \times size$
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(2 \times size - 1)} \leftarrow (wrA)_{i+size,\, i+(2 \times size - 1)} \times (wrB)_{i+size,\, i+(2 \times size - 1)}$
```

Each odd-numbered unsigned-integer byte or half-word of wrA is multiplied by the corresponding unsigned-integer byte or half-word of wrB, where the WW field determines if the 256-bit contents of wrA and wrB are treated as bytes or half-words. The resulting unsigned half-word or word products are placed, in the same order, into wrD, subject to participation. No condition codes are updated as a result of this operation.

Other registers altered:

- None

wnot*x* - WideWord NOT

WideWord Unit

wnot*pw*   wrD, wrA   (C = 0)

wnotc*pw*   wrD, wrA   (C = 1)

000010	wrD	wrA	00000	C	PPWW	101110
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow \neg(wrA)_{i,\, i+(size-1)}$
```

The 256-bit contents of wrA are bitwise inverted, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies and how condition codes are updated for this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wor*x* - WideWord OR

WideWord Unit

wor*pw* **wrD, wrA, wrB (C = 0)**

worc*pw* **wrD, wrA, wrB (C = 1)**

000010	wrD	wrA	wrB	C	PPWW	101100
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{i,\, i+(size-1)} \vee (wrB)_{i,\, i+(size-1)}$
```

The 256-bit contents of wrA are ORed with the 256-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies and how condition codes are updated for this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wpks*x* - WideWord Pack Signed

WideWord Unit

wpks*w* wrD, wrA, wrB

000010	wrD	wrA	wrB	0	00WW	001110
0 5	6 10	11 15	16 20	21	22 26	27 31

Variable values in the following equations are as follows:

WW Value	size	min	max
01	16	-2^7	$2^7 - 1$
10	32	-2^{15}	$2^{15} - 1$

```
for i = 0 to (128 - (size/2) ) by (size/2)
    if $(wrA)_{i \times 2, (i \times 2) + \text{size} - 1} < \text{min}$
        $wrD_{i, i + (\text{size}/2) - 1} \leftarrow \text{min}$
    else if $(wrA)_{i \times 2, (i \times 2) + \text{size} - 1} > \text{max}$
        $wrD_{i, i + (\text{size}/2) - 1} \leftarrow \text{max}$
    else
        $wrD_{i, i + (\text{size}/2) - 1} \leftarrow (wrA)_{(i \times 2) + (\text{size}/2), (i \times 2) + \text{size} - 1}$
    if $(wrB)_{i \times 2, (i \times 2) + \text{size} - 1} < \text{min}$
        $wrD_{128 + i, 128 + i + (\text{size}/2) - 1} \leftarrow \text{min}$
    else if $(wrB)_{i \times 2, (i \times 2) + \text{size} - 1} > \text{max}$
        $wrD_{128 + i, 128 + i + (\text{size}/2) - 1} \leftarrow \text{max}$
    else
        $wrD_{128 + i, 128 + i + (\text{size}/2) - 1} \leftarrow (wrB)_{(i \times 2) + (\text{size}/2), (i \times 2) + \text{size} - 1}$
```

Let the source vector be the concatenation of the contents of wrA followed by wrB. Each signed integer half-word or word, as specified by the WW bits, of the source vector is converted to a signed integer byte or half-word, respectively. If the value of the source element is outside the bounds that can be represented in the width of the result element, the result saturates to the minimum or maximum value appropriately. The aggregate result is placed into wrD. Note that participation is not supported for this instruction.

Other registers altered:

- None

wpku*x* - WideWord Pack Unsigned

WideWord Unit

wpku*w*   wrD, wrA, wrB

000010	wrD	wrA	wrB	1	00WW	001110
0 5	6 10	11 15	16 20	21	22 26	27 31

Variable values in the following equations are as follows:

WW Value	size	max
01	16	$2^8 - 1$
10	32	$2^{16} - 1$

```
for i = 0 to (128 - (size/2) ) by (size/2)
    if $(wrA)_{i \times 2, (i \times 2) + \text{size} - 1} > \text{max}$
        $wrD_{i, i + (\text{size}/2) - 1} \leftarrow \text{max}$
    else
        $wrD_{i, i + (\text{size}/2) - 1} \leftarrow (wrA)_{(i \times 2) + (\text{size}/2), (i \times 2) + \text{size} - 1}$
    if $(wrB)_{i \times 2, (i \times 2) + \text{size} - 1} > \text{max}$
        $wrD_{128 + i, 128 + i + (\text{size}/2) - 1} \leftarrow \text{max}$
    else
        $wrD_{128 + i, 128 + i + (\text{size}/2) - 1} \leftarrow (wrB)_{(i \times 2) + (\text{size}/2), (i \times 2) + \text{size} - 1}$
```

Let the source vector be the concatenation of the contents of wrA followed by wrB. Each unsigned integer half-word or word, as specified by the WW bits, of the source vector is converted to an unsigned integer byte or half-word, respectively. If the value of the source element is greater than the maximum value that can be represented in the width of the result element, the result saturates to the maximum value. The aggregate result is placed into wrD. Note that participation is not supported for this instruction.

Other registers altered:

- None

wprm*x* - WideWord Permute

WideWord Unit

wprm*p*   wrD, wrA, wrB

000010	wrD	wrA	wrB	0	PP00	001000
0 5	6 10	11 15	16 20	21	22 25	26 31

```
for i = 0 to 248 by 8
    $s \leftarrow (wrB)_{i+3,\,i+7}$
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+7} \leftarrow (wrA)_{s\times 8,\,(s\times 8)+7}$
```

The contents of wrA are the source vector for this permutation operation. Bits 3 to 7 of each byte element of the contents of wrB are used to select a byte element from the source vector for each byte element of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- None

wprmi*x* - WideWord Permute Indirect

WideWord Unit

wprmi*p*   wrD, wrA, rB

000010	wrD	wrA	rB	0	PP00	001001
0 5	6 10	11 15	16 20	21	22 25	26 31

The following lookup table is used for selecting a permutation vector:

index	vector
0x00	0x000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
0x01	0x0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00
0x02	0x02030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001
0x03	0x030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102
0x04	0x0405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203
0x05	0x05060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001020304
0x06	0x060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405
0x07	0x0708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203040506
0x08	0x08090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001020304050607
0x09	0x090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708
0x0A	0x0A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203040506070809
0x0B	0x0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A
0x0C	0x0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B
0x0D	0x0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C
0x0E	0x0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D
0x0F	0x0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E
0x10	0x101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F
0x11	0x1112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10
0x12	0x12131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011
0x13	0x131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112
0x14	0x1415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213

index	vector
0x15	0x15161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011121314
0x16	0x161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415
0x17	0x1718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213141516
0x18	0x18191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011121314151617
0x19	0x191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718
0x1A	0x1A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213141516171819
0x1B	0x1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A
0x1C	0x1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B
0x1D	0x1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C
0x1E	0x1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D
0x1F	0x1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E
0x20	0x00020406080A0C0E10121416181A1C1E01030507090B0D0F11131517191B1D1F
0x21	0x0100030205040706090808B0A0D0C0F0E111013121514171619181B1A1D1C1F1E
0x22	0x03020100070605040B0A09080F0E0D0C13121110171615141B1A19181F1E1D1C
0x23	0x07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918
0x24	0x0F0E0D0C0B0A090807060504030201001F1E1D1C1B1A19181716151413121110
0x25	0x1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
0x26	0x0002010304060507080A090B0C0E0D0F10121113141615171 81A191B1C1E1D1F
0x27	0x0004010502060307080C090D0A0E0B0F101411151216131718 1C191D1A1E1B1F
0x28	0x00080109020A030B040C050D060E070F101811191 21A131B141C151D161E171F
0x29	0x0001040508090C0D101114151819 1C1D020306070A0B0E0F121316171A1B1E1F
0x2A	0x020300010607040 50A0B08090E0F0C0D121310111617141 51A1B18191E1F1C1D
0x2B	0x06070405020300010E0F0C0D0A0B080916171415121310111E1F1C1D1A1B1819
0x2C	0x0E0F0C0D0A0B080906070405020300011E1F1C1D1A1B18191617141512131011
0x2D	0x1E1F1C1D1A1B18191617141512131011 0E0F0C0D0A0B08090607040502030001
0x2E	0x0001040502030607080 90C0D0A0B0E0F101114151213161718191C1D1A1B1E1F
0x2F	0x0001080902030A0B04050C0D06070E0F101118191213 1A1B14151C1D16171E1F
0x30	0x0001020308090A0B101112131819 1A1B040506070C0D0E0F141516171C1D1E1F
0x31	0x04050607000102030C0D0E0F08090A0B14151617101112131C1D1E1F18191A1B
0x32	0x0C0D0E0F08090A0B040506070001020 31C1D1E1F18191A1B1415161710111213
0x33	0x1C1D1E1F18191A1B14151617101112130C0D0E0F08090A0B0405060700010203
0x34	0x0001020308090A0B040506070C0D0E0F101112131819 1A1B141516171C1D1E1F
0x35	0x0001020310111213040506071415161708090A0B18191A1B0C0D0E0F1C1D1E1F

index	vector
0x36	0x101112130001020314151617040506071819 1A1B08090A0B1C1D1E1F0C0D0E0F
0x37	0x08090A0B0C0D0E0F000102030405060718191A1B1C1D1E1F1011121314151617

```
$index \leftarrow (rB)_{26,\ 31}$
$permvector \leftarrow vector[index]$
for i = 0 to 248 by 8
    $s \leftarrow permvector_{i+3,\ i+7}$
    if PP bits and conditions are set accordingly
        $wrD_{i,\ i+7} \leftarrow (wrA)_{s \times 8,\ (s \times 8)+7}$
```

The contents of wrA are the source vector for this permutation operation. The permutation vector is selected from a lookup table using the least significant bits of the contents of rB as an index into the table. Bits 3 to 7 of each byte element of the permutation vector are used to select a byte element from the source vector for each byte element of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- None

wsll*x* - WideWord Shift Left Logical

WideWord Unit

wsll*pw* **wrD, wrA, wrB** **(C = 0)**

wsllc*pw* **wrD, wrA, wrB** **(C = 1)**

000010	wrD	wrA	wrB	C	PPWW	000000
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	**size**	**bits**
00	8	3
01	16	4
10	32	5

```
for i = 0 to (256 - size) by size
    $s \leftarrow (wrB)_{i+size-bits,\, i+size-1}$
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow (wrA)_{i+s,\, i+(size-1)} \parallel 0^s$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted left by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, inserting zeros into the low order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wslli*x* - WideWord Shift Left Logical Immediate

WideWord Unit

wslli*pw* wrD, wrA, shift_amount (C = 0)

wsllic*pw* wrD, wrA, shift_amount (C = 1)

000010	wrD	wrA	shift_amount	C	PPWW	000010
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

```
$s \leftarrow \text{shift\_amount}_{5-bits,4}$
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,i+(size-1)} \leftarrow (wrA)_{i+s,i+(size-1)} \| 0^s$
```

The WW field determines if the 256-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted left by the number of bits specified by the appropriate bits of the shift_amount, inserting zeros into the low order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wsra*x* - WideWord Shift Right Arithmetic

WideWord Unit

wsra*pw*   wrD, wrA, wrB   (C = 0)

wsrac*pw*   wrD, wrA, wrB   (C = 1)

000010	wrD	wrA	wrB	C	PPWW	000101
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

```
for i = 0 to (256 - size) by size
    $s \leftarrow (wrB)_{i + size - bits,\, i + size - 1}$
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i + (size - 1)} \leftarrow ((wrA)_i)^s \,\|\, (wrA)_{i,\, i + size - s - 1}$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, sign-extending the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

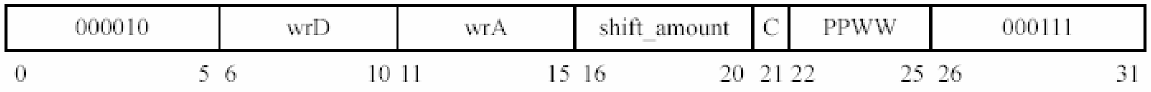
- If C =1, WideWord condition code registers: LT, GT, EQ

wsrai*x* - WideWord Shift Right Arithmetic Immediate

WideWord Unit

wsrai*pw*   wrD, wrA, shift_amount (C = 0)

wsraic*pw*   wrD, wrA, shift_amount (C = 1)



000010	wrD	wrA	shift_amount	C	PPWW	000111
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

```
$s \leftarrow \text{shift\_amount}_{5-bits,4}$
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,i+(size-1)} \leftarrow ((wrA)_i)^s \,\|\, (wrA)_{i,i+size-s-1}$
```

The WW field determines if the 256-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the appropriate bits of the shift_amount, sign-extending the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wsrl*x* - WideWord Shift Right Logical

WideWord Unit

wsrl*pw*   wrD, wrA, wrB   (C = 0)

wsrlc*pw*   wrD, wrA, wrB   (C = 1)

000010	wrD	wrA	wrB	C	PPWW	000001
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

```
for i = 0 to (256 - size) by size
    $s \leftarrow (wrB)_{i+size-bits,\, i+size-1}$
    if PP bits and conditions are set accordingly
        $wrD_{i,\, i+(size-1)} \leftarrow 0^{s} \,\|\, (wrA)_{i,\, i+size-s-1}$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the low order bits of the corresponding data field contained as contents of wrB, inserting zeros into the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wsrli*x* - WideWord Shift Right Logical Immediate

WideWord Unit

wsrli*pw* wrD, wrA, shift_amount (C = 0)

wsrlic*pw* wrD, wrA, shift_amount (C = 1)

000010	wrD	wrA	shift_amount	C	PPWW	000011
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size	bits
00	8	3
01	16	4
10	32	5

```
$s \leftarrow \text{shift\_amount}_{5-bits,4}$
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,i+(size-1)} \leftarrow 0^{s} \,\|\, (wrA)_{i,i+size-s-1}$
```

The WW field determines if the 256-bit contents of wrA are treated as 32 bytes, 16 half-words, or 8 words. The contents of each data field of wrA are shifted right by the number of bits specified by the appropriate bits of the shift_amount, inserting zeros into the high-order bits of each data field of the result. The result is placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

wst - Store WideWord Register

WideWord Unit

wst wrD, rA, offset

110101	wrD	rA	offset
0 – 5	6 – 10	11 – 15	16 – 31

$EA \leftarrow 0x\text{FFFFFFE0} \wedge ((rA) + ((offset_0)^{16} \| offset))$

$\text{MEM[EA]} \leftarrow wrD$

The 16-bit offset is sign-extended and added to the contents of rA to form the effective address EA. The 256-bit contents of wrD are stored at the memory location specified by EA (ignoring the least five significant bits to ensure a 256-bit aligned address).

Other registers altered:

- None

wsub*x* - WideWord Subtract

WideWord Unit

wsub*pw*   wrD, wrA, wrB (C = 0)

wsubc*pw*   wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	100010
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+(size-1)} \leftarrow (wrA)_{i,\,i+(size-1)} + \neg(wrB)_{i,\,i+(size-1)} + 1$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate differences of the aligned data fields of wrA and wrB are placed into wrD, subject to participation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ, CA
- A WideWord OV condition code bit is set if the operation in its corresponding datapath causes overflow.

wsube*x* - WideWord Subtract Extended

WideWord Unit

wsube*pw* wrD, wrA, wrB (C = 0)

wsubec*pw* wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	100011
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+(size-1)} \leftarrow (wrA)_{i,\,i+(size-1)} + \neg(wrB)_{i,\,i+(size-1)} + CA_{i/8}$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate differences of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. Each data field uses the associated bit of the WideWord Carry register as a carry in for the operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ, CA
- A WideWord OV condition code bit is set if the operation in its corresponding datapath causes overflow.

wsubu - WideWord Subtract Unsigned

WideWord Unit

wsubu*pw*   wrD, wrA, wrB

000010	wrD	wrA	wrB	1	PPWW	100100
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i,\,i+(size-1)} \leftarrow (wrA)_{i,\,i+(size-1)} + \neg(wrB)_{i,\,i+(size-1)} + 1$
```

The WW field determines if the 256-bit contents of wrA and wrB are treated as 32 bytes, 16 half-words, or 8 words. The aggregate differences of the aligned data fields of wrA and wrB are placed into wrD, subject to participation. This instruction is identical to **wsub** except that the OV condition codes are updated to reflect unsigned arithmetic.

Other registers altered:

- WideWord condition code registers: LT, GT, EQ, CA
- A WideWord OV condition code bit is set if the operation in its corresponding datapath causes overflow.

wupkh*x* - WideWord Unpack High

WideWord Unit

wupkhs*w* wrD, wrA (C = 0)

wupkhu*w* wrD, wrA (C = 1)

000010	wrD	wrA	00000	C	00WW	001101
0 5	6 10	11 15	16 20	21	22 26	27 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16

```
for i = 0 to (256 - (2 × size) ) by (2 × size)
    if C=1
        $wrD_{i,\, i+(2\times \text{size})-1} \leftarrow 0^{\text{size}} \,\|\, (wrA)_{i/2,\,(i/2)+\text{size}-1}$
    else
        $wrD_{i,\, i+(2\times \text{size})-1} \leftarrow ((wrA)_{i/2})^{\text{size}} \,\|\, (wrA)_{i/2,\,(i/2)+\text{size}-1}$
```

The most significant 128 bits of the contents of wrA are unpacked, or type promoted. For example, if WW=00 the 128-bit source vector is treated as 16 bytes, where each byte is promoted to a 16-bit half-word to form a 256-bit result that is placed into wrD. The C bit indicates whether sign extension or zero fill is used in the unpacking. Note that participation is not supported for this instruction.

Other registers altered:

- None

wupkl*x* - WideWord Unpack Low

WideWord Unit

wupklsw **wrD, wrA** **(C = 0)**

wupkluw **wrD, wrA** **(C = 1)**

000010	wrD	wrA	00000	C	00WW	001100
0 5	6 10	11 15	16 20	21	22 26	27 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16

```
for i = 0 to (256 - (2 × size) ) by (2 × size)
    if C=1
        $wrD_{i, i+(2\times \text{size})-1} \leftarrow 0^{\text{size}} \| (wrA)_{128+(i/2), 128+(i/2)+\text{size}-1}$
    else
        $wrD_{i, i+(2\times \text{size})-1} \leftarrow ((wrA)_{128+(i/2)})^{\text{size}} \| (wrA)_{128+(i/2), 128+(i/2)+\text{size}-1}$
```

The least significant 128 bits of the contents of wrA are unpacked, or type promoted. For example, if WW=00 the 128-bit source vector is treated as 16 bytes, where each byte is promoted to a 16-bit half-word to form a 256-bit result that is placed into wrD. The C bit indicates whether sign extension or zero fill is used in the unpacking. Note that participation is not supported for this instruction.

Other registers altered:

- None

wxor*x* - WideWord Exclusive-OR

WideWord Unit

wxor*pw*   wrD, wrA, wrB (C = 0)

wxorc*pw*   wrD, wrA, wrB (C = 1)

000010	wrD	wrA	wrB	C	PPWW	101010
0 5	6 10	11 15	16 20	21	22 25	26 31

Variable values in the following equations are as follows:

WW Value	size
00	8
01	16
10	32

```
for i = 0 to (256 - size) by size
    if PP bits and conditions are set accordingly
        $wrD_{i, i+(size-1)} \leftarrow (wrA)_{i, i+(size-1)} \oplus (wrB)_{i, i+(size-1)}$
```

The 256-bit contents of wrA are exclusive-ORed with the 256-bit contents of wrB, and the result is placed into wrD, subject to participation. The WW field simply effects how participation applies and how condition codes are updated for this operation.

Other registers altered:

- If C =1, WideWord condition code registers: LT, GT, EQ

xor*x* - Exclusive OR

Scalar Unit

xor rD, rA, rB (C = 0)

xorc rD, rA, rB (C = 1)

000011	rD	rA	rB	C		101010
0 5	6 10	11 15	16 20	21	22 25	26 31

$$rD \leftarrow (rA) \oplus (rB)$$

The contents of rA are exclusive-ORed with rB, and the result is placed into rD.

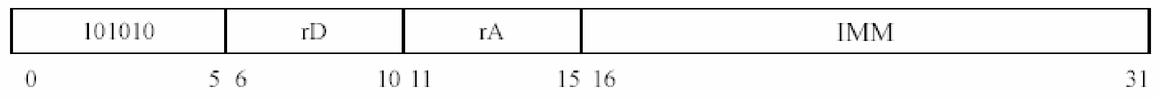
Other registers altered:

- If C =1, scalar condition code registers: LT, GT, EQ

xori - Exclusive OR Immediate

Scalar Unit

xori rD, rA, IMM



$$rD \leftarrow (rA) \oplus (0^{16} \parallel IMM)$$

The contents of rA are exclusive-ORed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- None

xoric - Exclusive OR Immediate Recording Condition Codes

Scalar Unit

xoric rD, rA, IMM

101011	rD	rA	IMM
0 – 5	6 – 10	11 – 15	16 – 31

$$rD \leftarrow (rA) \oplus (0^{16} \parallel IMM)$$

The contents of rA are exclusive-ORed with IMM (prepended with zeros to form a 32-bit value), and the result is placed into rD.

Other registers altered:

- Scalar condition code registers: LT, GT, EQ

Appendix B: DIVA PIM Node Architecture

DIVA PIM
Node Architecture

Jacqueline Chame, Jeff Draper, Mary Hall, Jeff LaCoss, Craig Steele

University of Southern California
Information Sciences Institute

DIVA
Data Intensive Architecture

Chapter 1 - Introduction and Rationale

Motivation

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 50-60% per year while memory access times improve at merely 5-7%. Recent VLSI technology trends offer a promising solution to bridging the processor-memory gap: *embedded-DRAM technology* integrates logic with high density memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (with hundreds of gigabit/second aggregate bandwidth available on a chip--up to 2 orders of magnitude over conventional DRAM) Latency to on-chip logic is also reduced, down to as little as one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data IntensiVe Architecture (DIVA) Project is an exploration of the potential benefits of making direct use of the high data bandwidth and low access latency available on memory devices. DIVA leverages embedded-DRAM technology to replace or augment the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. System bandwidth limitations are thus overcome in three ways: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processors and memory banks per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus.

The DIVA system architecture is focused on achieving the following four goals: (1) developing PIMs that can serve as the only memory in the system, assuming the dual roles of "smart memories" and conventional memory; (2) supporting a wide range of familiar programming paradigms, closely related to parallel computing; (3) targeting applications that are severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets; and, (4) developing a VLSI device to exploit memory and communications bandwidth in PIM-based systems while making efficient use of on-chip resources for target applications.

In DIVA, the PIM chips serve as the memory to a conventional host processor. A DIVA system is comprised of multiple interconnected PIM chips (on the order of 32 to 64). On each of these PIM VLSI devices, there may be multiple processors and memory banks. Each PIM processor has a specific memory bank associated with it. We refer to a single processor and its associated memory bank as a *node*.

This document describes the architecture of a single node in the DIVA system, presenting its key components in detail. This chapter provides a framework for understanding the role of a DIVA node by first describing the overall system architecture, as well as the architecture of the PIM VLSI device, followed by key features of the DIVA system architecture. Subsequently, it describes individual components of the node architecture, to be covered in much more detail in later chapters.

DIVA System Architecture

A driving principle of the DIVA system architecture is efficient use of PIM technology while requiring a smooth migration path for software. This principle demands integration of PIM features into conventional systems as seamlessly as possible. As a result, DIVA chips are designed to resemble commercial DRAMs, enabling PIM memory to be accessed by host software as if it were conventional memory. In Figure 1, we show a small set of PIMs connected to a single host processor through conventional memory control logic. Because of on-chip memory accesses, this memory controller can not be a commercially available device; while standard DRAMs are "slave" memories managed by the host, active PIMs may have to signal a "not ready" condition while access to the memory array is arbitrated.

Parcels which spawn computation, gather results, synchronize activity, or simply access non-local data are transmitted through a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic. This interconnect must have low latency and high bandwidth and be amenable to the dense packing requirement of memory devices. Furthermore, it must be scalable to allow the addition or removal of devices from the system. For system sizes of the scale expected for DIVA (32 to 64 PIM chips), this combination of requirements favors a one-dimensional network. The interconnection network of an earlier embedded scalable system, the Package-Driven Scalable System (PDSS) [Steele97], is used as a model. The interconnect is implemented by PIM Routing Components (PiRCs) - one per PIM chip. The PiRC inter-chip fabric is then a point-to-point bidirectional ring using wormhole routing and the Red Rover routing algorithm [Draper96] to effect deadlock-free, low-latency routing of fixed-sized packets. Future generations of DIVA systems will contain large numbers of PIM chips and will require a more complex network scheme.

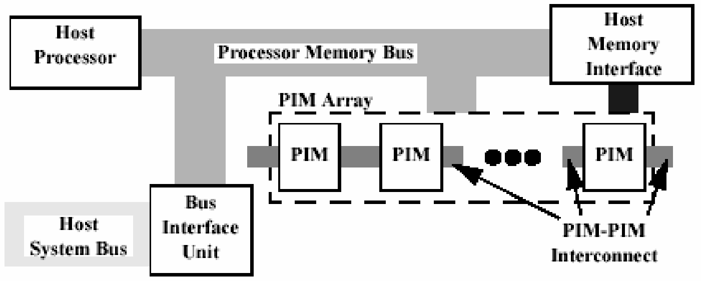


Figure 1: DIVA System Physical Organization

DIVA PIM Chip Architecture

Each DIVA PIM chip is a VLSI memory device augmented with general and special-purpose computing and networking/communication hardware. A PIM may consist of multiple *nodes*, each of which are primarily comprised of a few megabytes of memory and a node processor. Figure 2 shows a PIM with four nodes. The nodes on a chip share a single PiRC and a host interface. The PiRC is responsible for routing parcels on and off chip. The host interface supports conventional memory accesses from the host as well as parcels initiated by the host.

Figure 2 also shows two global interconnects that span the PIM chip for information flow between the nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. Within the host interface, a parcel buffer (PBUF) provides a buffer that is memory-mapped into the host processor's address space, permitting application-level communication through parcels. Each PIM node also has a PBUF, memory-mapped into the node's local address space (see discussion in next section). Although the PiRC also contains parcel ports, we do not label them PBUFs, as they are not memory-mapped.

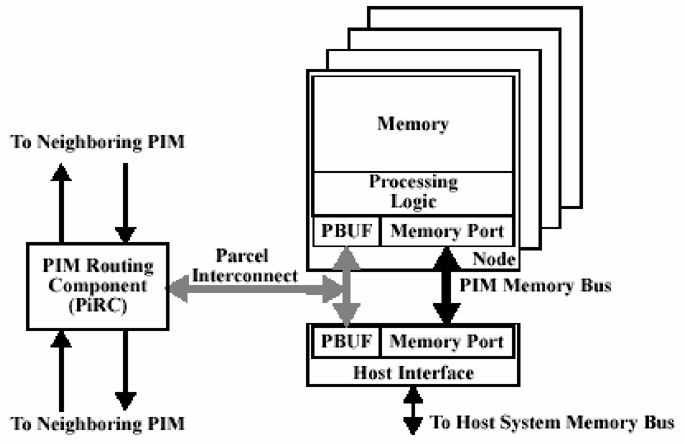


Figure 2: DIVA PIM Chip Organization

Overview of DIVA PIM Node

The DIVA PIM node processor supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two execution units, or datapaths: a *scalar datapath* performs sequential operations on 32-bit registers, and a *wide datapath* performs fine-grain parallel operations on 256-bit register. Both scalar and wide datapaths execute from a single instruction stream under the control of a single 5-stage pipeline. The instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap. Each datapath has its own independent register file, but special instructions permit direct transfers between register files without going through memory.

The combination of the execution control pipeline and scalar datapath may be viewed as a conventional microprocessor and may be programmed as such. This capability is essential to the evolutionary software development approach. Users may, with very little effort, exploit the coarse-grain parallelism offered by the PIM nodes by simply programming multiple nodes in a conventional sense. However, users may also exploit fine-grain parallelism by using the WideWord datapath.

Although not supported in the initial DIVA prototype, floating-point functionality will be provided in future systems as extensions to the WideWord unit to operate on eight 32-bit datapaths. The floating-point support will be mentioned throughout this document, but as it is subject to change, will not be presented in significant detail.

In addition to the execution units, each DIVA PIM node includes three other units. A *memory unit (MU)* is responsible for generating proper control signals to the memory macro. Its functions include initiating refresh cycles as needed and arbitrating between the host memory port and the execution control unit for access to the memory macro; priority of accesses goes to the host. Furthermore, it tracks and maintains an open row in the DRAM macro to enable page-mode accesses as often as possible.A small *instruction cache (IC)* is used to keep instruction accesses to the memory macro from interfering with data accesses as much as possible. Each node contains a memory-mapped location called a parcel buffer (PBUF) that serves as a port between the parcel interconnect and the node, permitting efficient application-level parcel sends and receives.

Key Features of DIVA Node

This section briefly highlights the most important features of the DIVA node architecture:

- High bandwidth and low latency access to node memory

  The wide datapath permits memory accesses of 256 bits with a single load or store operation. Further, the latency on these memory accesses is quite low. Two consecutive accesses to memory within the same 2k-bit row in the memory cell will be in page mode, with a latency on the order of just a few node cycles. If not on the same row in memory, accesses are in random mode, which is perhaps 3 times slower than a page mode access, but is still roughly 3-4 times faster than the latency that would be observed in a conventional system.

- Standard scalar instructions augmented with wide ALU and memory operations

  In addition to the high-bandwidth memory operations described in the previous paragraph, the wide datapath enables superword-level parallelism as available in multimedia extensions such as MMX and AltiVec on wide words of 256 bits. The functionality of the wide datapath is distinguished from other multimedia and subword parallelism ISAs in the following ways, which will be discussed in more detail later: DIVA supports selective execution of instructions on sub-fields with a WideWord, depending on the state of local and neighboring condition codes; it supports direct transfers to/from other register files; and, the wide datapath is tightly coupled with the inter-chip communication.

- Integrated scalar and wide datapaths, using a single control pipeline

  The scalar and wide datapaths share a single control pipeline, avoiding complications in keeping two separate pipelines synchronized. Direct transfers to/from the register files associated with each datapath facilitate efficient switching between scalar and fine-grain parallel portions of the computation. The two instruction sets share most of the same opcodes, and to further unify the instruction sets, use the same condition codes.

Block Diagram and Description of Node Components

Figure 3 shows the major control and data connections within a node. Information flows into and out of the node via the pbuf or the memory port. As shown in the figure, arbitration between external memory accesses by the host and node memory accesses is required. This arbitration adds an insignificant delay to the host memory access time when the PIM processor is not accessing memory; there is little difference in performance when the PIMs are simply used as conventional memory. If the PIM processor is accessing memory, the host memory access time includes the additional latency of waiting for the PIM memory cycle to complete.

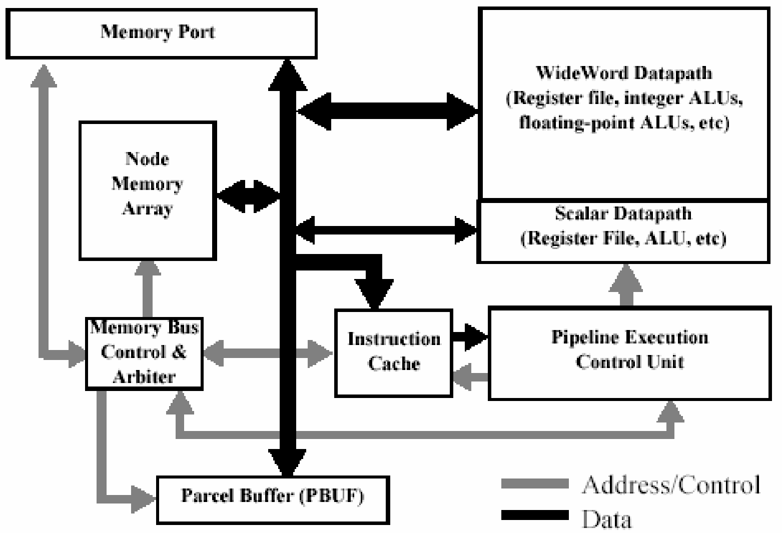

Memory Port
WideWord Datapath (Register file, integer ALUs, floating-point ALUs, etc)
Node Memory Array
Scalar Datapath (Register File, ALU, etc)
Memory Bus Control & Arbiter
Instruction Cache
Pipeline Execution Control Unit
Parcel Buffer (PBUF)
Address/Control
Data

Figure 3: DIVA PIM Node Architecture

Scalar Datapath and Execution Pipeline

The execution pipeline is shared between the scalar and wide datapaths. It is a standard 5-stage pipeline, with the following stages: (1) instruction fetch; (2) register decode; (3) execute; (4) memory; and, (5) write. There are three classes of pipeline hazards, which result in idle cycles: (1) long instruction sequences, such as for multiplies and divides; (2) register operations, involving data dependences between nearby instructions; and, (3) memory operations, which stall the pipeline due to multiple cycles latency to memory. The second class of hazards are sometimes avoided with pipeline forwarding. Other hazards can only be avoided through careful ordering of instructions by the compiler.

The scalar datapath is for the most part a standard RISC architecture, augmented with a few DIVA-specific functions for coordinating with the wide datapath. The wide datapath accesses the scalar registers for addressing operations, as well as for controlling subfield operations.

Wide Datapath

The WideWord datapath processes objects aggregated within a row of the local memory array by operating on 256 bits in a single processor cycle. This fine-grain parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. The WideWord unit can perform bit-level operations, such as simple pattern matching, or higher-order computations such as searches and reduction operations.

The WideWord datapath has several features to distinguish it from a conventional SIMD architecture. First is the ability to ***change ALU operand width*** on a per-instruction basis, enabling it to treat a WideWord as a packed array of objects of eight, sixteen, or thirty-two bits in size. This characteristic means the WideWord ALU is more accurately represented as parallel ALUs, where the number of ALUs depends on the operand size. Second, a ***permutation network*** enables applications to rapidly align and reorganize wide register operands. Third, it supports ***selective execution*** of instructions on sub-fields within a WideWord, depending on the state of local and neighboring condition codes.

Although similar designs support some type of conditional operation, the DIVA WideWord Unit provides a much richer functionality through the ability to specify selective execution in almost every wide instruction and the use of global condition code information in selection decisions. Fourth, even for applications where the WideWord ALU operations are not applicable, the wide datapath can be used to ***accelerate memory access time and communication***.

Memory Unit

The memory unit consists of the DRAM macro as well as the memory controller to arbitrate between various kinds of access requests to the memory. The memory macro includes the features typical of a standard DRAM; in addition, it supports a full address bus, rather than a row-column multiplexed one. The memory controller arbitrates memory requests, which come from several sources: memory refresh, the host interface, the memory stage of the node pipeline, and the node instruction cache. These sources are listed in the order in which they are granted priority.

Instruction Cache

A small instruction cache is included to avoid instruction accesses interfering with data requests, both to reduce the frequency of requests to memory and to maximize the opportunity for faster page mode accesses for the data requests. The instruction cache is direct mapped, and the size for the initial implementation is 4Kbytes with 32byte cache lines. Because it caches just instructions, which are not expected to be modified during program execution, there is no write back facility or other mechanisms for keeping cache lines coherent with memory. To support context switching, an invalidate instruction permits invalidation of individual cache lines.

Parcel Buffer

The basic mechanism used in the DIVA system to support parcel sending/receiving from/to an application is a parcel buffer (or *pbuf*). The pbuf has a virtual as well as a physical abstraction. To the application, the pbuf locations appear as regular memory locations that are manipulated through simple loads and stores. At a physical level, the pbuf is a set of memory-mapped registers. Each PIM node contains a pbuf that serves as a port between the on-chip parcel interconnect and the node (refer to Figure 2). Although the parcel buffer could be implemented as registers within the PIM node processor, a memory-mapped mechanism for the parcel buffer allows a uniform implementation for the node's pbuf as well as a host pbuf. Hence, a pbuf within the PIM chip host interface is memory-mapped into the host processor's address space to allow the host processor to communicate with PIM nodes via the parcel mechanism.

Other Node Features

Address Translation

DIVA partitions the virtual address space of the host processor into three classifications: dumb, which represents standard pages visible only to the host; global, which is shared by host and PIM; and local, which the PIM node uses for internal computation and is visible to the host only in supervisor mode. The local memory is further partitioned into segments; global memory for a particular PIM is also represented by one or more segments. To condense translation information, we use *segments*, each of which is defined by segment registers containing a physical base address and limit. The local memory region is partitioned into eight segments at fixed virtual bases, for kernel code, stack and data, user code and data/stack, and for kernel and user communication buffers. A small number of global segment registers are also used; since global segments must be able to map portions of a shared virtual address space much larger than the physical memory of an individual node, global segments must be represented by both a virtual and physical base address register.

Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation. Therefore, a node must maintain translation information for only eight local segments plus a small number of segments for its portion of the global memory, as well as for any remote data for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

Exceptions

Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external interrupt signal, are handled by a common mechanism. The exception handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, to maintain a flexible implementation platform. It provides an integrated mechanism for handling hardware and software exception sources. Additionally, it provides a flexible priority assignment scheme which minimizes the amount of time that exception recognition is disabled. While the hardware design supports traditional stack-based exception handlers, we also outline a non-recursive dispatching scheme which uses DIVA hardware features to allow preemption of lower-priority exception handlers using a mechanism which should be easier to debug.

Chapter 2 - Registers and Data Types

Introduction

This chapter describes DIVA's different registers and their usages, and how data is represented in these registers. The scalar and wide datapaths each have their own register file. Whether an instruction uses the scalar or wide datapath, arithmetic operations follow a 3-register format, with two sources and one destination. Transfers between register files is accomplished with explicit move instructions. Data is transferred between memory and registers with explicit load and store instructions only. Memory operations involving scalar and wide registers refer to memory locations aligned at 32-bit and 256-bit boundaries, respectively.

The general-purpose registers can be accessed in either user mode or supervisor mode. Some special-purpose registers can be accessed in user mode, but all remaining special-purpose registers may be accessed only in supervisor mode. For the most part, the registers in the scalar datapath follow standard RISC systems. The wide datapath, in contrast, has several novel types of registers to facilitate selective execution on specific subfields of the register. The condition codes have been extended on the wide datapath to maintain a result for each separate data field, and branch instructions have been added to the ISA to simultaneously check the conditions on all data fields. Another novel feature of the wide datapath is the ability to select an individual subfield of the wide register, using either an immediate or a scalar general-purpose register, and move the selected field in an explicit move instruction.

Beyond the standard supervisor-level registers required for interrupts, exceptions and protection, a few special-purpose registers in the system support DIVA-specific activities. Segment registers are used to support address translation. Also, an environment identifier (EID) identifies the currently active user program, for protection purposes, as well as to support inter-node communication.

Some additional registers on a DIVA PIM chip that are not part of a single node are included in the PiRC and host interface, but a discussion of these is beyond the scope of this document.

Description of Node Registers

The registers for a DIVA node are summarized in Table 1and graphically displayed in Figure 5. This section describes each type of register in detail. In the classification below, we first describe the general-purpose registers, both scalar and wide, then the special-purpose registers, distinguishing between supervisor-level registers and user-level registers. Access privileges are described by the mode field of the program status word (PSW) register. This organization is also reflected in Table 1and Figure 5. In Table 1, the "type" field describes the classification of each register. Type *scalar* and *WideWord* refer to the general-purpose registers, *SP* indicates the user-level special-purpose registers, *AT* refers to the address translation registers, and *P* refers to all other privileged registers.

User-Level General-Purpose Registers

This section describes the general-purpose scalar and wide registers that are accessible to user code.

General-Purpose Scalar Registers

There are 32 general-purpose scalar registers, each 32-bits wide, which we designate as R0-R31 in Figure 5. This register file is used as the source or destination for all integer scalar instructions. In addition, scalar registers are used to provide addresses for memory accesses to scalar and wide load/store instructions. Further, scalar general-purpose registers can be used to index subfields in a wide register during transfers between register files using the MVSWI and MVWSI instructions (see below). Memory operations to load and store objects to/from a general-purpose scalar register are aligned at 32-bit boundaries. For convenience in performing arithmetic operations where the immediate 0 is one of the operands, R0 is hardwired to hold the value 0.

General-Purpose Wide Registers

There are 32 general-purpose wide registers, each 256-bits wide, which we designate as WR0-WR31 in Figure 5. This register file is used as the source or destination of all wide instructions. Wide instructions perform the same operation on 8-, 16-, or 32-bit subfields of the wide register, as designated by the width (WW) field of the instruction (Future implementations may also support 64-bit subfields for wide double-precision floating point capability.) The mask register and participation mode register (described below) can optionally be used to designate which subfields will participate in an instruction, if the participation (PP) field of the instruction is set.

Wide registers are loaded from/stored to memory using addresses from the general-purpose scalar registers. Memory operations to load/store objects to/from a general-purpose wide register are aligned at 256-bit boundaries. Individual fields of wide word registers can also be set or read using MVSW, MVWS, MVSWI and MVWSI instructions that use a register or immediate index to specify the data field to be accessed. In addition to arithmetic and transfer operations, wide registers can be updated through the permutation instructions WPRM and WPRMI, which reorganize the data fields of the source register into a destination register. The former instruction uses a third wide register to specify how the data fields will be rearranged, and the latter performs a lookup into a table of hardcoded permutation patterns.

User-Level Special-Purpose Registers

A large number of special-purpose registers are directly or indirectly accessible to the user program, each described in this section.

- A single condition register for scalar condition codes, and a set of five condition registers for wide condition codes
- Scratch registers for scalar integer multiply and divide
- A participation mode register and mask register to support selective execution on the wide ALU

In addition to being read/written indirectly by other ALU operations, the DIVA node architecture permits user-level access to any special-purpose register through explicit moves to standard registers, using the MTSPR and MFSPR instructions.

Scalar Condition Register

The scalar condition code register, CC in Figure 5, consists of 5 bits. The first three bits of CC are set by an algebraic comparison of the result to zero; the other two bits have slightly more peculiar semantics. The condition codes have the CC bit labels and semantics as indicated in the table below. Note that LT, GT, EQ, and CA condition codes are updated only if the current instruction has its condition code enable bit set. The OV condition code is updated for any scalar add or subtract operation, regardless of the condition code enable bit setting, and is sticky;

that is, it is only cleared when the condition code register is read. They are accessed in conditional branch and call statements. Further, like any user-level special-purpose registers, they can be explicitly read and written with the MFSPR and MTSPR instructions, respectively.

Condition Code	CC bit	Description
LT	0	This bit is set when the result is negative.
GT	1	This bit is set when the result is positive and non-zero.
EQ	2	This bit is set when the result is zero.
OV	3	This bit is set to indicate overflow has occurred during execution of an add or subtract instruction. This bit is not altered by any other instructions. In practice, the OV bit is set if the carry out of bit 0 is not equal to the carry out of bit 1 (assuming big Endian bit labeling).
CA	4	In general, the carry bit (CA) is set to indicate that a carry out of bit 0 occurred during execution of an add or subtract instruction. This bit is not altered by any other instructions.

Figure 4: Scalar Condition Code Register

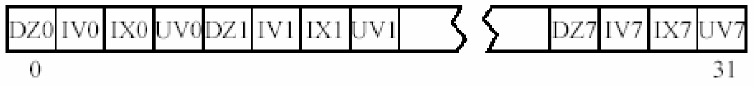
Wide Condition Registers

While the scalar codes are consolidated into a single condition register, the CC described above, each type of WideWord condition code is allocated an entire register so the results of parallel operations on objects as small as bytes may be recorded. Each one of these condition registers is 32-bits wide. Thus, wide condition registers are designated as LT, GT, EQ, OV, and CA. For an example of how the wide condition registers are used, a bit of the WideWord LT register is set if the result of its corresponding 8-bit datapath is negative. However, there are subtleties due to the configurability of the operand sizes. For example, if a WideWord instruction specifies that operands are to be treated as 32-bit values, the condition codes are grouped into eight groups of 4, where each bit of a group is updated with the same value to reflect a condition for the group's corresponding 32-bit result. Like the scalar CC register, the LT, GT, EQ, and CA wide condition registers are only set by instructions that have their C field enabled. The OV register is a sticky register that is updated on all WideWord add and subtract operations; bits of this registered are cleared only when the register is read using an mfspr instruction.

The wide condition codes are accessed by the branch instructions BAx and BNx, which represent Branch-On-All and Branch-On-None conditions for the appropriate wide condition register represented by x.

WideWord Floating-Point Status Register

Similar to condition codes, the WideWord floating-point status register (FPSR - special-purpose register 15) may be updated to reflect exception conditions for floating-point operations. This register is a 32-bit register arranged in group of 4 status conditions for each of the eight 32-bit floating-point units in the WideWord datapath. The 4 status conditions are: divide by zero (DZ), invalid (IV), inexact (IX), and unsupported value (UV). DZ, IV, and IX are typical IEEE-754 floating-point exceptions. Refer to the IEEE-754 standard for details. UV indicates that either overflow or underflow occurred at some point during the program. All bits of FPSR are sticky; once set, they remain set until FPSR is read via an mfspr instruction. The bit arrangement for FPSR is shown below.



FPSR Bit Arrangement

Scratch registers for integer multiplies and divides

Two registers, designated HI and LO in Figure 5, are automatically set as the result of a scalar integer multiply or divide. HI holds the most significant 32 bits of a multiplication result or the remainder of a division. LO has the least significant 32 bits of a multiplication result or the quotient of a division.

Participation Mode Register

The Participation Mode (PM) register is a 5-bit register that describes the conditions for selective execution of a wide instruction that has its PP field set. The conditions correspond to the four condition codes or the mask register M (as will be discussed in Chapter 5). The PM register is read/written using the `MFSPR and MTSPR` instructions. It is also updated automatically to select M for participation when the mask register M is updated.

Mask Register

The mask register is a 32-bit register used in participation, which we refer to as M in Figure 5. If the PP field of a wide instruction is set, and the M bit of the PM register is set, then the instruction is conditionally executed on each data field that has its corresponding bit in the M register set. Like the WideWord condition codes, if the width of each field is larger than 8 bits, multiple bits in the M register will be set corresponding to a single data field (2 for 16-bit widths, 4 for 32-bit widths). Update of the M register automatically causes the M bit of the PM register to be set.

Supervisor-Level Address Translation Registers

A total of 28 32-bit registers related to local and global segments are used to perform translation of virtual addresses to physical addresses by the node processor. A detailed description of how these registers are used in the address translation process can be found in Chapter 10. The registers are set by supervisor-level software using MTPR instructions, usually as a result of a context switch or a change in the size or location of current global segments. They are read either by MFPR instructions, or more commonly, directly by address translation hardware.

A set of 16 registers support local segments, referring to addresses local to the PIM node that are inaccessible to host user code or other PIMs nodes. There are eight local segments, with two registers representing each segment. The Local Segment Base registers (SB0-SB7) hold the physical base address of each local segment. The Local Segment Limit registers (SL0-SL7) hold the maximum offset from the base, for address bounds checking, as well as some additional bits to support access protection.

A set of 12 registers support global segments, referring to addresses that may be shared between host and PIM. There are four global segments, and each is supported by three separate registers. Global segments must be able to map portions of a shared virtual address space much larger than the physical memory of an individual node. For this reason, global segments have both Global Segment Physical Base registers (GPB0-GPB3), similar to local segments, as well as Global Segment Virtual Base Registers (GVB0-GVB3). Usages of the Global Segment Limit registers (GL0-GL3) are analogous to the SL0-SL7 registers for local segments.

Other Supervisor-Level Registers

A number of other supervisor-level registers are included to support the PIM run-time kernel activities. These can be classified into the following categories:

- Scratch registers
- The program counter
- The processor status word
- The environment identifier
- Timer registers, including two to hold current system clock and one used as a countdown timer
- Registers to support interrupts and exceptions, a total of seven

While in some cases these registers are updated as a result of a hardware event or upon execution of some other instruction, all of the registers can be read from/written to general-purpose registers by the supervisor-level instructions MFPR and MTPR. There are two exceptions to this. The Program Counter is set only by hardware, and cannot be accessed directly, even by supervisor-level code; for this reason, it is not given a register class in Table 1. Also, the Exception Source Word (ESW) is set in software only indirectly through the Exception Set Register and the Exception Reset Register, although it can be read by MFPR; MTPR to the ESW is undefined and is treated as a no op by the hardware.

Scratch registers

Four 32-bit scratch registers, designated SCR0-SCR3 in Figure 5, are used by the kernel for its various activities. The goal of having these additional registers is to avoid the need to save and restore context of general-purpose registers when switching between the kernel and user-level code. The kernel can instead copy the contents of up to four of the general-purpose registers into SR0-SR3, then use the general-purpose registers, and subsequently restore the contents of the general-purpose registers, thus avoiding more costly memory accesses.

Program counter

The program counter (PC) maintains the address to the current instruction to be executed. Although user code causes the PC register to be updated, it is updated indirectly through the execution instructions that change the flow of control in the program *(i.e.*, branches, procedure calls and interrupts and exceptions).

Upon execution of a branch instruction, the PC is updated by hardware to the target of the branch. For a CALL instruction, the current PC is copied into SR31, and then the PC is updated to the starting point of the called function. A subsequent RET instruction will cause R31 to be copied back to PC. On an interrupt or exception, the current PC is automatically copied into the FADR register (see description below), and is restored from FADR upon execution of a RFE instruction.

Processor status word

The processor status word is shown as PSW in Table 5. A detailed description of the PSW and its operation is given in Chapter 8.

Environment identifier

A 16-bit EID register records the currently active user context, and it is used to support communication between PIM nodes. A parcel arriving at a PBUF must have an EID in the header that matches the current EID register; otherwise, the parcel must be buffered, awaiting a PIM context switch. The EID register is set by the kernel upon PIM context switch.

Timer registers

Two 32-bit registers, RCL and RCH, hold the low-order and high-order bits, respectively, of the real-time clock. The real-time clock provides a high-resolution measure of real time for indicating the time of day and date. The combination of RCL and RCH may be viewed as a loadable 64-bit counter. At reset, the value of RCH and RCL are all 0s and begin incrementing when reset is released. The real-time clock is clocked by the CPU clock. Considering a probable CPU frequency range of 200MHz to 1GHz for implementations over the life of this architecture, the real-time clock will provide ranges of approximately 117 to 585 years at a 1ns to 5ns resolution, respectively. RCH and RCL values may be initialized to desired values through the use of the MTPR instruction and are read using the MFPR instruction.

The TIMER register is a 32-bit decrementing counter that provides a mechanism for causing an interrupt after a programmable delay. The frequency of the TIMER decrement is the same as the CPU clock frequency. The TIMER causes an exception (subject to masking) when it reaches 0 and begins immediately to count down the next interval without processor intervention. The interval is set by loading the TIMER register with the interval value by initially using an MTPR instruction. Subsequently, the TIMER returns to the interval value the next cycle after counting down to a 0 value.

Registers to support interrupts and exceptions

There are seven 32-bit registers, shown in Figure 5, that are used to support interrupts and exceptions. A detailed description of their usage can be found in Chapter 8.

The Stored PSW register (SSW) holds the value of the PSW immediately prior to the interrupt or exception. The MADR and FADR registers hold the address of the faulting memory address and/or faulting instruction, in the event of an exception. If the cause of the exception was just a normal timer-initiated interrupt, the FADR register will hold the next instruction to be executed. All three of these registers are set either by hardware in the event of a hardware exception, or by MTPR instructions at the beginning of a software exception. The PC and PSW registers are restored with the values of FADR and SSW, respectively, on execution of a RFE instruction.

The four additional registers to support exceptions are the Exception Enable Mask register (EMR), the Exception Source Word (ESW), the Exception Set register (ESR) and the Exception Reset register (ERR). The EMR register indicates which exceptions are currently enabled, and is set by the supervisor. Fields of the ESW are set to 1 either directly by hardware in the event of a hardware exception, or by software setting corresponding bits in the ESR register for software exceptions. Fields of the ESW are cleared to 0 by software setting corresponding bits in the ERR register. A description of the bit fields and their meaning can be found in Chapter 8.

NAME	Type	Number	Width	DESCRIPTION
SR0-SR31	scalar	0 - 31	32	General-purpose scalar registers
WR0-WR31	WideWord	0 - 31	256	General-purpose WideWord registers
CC	SP	0	5	LT, GT, EQ, OV, and CA bits of scalar processor
HI	SP	1	32	most significant 32 bits of multiplication result, remainder of division
LO	SP	2	32	least significant 32 bits of multiplication result, quotient of division
LT	SP	8	32	Less Than condition code register of WideWord Unit
GT	SP	9	32	Greater Than condition code register of WideWord Unit
EQ	SP	10	32	Equal condition code register of WideWord Unit
CA	SP	11	32	Carry condition code register of WideWord Unit
OV	SP	12	32	Overflow condition code register of WideWord Unit
M	SP	13	32	WideWord Mask register used in selective execution
PM	SP	14	5	WideWord Participation Mode register used in selective execution
FPSR	SP	15	32	WideWord floating-point status register
SB0-SB7	AT	0 - 7	32	Base registers for local segments, used for address translation
SL0-SL7	AT	8 - 15	32	Limit registers for local segments, used for address translation
GVB0-GVB3	AT	16 - 19	32	Virtual base registers for global segments, used for address translation
GL0-GL3	AT	20 - 23	32	Limit registers for global segments, used for address translation
GPB0-GPB3	AT	24 - 27	32	Physical base registers for global segments, used for address translation
PSW	P	0	5	Processor status word
SSW	P	1	5	Stored value of PSW, used in exception handling
EID	P	2	16	Environment identifier
FADR	P	3	32	Stored value of PC, used in exception handling
SCR0-SCR3	P	4 - 7	32	Supervisor-level scratch registers
ESW	P	8	32	Exception source word
EMR	P	9	32	Exception mask register
ESR	P	10	32	Exception set register
ERR	P	11	32	Exception reset register
MADR	P	12	32	Faulting memory address, used in exception handling
TIMER	P	13	32	Timer for programmable delay interrupts
RCL	P	14	32	Low-order bits of real-time clock
RCH	P	15	32	High-order bits of real-time clock
PC	NA	NA	32	Program counter

TABLE 1. Summary of registers

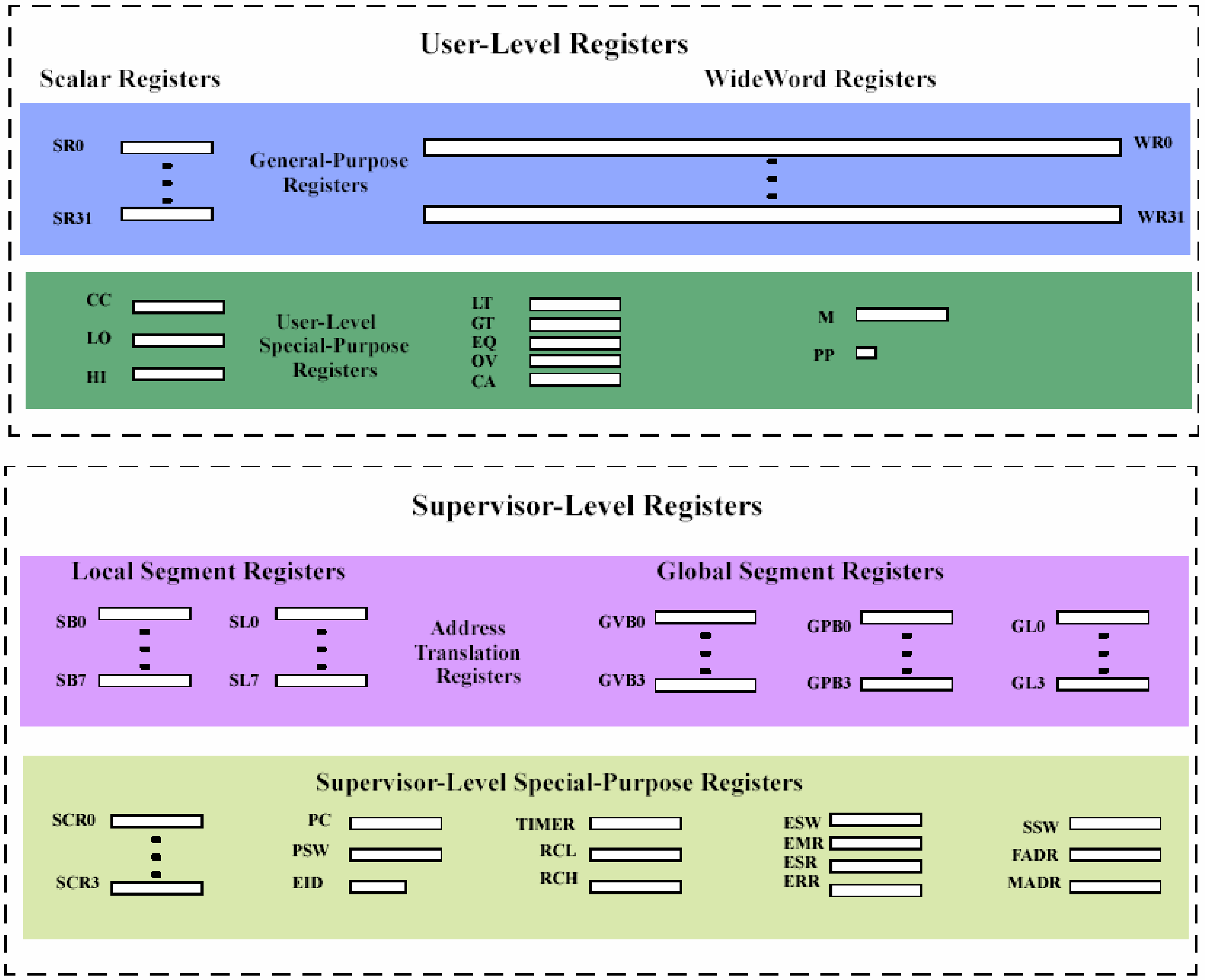


Figure 5: DIVA Node Registers

Operand Conventions

As stated earlier, memory operations are assumed to be aligned at 32-bit boundaries for the scalar datapath, and 256-bit boundaries for the wide datapath. Thus, on memory operations, the appropriate number of least significant bits in the address should be 0 (last 2 for scalar datapath, last 5 for WideWord datapath). Addresses in memory operations that do not conform to these rules will trigger an exception.

Following the convention of the PowerPC host, bits and bytes are stored in BigEndian order in memory.

Chapter 3 - ISA Summary

Scalar Instruction Formats

As shown in Figure 6, the DIVA scalar instruction uses a three-operand format to specify two 32-bit source registers and a 32-bit target register. For arithmetic/logical instructions using this format, there is also a **C** bit to indicate whether the current instruction updates condition codes. However, the **C** bit indicates signed/unsigned arithmetic for multiply/divide instructions, since these instructions never update condition codes by definition. In lieu of a second source register, a 16-bit immediate value may be specified, as shown in Figure 7.

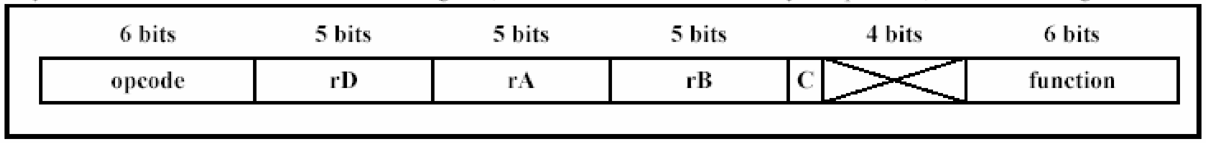


Figure 6: Format R for Scalar Register Operations

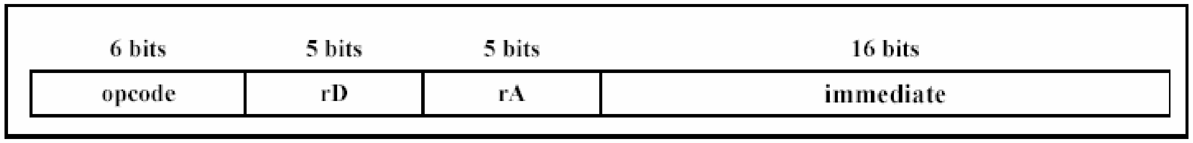


Figure 7: Format I for Scalar Immediate Operations

The branch instruction formats are shown in Figure 8. The branch target address may be PC-relative or calculated using a base register ORed with an offset. In both formats, the offset is in units of words, or 4 bytes, since instructions must be on a 4-byte boundary. Furthermore, the **L** bit specifies linkage, that is, whether a return instruction address should be saved in R31, referred to as a call instruction. Also, the **CCC** field specifies one of eight branch conditions: always, equal, not equal, less than, less than or equal, greater than, greater than or equal, or overflow. See the branch and call instruction descriptions in the DIVA ISA document for details.

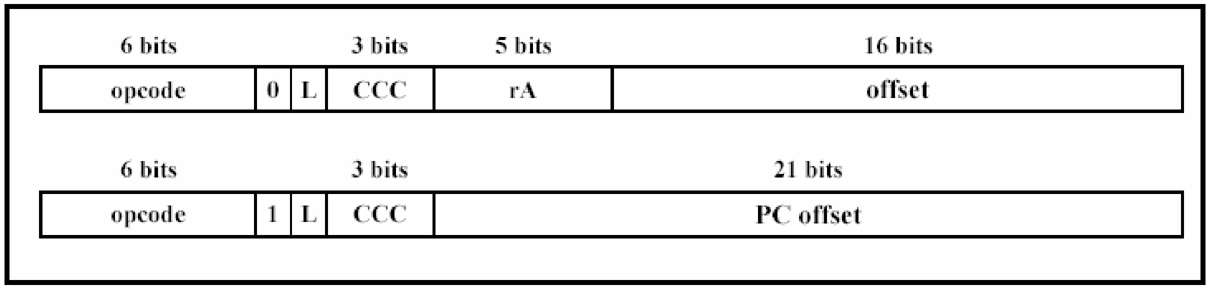


Figure 8: Format B for Branches

WideWord Instruction Formats

As shown in Figure 9, "WideWord Arithmetic/Logical Format," WideWord instructions follow the general form of scalar instructions. Additional control information is included to manage the data fields of the WideWord, and to modify the execution of the instruction. Figure 10 shows the format for transfers within the WideWord register file and across the scalar, FP, and WideWord register files.

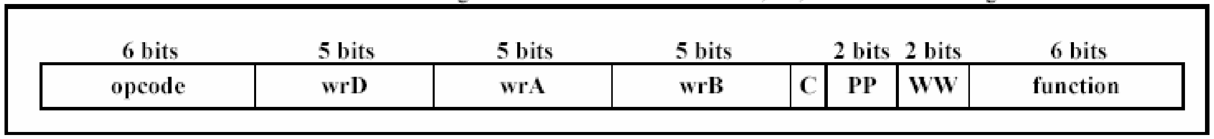


Figure 9: Format W for WideWord Arithmetic/Logical Operations

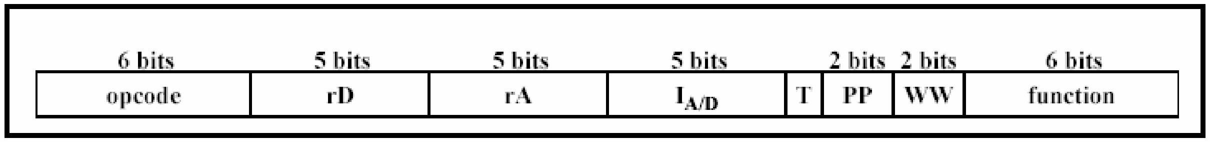


Figure 10: Format T for Wide-Word and Inter-Register File Transfers

The control fields are defined as follows:

WW (width)

The ***WW*** field sets the width of the WideWord operands to eight, sixteen, or thirty-two bits, which primarily affects the shift operations and the configuration of the carry chain for additions and subtractions. For the merge instruction, these bits specify the condition on which the merge is based. The encoding of these bits is listed in the following table:

WW Value	Operand Width	Assembler Mnemonic
00	8 bits	b
01	16 bits	h
10	32 bits	w
11	Reserved	NA

C (condition code enable)

The ***C*** bit indicates whether condition codes will be updated as a result of the current instruction's execution. However, the ***C*** bit indicates signed/unsigned arithmetic for multiply, pack, and unpack instructions.

PP (participation)

The ***PP*** field interacts with condition codes to control whether a computation is performed on a given data field. The participation field can specify that a data field participate always, only if a condition local to its own data field is true, only if the data field is the leftmost field with a condition that is true, or only if the data field is the rightmost field with a condition that is true. The condition that is inspected for participation depends on the value of the ***PM*** (participation mode) register. Refer to Chapter 5 for more details. The encoding of the ***PP*** bits is listed in the following table:

PP Value	Participation Definition	Assembler Mnemonic
00	Always participate	a
01	Specified by local condition	o
10	Leftmost participation	l
11	Rightmost participation	r

T (type)

The ***T*** bit governs whether the current instruction operates on a vector or scalar. Depending on the function, ***rD*** or ***rA*** may specify a WideWord register. In this case, the ***T*** bit specifies whether the current transfer instruction refers to the WideWord register as a whole vector or instead uses $I_{A/D}$ to index a sub-field of the WideWord register.

$I_{A/D}$

Value to be used as an index when a sub-field of a WideWord is involved in a transfer. Depending on the function, this index field may be an immediate or a scalar GPR specifier. Also, $I_{A/D}$ may be coupled with either ***rD*** or ***rA*** depending on the direction of the transfer as specified by the function.

Concise List

A concise list of the instructions in the DIVA Instruction Set Architecture (ISA) is given in Table 2.

TABLE 2. DIVA Instruction Set

FUNC	DESCRIPTION	FUNC	DESCRIPTION	FUNC	DESCRIPTION
SYS	System Call	**MTSPR**	Move to special-purpose reg	**B*x***	Branch on scalar condition
ICLI	Instruction Cache Line Invalidate	**MFSPR**	Move from special-purpose reg	**BA*x***	Branch on all WideWord conditions
RFE	Return from Exception	**MTPR**	Move to protected reg	**BN*x***	Branch on no WideWord condition
		MFPR	Move from protected reg	**CALL*x***	Call on scalar condition
	Scalar Instructions	**MTATR**	Move to address translation reg	**CALLA*x***	Call on all WideWord conditions
ADD	Add	**MFATR**	Move from address translation reg	**CALLN*x***	Call on no WideWord condition
ADDE	Add extended				
ADDI	Add immediate		**WideWord Instructions**		
ADDIC	Add immediate w/ condition codes	**WADD**	Add		
SUB	Subtract	**WADDE**	Add extended		
SUBE	Subtract extended	**WSUB**	Subtract		
SUBU	Subtract unsigned	**WSUBE**	Subtract extended		**Special WideWord Instructions**
MUL	Multiply	**WSUBU**	Subtract unsigned	**WPRM**	Permute
MULU	Multiply unsigned	**WMULES**	Multiply even signed	**WPRMI**	Permute immediate
DIV	Divide	**WMULEU**	Multiply even unsigned	**WMRG**	Merge based on condition codes
DIVU	Divide unsigned	**WMULOS**	Multiply odd signed	**WPKS**	Pack using signed arithmetic
AND	And	**WMULOU**	Multiply odd unsigned	**WPKU**	Pack using unsigned arithmetic
ANDI	And immediate	**WAND**	And	**WUPKH**	Unpack high-order byte/halfword
ANDIC	And immediate w/ condition codes	**WNOT**	Bitwise inversion	**WUPKL**	Unpack low-order byte/halfword
NOT	Bitwise inversion	**WOR**	Or		
OR	Or	**WXOR**	Xor		**Transfer Instructions**
ORI	Or immediate	**WSLL**	Shift left logical	**MVSW**	Move scalar to WW
ORIC	Or immediate w/ condition codes	**WSLLI**	Shift left logical immediate	**MVSWI**	Move scalar to WW, indirect
ORIS	Or immediate shifted	**WSRA**	Shift right arithmetic	**MVWS**	Move WW to scalar
XOR	Xor	**WSRAI**	Shift right arithmetic immediate	**MVWSI**	Move WW to scalar, indirect
XORI	Xor immediate	**WSRL**	Shift right logical	**MVWW**	Move WW to WW
XORIC	Xor immediate w/ condition codes	**WSRLI**	Shift right logical immediate	**MVWWI**	Move WW to WW, indirect
SLL	Shift left logical	**WLD**	Load Reg from Mem		
SLLI	Shift left logical immediate	**WST**	Store Reg to Mem		
SRA	Shift right arithmetic	**WFABS**	Floating-point absolute value		
SRAI	Shift right arithmetic immediate	**WFADD**	Floating-point add		**Miscellaneous Instructions**
SRL	Shift right logical	**WFDIV**	Floating-point divide	**LOKL**	Lock Load
SRLI	Shift right logical immediate	**WFMUL**	Floating-point multiply	**LOKS**	Lock Store
LD	Load Reg from Mem	**WFNEG**	Floating-point negate	**PROBE**	Probe address to determine
ST	Store Reg to Mem	**WFSUB**	Floating-point subtract		locality
		WFTI	Floating-point to integer conversion		
ELO	Encode leftmost one	**WITF**	Integer to floating-point conversion		
CLO	Clear leftmost one				

Chapter 4 - Execution Pipeline and Scalar Datapath

Introduction

The DIVA execution pipeline is modeled as a five-stage architecture, and is used to control the operation of the scalar and WideWord datapaths. Because the combined pipeline and scalar datapath are quite similar to familiar RISC processor architectures, the operation of these units are detailed together to simplify description. A later section will describe the operations of the WideWord datapath. The stages of the pipeline are named here, with an explanation of the major events occurring within that stage of execution.

Pipeline Stages

We establish the convention that each stage views it's local instruction and output to be synchronized at the next clock edge as the *current* instruction. While from an external view, there are *five* instructions "currently" executing, the ALU stage sees an opcode, two operands, and control and stored state as components of the "current" instruction. This view of execution local to each stage is the convention used in all descriptions of the pipeline.

F - instruction fetch

The F stage of the pipeline is where the address of the current instruction is applied to the instruction cache and the instruction is located. At the end of the cycle the output of the instruction cache is latched into the first register stage of the pipeline.

During the F stage, the address for the next instruction is calculated. Note that the calculation applies to sequential addresses as well as branches.

D - register decode

During the R stage, operands for the current instruction are selected from the register file or the most recent value in the pipeline forwarding logic. In the case of an immediate instruction, immediate field of the current instruction is routed to the SRC2 pipeline. The result is latched into the datapath D-stage registers.

X - execute

Depending on the instruction, the X stage selects either the operands from the local register file, or an operand from the WideWord register file, and forwards the result to the ALU, which performs the computation defined by the opcode and value.fields of the current instruction.

M - memory

Register load and store instructions require memory accesses. To maintain consistency with the normal register-write logic, memory operations are begun during the M cycle, and the pipeline is stalled until memory arbitration and the required read operation has been performed. During memory write operations, the pipeline is released as soon as arbitration grants access to the memory.

W - write

During the W stage, the register file is written with the result of the current operation, whether a computation or a memory read. During the W stage, memory write operations are allowed to complete.

Major Signal Paths

Major data and control paths of the DIVA node processors are shown in Figure 11 and Figure 12. Execution pipeline logic is depicted in the shaded area of the figures, while the unshaded area of the figures shows the control pipeline and scalar datapath.

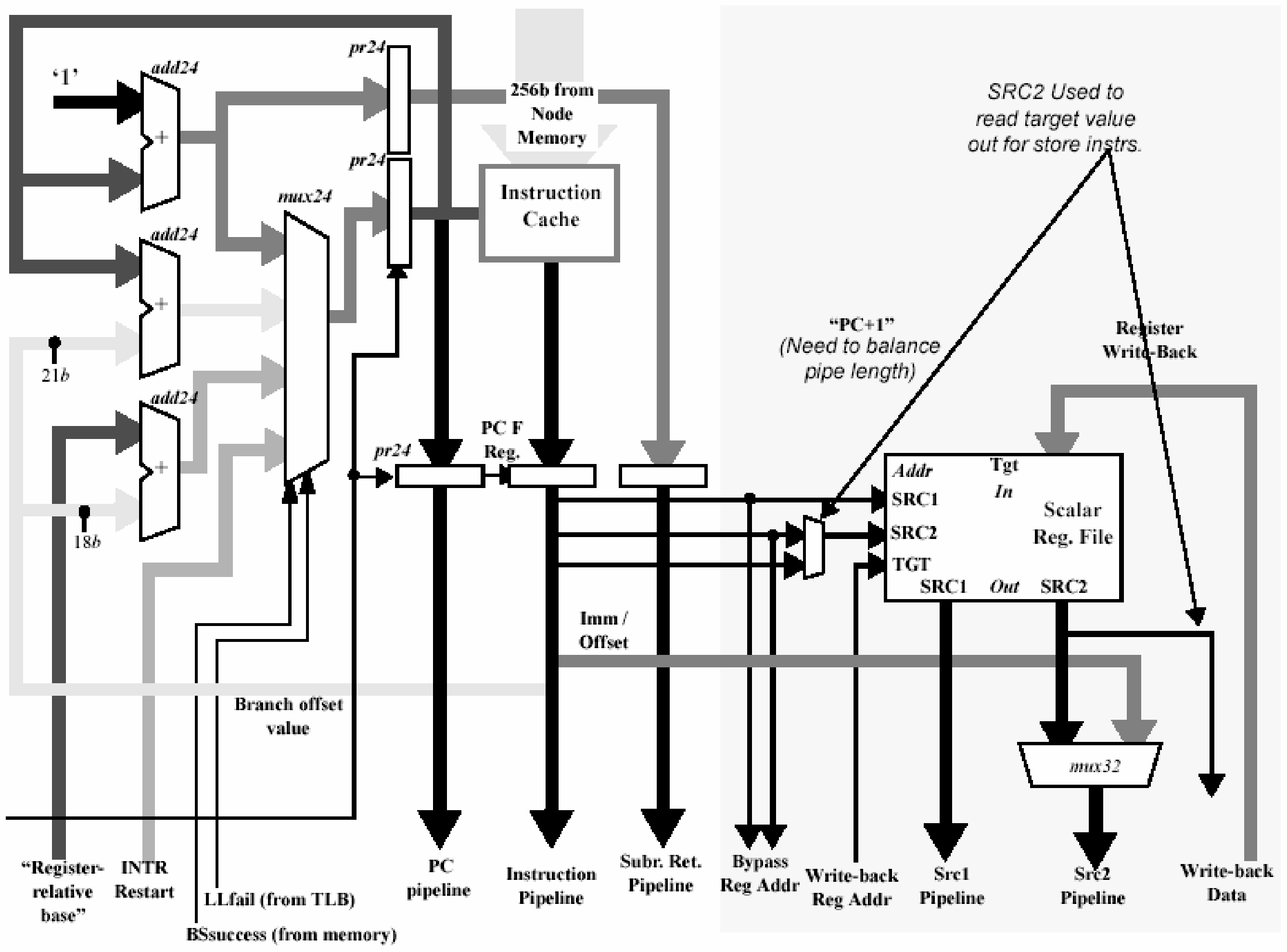


Figure 11: DIVA 5-Stage Execution Pipeline (F & D Stages)

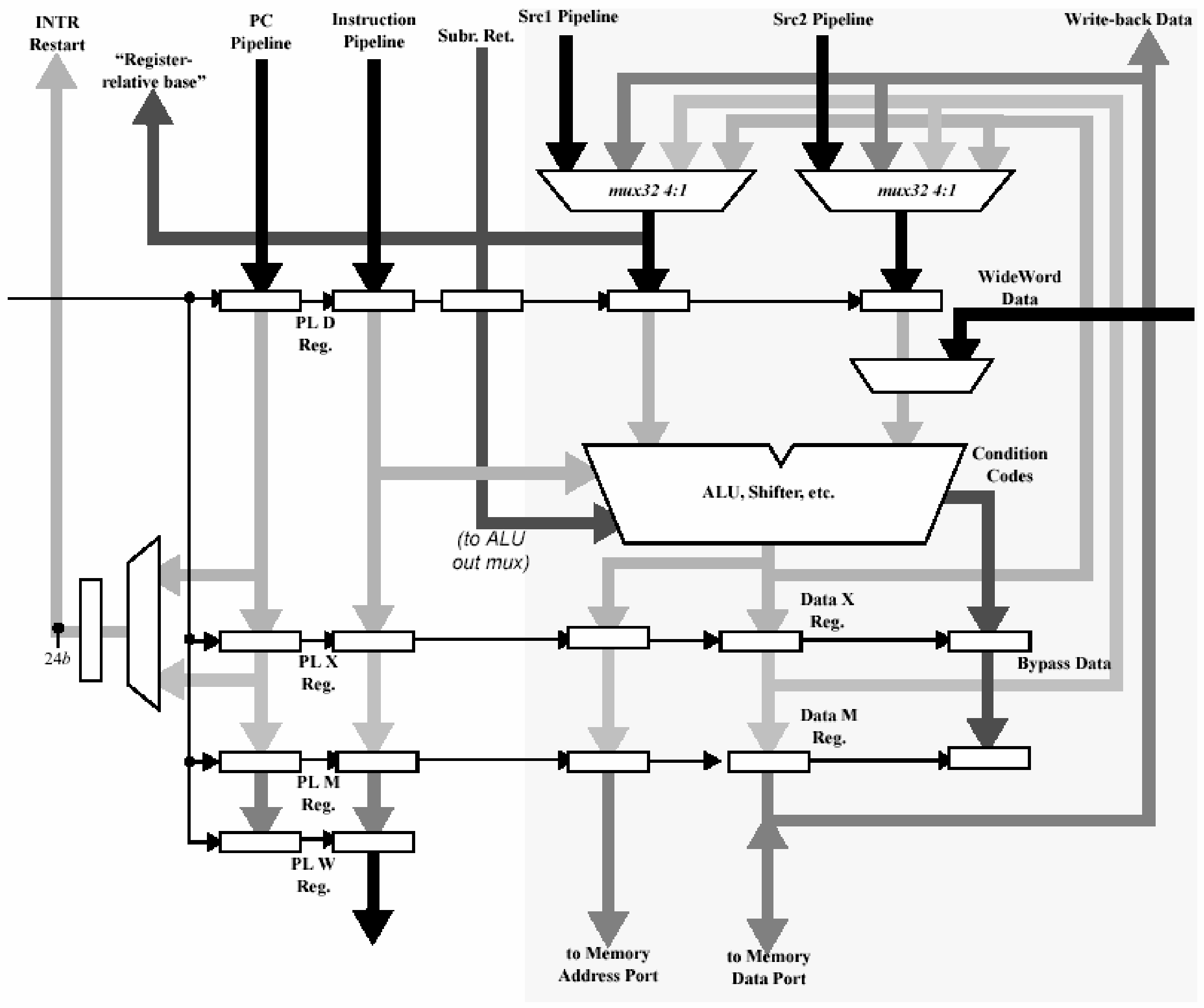


Figure 12: DIVA 5-Stage Execution Pipeline (D through W Stages)

Scalar Computing Functions

The scalar datapath performs operations on objects of 32 bits or less. Refer to the DIVA Instruction Set Architecture document for a complete description of these operations.

DIVA Pipeline Analysis

Numerous examples of a five-stage pipeline exist in the literature, providing a starting design-point for new machines, including DIVA. We perform an analysis of the DIVA pipe to ensure no undue overhead is incurred by branches or other changes in program flow.

Address Calculations

Figure 13 below is excerpted from the earlier execution pipeline illustration, Figure 11. The address calculation portion of the pipeline has been highlighted to clarify the several parallel paths used to develop the address of the next instruction to be executed. Address computations are performed in parallel to guarantee the fastest possible operations. The address calculations indicated in the figure are: *pc_increment*, *pc_offset*, and *register_offset*, which correspond to the types of branches supported by DIVA.

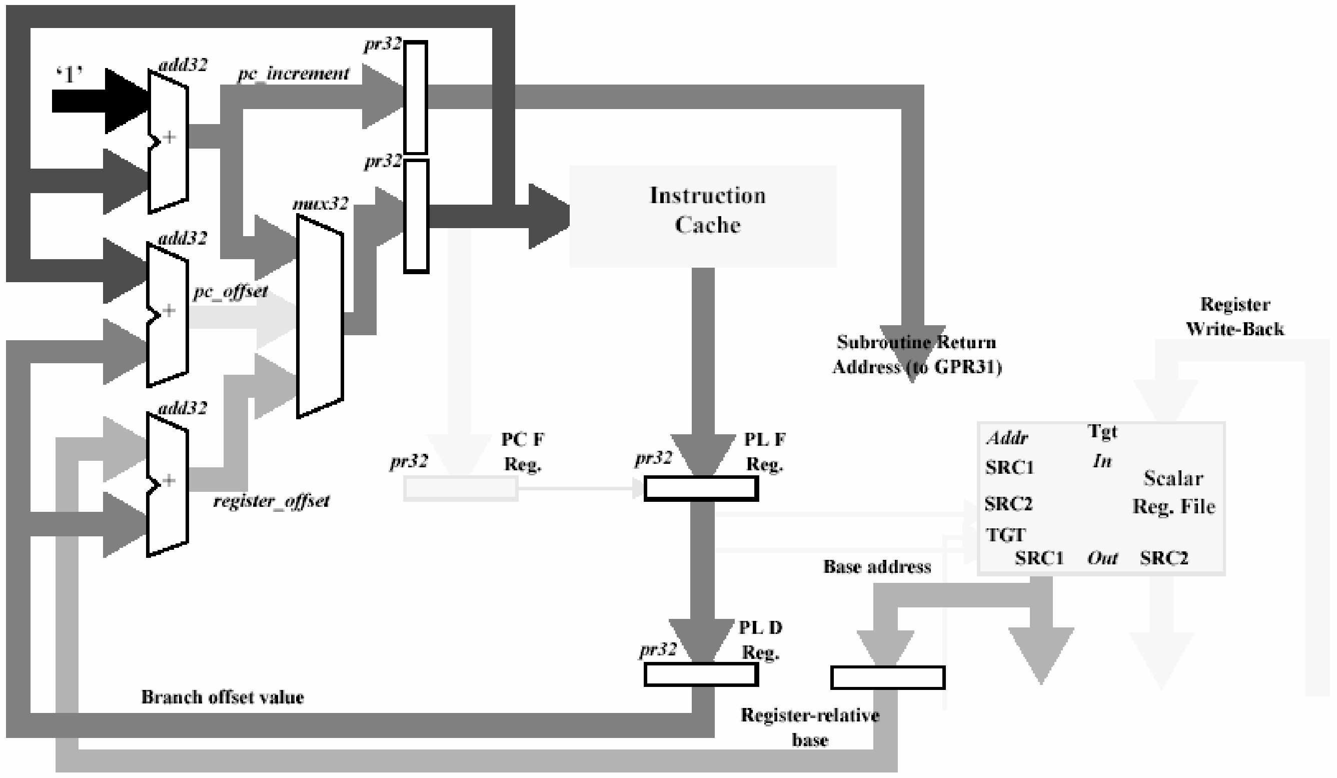


Figure 13: DIVA Instruction-Address Pipeline

Branch Pipeline States

As shown in Figure 13, a branch instruction incurs a two-clock delay, or stall, before the first post-branch instruction can be accessed from the instruction cache and loaded into the execution pipeline. Because the branch instruction doesn't depend on the two ADD instructions, it should be possible for the compiler to move the branch instruction to the point before the first ADD in order to avoid totally wasted dead spots (or "bubbles") in the flow of program execution. In the event a code sequence cannot be rescheduled, either logic is required to keep the pipeline executing correctly, or NOP instructions inserted into the program to ensure proper operation. Obviously, it is simplest to insert the NOP instructions as they require minimal pipeline control logic.

Pipeline Hazards

In pipelined systems, hazards occur when an operation is begun before another has completed, or before required results are available. In DIVA, these are broken down into three classes: *instruction sequences*, *register operations*, and *memory operations*. Each of these hazard classes is described below.

Instruction Sequences

There are several instances of instructions that incur hazards due to "extra" time required for completion. Among these instructions are integer multiply and divide. When these instructions reach the execute (X) stage of the pipeline, the pipeline is stalled for the required number of clock cycles.

Register Operations

Register hazards occur when an instruction requires an operand that is currently in the data pipeline. In the simplest case, consider a stream of instructions where a register is required in the same clock cycle where it is being written into the register file. This hazard can be very simply eliminated by requiring register writes to complete in the first half of each clock cycle, and performing all register reads during the second half. This is well within the capabilities of the technology.

Consider the following code sequence, where an operand is not ready:

```
ADD   R3, R1, R2                  /* R3 = R1 + R2 */
ADD   R5, R3, R4                  /* R5 = R3 + R4 */
```

Because R3 is emerging from the ALU as the first instruction finishes execution, it is not available to be fetched from the register file. This hazard requires *bypassing* or *forwarding* to get the most recent copy of a register from a later stage in the pipeline, and move it to the ALU inputs. Selection is performed by comparing the destination address of every register in the pipeline against the register specifications accessing the register file. The most recent copy (closest to the ALU) is selected, resolving events where several copies of a register are in the pipeline.

Memory Operations

Memory-related hazards can occur in DIVA. These are caused by the proximity of register load and store instructions. Consider the following code sequence, which is typical of moving data for further processing:

```
MOV   R1, R0                      /* initialize the index */
LD    R2, TABL1, R1               /* */
ST    R2, TABL2, R1               /* */
ADD   R1, 0x1
```

Now it is impossible for both the execution pipeline and the memory to respond to these two instructions as written. First, the pipeline can't store a value that has not yet loaded: the register write-back stage is *after* the memory write stage. Second, there is no guarantee that the objects TABL1 and TABL2 are located in the same open row in memory. As a result, an unknown number of delays will occur before the store request will start in the memory.

Chapter 5 - WideWord Datapath

Participation

The WideWord ALU supports ***selective execution*** of instructions on sub-fields within a WideWord. Under selective execution, only the results corresponding to the data paths that participate in the computation are written back, or committed, to the instruction's destination registers. The data fields that participate in the conditional execution of a given instruction are derived from the condition codes or the mask register, plus the instruction's ***participation field***. The conditions used (condition codes or mask register) are specified in the ***participation mode*** register. The instruction's participation field determines how the condition code (or mask register) bits are combined to specify the participation of each data path.

Participation field

Each WideWord instruction with support for conditional execution has a 2-bit participation field. The participation field specifies four ways in which the condition code (or mask register) bits are combined for determining participation of each data path: (1) ***Always participate***, where all data fields participate; (2) ***Local participation***, where a data field participates only if a condition local to its own data path is true; (3) ***Leftmost participation***, where only the leftmost data field with a condition that is true participates; and (4) ***Rightmost participation***, where only the rightmost data field with a condition that is true participates. The encoding of the participation field (***PP***) bits is described in the document "DIVA ISA Overview", and is also listed in the following table:

PP Value	Participation Definition
00	Always participate
01	Local participation
10	Leftmost participation
11	Rightmost participation

Participation Mode

The conditions that are inspected for participation depend on the value of the Participation Mode (***PM)*** register. The PM register is a 5-bit register that is read/written using the `mfspr`/`mtspr` instructions. The conditions correspond to the condition codes EQ, GT, LT, OV or the mask register M. The encoding of the Participation Mode is shown in the following table:

PM Value	Mask/Condition Code
00001	M
00010	EQ
00100	GT
01000	LT
10000	OV

Any combination of the 5 conditions listed in the table can be used to determine participation. For instance, if the PM value is 00110, the EQ and GT condition codes are ORed together to determine participation.

In addition, if the mask register is updated, the participation mode register is automatically updated to select M for participation.

The figure below illustrates an implementation of local participation for data path *i* (note that this simple example is not a complete implementation of a participation bit and does not include the participation field bits):

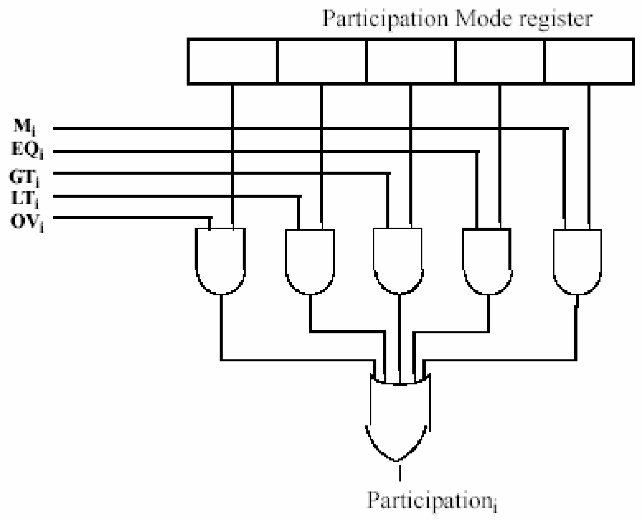


Figure 14: Example of participation bit derived from PM register and condition codes

Setting the condition bits for participation

For simplicity, the WideWord ALU performs conditional write-backs (commits the results) on 8-bit datapaths, independently of the datapath width of the instruction. Conditional operations on 16-bit or 32-bit data paths assume that the condition bits for participation (condition codes or mask register) are set consistently with the current datapath width. For example, an instruction that operates on 32-bit data fields should have a 32-bit result written back to the destination register, for each participating 32-bit data field. Therefore, since the WideWord ALU performs conditional write-backs of 8-bit values, the 4 consecutive bits of the condition code/mask register corresponding to a 32-bit datapath should be set consistently (either all ones, for participation, or all zeros). It is the programmer's responsibility to ensure that the conditions for participation are consistent with the datapath width, either by setting the mask register or by performing a previous operation with the same datapath width to set the condition codes.

Permutation

The WideWord permutation network supports fast alignment and reorganization of data in wide registers. The permutation network supports general permutations of 8-bit data fields, that is, any 8-bit data field of the source register can be moved into any 8-bit data field of the destination register. A permutation is specified by a *permutation vector*, which is a 256-bit object containing 32 indices corresponding to the 32 8-bit data fields of a WideWord. Each 8-bit field of a permutation vector corresponds to the same 8-bit data field of the destination register, and contains the index of the source data field to be moved into that destination field. The figure below illustrates a permutation on 8-bit and 16-bit data paths, and the corresponding permutation vectors.

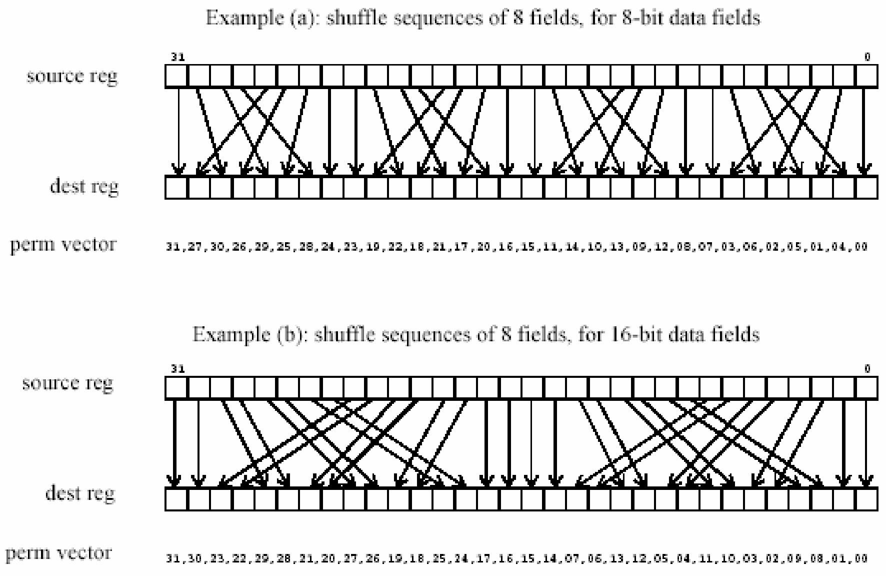
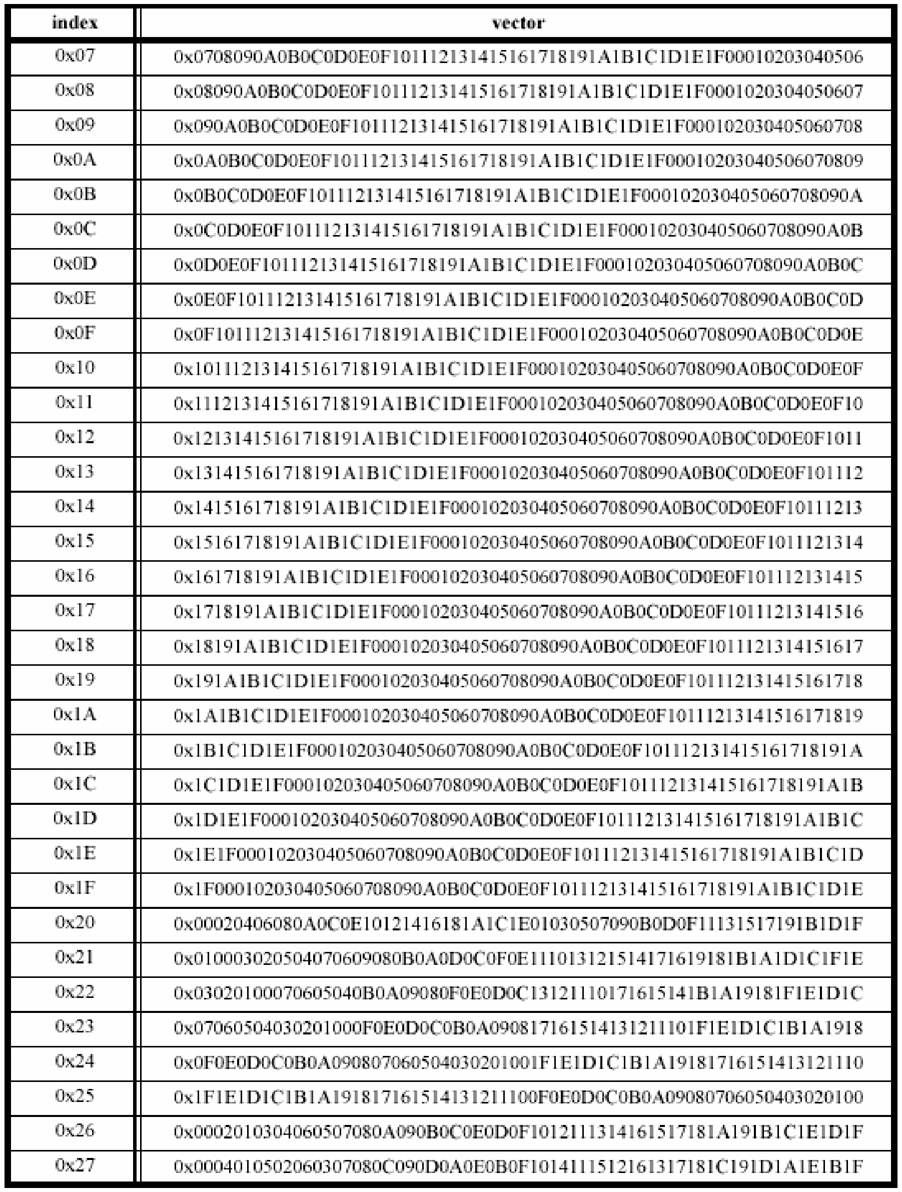


Figure 15: Example of permutation vectors for 8-bit and 16-bit data paths

The WideWord supports two types of permutation operations, `wprm` and `wprmi`. In `wprm` the permutation vector is in a general-purpose wide register, allowing permutation vectors to be loaded from memory and manipulated using WideWord operations. `wprmi` selects a permutation vector from a lookup table, supporting faster permutations (one operation) for the set of frequently used permutation vectors in the table. The hardwired permutation vectors are listed in the following table, and the permute instructions are described in more detail in the document "DIVA ISA Overview".

index	vector
0x00	0x000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
0x01	0x0102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00
0x02	0x02030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001
0x03	0x030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102
0x04	0x0405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203
0x05	0x05060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001020304
0x06	0x060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405



index	vector
0x07	0x0708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203040506
0x08	0x08090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F0001020304050607
0x09	0x090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708
0x0A	0x0A0B0C0D0E0F101112131415161718191A1B1C1D1E1F00010203040506070809
0x0B	0x0B0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A
0x0C	0x0C0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B
0x0D	0x0D0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C
0x0E	0x0E0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D
0x0F	0x0F101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E
0x10	0x101112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F
0x11	0x1112131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10
0x12	0x12131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011
0x13	0x131415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112
0x14	0x1415161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213
0x15	0x15161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011121314
0x16	0x161718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415
0x17	0x1718191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213141516
0x18	0x18191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F1011121314151617
0x19	0x191A1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718
0x1A	0x1A1B1C1D1E1F000102030405060708090A0B0C0D0E0F10111213141516171819
0x1B	0x1B1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A
0x1C	0x1C1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B
0x1D	0x1D1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C
0x1E	0x1E1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D
0x1F	0x1F000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E
0x20	0x00020406080A0C0E10121416181A1C1E01030507090B0D0F11131517191B1D1F
0x21	0x010003020504070609080B0A0D0C0F0E111013121514171619181B1A1D1C1F1E
0x22	0x03020100070605040B0A09080F0E0D0C13121110171615141B1A19181F1E1D1C
0x23	0x07060504030201000F0E0D0C0B0A09081716151413121110 1F1E1D1C1B1A1918
0x24	0x0F0E0D0C0B0A090807060504030201001F1E1D1C1B1A19181716151413121110
0x25	0x1F1E1D1C1B1A191817161514131211100F0E0D0C0B0A09080706050403020100
0x26	0x0002010304060507080A090B0C0E0D0F10121113141615171 81A191B1C1E1D1F
0x27	0x0004010502060307080C090D0A0E0B0F101411151216131718 1C191D1A1E1B1F

index	vector
0x28	0x00080109020A030B040C050D060E070F1018111912 1A131B141C151D161E171F
0x29	0x0001040508090C0D1011141518191C1D020306070A0B0E0F121316171A1B1E1F
0x2A	0x02030001060704050A0B08090E0F0C0D1213101116171415 1A1B18191E1F1C1D
0x2B	0x06070405020300010E0F0C0D0A0B080916171415121310111E1F1C1D1A1B1819
0x2C	0x0E0F0C0D0A0B080906070405020300011E1F1C1D1A1B18191617141512131011
0x2D	0x1E1F1C1D1A1B181916171415121310110E0F0C0D0A0B08090607040502030001
0x2E	0x000104050203060708090C0D0A0B0E0F101114151213161718191C1D1A1B1E1F
0x2F	0x000108090203 0A0B04050C0D06070E0F101118191213 1A1B14151C1D16171E1F
0x30	0x0001020308090A0B1011121318191A1B040506070C0D0E0F141516171C1D1E1F
0x31	0x04050607000102030C0D0E0F08090A0B14151617101112131C1D1E1F18191A1B
0x32	0x0C0D0E0F08090A0B04050607000102031C1D1E1F18191A1B1415161710111213
0x33	0x1C1D1E1F18191A1B14151617101112130C0D0E0F08090A0B0405060700010203
0x34	0x0001020308090A0B040506070C0D0E0F1011121318191A1B141516171C1D1E1F

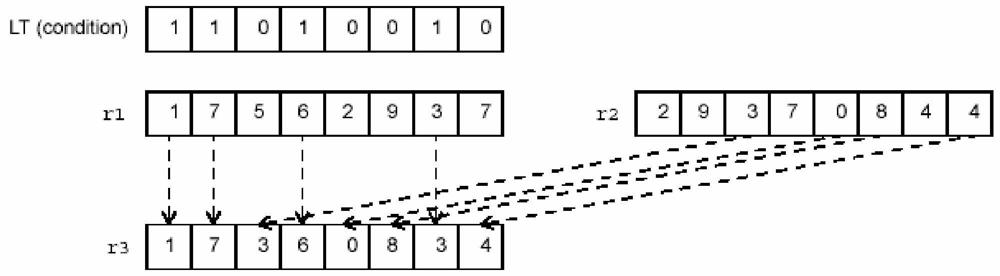
Merge

The WideWord unit supports a special instruction (`wmrg`) for merging data from two source registers according to a given condition. The condition is specified by the WW field of the instruction, and can be one of the condition codes EQ, LT or GT, or the M register. The following table shows the encoding of the WW field.

WW Value	CC
00	EQ
01	LT
10	GT
11	M

The figure below illustrates a merge operation using the condition LT. The condition codes are set by a previous `wsubc` instruction with the same data path width as the `wmrg` instruction.

```
wsubcw r4, r1, r2
wmrgltw r3, r1, r2
```

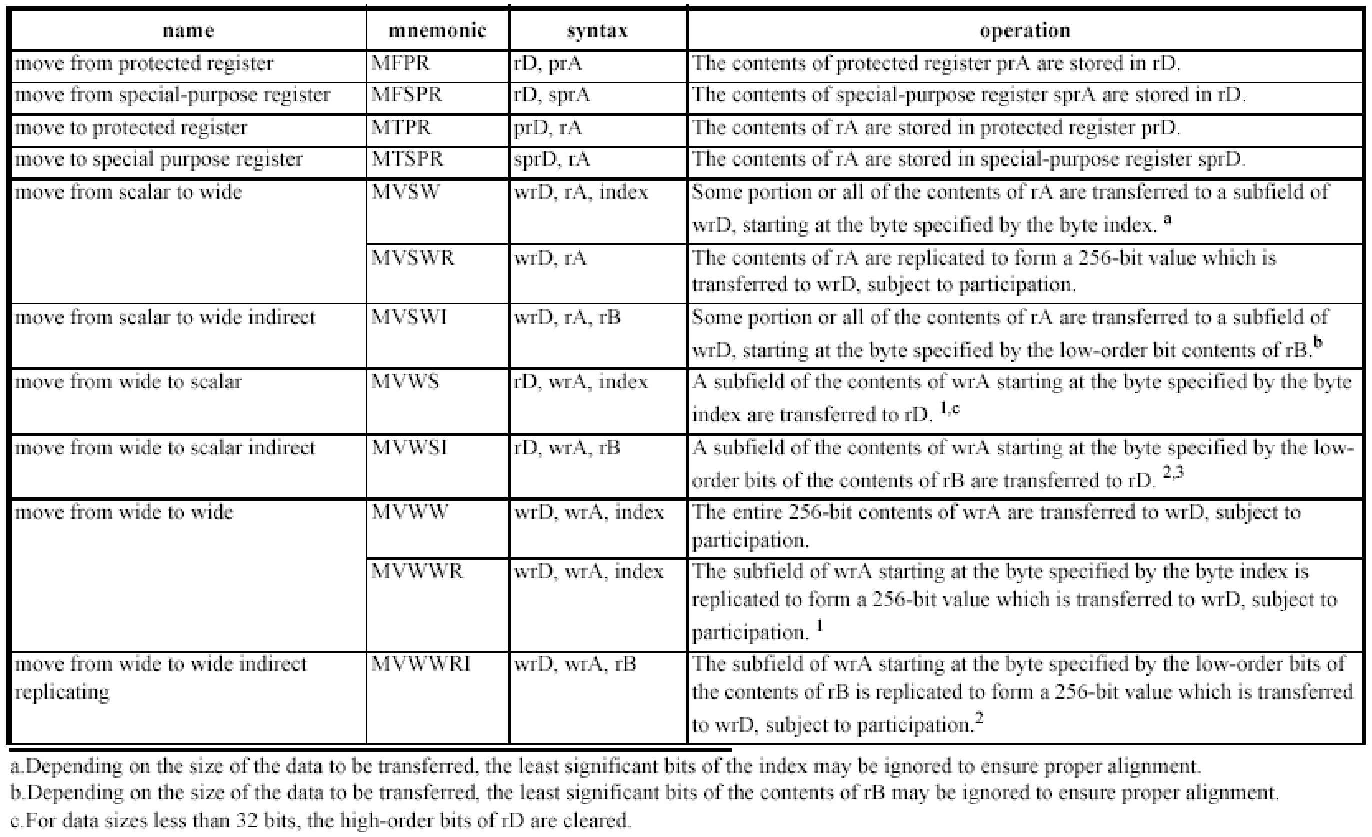


Pack/Unpack

Transfers

A set of ***transfer instructions*** allows data to be moved between the several register files: (1) between wide registers and general-purpose scalar registers; (2) from wide register to wide register; and (3) between general-purpose integer registers and special-purpose or protected registers. The transfer functions where the source is a scalar value (scalar register or a data field in a wide register), and the destination is a wide register allow the source data to be replicated and stored into all the fields of the destination.

The complete set of transfer instructions is listed in the table below, and each instruction is described in detail in the document "DIVA ISA Overview".



name	mnemonic	syntax	operation
move from protected register	MFPR	rD, prA	The contents of protected register prA are stored in rD.
move from special-purpose register	MFSPR	rD, sprA	The contents of special-purpose register sprA are stored in rD.
move to protected register	MTPR	prD, rA	The contents of rA are stored in protected register prD.
move to special purpose register	MTSPR	sprD, rA	The contents of rA are stored in special-purpose register sprD.
move from scalar to wide	MVSW	wrD, rA, index	Some portion or all of the contents of rA are transferred to a subfield of wrD, starting at the byte specified by the byte index. [a]
	MVSWR	wrD, rA	The contents of rA are replicated to form a 256-bit value which is transferred to wrD, subject to participation.
move from scalar to wide indirect	MVSWI	wrD, rA, rB	Some portion or all of the contents of rA are transferred to a subfield of wrD, starting at the byte specified by the low-order bit contents of rB.[b]
move from wide to scalar	MVWS	rD, wrA, index	A subfield of the contents of wrA starting at the byte specified by the byte index are transferred to rD. [1,c]
move from wide to scalar indirect	MVWSI	rD, wrA, rB	A subfield of the contents of wrA starting at the byte specified by the low-order bits of the contents of rB are transferred to rD. [2,3]
move from wide to wide	MVWW	wrD, wrA, index	The entire 256-bit contents of wrA are transferred to wrD, subject to participation.
	MVWWR	wrD, wrA, index	The subfield of wrA starting at the byte specified by the byte index is replicated to form a 256-bit value which is transferred to wrD, subject to participation. [1]
move from wide to wide indirect replicating	MVWWRI	wrD, wrA, rB	The subfield of wrA starting at the byte specified by the low-order bits of the contents of rB is replicated to form a 256-bit value which is transferred to wrD, subject to participation.[2]

a.Depending on the size of the data to be transferred, the least significant bits of the index may be ignored to ensure proper alignment.
b.Depending on the size of the data to be transferred, the least significant bits of the contents of rB may be ignored to ensure proper alignment.
c.For data sizes less than 32 bits, the high-order bits of rD are cleared.

Chapter 6 - Memory Unit

Introduction

This chapter presents the basic functionality of the assumed DRAM memory macro as well as the essentials of a memory controller needed with each macro on a DIVA PIM chip. This controller serves to arbitrate among the various requests for access to a DRAM macro and take advantage of page-mode accesses wherever possible.

Memory Macro Description

A DRAM array similar to the DRAM macro provided by the IBM SA27-E process is assumed. This macro exhibits features of typical DRAM: page-mode accesses, refresh, etc. Unlike conventional DRAM, however, it supports a full address bus, rather than a row-column multiplexed one, and a very wide 256-bit data bus. Specifically, the input signals to the macro are: macro select (similar to RAS in conventional DRAM), page-mode select (similar to CAS in conventional DRAM), write enable, refresh enable, an address bus where 3 bits of the bus are treated as a column address, a 256-bit input data bus, and a 256-bit write enable bus. The only output signals are a 256-bit output data bus. There are also some test input/outputs, but these are not crucial to the DIVA architecture design. The macro page size is 2048 bits; each page contains 8 distinct addressable 256-bit units of data. For an example of the timing benefits of page-mode accesses, the page-mode cycle time in the SA27-E technology is 6.6ns while the random mode cycle time is 20ns.

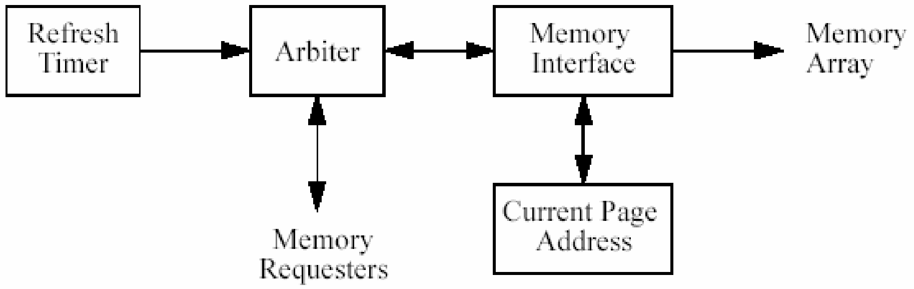


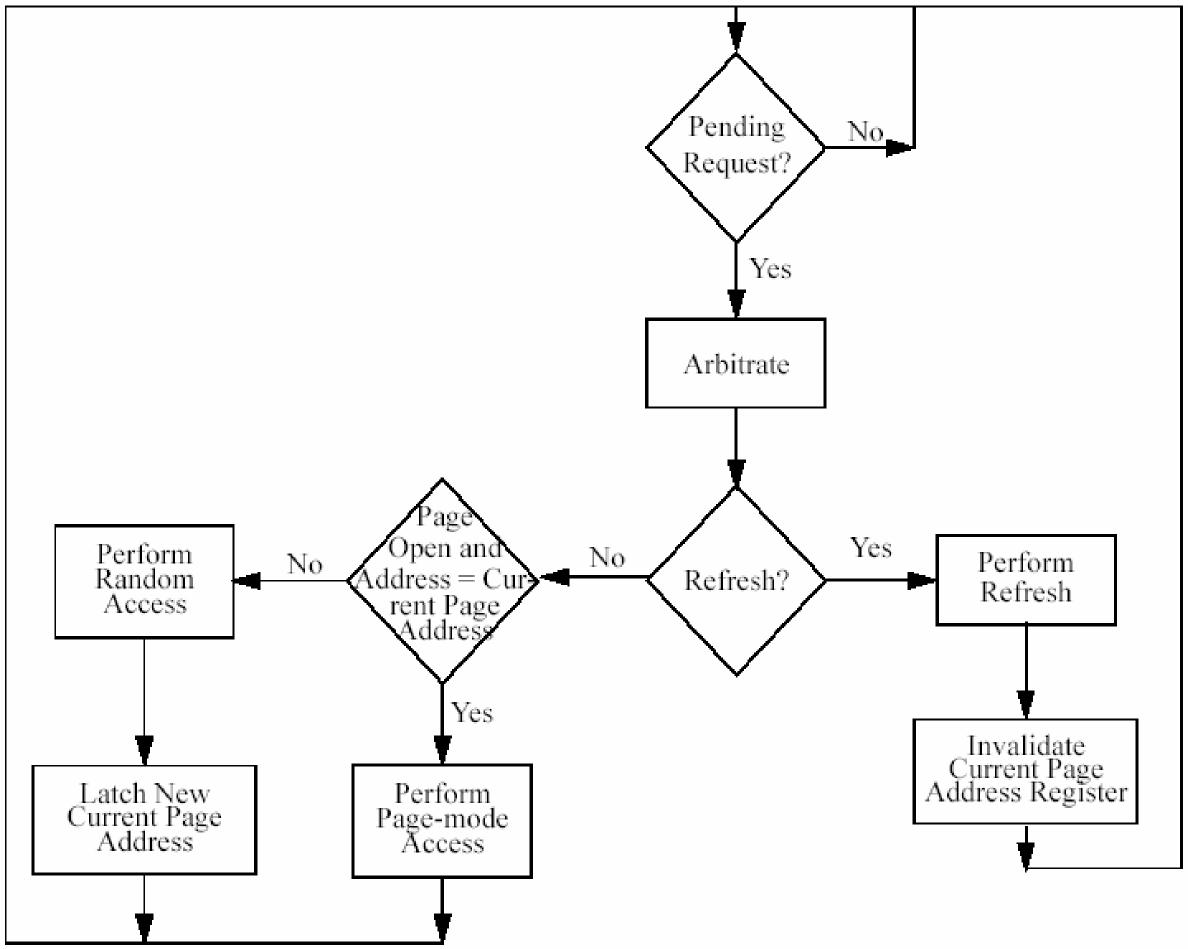
Figure 16: DIVA Memory Controller

Memory Controller Description

A diagram of the memory controller is given in Figure 16. The basic components of the memory controller are an arbiter, a refresh timer, the Current Page Address register, and the Memory Interface. The arbiter is responsible for handshaking with all possible requesters of access to the memory array and determining the priority of competing requests. It communicates closely with the Memory Interface, which is responsible for generating all control signals to the memory array, such as address bits, macro select and page-mode select strobes, refresh pulses, write enables, etc. The Current Page Address register contains the address of the page which is currently held in the sense amps of the memory array.

The operation of the memory controller is best described by the flowchart given in Figure 17. Upon reset, the memory controller is in an idle state awaiting access requests. If any request(s) occurs, the controller performs an arbitration phase. There are two basic types of requests: refresh and normal access (read or write). If a refresh cycle is pending, it is performed. Note that no address is needed for refresh cycles. However, the refresh cycle corrupts the sense amps so the contents of the Current Page Address register are no longer valid.

If the access request that "wins" the arbitration phase is not a refresh request, i.e. it is a normal access, the address presented with the request is compared against the contents of the Current Page Address register, assuming that a page is currently open. If the portion of the requesting address which designates the DRAM page matches the value of the Current Page Address register, the access is performed as a page-mode access, minimizing latency. If the two values are unequal, a random access must be performed, which entails restoring the currently open page and strobing in the new page corresponding to the access request. Simultaneously with this access, the new page address is latched into the Current Page Address register.



Pending Request?
No
Yes
Arbitrate
Refresh?
Yes
No
Perform Refresh
Invalidate Current Page Address Register
Page Open and Address = Current Page Address
No
Yes
Perform Random Access
Latch New Current Page Address
Perform Page-mode Access

Figure 17: DIVA Memory Controller Flowchart

Sources of Requests and Arbitration Priorities

Requests for normal accesses, i.e. reads and writes, may originate from several sources within the PIM node. The possible sources are the host interface port, the processor instruction cache, and memory stage of the processor pipeline. With the possibility of these sources competing for memory access, arbitration priorities must be formulated. As indicated in the flowchart of Figure 17, refresh cycles have the highest priority. The following priority includes the remaining sources:

1. Refresh
2. Host interface
3. Processor memory stage

- Processor instruction cache

After refresh, the host interface has the highest priority since minimal latency penalties for conventional DRAM accesses are highly desired. Since a memory request from either the memory stage or the instruction cache of the processor will stall the processor pipeline, the priority between these makes little difference. If there are requests pending from both, they must both be satisfied before the pipeline can advance. However, the processor memory stage is assigned a higher priority to simplify the pipeline control logic since the memory stage is deeper in the pipeline than the instruction fetch stage.

To request a memory access, each of these units must provide an address, type of access (read or write), and data (for write operations). In addition to these signals, there are handshaking signals between the arbiter and these units to indicate when requests are pending and when they have been granted.

Chapter 7 - Instruction Cache

Introduction

It is of critical importance to keep instruction fetches from interfering with the flow of operand data from the node memories. In addition to the reduction of operand data bandwidth due simply to contention, instruction fetches from memory reduce bandwidth even further due to the resulting increase in memory latency because they disrupt reference locality. Since the code segment of an application is placed in a different area of memory from the data segment, interleaving instruction fetches with operand fetches from memory would cause many random memory accesses that could have otherwise been satisfied in a page-mode fashion. DIVA avoids most of the bandwidth losses by implementing a small instruction cache.

Instruction Cache Description

The DIVA PIM node processor contains a 4-Kbyte, direct-mapped instruction cache. The cache line size is 32 bytes, each of which can be loaded or invalidated individually. In addition, the entire cache can be invalidated by disabling the cache. The DIVA architecture does not support self-modifying code, so the instruction cache does not require any write-back capability. The cache does not contain a snooping port and is therefore not kept coherent with memory automatically. Kernel software is responsible for invalidating stale cache lines when the backing memory for those lines is being loaded with new code.

Instruction Cache Organization

The cache consists of three major components: core ram, tag ram, and the controller. A diagram showing the organization of the core ram and tag ram is shown in Figure 18. The core RAM consists of 128 lines, where each line is 256 bits long. Each line is then capable of storing eight 32-bit instructions. The tag RAM contains a 20-bit tag for each line of core RAM, although the tag size could be reduced to match the amount of physical memory actually present and thereby optimize the storage and performance of tag accesses. Each tag RAM line also contains a valid-bit to indicate whether the line contents is empty or it actually contains valid information.

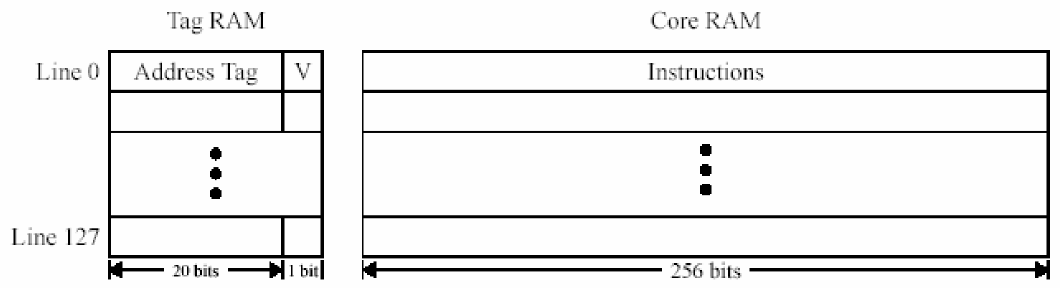


Figure 18: Instruction Cache Organization

A physical address is decoded as shown in Figure 19 for determining placement or validity within the cache. The least two significant bits are ignored as they should always be zero because instructions are 32 bits in size and aligned to 32-bit boundaries. Bits 27 through 29 are used to select a specific instruction within a cache line, and bits 20 through 26 are used to specify the cache line. The upper 20 bits are then used as the tag information for a cache line. The instruction cache unit operates closely with the address translation unit. For example, the least significant 12 bits of instruction virtual addresses are assumed to be unaffected by the address translation process. Therefore, these bits can be used to index into the cache simultaneously with the translation of the upper 20 bits. By the time the appropriate tag has been accessed, the translation has taken place, so that the tag contents can be compared with the physical address.

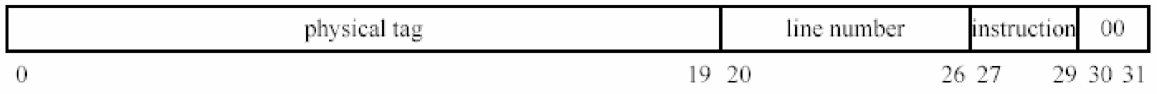


Figure 19: Instruction Cache Address Interpretation

Instruction Cache Operation

The operation of the instruction cache is best described by defining the tasks of the cache controller. The controller is responsible for managing all activity of the cache, including instruction fetches from the cache, loading cache lines from memory, and invalidating cache lines. The controller is basically a finite state machine (FSM) with three states, where each state has sub-states. The FSM diagram is shown in Figure 20.

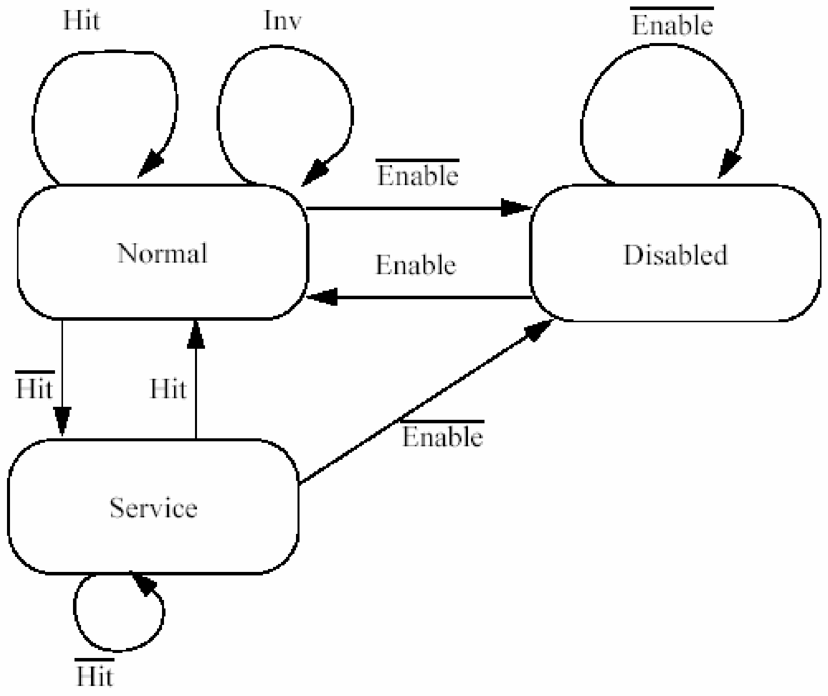


Figure 20: Cache Controller Finite State Machine

At processor boot time, the cache controller in the disabled state. In this state, when the processor makes an instruction request, a 256-bit data item including the desired instruction is fetched from the memory, the requested instruction is selected from the incoming data, and placed onto the instruction bus. All the valid bits are also reset when the controller enters the disabled state.

When code enables the cache, which asserts the enable signal, the controller enters the normal state. In this state two operations are possible: read and invalidate. During a read operation the controller performs an instruction fetch by comparing the tag portion of the supplied address with the tag of the appropriate line of the tag RAM. If they match and the valid bit is set, then the desired word is selected, placed onto the instruction bus, and the hit signal is asserted. Otherwise, the hit signal is negated, and the controller enters the memory service state. If the INV signal is high, then the valid bit of the cache line specified by the instruction address is reset if the tag of the address matches the tag of the line.

The memory service state is very similar to the disabled state. The only difference is that when the data is fetched from the memory, it is also written to the appropriate core RAM line, the tag is written to the corresponding line of the tag RAM, and the valid bit of that line is asserted.

Cache Control Instructions

The only cache control instruction supported by the DIVA instruction set is the icli (instruction cache line invalidate) instruction. This instruction supplies an address using the register plus offset addressing mode. If the address is found in the cache, the corresponding cache line is invalidated.

Chapter 8 - Exceptions

This chapter defines the exceptions and exception-handling mechanism for the DIVA PIM node. Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external interrupt signal, are handled by a common mechanism. For the most part this document will refer to both exceptions and interrupts as exceptions.

Traditionally RISC processors have had relatively primitive mechanisms for exception handling compared to CISC processors which may have multiple stack registers, extensive hardware-supported vectoring and priority-level controls of enabling exceptions. Even with these supporting hardware features, it's common to find problems of priority inversion and stack management errors in interrupt-service software. Errors in priority assignment are not easily fixed once cast in hardware. Exception handling hardware is difficult to implement and integrate with high-performance hardware.

The exception handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, which is easier to mend. It does provide an integrated mechanism for handling hardware and software exception sources. Additionally, it provides a flexible priority assignment scheme which minimizes the amount of time that exception recognition is disabled. While the hardware design supports traditional stack-based exception handlers, we also outline a non-recursive dispatching scheme which uses DIVA hardware features to allow preemption of lower-priority exception handlers using a mechanism which should be easier to debug.

Hardware-Vectored Exceptions

The DIVA node processor must respond to a variety of exceptions due to internal instruction processing conditions and interrupts due to external stimuli. The PIM node processor has only four hardware-vectored exceptions, all others are dispatched by software with some hardware assistance. The exceptions are listed in descending priority order.

TABLE 3. Hardware-Vectored Exceptions

Exception	Vector Address	**Notes**
Hard RESET	TBD	Power-on clear and/or diagnostics
Soft RESET	0x08000000	External reset
Undefined Instruction (incl. BRK)	0x08000100	
Software-vectored exceptions	0x08000200	

The assignment of a vector address to the hard RESET exception depends on a specification of the initial program load bootstrap mechanism.

Note that the three vector addresses other than the hard RESET point to exception handler routines located at the start of node DRAM, so the node DRAM must be initialized and functional for any operation beyond hard RESET.

All exceptions other than reset and undefined-instruction exceptions are vectored by hardware to the catchall "software-vectored exception" handler, which examines the exception source word to perform a software-vectored dispatch to the appropriate exception handler.

Hardware Support for Hardware-Vectored Exceptions

The node processor has several privileged registers and a privileged instruction, RFE, used to return from exception handlers to normal processing.

All exceptions operate in supervisor mode. The program counter and processor status words are copied to privileged temporary registers before exception processing is begun The exception handling code runs in the same address map as the preceding code. Other state changes are performed at the exception handler if necessary. Other registers are set by specific exception conditions, e.g., MADR is set in the event of a memory-access exception. The exception source word is set to indicate the cause of all but the reset and undefined-instruction exceptions, which are implicitly identified by the hardware vectoring to associated exception handlers. The exception source word and its associated enable mask register are discussed at more length in the "Software-Vectored Exceptions" section. The reset and undefined-instruction exceptions may not be disabled. All other exceptions may be disabled in aggregate by setting a bit in the PSW or selectively, by setting a bit in the exception-enable mask register.

TABLE 4. Hardware State at Start of Exception Processing

Register	Field	Value	Notes
PSW	MD	0	Mode is set to supervisor, other fields unchanged
PC		handler	Address of exception handler
FADR		old PC	Address of faulting instruction or next instruction
SSW		old PSW	Saved copy of prior PSW

Upon completion of exception handling, the RFE instruction will copy the FADR to the PC and the SSW to the PSW to resume normal processing. Depending on the cause of the exception, the FADR may point to the instruction that caused the exception, if the exception prevented the instruction from completing, or to the next instruction in the code sequence, if the prior instruction did complete. For example, a memory access fault would load the FADR with the address of the load or store instruction which caused the access exception, while a timer interrupt or external interrupt would load the FADR with the next instruction to be executed. The exception handling code is responsible for adjusting the FADR as needed prior to executing the RFE instruction. Depending on the nature of the exception, the faulting instruction may be retried, for example a WideWord instruction after a lazy register save, or a memory access instruction after an address-translation adjustment.

The node processor provides four scalar system scratch registers to be used by exception handlers. Exception handling code requiring more registers are responsible for saving and restoring node processor registers as needed.

Hardware-Vectored Exception Descriptions

Hard RESET (0xTBD)

This exception provides a starting point for power-on initialization and (optionally) self-test and diagnostic functions for the node. It can be triggered by internal power-on detection circuitry or an external source. At the conclusion of initialization and testing the node processor is ready for initial program loading *by a mechanism TBD. This mechanism may be simplified for a node attached to a host via its system interface.*

In contrast to the soft reset, which generally preserves present hardware state and register contents, a hard reset will set certain hardware to a known state to allow straightforward initialization.

TABLE 5. PSW State at Hard or Soft RESET

Bit	Field	Value	Notes
0	MD	0	Mode is set to supervisor
1	Unused	X	Reserved
2	IC	0	Instruction cache is disabled
3	EE	0	Exception recognition is disabled
4	WW	0	WideWord instruction processing is disabled
5	FP	0	Floating-Point Instruction processing is disabled
6 - 7	Unused	X	Reserved
8	IA	0	Instruction address translation is disabled
9	DA	0	Data address translation is disabled
10 - 31	Unused	X	Reserved

Soft RESET (0x08000000)

After a kernel or monitor program has been loaded into functional DRAM, the external RESET input causes instruction execution to begin at this DRAM address. It is anticipated that this RESET can be triggered either by an external input or by the host processor accessing the node via its system interface.

It is expected that a soft reset handler will dump a detailed snapshot of node status to memory to aid debugging before reinitializing the kernel or monitor data structures.

Undefined Instruction (0x08000100)

This vector services all undefined instruction exceptions and also serves as the primary exception handler for breakpoint instructions. Breakpoint instructions are implemented by a software convention defining one or more undefined instruction opcodes as BRK_x. The FADR register points to the address of the undefined instruction. To allow the BRK mechanism to debug exception handling code, we adopt the convention that SR3 is reserved exclusively for use by this exception handler, which does not use other scratch registers. This is not adequate to allow use of BRK prior to copying of FADR and SSW however.

Software-vectored exceptions (0x08000200)

This vector provides the initial exception handling for all other exceptions and interrupts in the system. Recognition of this aggregate exception may be disabled by privileged code altering the PSW and is automatically disabled upon exception recognition, to remove any hardware requirement to support nested exceptions.

Software-Vectored Exceptions

Most exception sources in the DIVA PIM are serviced by a software-vectored exception handler. Determination of the exception cause requires examination of the 32-bit exception source word, which constantly monitors hardware which may cause exceptions and also provides the ability for software to trigger exceptions.

Nested exceptions can be supported if the exception handler saves essential state, notably FADR and SSW, prior to reenabling exceptions. The software-vectored exception handling procedure supports nesting of exceptions for some potentially lengthy handlers by splitting the exception handler into primary and secondary parts. Primary exception handlers are non-interruptible except for reset and undefined-instruction exceptions. Secondary exception handlers may be interrupted by other exceptions. They may or may not be re-entrantly interrupted by other instances of the same exception type, depending on the handler code treatment of the mask register.

Lightweight Exceptions

Lightweight exceptions are those which can be serviced completely within the primary exception handler, and do not require saving of transient exception state. Hardware disables further exceptions until reenabled by execution of RFE.

An example of a lightweight exception is the timer tick exception, which increments a counter in memory. If the tick does not end a scheduling quantum, no further processing is required. If the tick does end a scheduling quantum, it triggers a quantum-expiration exception, but does no further processing itself.

Heavyweight Exceptions

Heavyweight exceptions are those which cannot be serviced entirely within a primary exception handler. The primary exception handler saves necessary exception state in one of three locations. Temporary use is made of the system scratch registers. Processor context is saved, as necessary, in a register save area in a fixed-location memory area common to all primary exception handlers. Information specific to the particular exception, which is required for later processing by the secondary exception handler is saved in a fixed-location memory area specific to that particular exception type.

Primary Exception Handlers

Primary exception handlers perform all of the processing for lightweight exceptions and the initial time-critical portion of heavyweight exceptions.

The environment of primary exception handlers is highly constrained. They may use the system scratch registers SR0-SR3 freely but must save and restore any other GPRs. Primary handlers may call other routines conforming to the constraints, but must use the exception stack, which is located at the top of the kernel stack segment. Calling a subroutine in the primary exception handler environment requires initializing the stack pointer to the fixed top of the exception stack area. Primary handlers are written in assembly language.

Secondary Exception Handlers

Secondary exception handlers perform the non-initial processing of heavyweight exceptions. They may not use the system scratch registers SR0-SR3, since exceptions are enabled during most of the execution of the secondary handler. Secondary handlers may be written in a restricted subset of the C language. Secondary handlers are written in a stylized form providing functions to suspend and resume their processing if preempted by higher priority exceptions.

Hardware Support for Software-Vectored Exceptions

All software-vectored exception sources have an associated bit defined in the 32-bit exception source word, ESW, and corresponding bits in the exception-enable mask register, EMR, the exception set register, ESR, and the exception reset register, ERR. When a software-vectored exception is recognized, the global exception enable bit in the processor status word, PSW, is cleared, so that hardware events which cause changes to the ESW cannot trigger a nested exception. Reset and undefined instruction exceptions may preempt primary exception handling code, but other exceptions will not be recognized.

The exception source word is a 32b register recording exceptions initiated both by hardware and software sources. Hardware-source bits in the exception source word may be set to one by hardware conditions, such as a pbuf interrupt, while software-source fields are set by software writing a one to the corresponding bit location in the exception set register. Once set, a bit in the exception source word can be cleared only by writing a one to the corresponding bit of the exception reset register. Although labeled registers, both ESR and ERR are really reg-

TABLE 6. Exception-Related Registers

Name	PR#	Description
Exception Source Word (ESW)	8	Specifies sources of exceptions
Exception Enable Mask Register (EMR)	9	Bitwise exception enabling mask, 1 = enabled
Exception Set Register (ESR)	10	Write 1 to set corresponding bit in source word, SW-source fields only
Exception Reset Register (ERR)	11	Write 1 to clear corresponding bit in word register

ister-address triggering functions. That is, a one written to any bit in either of these registers causes an immediate and one-time effect on the corresponding bit in the exception source word; ESR and ERR do not maintain any state.

Bits in ESW are affected by hardware conditions and ESR and ERR actions regardless of settings of the exception enable mask register, EMR. The bits of EMR merely enable, or disable, corresponding bits of ESW to cause exceptions. Therefore, there is a global exception enable control via the exception enable bit in PSW and individually maskable controls for each bit of the ESW via the EMR.

The Exception Source Word has 32 possible hardware- and software-initiated exception sources. The priority of the sources decreases with increasing bit number.

TABLE 7. Exception Source Word

Exception Name	Initiator	Bit#	Description
Watchdog Timer	HW	0	May not be implemented
Unmapped Instruction Access	HW	1	Instruction access not within segment boundaries
Invalid Instruction Access	HW	2	Instruction access not permitted
Unmapped Data Access	HW	3	Data access not within segment boundaries
Invalid Data Access	HW	4	Data access not permitted
PBuf Receive Interrupt	HW	5	
PBuf Send Error	HW	6	
Interval Timer	HW	7	Tick counter
WideWord Not Available	HW	8	WideWord instructions attempted without enable
Floating Point Not Available	HW	9	Floating-point instructions attempted without enable
Address Fault Fix-up	SW	10	
Received Packet Processing	SW	11	
Send Error Processing	SW	12	

TABLE 7. Exception Source Word

Exception Name	Initiator	Bit#	**Description**
Reserved	SW	13	
Host Interrupt	HW	14	May not be implemented
FP Divide by Zero	HW	15	Refer to FPSR description
Host Interrupt Processing	SW	16	
FP Unsupported Value	HW	17	Refer to FPSR description
Context Swapper	SW	18	
System Call	HW	19	
Privileged Instruction Violation	HW	20	
Scalar Integer ALU Exception	HW	21	
WideWord Integer ALU Exception	HW	22	
FP Inexact/Invalid	HW	23	Refer to FPSR description
Integer ALU Fix-up	SW	24	
WideWord ALU Fix-up	SW	25	
Floating Point Fix-up	SW	26	
Reserved	SW	27	
Lock Buzzer	SW	28	May not be implemented
Thread Rescheduler	SW	29	
Thread Dispatcher	SW	30	
Return to User Mode	SW	31	Full register restore as necessary

Dispatch of Software-Vectored Exception Handlers

The DIVA PIM node exception handling mechanism requires little specialized hardware support and supports preemption of lengthy low priority handlers without requiring LIFO processing due to stack mechanisms. Dispatch is always to the highest priority exception handler. There is no possibility of pathological stack growth under high rates of exceptions. System overload due to design problems will manifest as overruns, which can be evident and recoverable, rather than stack explosion, which is typically obscure and fatal.

Dispatch to the primary handler

A new exception condition will be recognized if exceptions are enabled in the PSW and if the particular source is enabled by the mask register. The hardware begins execution of code at the software-vectored exception vector address. Exceptions are disabled in the new PSW. Since the primary handlers are non-recursive and run to completion, processor state can be saved to a reserved temporary area at a fixed address (rather than a true stack) as needed by the particular handler.

The exception source word is copied into a scalar GPR and the ELO instruction is used to encode the bit number of the leftmost (smallest numbered) set bit. This operation selects the highest priority source. The encoded source bit number is used as the index into a vector of handler addresses, and the processor branches to that primary handler.

Completion of a primary handler

The selected primary handler determines whether the exception is lightweight enough to be handled in the primary handler or whether additional processing must be deferred to the secondary handler.

If the primary handler can complete the exception processing, it does so and then restores the saved GPRs and status before reenabling exception recognition by executing the RFE instruction. Prior to completion it will reset its associated exception source bit.

If the primary handler cannot complete the exception processing, it will copy the necessary state to a structure associated with its secondary handler, and set the bit associated with the secondary handler by writing to the exception set register. After restoring saved GPRs and status and resetting its source bit, it reenables exception recognition by executing the RFE instruction. The highest-priority exception source will subsequently be recognized and begin exception processing. This may be the secondary handler just scheduled or a higher priority hardware or software exception handler.

We can optimize restoring GPRs by delaying this until all secondary handlers have completed and no further exceptions are present. The deferred GPR restore can be accomplished by setting a software exception for the "return to user mode" handler. It remains to be seen whether this is generally advantageous for performance.

Dispatch of a secondary handler

The initial part of the software vectoring of a secondary handler is the same as a primary handler. After the handler branches to the specific secondary handler code, the secondary handler is required to perform more elaborate state saving due to the possibility of preemption by higher-priority sources. The first portion of the secondary handler runs with exceptions disabled.

When a secondary handler begins execution it installs a pointer to its environment structure in the privileged register SR2. If the prior value of SR2 is zero, it is not preempting another secondary handler. If the prior value is nonzero, it is preempting another lower-priority secondary handler. To preempt, the current handler saves the state of the prior secondary handler by calling its suspend routine, the address of which is at a fixed offset within the environment The suspend routine copies the necessary state into the environment and returns. The environment will typically hold only one instance of a given type of suspended secondary handler. This means that while exceptions can interrupt and preempt secondary handlers of a different type, we don't support reentrant handling of multiple exceptions of the same type. While it is a straightforward extension to support a per-type stack or queue of multiple exception instances, in most circumstances the inability to complete exception processing prior to encountering a subsequent exception of the same type reflects an underlying system-design problem.

The handler is coded to record its essential state at periodic intervals. In effect, it stores a checkpoint record of its progress in its environment with sufficient detail to allow processing to resume in the event of a preemption. A technique sufficient to maintain atomicity is to "double buffer" a structure with essential information and "flip" between the consistent and working copies with a write to an index or pointer variable. Code progress can be recorded by using a state variable for a software state machine or by updating function pointers.

In contrast to a traditional stack-based system, which keeps activation records on a stack which must be unwound in a LIFO order, our dispatch scheme records the activation of the handler by a bit in the exception source vector, while storing the associated saved state of preempted handlers in handler-specific environment structures. This ensures completion of handlers in priority order without requiring hardware support of multiple priority levels for exception recognition. It may also reduce the amount of saved state. The handler itself can be coded to record the bare minimum of state to allow a resumption, rather than being forced to assume the worst case and save entire register sets which may or may not have been altered. This is particularly significant for the large register sets of the DIVA PIM node.

The "checkpointed" exceptions scheme is much easier to debug via an interactive debugger or memory dump, since the state of each active exception handler is recorded at fixed locations in a form which may be conveniently examined as a high-level structure. This is in contrast to a preemptive stack-based record, where the states of several handlers may be distributed in their lowest-level bindings across large chunks of stack at highly variable locations.

Completion of a secondary handler

The secondary handler completes by reinitializing its checkpoint record to its starting state, resetting its associated exception source word bit, and executing an RFE.

If no other exception is recognized, the lowest-priority software exception will restore all disturbed register states and return to user mode code.

Software-Vectored Exception Descriptions

Each exception needs to have detailed the relevant hardware status registers and in particular the correct interpretation of FADR. This will require some iteration to converge on a good set of hardware exceptions.

The description of software exceptions is less helpful, since the details of the runtime kernel will define both the priority and meaning of software-initiated secondary exceptions.

Chapter 9 - Parcel Buffer

Introduction

The communication abstraction in DIVA is a *parcel (PARallel Computing ELement)*. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received. Parcels are distinguished from active messages in that the destination of a parcel is an *object in memory*, not a specific processor. From a programmer's view, parcels, together with the global address space supported in DIVA, provide a compromise between the ease of programming a shared-memory system and the architectural simplicity of pure message passing. Remote operations or accesses can be accomplished through parcel sends and receives; application programs need specify only the address of an object, and not the processor upon which the object resides.

The basic mechanism used in the DIVA system to support parcel sending/receiving from/to an application is a parcel buffer (or *pbuf*). The pbuf has a virtual as well as a physical abstraction. To the application, the pbuf locations appear as regular memory locations that are manipulated through simple loads and stores. At a physical level, the pbuf is a set of memory-mapped registers. Each PIM node contains a pbuf that serves as a port between the on-chip parcel interconnect and the node. (The on-chip parcel interconnect connects node pbufs, the host interface pbuf, and the PIM Routing Component, or PiRC.) Although the parcel buffer could be implemented as registers within the PIM node processor, a memory-mapped mechanism for the parcel buffer allows a uniform implementation for the node's pbuf as well as a host pbuf. Hence, a pbuf within the PIM chip host interface is memory-mapped into the host processor's address space to allow the host processor to communicate with PIM nodes via the parcel mechanism.

To launch parcels, a user simply writes to appropriate fields in the pbuf. To receive parcels, users may either use an interrupt or polling methodology to know when to read parcels from the pbuf. Parcel buffer access is managed by the system in the same fashion as any other region of memory in the node's local address space (refer to Chapter 10).

Parcel Format

The physical parcel format is shown in Figure 21. A parcel consists of a 96-bit header and 256-bit payload. Most of the parcel contents are

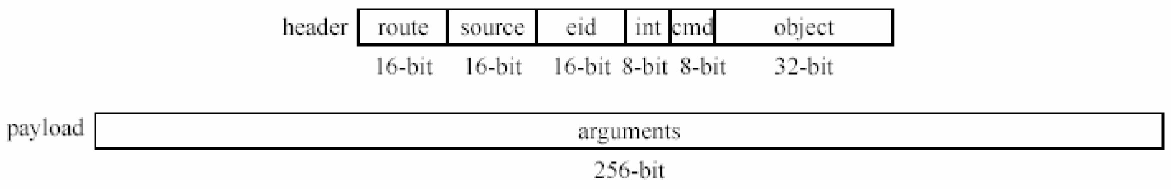


Figure 21: Physical Parcel Format and Fields

written by the user during a parcel launch; however, the system is responsible for generating the route, source, eid, and int fields. The *route* field is a 16-bit value that is used by the PiRC to direct a parcel to the correct PIM chip and node. The *source* field is a 16-bit value that represents the node ID of the sender and can be used by kernel software to correct routing errors. The *eid* field is the 16-bit environment identifier of the process that launched the parcel, and the 8-bit *int* field indicates whether the parcel should generate an interrupt at the receiving pbuf. The *object* field is a 32-bit virtual address of the object to which the parcel is directed, and the *cmd* is an 8-bit identifier that the user can use to index into a table of commands for the specified object. The 256-bit payload consists of arguments for the command task or other data associated with the action specified by the parcel.

Parcel Buffer Addressing

Data is written to or read from the pbuf in 256-bit increments via the WideWord Unit registers. The pbuf address space can then be viewed as a set of 256-bit registers. Besides the header and payload registers, there are also status and configuration registers. Although the payload is the only true physical 256-bit register, each register is allocated 256 bits of the address space and is aligned to the least significant bit boundary. For example, the 96-bit header is aligned to the least significant 96 bits of the 256-bit register space it is allocated. At least two register sets are needed: one for sending and one for receiving. In addition, it is desirable to have multiple address mappings (aliases) of these sets to support different access privileges and modes, as described later. To minimize interface issues with standard CPU cache line sizes in supporting this feature, 256 bytes of address space are allocated for each virtual copy of a pbuf register set, with the 256 bytes distributed to eight 256-bit registers. The 256-byte register set space for the pbuf send side is shown in Table 8. The payload and header are as described

TABLE 8. Parcel Buffer Send Register Set

Address Relative to Register Set Base	Register Description	Physical Size (bits)	Access Privilege
0x00	payload	256	supervisor/user
0x20	header	96	supervisor/limited user
0x40	status	3	supervisor/limited user
0x60	reserved	NA	NA
0x80	source	16	supervisor
0xA0	eid	16	supervisor
0xC0	route cache entry	96	supervisor
0xE0	route cache invalidate	NA	supervisor

in Figure 21. The status bits and route cache entry/invalidate are described in later subsections. The source and eid registers are intended to be accessed only by the trusted supervisor kernel. Such access protection is accomplished through address aliases for the pbuf, as described in the following paragraphs. At PIM node boot time, the kernel writes a node ID to the source register. This value is copied into the source field of every outgoing user parcel when launched. When any new application is swapped in, the kernel should also write the application's eid value into the eid register in the pbuf send register set. The value of the eid register is copied into the eid field of every outgoing user parcel when launched.

The corresponding register space for the pbuf receive side is shown in Table 9.

TABLE 9. Parcel Buffer Receive Register Set

Address Relative to Register Set Base	Register Description	Physical Size (bits)	Access Privilege
0x00	payload	256	supervisor/user
0x20	header	96	supervisor/user

TABLE 9. Parcel Buffer Receive Register Set

Address Relative to Register Set Base	Register Description	Physical Size (bits)	Access Privilege
0x40	status	5	supervisor/limited user
0x60	reserved	NA	NA
0x80	reserved	NA	NA
0xA0	reserved	NA	NA
0xC0	reserved	NA	NA
0xE0	reserved	NA	NA

As mentioned previously, the 256-byte pbuf send and receive register sets are multiply mapped to support a number of desired features. First, two aliases of the send register set are desired to support different functions: one address space for non-triggering writes, one address space for triggering writes. The distinction is that writes to the non-triggering address space simply enter new data into the send register set but do not cause a parcel launch. In contrast, a write to a register within the triggering address space not only causes new data to be written into the specified register but also initiates a parcel launch, which results in the parcel contents of the pbuf being forwarded to the PiRC. The provision of triggering and non-triggering spaces supports several nice capabilities but is also necessary for restoration of the pbuf state upon context switches. With this support, it is not necessary to write both the header and payload to launch a parcel. For instance, if a multicasting operation is desired, it is only necessary to write the payload once to the non-triggering address and then trigger a parcel to each destination of the multicast by writing the appropriate header to the triggering address for each destination object. Similarly, if it is desired to send multiple parcel payloads to the same object, only the payload need be written to the triggering address for each send once the header has been initialized.

Similarly, there are two aliases of the receive register set: one space for non-destructive reads, one space for destructive reads. The non-destructive read merely reads from the pbuf location but does not cause the data to be removed from the pbuf. The destructive read also returns the specified payload or header register contents, but it also causes the status of the receive register set to be marked empty so that any parcel waiting at the PiRC may then be forwarded to the pbuf. In a sense, the parcel that is read from the destructive read address is then removed from the pbuf.

There are other capabilities that are also desired that are accomplished through even more aliases of the pbuf hardware. For instance, it is useful for a process that is launching a parcel to be able to specify whether that parcel should generate an interrupt when it arrives at its destination node. By using another set of aliases for this function, one address bit can be decoded to determine if the parcel should generate an interrupt once it arrives at a destination pbuf. The system should set up the address translation unit appropriately to grant or revoke such privileges from users. Any write to a sending header register address, whether it is triggering or non-triggering, updates the *int* field of the current pbuf header. If the system or user writes to the pbuf header field at the interrupting address, the *int* field of the parcel gets set to indicate the parcel is to generate an interrupt at the receiving pbuf; otherwise, the int field indicates no interrupt.

Additionally, it is desired that the supervisor be able to explicitly write the bits that are to be contained in an outgoing parcel. A user has no control over the route, source, eid, and int fields; these fields are generated by mechanisms set up by the supervisor kernel. However, the supervisor should be able to circumvent such mechanisms to write to these fields directly. To implement such a capability, another set of aliases is used. It is assumed that kernel software sets up the address translation tables appropriately so that only the supervisor may access the pbuf hardware at the supervisor addresses. In addition to the ability of the supervisor to write bits explicitly into an outgoing parcel, there are parts of the pbuf hardware that are intended to be accessible only to a trusted supervisor, as indicated in Table 8 and Table 9. For example, some of the status bits should only change state when accessed by the supervisor kernel (refer to the subsection on status bits). The supervisor aliases of the pbuf hardware accomplish this task as well.

Combining support for triggering/non-triggering writes, destructive/non-destructive reads, interrupt specification, and supervisor/user capability, 6 aliases are used for each of the pbuf send and receive register sets, as shown in Table 10. The total address space required to support

TABLE 10. Address Mapping of Pbuf Aliases

Alias Address Relative to Pbuf Base	Register Set Type	Type of Write (send) or Read (receive)	Parcel Type	Access Level
0x000	send	non-triggering	interrupting	user
0x100	send	triggering		
0x200	receive	non-destructive	NA	
0x300	receive	destructive		
0x400	send	non-triggering	non-interrupting	
0x500	send	triggering		
0x600	receive	non-destructive	NA	
0x700	receive	destructive		
0x800	send	non-triggering	explicitly specified	supervisor
0x900	send	triggering		
0xA00	receive	non-destructive	NA	
0xB00	receive	destructive		

this pbuf functionality is 3 Kbytes. Recall that there need be only one set of parcel buffer hardware for send and receive functions; the multiple address mappings exist only to use address bits to impart information to the pbuf control hardware. By using address bits to control such features, access privileges to the pbuf hardware can be granted or revoked by the supervisor kernel by normal management of the address translation unit. Many of the register aliases are not needed to support some of this functionality; for example, a copy of the receive register set for the interrupting/non-interrupting launching capability is unnecessary. However, to accommodate protection through the segmented memory management scheme, it is necessary for the alias sets to be arranged as shown in Table 10.

Parcel Buffer Status Bits

The pbuf send and receive hardware maintain a handful of bits to indicate the state of the pbuf. The three status bits associated with the send side are shown in Figure 22. (Note this 3-bit status register is aligned to bit positions 253 - 255 of the status register address space within the send register set.) The buffer empty bit indicates when it is possible for a parcel to be written into the pbuf. When this bit is set, the buffer is empty and a new parcel may be written to it. The bit is reset indicating the buffer is full when an application (user or supervisor) writes to a triggering address to launch a parcel. It is then once again set when the parcel transits out of the pbuf to the PiRC or host pbuf. This status bit can be used to support a "safe mode" for sending parcels. Before writing a new parcel to the pbuf, an application can check this bit to ensure the pbuf is available, i.e., the last parcel written to it has exited.

The overrun bit is used to indicate that an application has attempted to write a new parcel to the pbuf, although the pbuf is not empty. This bit is a sticky error bit. It gets set when an overrun occurs and is reset only when the status bits are read. Since it is combined with other status bits, it is important that anytime the send status is read, the state of the overrun bit should be checked if applications are not using the safe mode to send parcels. User writes to the pbuf send registers are ignored if the overrun bit is set. The application must clear the overrun error to resume launching parcels. This bit may also be set by a write to any send status register address contained in a supervisor mapping of the pbuf. This capability is needed to restore the state of the pbuf upon context switch.

The route error bit indicates that the system does not have sufficient information to translate an object address to a route. When an application writes to the triggering address of a send register to launch a parcel, a certain amount of system processing is applied to the parcel before it is forwarded to the PiRC or host pbuf. One of the tasks is the generation of a route from the object address. More information about this translation is given in the last section of this chapter. If the pbuf hardware cannot automatically generate this route, the route error bit is set causing an exception. This bit is reset when the supervisor reads the status register from a designated supervisor alias for the pbuf. When a

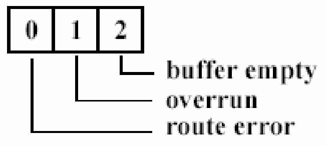


Figure 22: Pbuf Send Status Bits

route error occurs, the buffer empty bit is re-asserted, even though the parcel has not exited the pbuf. This allows the supervisor to read the contents of the pbuf, construct the proper route if possible, and then re-launch the parcel on behalf of the user without incurring an overrun error.

The five status bits associated with the pbuf receive register set are shown in Figure 23. (Note this 5-bit status register is aligned to bit positions 251 - 255 of the status register address space within the receive register set.) The buffer full bit indicates that a parcel has been loaded into the pbuf register set from the PiRC or host pbuf and is available for reading. This bit is set when the PiRC or host pbuf forwards data to the node pbuf and is reset when an application performs a destructive read by reading from the appropriate pbuf alias. When the bit is reset, it serves as a signal to the PiRC or host pbuf that the next parcel destined for this pbuf may be forwarded. If an application performs a read (destructive or non-destructive) when the buffer is empty, all 0s are returned and the underrun status bit is set. Similar to the send overrun bit, the receive underrun bit is also sticky and remains set until the user reads the status register.

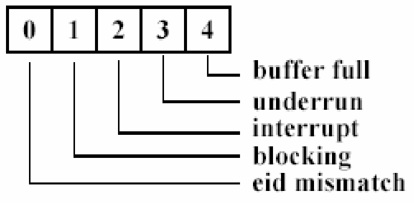


Figure 23: Pbuf Receive Status Bits

The interrupt, blocking, and eid mismatch bits all generate exceptions when set and remain set until the supervisor reads the status register from a designated supervisor pbuf alias. The interrupt bit is set when a parcel that has its interrupt field set arrives to the receive registers. The blocking bit is set when the PiRC or host pbuf is attempting to forward a parcel to the receive registers but cannot do so because the receive registers still contain the previous parcel for an extended period of time. A buried 10-bit timeout counter is associated with this function. The counter starts incrementing anytime the pbuf receive buffer is full and a request for parcel forwarding from the PiRC or host pbuf occurs. If the counter reaches its maximum value, the blocking bit is set. If the parcel is destructively read before the counter expires, then the counter is stopped and cleared, and no blocking is recorded. The eid mismatch bit is set when the eid field of the parcel in the receive registers does not match the pbuf eid register contents when a user read is attempted on any of the pbuf receive registers, including the status registers. In such a case, the buffer full status bit should also be masked when the status is delivered to the reading process. The concept is that a user application should view the pbuf receive registers as empty if the parcel contained in them is for a different application, as indicated by the eid values. However, the supervisor should always be able to read the pbuf contents without encountering any eid mismatch error, regardless of the eid values in the parcel and pbuf configuration. Therefore, eid checking does not apply to supervisor reads from the pbuf.

Parcel Buffer Exceptions

As described in the previous section there are a number of pbuf events that may cause exceptions. The pbuf receive events that cause an exception are indicated by the interrupt, blocking, and eid mismatch status bits. These bits are simply ORed together to set bit 5 of the Exception Source Word (ESW) of the DIVA processor (refer to Chapter 8). When the kernel is invoked by this exception, the kernel must first read the pbuf receive status register to determine which type of event caused the exception and then take appropriate measures to respond to the exception. Similarly, a pbuf send event that causes an exception is indicated by the route error bit. When set, this status bit also causes bit 6 of the ESW to be set. Since this is the only send event that may cause an error, the kernel does not need to perform any extra decoding; however, it will still need to read the send status register to clear the route error bit.

Object Address to Route Translation

The architecture allows for hardware support to facilitate the generation of routes from object addresses. The most flexible mechanism for supporting this capability is a route cache which simply contains mappings from objects to routes. The supervisor kernel manages entering, placement, replacement, and invalidation of all entries explicitly. The form of a route cache entry is shown in Figure 24. (Note that the 96-bit entry is aligned to bits 160 - 255 of the route cache register space.) The supervisor makes such an entry into the route cache by simply writing the required data to the route cache entry register space of the pbuf. Since this space is designated for supervisor access only, the

index	route	mask	object address
16-bit	16-bit	32-bit	32-bit

Figure 24: Route Cache Entry Format

supervisor pbuf alias address must be used to accomplish a successful entry. The 16-bit index specifies which location of the route cache to store the entry; in this manner, the supervisor explicitly manages placement and replacement activity. Although 16 bits are shown for the index, a smaller number of bits may actually be used in specific implementations of this architecture. For instance, if the route cache allows for only 16 entries, only the 4 least significant bits of the index field are used.

For implementations which support a route cache, when a parcel is launched, the cache is searched for the object address specified in the parcel. The mask field of a valid route cache entry indicates which bits of the corresponding object address should be compared to the parcel object address to determine a successful match. An equation specifying a match M where *parob* is the parcel object address and *rcob* is the route cache object address is given by $\overline{M} = (\mathrm{mask}_0 \wedge (\mathrm{rcob}_0 \oplus \mathrm{parob}_0)) \vee (\mathrm{mask}_1 \wedge (\mathrm{rcob}_1 \oplus \mathrm{parob}_1)) \vee \ldots \vee (\mathrm{mask}_{23} \wedge (\mathrm{rcob}_{23} \oplus \mathrm{parob}_{23}))$. If a match is found, the corresponding route is written into the route field of the parcel. The hardware does not protect against matches on multiple entries in the route cache, i.e., system software must set up the route cache appropriately so that only one entry will match any given object address. The route cache behavior is undefined if multiple entries match. The match does not actually have to include the full 32-bit address. Since the smallest allowable segment from the perspective of a DIVA PIM processor is 256 bytes, the match can be performed using the most significant 24 bits of the parcel object address and route cache object address and mask.

The route cache contains buried valid state bits---one for each entry. These valid bits are negated upon reset and any time a route cache invalidate is executed. A valid bit for a particular entry is set when the index corresponding to that entry is written to. For the supervisor to be able to manage the route buffer between multitasking user processes, supervisor-controlled invalidation is provided via an address-mapped mechanism, similar to other pbuf functions. Anytime the supervisor writes to the route cache invalidation address (offset 0xE0), the entire route cache is invalidated. For this mechanism, the data contained in the write is irrelevant since it is not used in any meaningful manner.

The contents of the route cache may be read for debugging purposes. An internal address counter is maintained to provide this capability. Upon reset, this counter points to index 0 of the route cache. Upon each read from the route cache entry address (offset 0xC0), data corresponding to the indexed entry indicated by the current contents of the counter are returned, and the counter increments to point to the next data. The counter value representing the index and the valid bit status are also returned for the entry. The format of the 97-bit data returned upon such a read is shown in Figure 25. (Note that the 97-bit data is aligned to bits 159 - 255 of the node data bus.)

v	index	route	mask	object address
	16-bit	16-bit	32-bit	32-bit

Figure 25: Data Format for Route Cache Read

In lieu of any such hardware or if the cache does not contain a translation for the object of a parcel to be launched, a route exception occurs and the kernel must explicitly set up the route field of the parcel by using the supervisor alias addresses for the pbuf. As part of the exception handling, the kernel may wish to make an entry into the route cache for the object address segment which caused the exception to prevent further exceptions due to that segment. If a specific implementation of the pbuf architecture does not contain hardware support for route generation, parcel launching will always invoke the supervisor kernel to generate the route.

Chapter 10 - Address Translation

Introduction

Parcels, application code, and data contain virtual addresses. To interpret these addresses, a PIM processor must support a translation mechanism. However, the overhead of maintaining conventional page tables at each node is prohibitive. To simplify translation, we classify DIVA memory according to usage:

- *global memory* is composed of contiguous segments distributed across nodes, visible to applications running on the host and PIM nodes.
- *dumb memory* is a region of a node's memory allocated as conventional pages in a host application's virtual space and untouched by PIM node processing.
- *local memory* is a region of a node's memory used exclusively by node routines. This rule is excepted during initialization when the host system boot process loads node software.

A node must be able to rapidly determine if an address is located in its own memory, and if so, find the physical address. To condense translation information, we use *segments*, each of which is defined by segment registers containing a base address and size. The local memory region is partitioned into eight segments in the initial DIVA architecture, although this number could change in future DIVA architectures. Like pages in a conventional system, the segment descriptors are generic in nature. It is only through system programming that the segments serve a specific purpose. For example, a logical allocation of the eight segments would be to assign one segment for each of the following:

1. Kernel code
2. Kernel data
3. Kernel stack
4. Kernel parcel buffer
5. User code
6. User data
7. User stack
8. User parcel buffer

Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation. In addition to the local segments, a node maintains translation information for its resident portion of the global memory, as well as for any remote data for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

The primary functions of the node address translation unit are to translate virtual addresses to physical addresses for those accesses which are locally resident and to provide access protection. The types of accesses generated by a DIVA PIM processor that require translation include instruction fetches and data accesses to memory or memory-mapped devices such as parcel buffers, generated by load or store instructions.

Given the simplicity of the address translation scheme discussed above, very little hardware support is needed to effect efficient translation. A segment base address register and limit register is needed for each of the eight local segments. Also, one virtual base, limit, and physical base register are needed for each resident global segment. The initial DIVA architecture provides four sets of global segment registers, although alternative architectures could provide more. The address translation unit contains no direct support for home node translation, although the preferred system programming is such that the global segments resident on a node form the portion of global memory for which that node is the home node. If this is not the case, address faults invoke system software which performs the home node translation.

Address Translation Mechanisms

The DIVA PIM processor provides 4 Gbytes of virtual address space accessible to kernel and user applications via segments that are a power of 2 in size. Segment sizes can range from 256 bytes to the maximum amount of physical memory available to a node. The initial DIVA architecture supports a maximum segment size of 16 MBytes. Each virtual address generated by the PIM processor is 32 bits wide, and the resulting physical address generated by the address translation unit is also 32 bits wide, although implementations may reduce this width to optimize for the actual amount of physical memory present.

The PIM processor address translation unit supports three main types of address translation:

- direct address translation
- local address translation
- global address translation

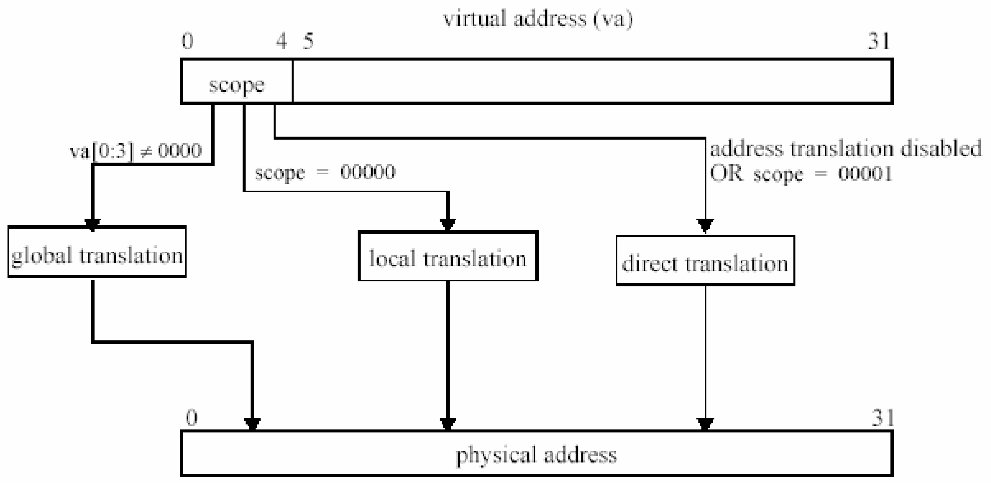


Figure 26: Address Translation Types

Figure 26 shows the three main address translation mechanisms provided. When the address translation unit is disabled, direct address translation occurs, and the address translation unit will not generate any exceptions. In this case, the resulting physical address is identical to the virtual address. If address translation is enabled, then the scope field of the virtual address must be inspected to determine what type of translation should be used. In the initial DIVA architecture, the scope field is the most significant five bits of the virtual address VA. If this 5-bit value is zero, then local translation is used. If the scope field equals binary value 00001, i.e., the virtual address falls in the range of 0x08000000 to 0x0FFFFFFF, direct translation is used to generate the physical address; however, unlike the mode where address translation is disabled, an exception can be generated in this case if access privileges are violated. By definition, the address region 0x08000000 to

0x0FFFFFFF is a supervisor-level region. Therefore, any user-level attempt to access this region while address translation is enabled will trigger an exception. Lastly, if any of the four most significant bits of the virtual address are non-zero, i.e., va[0:3] != 0, then global translation is used.

Figure 27 shows the steps involved in local address translation. The 3-bit index field of the virtual address is used to select a set of local segment registers for the translation. The segment base is simply bitwise-ORed with the zero-padded offset of the virtual address to form the physical address. The specified segment limit register is also accessed and manipulated in conjunction with the offset to determine if the virtual address is valid. More information on protection is given in the next section.

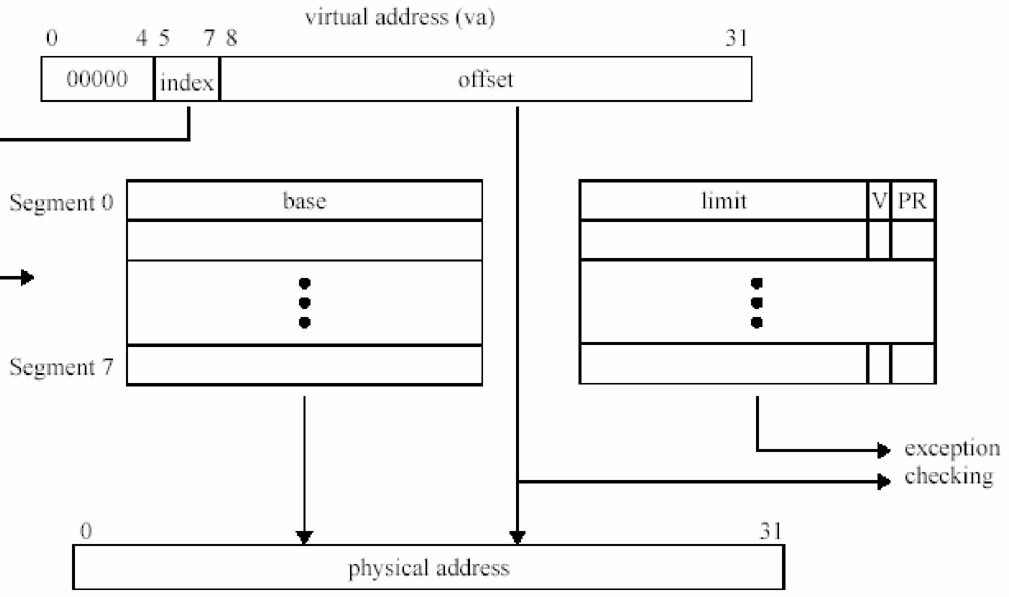


Figure 27: Local Address Translation

Figure 28 shows the steps involved in global address translation, which is a reverse address translation style. In this case, the address is checked to see if it is mapped locally by simply ensuring that the address is within the range specified by a valid set of the global segment base address and limit registers. The hardware does not protect against overlapping global segments, i.e., system software must set up the global segment registers appropriately so that any global virtual address is contained in at most one global segment. The multiple sets of global segment registers are checked concurrently to see if any one of them should be used for the translation, similar to a fully associative cache. If there is no match, a translation exception occurs. More detail on this matching and protection checking is given in the next section. If there is a match, the virtual address is simply translated into a physical address by a bitwise-OR of an offset with the global segment physical base register of the matching global segment. The offset is formed by using the limit register of the matching segment to mask off the appropriate part of the virtual address.

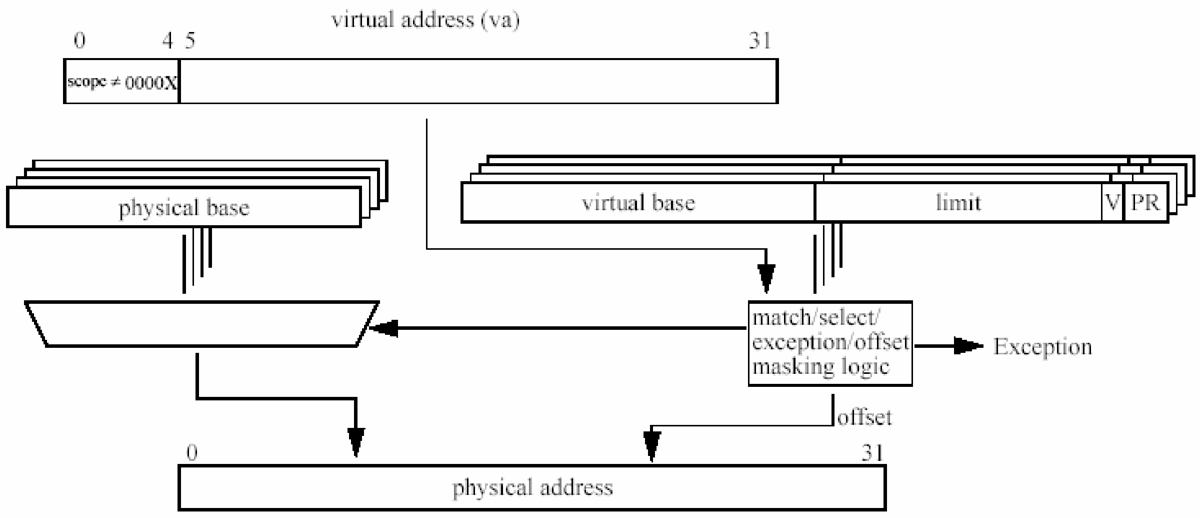


Figure 28: Global Address Translation

Memory Access Protection

In addition to the translation of virtual addresses to physical addresses, the address translation unit provides access protection and bounds checking to ensure that the offset portion of an address is not outside the range of the segment. The 2 PR bits of a segment limit register specify the access protection mode for that segment. Table 11 shows the possible access modes and their corresponding encodings.

TABLE 11. Segment Access Modes and Corresponding PR Bit Encodings

Encoding of PR Bits	Supervisor Privilege	User Privilege
00	RW (read-write)	RW
01	RW	RO (read only)
10	RW	none
11	RO	none

Each local segment limit register consists of a limit value, a valid bit, and the two PR bits. The first level of protection for local addresses is provided by ensuring that a valid set of segment registers is used. If the V bit of the selected local segment is not asserted, an unmapped access exception occurs (refer to Chapter 8). The second level of protection is provided by the PR bits. If the PIM processor mode (supervisor or user) and access type (read or write) are not allowed by the PR bit setting of the selected segment, an invalid access exception occurs (refer to Chapter 8). The final level of protection for local addresses is provided with bounds checking. The limit value of the specified segment is used to inspect bits in the virtual address offset to ensure that the offset has not exceeded the segment size. If the segment size is exceeded, an unmapped access exception occurs (refer to Chapter 8). Assuming the limit value has been set according to the Implications section at the end of this chapter, an equation specifying the exception condition E is:

$$E = (va_8 \wedge \overline{\text{limit[index]}_8}) \vee (va_9 \wedge \overline{\text{limit[index]}_9}) \vee \ldots \vee (va_{23} \wedge \overline{\text{limit[index]}_{23}})$$

Although the conditions for address translation exceptions for global virtual addresses are similar to that of local addresses, the mechanism is quite different due to the fully associative nature of the global segment hardware. Basically, if one of the four sets of global segment registers does not "match" an attempted global address access, an exception occurs. A successful match occurs when a set of segment registers is valid, the PR bit setting allows the access type being attempted, and the address range specified by the global virtual base and limit encompasses the global address of the operation. An equation specifying the range match condition RM, where va is the virtual address and base is the contents of the global virtual base register, is:

$$\overline{RM} = (\overline{\text{limit}_0} \wedge (va_0 \oplus \text{base}_0)) \vee (\overline{\text{limit}_1} \wedge (va_1 \oplus \text{base}_1)) \vee \ldots \vee (\overline{\text{limit}_{23}} \wedge (va_{23} \oplus \text{base}_{23}))$$

An unmapped access exception is triggered if there is no valid set of registers that pass the range match test. If there is a valid set of registers that passes the range match test, but the PR bits for that segment do not allow the attempted access, an invalid access exception occurs (refer to Chapter 8).

Address Translation Unit Instructions

The primary instruction supported by the DIVA instruction set which affects address translation operation is the MTATR (move to address translation register) instruction. The destination field of this instruction can be set to specify any local base register, local protection register, global physical base register, global limit register, or global physical base register. Since the contents of a GPR is the data source for an MTATR instruction, each of these address translation unit registers is defined to be 32 bits wide, although implementations may truncate some segment registers to optimize for the actual amount of physical memory present. Furthermore, each limit register is a concatenation of a limit value, a valid bit, and the two PR bits. The MTPR instruction is also used to enable/disable address translation by writing to the appropriate bit of the PSW register.

Implications

There are a number of stipulations implied for the address translation mechanisms described in this chapter to operate correctly. First, every segment size must be a power of 2, and the base address for each segment must be aligned to a value that is a multiple of the segment size. Also, the limit value must be set so that simple logic functions can be used for translation and protection checking. For example, a segment size of 2^n should have a limit register value that is $(2^n - 1)$. Finally, the virtual to physical translation for code segments must not affect the 12 least significant bits so that instruction cache look-ups can proceed concurrently with translation. While stipulating that code segment base addresses must be some multiple of 4Kbtyes is sufficient, it is not necessary, and less strict policies can be used to ensure the requirement is met.

The exception portion of the architecture assumes that instruction and data address translations are independent. Thus, the PSW contains two address translation enable bits (one for instruction addresses and one for data addresses). Likewise, the exception source word contains separate status bits for instruction and data translation exceptions (refer to Chapter 8). There are also implications for better performance. For example, to allow address translation for both instruction fetches and data fetches to proceed concurrently, the address translation hardware must be dual-ported.

Appendix C: HiDISC Final Report

HiDISC: A Decoupled Architecture for Applications in Data Intensive Computing

Drs. Alvin Despain and Jean-Luc Gaudiot

Final Report

Abstract

The ever growing speed gap between processor and main memory has been a major performance bottleneck of modern computer systems. As a result, today's data intensive applications suffer from frequent cache misses and lose many CPU cycles due to pipeline stalling. Although traditional prefetching methods reduce cache misses considerably, most of them strongly depend on the access pattern being predicted and fail when faced with irregular memory access patterns with low locality.

This report presents our design and performance evaluation of a novel, high-performance decoupled architecture called HiDISC (Hierarchical Decoupled Instruction Stream Computer). HiDISC provides low memory access latency by introducing enhanced data prefetching techniques at both hardware and software levels. Three dedicated processors for each level of the memory hierarchy act in concert to mask the memory latency.

As required by the DARPA Data Intensive program, we used as our performance evaluation benchmarks the Data-intensive Systems Benchmark Suite and the DIS Stressmark suite. The simulation results for both benchmarks show a distinct advantage of the HiDISC system over current prevailing superscalar architectures.

Table of Contents

List of Figures

List of Tables

1. Introduction

The speed mismatch between processor and main memory has been a major performance bottleneck in modern processor architectures. Processor speed has been improving at a rate of 60% per year during the last decade. Conversely, access latency to main memory has been improving at less than 10% per year [24]. This speed mismatch – the Memory Wall problem - results in considerable cost in terms of cache misses and severely degrades processor performance. The problem becomes even more acute when faced with highly data intensive applications. Indeed, these applications are becoming more prevalent. By definition, they have a higher memory access/computation ratio than "conventional" applications. Moreover, the access pattern tends to be more irregular. As a result, the penalty caused by cache misses is becoming even more serious. This means that the architect must either reduce pipeline stalling upon cache misses or reduce the number of those cache misses (incidentally, this latter objective is the main goal of the HiDISC project).

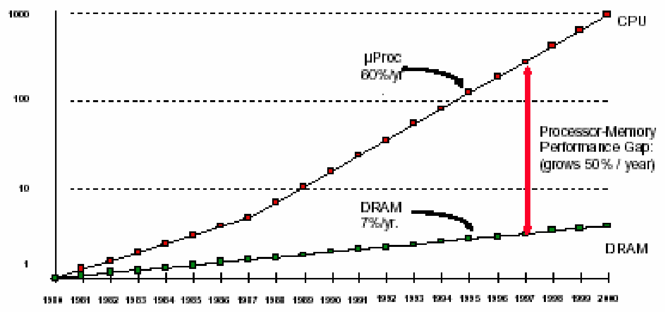


Figure 1: The speed mismatch between CPU cycle and DRAM speed

Reaching higher Instruction-Level Parallelism (ILP) through multiple instruction issue and out-of-order execution has been an essential part of modern processor design for many years. Moreover, sophisticated branch prediction and speculative execution techniques provide more opportunities for the discovery of independent instructions across basic blocks [31]. Various approaches using Thread-Level Parallelism (TLP) have also been introduced to deliver more ILP. During the last decade, superscalar and very long instruction word (VLIW) architectures have played an important role in ILP research. Although both models are designed to deliver higher levels of parallelism through multiple instruction issue, the ever increasing memory access latency has become a major obstacle to the exploitation of higher degrees of ILP.

To solve the *memory wall* problem, current high performance processors are designed with large amounts of integrated on-chip cache. However, this large cache strategy works efficiently only for applications which exhibit sufficient temporal or spatial locality. Newer applications such as multi-media processing, database, embedded processor, automatic target recognition, and any other data intensive programs exhibit irregular memory access patterns [15] and result in considerable numbers of cache misses which cause significant performance degradation.

To reduce the occurrence of cache misses, various prefetching methods have been developed. Prefetching is a mechanism by which data is fetched from memory to cache before it is even requested by the CPU. It can be implemented either in hardware or in software. Hardware prefetching [6] dynamically adapts to the runtime memory access behavior and decides the next cache block to prefetch. Software prefetching [20] usually inserts the prefetching instructions inside the code. Although previous prefetching research considerably contributed to improvements in cache performance, prefetching techniques still suffer from irregular memory access patterns. Indeed, typical prefetching strategies strongly depend on the predictability of the future data addresses. This is very difficult to predict when the access patterns are random [19]. Moreover, many current applications use sophisticated data structures with pointers which dramatically lower the regularity of memory accesses.

The Data Intensive Systems Benchmark Suite and the DIS Stressmark Suite are used in this project as our performance evaluation benchmarks. Both benchmarks are provided by Atlantic Aerospace Electronics Corporation [38][39] and supported by the Data Intensive Systems project of the DARPA Information Technology Office. Stressmark includes seven small data intensive benchmarks. Conversely, the DIS benchmarks consist of five codes more realistic than Stressmark. The five benchmarks can be categorized into three groups:

1. The Model based image generation group has two benchmarks – Method of Moments and Simulated SAR Ray Tracing.
2. The Target detection includes Image Understanding and Multidimensional Fourier Transform.
3. The Data Management benchmark

2. Method, Assumptions, and Procedures

In order to counter the inherently low locality in Data Intensive applications, our design philosophy is to emphasize the importance of memory-related circuitry and even employ two dedicated processors to respectively manage the memory hierarchy and prefect the data stream.

2.1 The HiDISC System

Access/Execute decoupled architectures have been developed as alternate processor architectures which exploit the parallelism between data access operations and "normal" computation. Concurrency is achieved by separating the original, single instruction stream into two streams based on the functionality of instructions. Asynchronous operation of the streams provides for a certain distance between the streams and makes data prefetching possible. The HiDISC architecture is an enhanced variation of conventional decoupled architectures.

Decoupled architectures (also called Access/Execute architectures) deliver higher degrees of Instruction-Level Parallelism by separating the sequential code into two instruction streams - *Access Stream* and *Execute Stream* - based on memory access functionality. Each stream runs almost independently of the other. The model was originally developed to tolerate long memory latencies: hopefully, the Access Stream will run ahead of the Execute Stream in an asynchronous manner, thereby allowing timely prefetching. It should be noted at this point that an extremely important parameter will be the "distance" between the instruction currently producing a data element in the Access Stream and the instruction which uses it in the Execute Stream. This is also called the *slip distance*, and it will be shown how it is a measure of tolerance to high memory latencies. Communication is achieved via a set of FIFO queues (they are architectural queues between the two processors to guarantee the correctness of program flow).

Our HiDISC (Hierarchical Decoupled Instruction Stream Computer) architecture is a variation of the traditional decoupled architecture model. In addition to the two processors of the original design, the HiDISC comprises one more processor for data prefetching [6][8] (Figure 2). A dedicated processor for each level of the memory hierarchy timely supplies the necessary data for the above processor. Thus, three individual processors are combined in this high-performance decoupled architecture. They are used respectively for computing, memory access, and cache management:

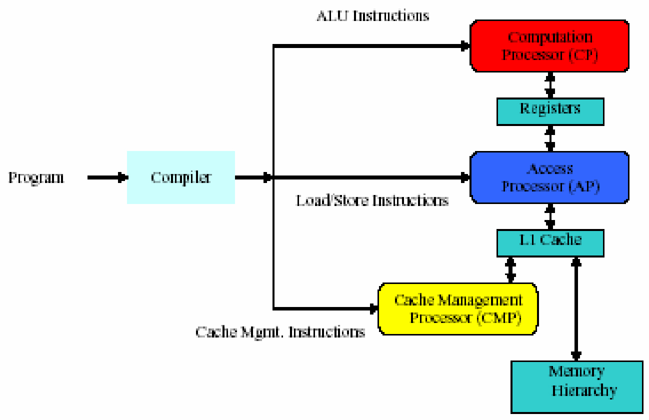


Figure 2: The HiDISC System

- Computation Processor (CP): executes all primary computations except for memory access instructions.

- Access Processor (AP): performs basic memory access operations such as loads and stores. It is responsible for passing data from the cache to the CP.

- Cache Management Processor (CMP): keeps the cache supplied with data which will be soon used by the AP and reduces the cache misses, which would otherwise severely degrade the data preloading capability of the AP.

By allocating additional processors to each level of the memory hierarchy, the overhead of generating addresses, accessing memory, and prefetching is removed from the task of the CP: the processors are decoupled and work relatively independently of one another.

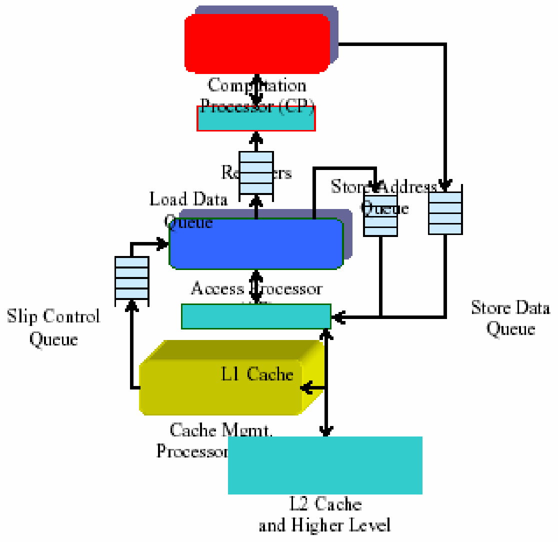
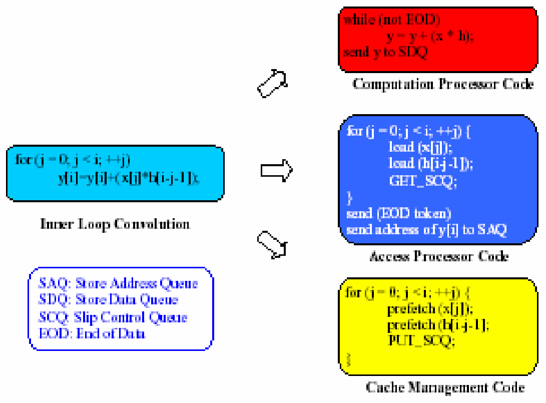


Figure 3: Inside the HiDISC architecture

Now, our compiler must appropriately form three streams from the original program: the computing stream, the memory access stream, and the cache management stream are created by the HiDISC compiler and stored into the program memory of each of the processors. As an example, Figure 4 shows the stream separation for the inner loop of the discrete convolution algorithm.

The control flow instructions are executed by the AP. Incidentally, it should be noted that additional instructions are required in order to facilitate the synchronization between the processors. Also, the AP and the CP use specially designed tokens to ensure correct control flow: for instance, when the AP terminates a loop operation, it simply deposits the End-Of-Data (EOD) token into the load data queue. When the CP sees an EOD token in the load data queue, it exits the loop.



```
for (j = 0; j < i; ++j)
        y[i]=y[i]+(x[j]*h[i-j-1]);
```

```
while (not EOD)
        y = y + (x * h);
send y to SDQ
```

```
for (j = 0; j < i; ++j) {
        load (x[j]);
        load (h[i-j-1]);
        GET_SCQ;
}
send (EOD token)
send address of y[i] to SAQ
```

```
for (j = 0; j < i; ++j) {
        prefetch (x[j]);
        prefetch (h[i-j-1];
        PUT_SCQ;
}
```

Figure 4: Discrete Convolution as processed by the HiDISC Compiler

2.2 Experimental Environment

In order to evaluate the performance of our proposed architecture, we have designed a simulator for our HiDISC architecture. It is based on the SimpleScalar 3.0 tool set [5] and it is an execution-based simulator which describes the architecture at a level as low as the pipeline states in order to accurately calculate the various architectural delays.

Figure 5 shows a high-level block diagram of the simulation procedure. Each benchmark program follows the two steps described. The first step consists in compiling the target benchmark using the HiDISC compiler which we have designed, while the second step is the simulation and performance evaluation phase.

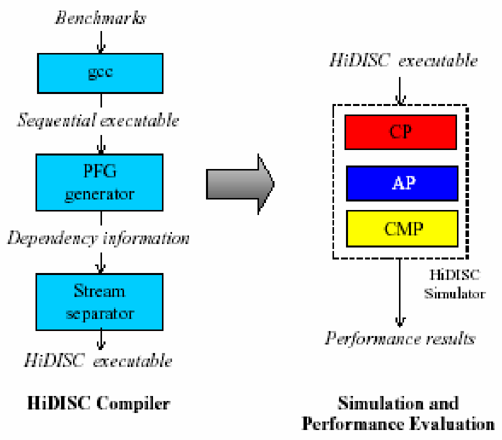


Figure 5: Simulation Procedure

2.3 Operation of the HiDISC Compiler

The HiDISC executables are produced by our HiDISC compiler. The core operation of the HiDISC compiler is stream separation. Stream separation is achieved by backward chasing of load/store instructions based on the register dependencies. This means that, in order to obtain the register dependencies between instructions, a Program Flow Graph (PFG) must be derived. Indeed, the PFG generator and the stream separator are two major operations of the HiDISC compiler. The PFG generator and the stream separator are adopted after some modifications from the SimpleScalar 3.0 tool set and integrated in the HiDISC compiler.

Figure 6 depicts the overall HiDISC compiler. Its detailed operation is described below.

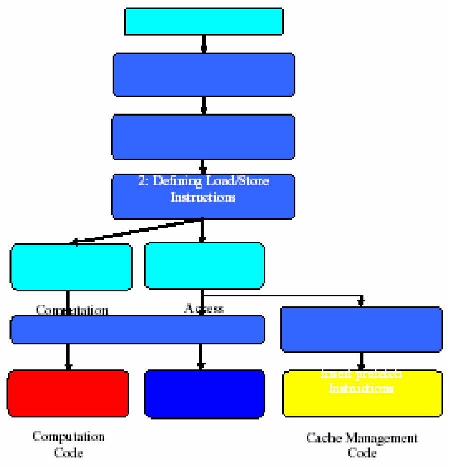

2: Defining Load/Store Instructions
Computation
Access
Insert prefetch Instructions
Computation Code
Cache Management Code

Figure 6: Overall HiDISC stream separator

The input to the HiDISC compiler is a conventional sequential binary code. The first step (1: *Deriving the Program Flow Graph* in Figure 6) consists in uncovering the data dependencies between the instructions. Each instruction is analyzed so as to determine which its parent instructions are. This determination is based on the source register names. Whenever the stream separator meets any load/store instruction in step 2 (2: *Defining Load/Store Instructions*), it defines the instruction as the Access Stream (AS) and chases backward to discover its parents instruction. The next step (3: *Instruction Chasing for Backward Slice*) is designed to handle the backward chasing of pointers. The instructions which are chased according to the data dependencies are called the *backward slice* of the instruction from which we started.

Since the Access Stream should contain all access-related instructions, as well as the address calculation and index generation instructions, the backward slice should be included in the Access Stream as well. It should be noted that all the control-related instructions are also part of the Access Stream. The instructions which should belong to the control flow are determined by a similar method. After defining all the Access Stream, the remaining instructions are, by default, classified as belonging to the Computation Stream (CS).

In addition to the stream separation, appropriate communication instructions should be placed in each stream in order to synchronize the two streams. Finding what the required communications are is also based on the register dependencies between the streams. Essentially, when it is determined that some required source data is produced by the other stream, some kind of communication should take place. For instance, when a memory load (inside the Access Stream) produces a result which should be used by the Computation Stream, a Load instruction would be inserted in the Access Stream. It would send the data to the Load Data Queue (LDQ). However, if the result of that load was not needed by the Computation Stream, then obviously no such insertion would be needed. Similarly, when the result produced by the Computation Stream is used by a store instruction (inside the Access Stream), it should be sent to the store data queue (SDQ) by inserting an appropriate communication instruction.

The backward chasing starts whenever we encounter new load/store instruction. The backward chasing ends when the procedure meets any instruction which already has been defined as the Access Stream. The parent instructions of any defined Access Stream have already been chased.

After separating the Access Stream and the Computation Stream, the CMP stream is constructed by modifying the Access Stream. The instruction stream for the CMP is indeed quite similar to the Access Stream. Only the load instructions are replaced with the prefetch instructions for the CMP stream.

Figure 7 shows an example of the operation of the backward slicing mechanism in the HiDISC compiler. The assembly code input to the HiDISC compiler is the PISA (Portable Instruction Set Architecture) which is the instruction set of the SimpleScalar simulator [5]. We have selected for this example the inner product of Livermore loop (lll1). The PISA code is compiled into SimpleScalar binary by first using a version of *gcc* which targets SimpleScalar.

Initially, each memory access instruction is defined as belonging to the Access in the Stream. For example, the ***ld*** instruction in the fifth line (pointed to by an arrow ① left margin) can be immediately determined as belonging to the AS. Moreover, every parent instruction of a memory access instruction should be identified. In the example, the ***addu*** instruction in the fourth line (pointed to by an arrow ②) - due to the register $9

- and the ***mul*** instruction in the second line (pointed to by an arrow ③) -due to the register $25 - are also chased and marked as belonging to the AS. Likewise, other instructions are examined based on the above approach. The instructions in the shaded box in Figure 7 belong to the Access Stream.

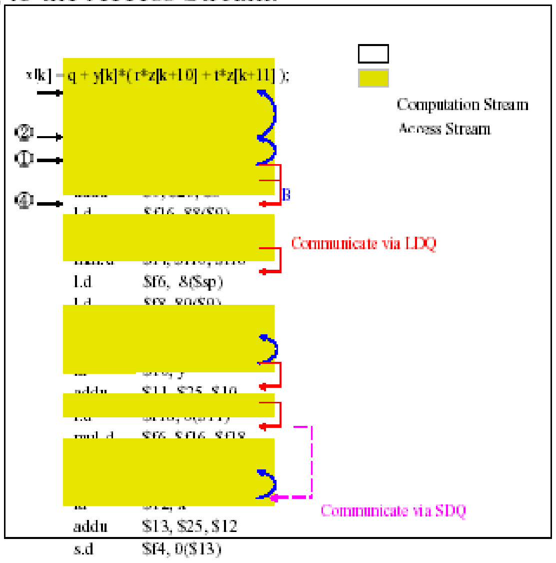
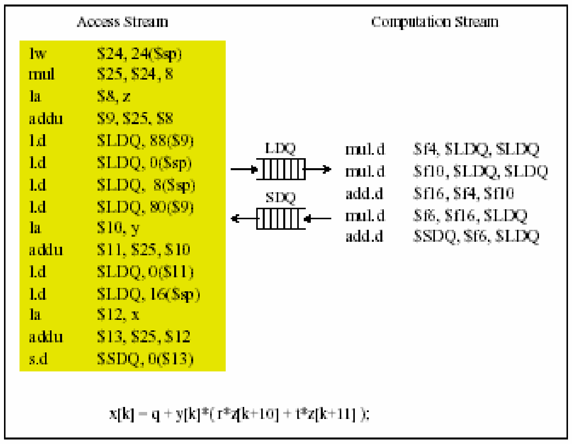


Figure 7: Backward chasing of load/store instructions

After defining each stream, the communication instructions should be inserted. The red lines in Figure 7 (forward arrows, solid lines) show the necessary communications from the AS to CS. For example, the ***mul.d*** instruction (which is marked as being inside the Computation Stream, pointed to by arrow ④) in the seventh line requires data from the other instruction stream (The Access Stream). Therefore, both ***l.d*** instructions in the fifth and sixth line need to send data to LDQ. Likewise, the purple line at the bottom (forward arrow, dotted line) also shows the communication from the CS to the AS via the Store Data Queue (SDQ).

Figure 8 shows the complete separation of the two streams and insertion of the communication instructions.



```
lw      $24, 24($sp)
mul     $25, $24, 8
la      $8, z
addu    $9, $25, $8
l.d     $LDQ, 88($9)
l.d     $LDQ, 0($sp)
l.d     $LDQ,  8($sp)
l.d     $LDQ, 80($9)
la      $10, y
addu    $11, $25, $10
l.d     $LDQ, 0($11)
l.d     $LDQ, 16($sp)
la      $12, x
addu    $13, $25, $12
s.d     $SDQ, 0($13)
```

```
mul.d   $f4, $LDQ, $LDQ
mul.d   $f10, $LDQ, $LDQ
add.d   $f16, $f4, $f10
mul.d   $f6, $f16, $LDQ
add.d   $SDQ, $f6, $LDQ
```

```
x[k] = q + y[k]*( r*z[k+10] + t*z[k+11] );
```

Figure 8: Separation of sequential code

2.4 Benchmark Description

Applications causing large amounts of data traffic are often referred to as data-intensive applications as opposed to computation intensive applications. Inherently, data-intensive applications use the majority of the resources (time and hardware) to transport data between the CPU and the main memory. The tendency for a higher number of applications to become data intensive has become quite pronounced in a variety of environments [39]. Indeed, many applications such as Automatic Target Recognition (ATR) and database management show non-contiguous memory access patterns and currently result in idle processors due to data starvation. These applications are more stream-based and result in more cache misses due to lack of locality.

Frequent use of memory dereferencing and pointer chasing also creates an enhanced pressure on the memory system. Pointer-based linked data structures such as lists and trees are used in many current applications. For one thing, the increasing popularity of Object Orient Programming correspondingly increases the underlying use of pointers. Due to the serial natural of pointer processing, memory accesses become a severe performance bottleneck of existing computer systems. Flexible, dynamic construction allows linked structures to grow large and difficult to cache. At the same time, linked data structures are traversed in a way that prevents individual accesses from being overlapped since they are strictly dependent upon one another [26].

The applications for which our HiDISC is designed are obviously data intensive programs, the performance of which is strongly affected by the memory latency. As required by the Data Intensive Systems project of the DARPA Information Technology Office, we used for our benchmarks the Data-intensive Systems Benchmark Suite [39] and DIS Stressmark Suite [38] provided by the Atlantic Aerospace Electronics Corporation. Both of the benchmarks are targeting data intensive applications. The DIS benchmarks are five benchmarks codes, which are more realistic and larger than Stressmark. Stressmark includes seven small data intensive benchmarks, which extracts and shows the kernel operation of data intensive programs.

Due to problems with the input data file, the Image Understanding benchmark cannot be executed. Also, since the Corner-Turn benchmark among seven Stressmarks is not provided with the source code, we only simulated the other six Stressmarks.

Table 1 shows the characteristics of each of the benchmarks simulated.

Table 1. Simulated Benchmark Description

Benchmark	Name	Problem	Characteristic
DIS benchmarks	Method of Moments	Computing the electromagnetic scattering from complex objects	Containing computational complexity and requesting high memory speed
	Multidimensional Fourier Transform	Fourier Transform	Wide range of application usage
	Data Management	Traditional DBMS processing	Index algorithms and ad hoc query processing
	SAR Ray Tracing	SAR image simulation	Utilizes Image-domain approach

Stressmark	Pointer	Pointer following	Small blocks at unpredictable locations. Can be parallelized
	Update	Pointer following with memory update	Small blocks at unpredictable location
	Matrix	Conjugate gradient simultaneous equation solver	Dependent on matrix representation Likely to be irregular or mixed, with mixed levels of reuse
	Neighborhood	Calculate image texture measures by finding sum and difference histograms	Regular access to pairs of words at arbitrary distances
	Field	Collect statistics on large field of words	Regular, with little re-use
	Transitive Closure	Find all-pairs-shortest-path solution for a directed graph	Dependent on matrix representation, but requires reads and writes to different matrices concurrently

3. Results and Discussion

We used our architectural simulator of the HiDISC machine to evaluate the performance of all the benchmarks except two.

3.1 Simulation Parameters

In our benchmark simulations, we assumed the architectural parameters outlined in Table 2. The baseline architecture for the comparison is a 4-way superscalar architecture, which is implemented as *sim-outorder* in the SimpleScalar 3.0 tool set. In both cases, the memory access latency has been made to vary between 20 and 120 CPU cycles. The baseline superscalar architecture supports out-of-order issue with 16 register update units and 8 load store queues.

Table 2: Simulation Parameters

Branch predict mode	Bimodal
Branch table size	2048
Issue width	4
Window size for superscalar	RUU: 16 LSQ: 8
Slip distance for AP/CP	50
Data L1 cache configuration	128 sets, 32 block, 4 -way set associative , LRU
Data L1 cache latency	1
Unified L2 cache configuration	1024 sets, 64 block, 4 - way set associative, LRU
Unified L2 cache latency	6
Integer functional unit	ALU(x 4), MUL/DIV
Floating point functional unit	ALU(x 4), MUL/DIV
Number of memory port	2

3.2 Benchmarks Results

Figure 9 and Figure 10 show the simulation results of the DIS Benchmark Suite and the Stressmark Suite. The performance results of the HiDISC architecture are compared to a 4-way superscalar architecture. The far left bar indicates the performance results of the superscalar architectures. The second bar expresses the performance results of the basic HiDISC architecture. The remaining two bars show the possible performance results when enhancing the prefetching capability of the CMP processor. The numbers in parenthesis express the cache miss reduction ratio. The enhancements will be explained in more detail in the next section.

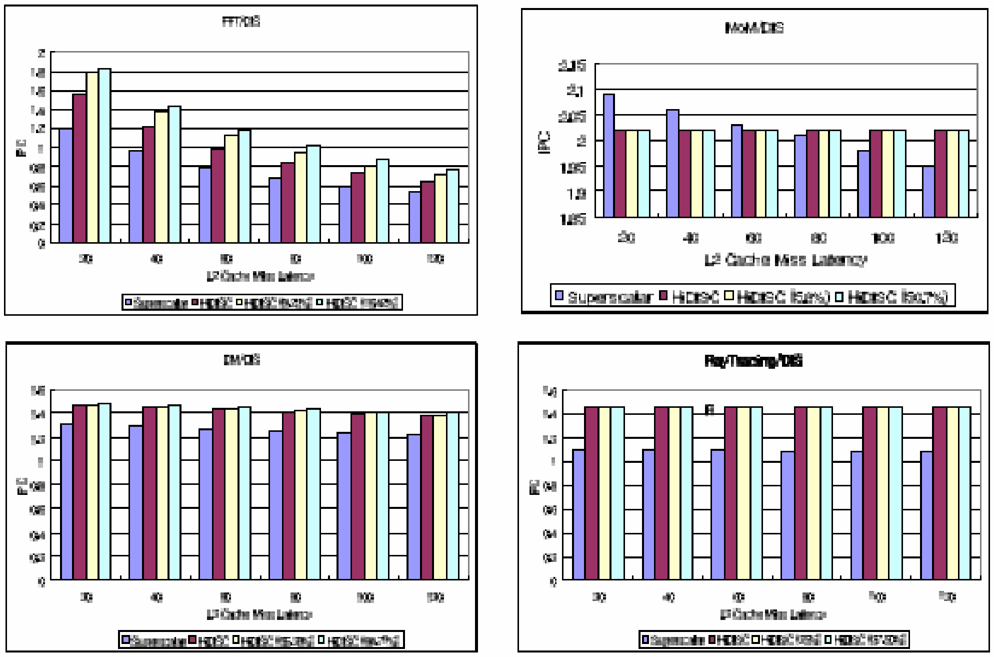


Figure 9: DIS benchmark performance results

All four DIS benchmarks show better performance than the baseline superscalar architecture. However, with the Stressmark, only two of the six cases show better performance for the HiDISC. The remaining four benchmarks do not show any performance advantage for the HiDISC architecture.

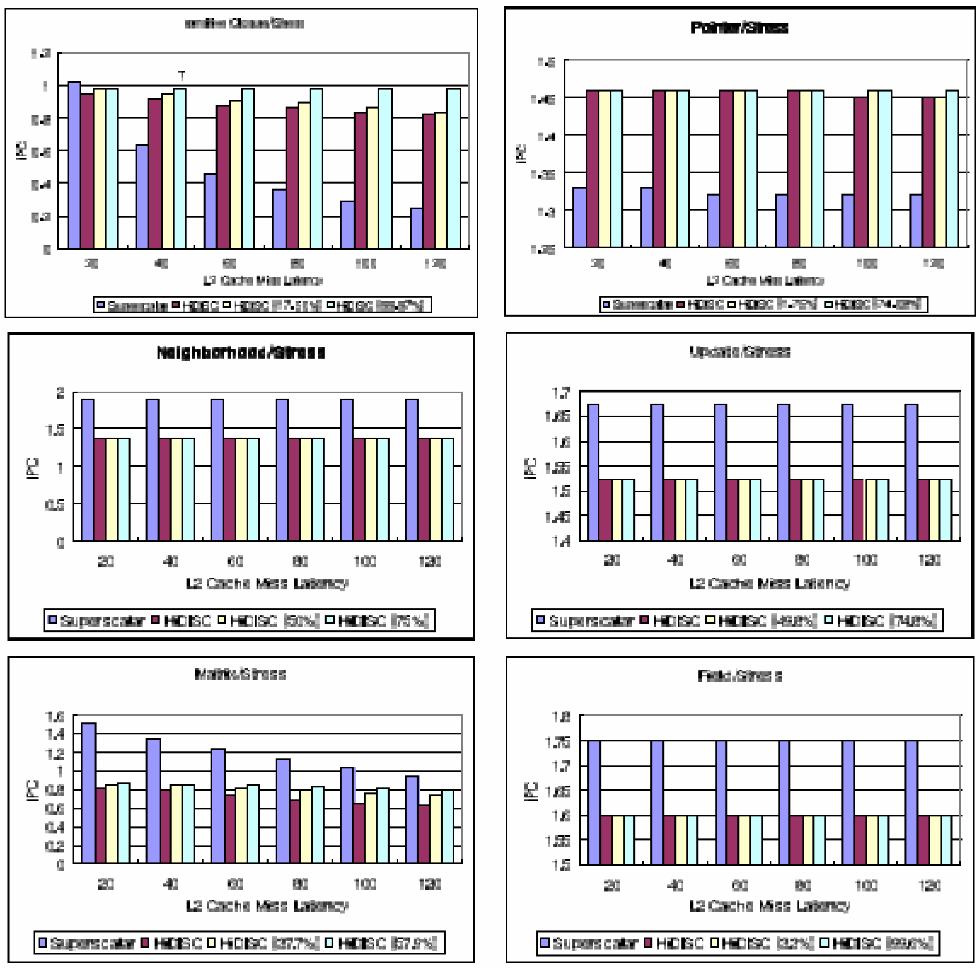


Figure 10: Stressmark performance results

3.3 Discussion

The simulation results show that the HiDISC system performs quite well in general with the DIS benchmarks. This is because the DIS benchmarks contain many long latency floating-point operations which can effectively hide any long memory latency. In other words, the amount of computation code and that of memory access code are well balanced in the DIS benchmark Suite. Conversely, the size of the Stressmark computation code is much smaller than that of the memory access code. It is one of the main reasons for the somewhat weaker performance results observed in the case of the Stressmark Suite.

Four DIS Benchmarks Results (Figure 9)

Four DIS benchmarks outperform the baseline superscalar architecture particularly with higher memory latencies. More particularly, the Method of Moments is quite robust when faced with longer memory latencies. It contains enough computation code which can hide the longer access latency. Also, the dependencies between the Computation Stream and the Access Stream are comparatively not heavy and provide enough slip distance to hide any long memory latency.

In the case of the Multidimensional Fast Fourier Transform, HiDISC also outperforms the superscalar architecture. However, the results show a weaker performance for long memory latencies even with the HiDISC model. Indeed, the synchronization between the AS and the CS limits the possible slip distance between the two streams. It is due to the data dependencies between the two streams: frequent data dependencies between the Access Stream and the Computation Stream cause *loss of decoupling* events. Usually, it is the CS which has to wait for a data element to be produced by the AS (although the converse is also sometimes true). When this happens, the slip distance between the two processors is reduced significantly, one processor must wait for the other and any advantage is negated since there is no more parallelism between the two processors.

The Data Management and the Ray-Tracing benchmarks are not affected by longer memory latencies in either case. It should be noted that the working set for the Data Management benchmark fits quite well in the cache. As should be expected, a program with a small working set is not a good candidate for a prefetching architecture such as the HiDISC. Conversely, due to the prefetching of the CMP, FFT exhibits better performance.

Six Stressmarks Results (Figure 10)

Generally, the Stressmark codes are too small and contain too many operations which are concerned only with data access. Therefore, the amount of computation code to hide data access is not sufficient. The HiDISC produces weaker results in four Stressmarks – Update, Field, Matrix and Neighborhood - out of the six Stressmarks. However, the remaining two Stressmarks - the Pointer and the Transitive Closure – advantageously exploit the characteristics of our architecture.

Besides the unbalanced computation and access code ratio, frequent *loss of decoupling* is another main reason for the weak performance we observe in several Stressmarks. Indeed, four Stressmarks except Pointer and Transitive Closure contain too much data dependencies and frequent synchronizations between two streams.

However, in the Pointer Stressmark case, pointer chasing can be executed far ahead since it does not require the computation results from the CP. The Transitive Closure benchmark also produces good results because not much in the AP depends on the results of the CP. In both cases, the Access Stream can run far ahead of the Computation Stream: a sufficient slip distance is guaranteed in both benchmarks.

The slip distance is truly inherent to the instruction mix pattern of the application: if the Access Stream does not depend much on results from the Computation Stream, the Access Stream can run earlier and maintain a high slip distance. Pointer and Transitive Closure exhibit good performance for the same reasons. In addition to the possible slip-distance between the two streams, the Stressmark results suggest that applications which are ideal for the HiDISC would be well balanced in terms of the ratio of computation operations over memory operations.

Finally, the working set for the Stressmark is quite small and the baseline superscalar architecture does not suffer from many cache misses. Three Stressmarks (Update, Field and Neighborhood) cannot improve even with the prefeching of the CMP.

Although some of the benchmarks show weak performance, the fact that the Pointer Stressmark and the Transitive Closure Stressmark perform better that the baseline superscalar architecture is quite encouraging and suggests the type of the candidate applications for the HiDISC architecture.

4. Conclusions

Current high-level programming languages and all supporting compilers are based on an underlying sequential programming behavior. This is confirmed at the lower level where the instruction set of modern microprocessors are based on a sequential model. However, in order to exploit some parallelism at the instruction level, manufacturers of current prevailing high performance processors have considerable changed the processor internal structure. Also, several features of dataflow models have found their way in modern processor architectures and compiler technologies such as register renaming and dynamic scheduling [17]. Decoupled architecture is one such technique which promises to bring improvement to the performance.

The effectiveness of the HiDISC decoupled architecture has been demonstrated here with data intensive applications. It has been eloquently shown that the proposed prefetching method provides better ILP compared to conventional superscalar architectures. However, the possible *loss of decoupling*, which is inherited from the sequential behavior of the programs, stalls the processors and drops utilization in some cases. The results also point to some future modifications of the current CMP for effective prefetching.

Clearly, the HiDISC architecture, as designed, will shine when executing data intensive applications because they contain enough computation to hide long memory latencies. In addition to that, the slip distance is another important factor which determines overall performance. Too many data dependencies of the access processor on the computation processor prevent a sufficient slip distance from developing. Therefore, stream-like applications are favored for the HiDISC system.

5. Recommendations

Based upon these performance results, we propose some improvements to the basic HiDISC architectures in order to make it fit a wider variety of applications.

5.1 Future Enhancements to the HiDISC

Although the independent management of the memory hierarchy provides an opportunity to implement novel prefetching techniques, the HiDISC architecture suffers from two significant weaknesses. First, the frequent synchronizations between the AP and the CP cause stalling of the processors and result in low utilization. Second, the CMP code is essentially not different from the AP code. Therefore, all the load instructions are forced to run on the CMP as prefetching. However, not every prefetching by the CMP is necessary and helpful. Necessary enhancements regarding the above two problems will follow.

The frequent synchronizations cause *loss of decupling* and prevent timely prefetching. Therefore, each processor of the HiDISC loses many CPU cycles to wait until the necessary data arrives. To solve this problem, Simultaneous MultiThreading (*SMT*) should be added to the HiDISC architecture. SMT will raise the utilization by running multiple threads simultaneously. In other words, in a multithreaded HiDISC system, SMT would raise the utilization of the processors, while decoupling would reduce the memory latency [22][23].

The second modification is related to the current CMP design. The main motivation for the existence of the CMP processor is to reduce the cache miss rate by the Access Processor by timely prefetching. Therefore, the CMP should run ahead of the AP, just like the AP runs ahead of the CP. However, in the basic HiDISC design, the instruction stream for the CMP is quite similar to the Access Stream, which is a significant limitation as far as the effectiveness of the prefetching is concerned. Our original design executes every load instruction on CMP. However, if the cache line already resides in cache, those prefetches become redundant operations.

Only future probable miss instructions can benefit from the prefetches by the CMP. However, the current CMP is too heavy and involves performing too many redundant operations. Hence, in order to prefetch more efficiently into the cache, we must develop better methods so that we execute only probable miss instructions.

We define Cache Miss Access Slice (CMAS), which is a part of the Access Stream, consisting of the probable cache miss instruction and its parent instructions. The probable cache miss instructions can be found using the cache access profile [27][28]. The CMAS is executed on existing CMP in a multithreaded manner. Indeed, the CMP is an auxiliary processor for speculative execution of probable cache miss instructions.

5.2 Flexi-DISC

One of the most striking characteristics of the HiDISC architecture is its inherent flexibility and how it yields highly efficient execution of a large variety of loop-based programs with little or no temporal locality. This fundamental feature is further extended in the proposed Flexi-DISC. This new architecture will be targeted to a wide variety of more complex, numerical and non-numerical applications (such as Automatic Target Recognition).

While the original HiDISC is centered around three processors with well defined roles, the Flexi-DISC maintains the three roles of the CP, the AP, and the CMP at the kernel of its fundamental machine model but elevates it to a more sophisticated concept: the two highest levels (Access and Cache Management) are still handling the transfer of data between the memory system and the Computation level while the third level remains in charge of the computation per se. This can be represented as the three concentric rings on Figure 11: the Computation Kernel (CK), the Low-level Cache Access Ring (LCAR), and the Memory Interface Ring (MIR).

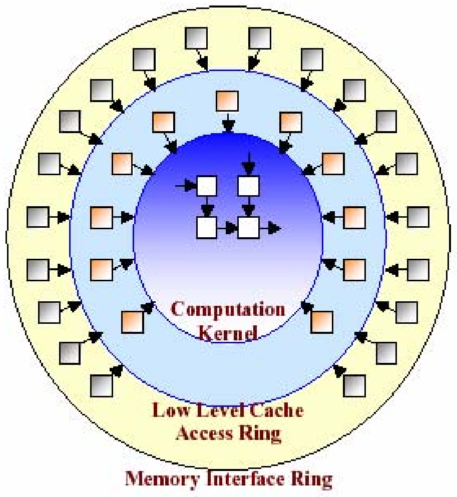


Figure 11: The three-Ring Flexi-DISC Architecture

The fundamental observation which leads to this partitioning comes from the fact that the types of applications (Memory Intensive) we have been targeting are both varied in nature and also inherently highly dynamic at execution time. This may mean that memory access patterns could range from, say, single use of any data element (no temporal locality), to multiple reuses (high temporal locality). Consequently, the bandwidth and types of pipes to and from the memory system must adapt to the changes, whether they be static or dynamic. We plan on centering the whole architecture around a highly reconfigurable Computation Kernel.

The central Computation Kernel is based on an array of simple processors which can be dynamically rearranged to meet the demands of the current application. It can even be partitioned into sub-arrays which are allocated to different portions of the application (or even to different applications as needed). Such a powerful computation kernel requires an equally powerful "pipeline" to feed it information to and from the memory system. Further, the variety of target applications makes the memory accesses unpredictable. This means that depending on the application (or even the phase of a given computation), the amount of memory traffic may fluctuate, and the prefetching mechanisms must be allowed to adapt to the situation at hand. This also means that instead of allowing a single processor for the Cache Access role and another for the Cache Management role, a *pool* of identical processing units must be made available to the two roles combined. This sharing enables a highly efficient dynamic partitioning of the resources and their run-time allocation to the two outer rings (the Low-level Cache Access Ring, and the Memory Interface Ring).

The technology developed for the HiDISC compiler can be expanded to include the rearrange ability of the machine, as well as the partitioning it will undergo in the presence of multi-headed applications.

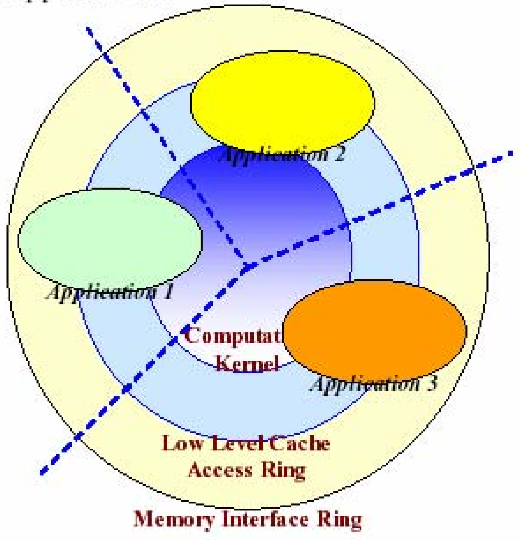


Figure 12: Multiple application sharing of the Flexi-DISC model

References

[1] Murali Annavaram, Jignesh M. Patel, Edward S. Davidson, Data Prefetching by Dependence Graph Precoumputation, *28th International Symposium on Computer* Architecture, June, 2001

[2] J.Arul, Execution Performance of the Scheduled Dataflow Architecture(SDF), International Conference on Parallel Architecture and Compilation Techniques: *MEDEA Workshop* Oct 13-15th 2000.

[3] A. Bakshi, Jean-Luc Gaudiot, Wen-Yen Lin, M. Makhija, V. K. Prasanna, Wonwoo Ro, Chulho Shin , Memory Latency: to Tolerate or to Reduce?, *The 12th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD 2000* Oct 24-27, 2000

[4] P. Bird, A. Rawsthorne, and N. Topham. The effectiveness of decoupling. In *Int. Conf. on* Supercomputing, pages 47--56, 1993

[5] Doug Burger and Todd M. Austin. The simplescalar tool set, version 2.0. *Technical report,* University of Wisconsin-Madison, Computer Science Department, 1997

[6] T.-F. Chen and J.-L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *Transactions on Computers*, 44(5):609--623, May 1995

[7] Stephen P. Crago, HiDISC: a high-performance hierarchical, decoupled computer architecture, , *Ph.D. Dissertation*, University of Southern California, 1997

[8] Stephen P. Crago, Alvin Despain, Jean-Luc Gaudiot, Manil Makhija, Wonwoo Ro, and Apoorv Srivastava, A High-Performance, Hierarchical Decoupled Architecture, *In Proceedings of MEDEA Workshop, Philadelphia, October 15*, 2000

[9] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, John P. Shen, Speculative Precomputation: Long-range Prefetching of Delinquent Loads, *28th International Symposium on Computer* Architecture, June, 2001

[10] Haitao Du, Analysis of Memory Access Behavior of DIS Stressmark Suite and Optimization, *Tech . Report*, Center for Embedded Computer Systems, University of California, Irvine

[11] S. Eggers, J. Emer, H. Levy, J. Lo, R. Stamm and D. Tullsen, Simultaneous Multithreading: A Platform for Next-generation Processors, In *IEEE Micro*, 0. 12-19, Oct. 1997

[12] M. Farrens, P. Nico and P. Ng, A Comparison of Superscalar and Decoupled Access/Execute Architectures, In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, Dec. 1993

[13] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter, and H. C. Young, PIPE: A VLSI Decoupled Architecture, In *Proc. 12th International Symposium on Computer Architecture,* pp. 20-27, June 1985

[14] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 1992

[15] S.I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor and W.A. Wulf, Access Order and Effective Bandwidth for Streams on a Direct Rambus Memory, *5th International Symposium on High-performance Computer Architecture*, 1999.

[16] G. P. Jones and N. P. Topham, A Comparison of Data Prefetching on an Access Decoupled and Superscalar Machine, In *Proc. 30th International Symposium on Microarchitecture*, Dec. 1997

[17] K.M. Kavi, J.Arul and R.Giorgi. Execution and cache performance of the Scheduled Dataflow Architecture, *Journal of Universal Computer Science, Special Issue on Multithreaded and Chip Multiprocessors*, Oct. 2000, pp 948-967, Vol. 6, No. 10.

[18] Ronny Krashinsky and Mike Sung, Decoupled Architectures for Complexity-Effective General Purpose Processors, *Tech. Report*, MIT Laboratory for Computer Science, Dec. 2000

[19] Lizy Kurian, Paul T. Hulina and Lee D. Coraor, Memory Latency Effects in Decoupled Architectures, *IEEE Transactions on Computers*, vol. 43, no. 10, Oct. 1994

[20] C.-K. Luk and T. C. Mowry. Compiler based prefetching for recursive data structures. In 7th *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996

[21] Chi-Keung Luk, Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processor, *In Proc. of International Symposium on Computer Architecture, 2001*

[22] Joan-Manuel Parcerisa and Antonio González, The Synergy of Multithreading and Access/Execute Decoupling, In Proc. *5th. Int. Symp. on High-Performance Computer Architecture (HPCA-5),* Jan. 1998

[23] Joan-Manuel Parcerisa and Antonio González, Improving Latency Tolerance of Multithreading through Decoupling, *IEEE Transactions on Computers*, October, 2001

[24] D. Patterson, et al. A Case for Intelligent DRAM: IRAM, *IEEE Micro*, April 1997

[25] Kevin Rich, Decoupled Architectures: A Thorough Analysis, Computer Science Department: *Qualifying Examination Paper*, University of California at Davis, Davis, California (May 1995)

[26] Amir Roth, Andreas Moshovos and Guri S. Sohi, Dependence Based Prefetching for Linked Data Structures. In *proc. ASPLOS-8,* October 4-7, 1998.

[27] Amir Roth and Gurindar S. Sohi, Speculative Data-Driven Multithreading, *In proc. of HPCA-7*, Jan. 2001

[28] Amir Roth, Craig B. Zilles and Gurindar S. Sohi, Speculative Miss/Execute Decoupling*. In proc. of MEDEA Workshop*, Oct. 19, 2000

[29] Jurij Silc, Borut Robic and Theo Ungerer, *Processor Architecture*, Springer, 1999

[30] J. Smith. Decoupled Access/Execute Computer Architecture. In *Proc. 9th International Symposium on Computer Architecture*, Jul. 1982

[31] Srikanth T. Srinivasan and Alvin R. Lebeck, Load latency tolerance in dynamically scheduled processors, In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, 1998.

[32] D. M. Tullsen, S. J. Eggers, and H.M. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism. *In Proc. 22nd International Symposium on Computer Architecture* , June 1995.

[33] G. Tyson, M. Farrens and A. Pleszkun, MISC: A Multiple Instruction Stream Computer, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, Dec. 1992

[34] Wm. A. Wulf, Evaluation of the WM Architecture, In *Proc. 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992

[35] Yinong Zhang, G. B. Adams III, Performance Modeling and Code Partitioning for the DS Architecture, In *Proc. 25th Annual International Symposium on Computer Architecture*, 1998

[36] Yinong Zhang, G. B. Adams III, Exploiting Instruction Level Parallelism with the DS Architecture. In *Proc. of the 1996 Int'l Conf. on Parallel Processing*, Aug. 1996

[37] Craig B. Zilles and Gurindar S. Sohi, Understanding the Backward Slices of Performance Degrading Instructions. *ISCA-2000*, June 2000.

[38] DIS Stressmark Suite, http://www.aaec.com/projectweb/dis/DIS_Stressmarks_v1_0.pdf

[39] Data-Intensive Systems Benchmarks Suite Analysis and Specification , http://www.aaec.com/projectweb/dis/

Appendix A: Compiler and Simulator Description

The compiler and the simulator are based on the SimpleScalar 3.0 tool set. The two tools have been designed by modifying *sim-outorder.c*. The first tool is *sim-pfg.c*, which takes care of the whole compiling procedure and the other one is *sim-dumas.c*, which exactly matches the HiDISC simulator. This appendix gives a detailed description of the tools.

A.1. Compiler Tool: *sim-pfg.c*

sim-pfg.c is the source code (C) for the HiDISC compiler. The main tasks of *sim-pfg.c* are: 1. Deriving the Program Flow Graph and 2. Separating the streams. The input for *sim-pfg.c* is a binary executable for SimpleScalar while the output is a binary executable for the HiDISC architecture with the separation information.

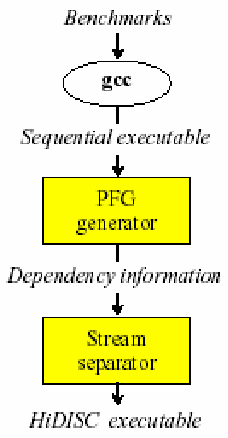


Figure 13: The HiDISC Compiler

Figure 13 shows the procedure inside the HiDISC compiler. The two boxes perform the operations mentioned earlier.

Deriving Program Flow Graph (PFG)

The Program Flow Graph delivers the data dependency information between instructions. The dataflow relationship between instructions must first be defined in order to get the backward slice of a certain target instruction. After this procedure, each access related instruction can point to the parent instructions based on the source register name. The main procedure is named *pfg_const()*. Its detailed mechanism is described in Figure 14.

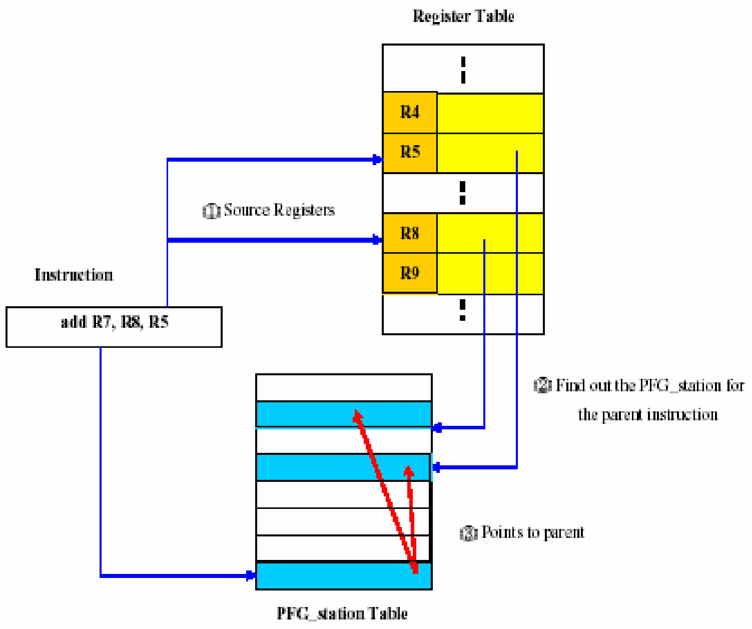


Figure 14: Deriving PFG Graph

The data structure for each instruction has been defined as *pfg_station*. After the instruction is decoded, a dedicated *pfg_station* is assigned. The first procedure consists in accessing the register table based on the source register name. (referred to as ① in Figure 14). The register table gives the pointer to the instruction (actually, the pointer to *pfg_station* of the instruction, referred to as ② in Figure 14) which last updated the source register. Finally, the decoded instruction can have the pointer for the parent instructions referred to as ③ in Figure 14.

This is how we uncover the parent instructions of a load/store instruction. Later, we can proceed with a backward chasing procedure in order to extract the backward slice based on the PFG information.

Separating Stream

The stream separation is based on the register dependencies. First, when the decoded instruction is either a load or a store instruction, it is immediately assigned to the Access Stream. After that, the backward chasing procedure is initialized (procedure named *chasing_parents( )* is called). Essentially, it is function call which is recursively applied until it reaches an instruction which has been pre-determined to belong to the Access Stream.

The PFG information from the previous step yields the pointers to the parent instructions. Therefore, the *chasing_parents( )* procedure basically returns all the pointers to the parent instructions.

After the instruction is detected as belonging to the Access Stream, the stream separation information is updated inside the binary file. Since each instruction of the SimpleScalar binary includes an additional annotation field, those extra bits can be used to carry the separation information.

A.2. Simulator: *sim-dumas.c*

The HiDISC simulator has been designed by modifying the *sim-outorder.c* module of the SimpleScalar 3.0 tool set [5]. The major modifications consist in: 1. implementing the three processors of the HiDISC and 2. implementing the communication mechanisms (queues) between those three processors. As in the original SimpleScalar simulator, the HiDISC simulator is also an execution-driven, cycle- time simulator.

To implement the three processors of the HiDISC, we basically copied three times the pipelined RISC processor of the SimpleScalar tool set and tailored each so they would correspond to the architecture of each HiDISC processor.

After the decoding stage, each processor has a corresponding *ready list*, which is the instruction stream for each processor. We implement three different functional units which are unique to each processor. Procedure *ruu_issue( )* of the *sim-outorder.c* has been copied and changed to *ruu_issup_cp( )*, *ruu_issue_ap( )*, and *ruu_issue_cmp( )*.

Each function detects each *ready list* and finds the available functional unit that is assigned to the corresponding processor.

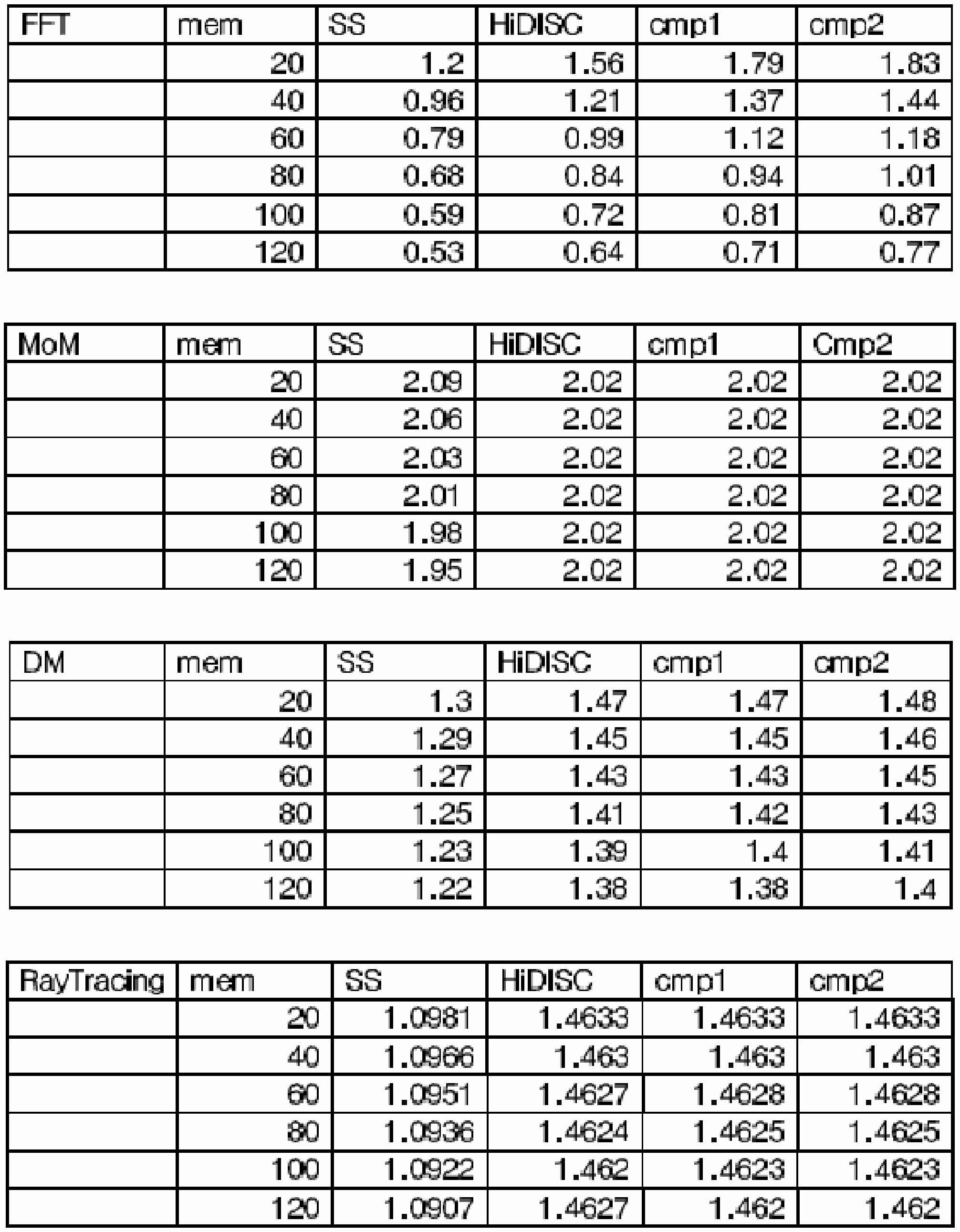
The need for communication can also be detected at the decoding stage. If an instruction requires data from the other processor, it should be blocked and it should wait until the other processor sends the data. The queue implementation is quite easily handled using the existing link operations of the SimpleScalar tool set. All the necessary source data is linked after the *ruu_dispatch( )* procedure. Therefore, the sending processor can "wake up" the waiting processor just like *ruu station* in *sim-outorder.c.*

Communications between the AP and the CMP are achieved through the data cache. Therefore, the data cache is designed and implemented to be shared and accessed by both processors.

Appendix B: Raw Performance Data

This appendix contains all the simulation results. The column denoted as *mem* corresponds to the various memory latencies. The column marked SS contains the performance of the base line superscalar architectures. The fourth column denoted as HiDISC contains the performance results of the HiDISC architecture without the CMP processor. The remaining two contain the performance results with the CMP enhanced pre-fetching algorithms. The performance measures are all in IPC (instructions per clock).

< DIS benchmarks >



FFT	mem	SS	HiDISC	cmp1	cmp2
	20	1.2	1.56	1.79	1.83
	40	0.96	1.21	1.37	1.44
	60	0.79	0.99	1.12	1.18
	80	0.68	0.84	0.94	1.01
	100	0.59	0.72	0.81	0.87
	120	0.53	0.64	0.71	0.77

MoM	mem	SS	HiDISC	cmp1	Cmp2
	20	2.09	2.02	2.02	2.02
	40	2.06	2.02	2.02	2.02
	60	2.03	2.02	2.02	2.02
	80	2.01	2.02	2.02	2.02
	100	1.98	2.02	2.02	2.02
	120	1.95	2.02	2.02	2.02

DM	mem	SS	HiDISC	cmp1	cmp2
	20	1.3	1.47	1.47	1.48
	40	1.29	1.45	1.45	1.46
	60	1.27	1.43	1.43	1.45
	80	1.25	1.41	1.42	1.43
	100	1.23	1.39	1.4	1.41
	120	1.22	1.38	1.38	1.4

RayTracing	mem	SS	HiDISC	cmp1	cmp2
	20	1.0981	1.4633	1.4633	1.4633
	40	1.0966	1.463	1.463	1.463
	60	1.0951	1.4627	1.4628	1.4628
	80	1.0936	1.4624	1.4625	1.4625
	100	1.0922	1.462	1.4623	1.4623
	120	1.0907	1.4627	1.462	1.462

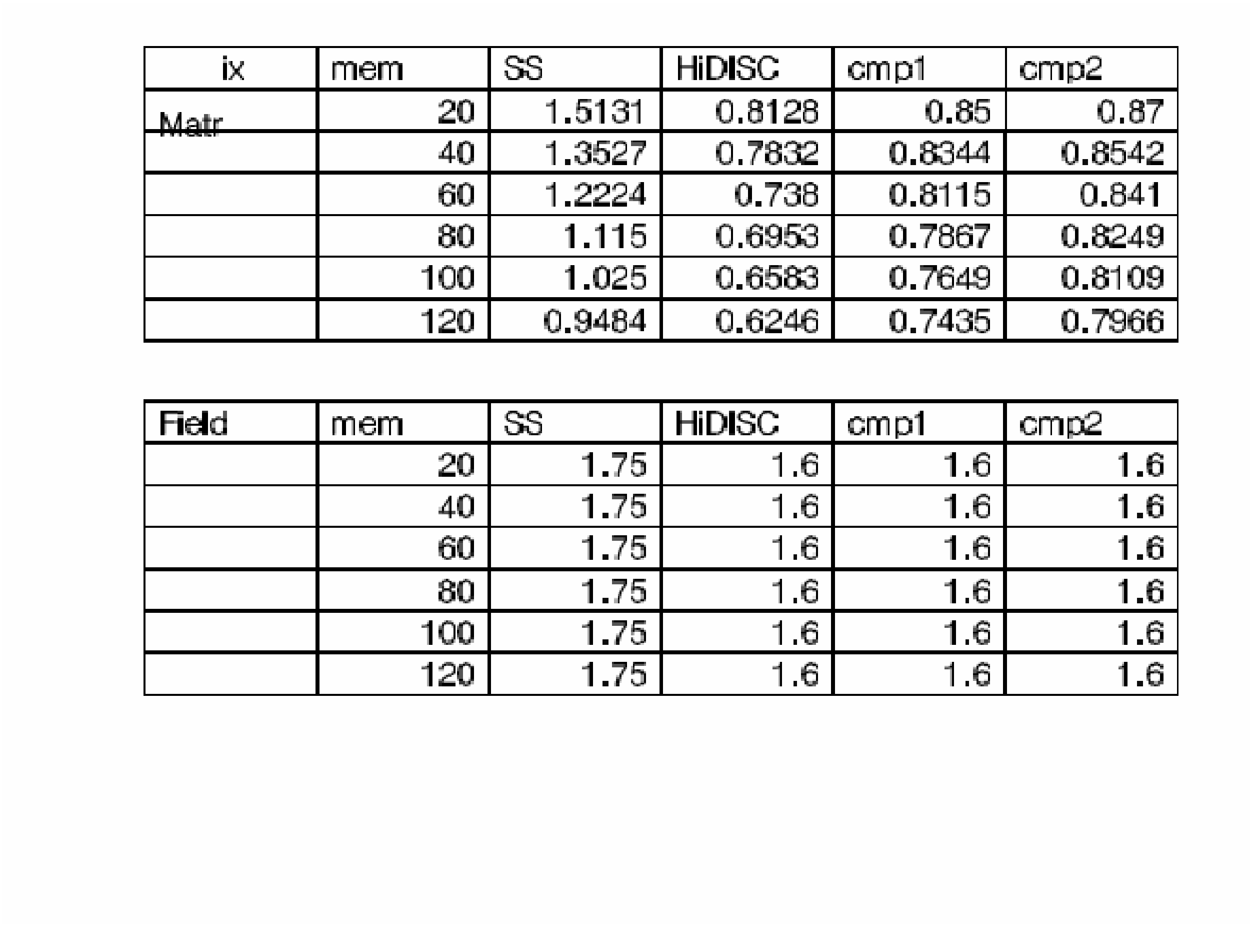
< Stressmark >

Transitive Closure	mem	SS	HiDISC	Cmp1	cmp2
	20	1.02	0.94	0.98	0.98
	40	0.63	0.91	0.98	0.95
	60	0.46	0.87	0.98	0.9
	80	0.36	0.86	0.98	0.89
	100	0.29	0.83	0.98	0.86
	120	0.25	0.82	0.98	0.83

Pointer	mem	SS	HiDISC	cmp1	cmp2
	20	1.33	1.46	1.46	1.46
	40	1.33	1.46	1.46	1.46
	60	1.32	1.46	1.46	1.46
	80	1.32	1.46	1.46	1.46
	100	1.32	1.45	1.46	1.46
	120	1.32	1.45	1.45	1.46

Neighbor-hood	mem	SS	HiDISC	cmp1	cmp2
	20	1.9	1.36	1.36	1.36
	40	1.9	1.36	1.36	1.36
	60	1.9	1.36	1.36	1.36
	80	1.9	1.36	1.36	1.36
	100	1.9	1.36	1.36	1.36
	120	1.9	1.36	1.36	1.36

Update	mem	SS	HiDISC	cmp1	cmp2
	20	1.676	1.5225	1.5225	1.5225
	40	1.6759	1.5225	1.5223	1.5223
	60	1.6759	1.5225	1.522	1.5221
	80	1.6759	1.5225	1.5218	1.5218
	100	1.6758	1.5224	1.5216	1.5216
	120	1.6758	1.5224	1.5214	1.5214



ix	mem	SS	HiDISC	cmp1	cmp2
Matr	20	1.5131	0.8128	0.85	0.87
	40	1.3527	0.7832	0.8344	0.8542
	60	1.2224	0.738	0.8115	0.841
	80	1.115	0.6953	0.7867	0.8249
	100	1.025	0.6583	0.7649	0.8109
	120	0.9484	0.6246	0.7435	0.7966

Field	mem	SS	HiDISC	cmp1	cmp2
	20	1.75	1.6	1.6	1.6
	40	1.75	1.6	1.6	1.6
	60	1.75	1.6	1.6	1.6
	80	1.75	1.6	1.6	1.6
	100	1.75	1.6	1.6	1.6
	120	1.75	1.6	1.6	1.6

List of Acronyms

Acronym	Meaning
ASIC	Application-specific integrated circuit
ASNT	Advanced scalable networking technology
BIOS	Basic input/output system
CPCI	Compact peripheral component interconnect
CPLD	Complex programmable logic device
DLX	Pronounced "Deluxe", a representative load-store architecture described in Computer Architecture: A Quantitative Approach, by John Hennessy and David Patterson, Morgan Kaufmann Publishers, Inc., 1990.
DRAM	Dynamic random access memory
DDR	Double data rate, refers to DRAM
DRC	Design rule check
GCC	Gnu C compiler
ISA	Instruction-set architecture
JEDEC	Joint Electron Device Engineering Council
LVS	Layout versus schematic
MMX	Intel's multimedia instruction-set architecture
NAS	NASA Advanced Supercomputing Division
PCB	Printed circuit board
PCI	Peripheral component interconnect
RSIM	Rice Simulator for ILP Multiprocessors
RTEMS	Real-Time Operating System for Multiprocessor Systems
SALU	Scalar arithmetic logic unit
SIMD	Single instruction, multiple data
SLP	Superword-level parallelism
SODIMM	Small Outline, Dual Inline Memory Module
SRAM	Synchronous Random Access Memory
VDC	Voltage Direct Current
VLSI	Very Large Scale Integration

Code Transformations for Exploiting Bandwidth in PIM-Based Systems

Jacqueline Chame, Jaewook Shin, Mary Hall

USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292
email: {jchame, jaewook, mhall}@isi.edu

Abstract

This paper presents code transformations designed to take advantage of the potential 2 orders of magnitude bandwidth increase available in a PIM-based architecture. Using an image processing application as a case study, we demonstrate how code transformations can exploit: (1) fine-grain parallelism in the wide-word processing unit to maximize the computation performed on each processor cycle; (2) data reuse in the large register file to avoid unnecessary memory accesses that stall the processor; and, (3) page mode accesses in the memory array to minimize the cost of each remaining memory access. While most of the transformations described here are well-known compiler techniques, in PIM-based systems we require a new optimization strategy to meet a very different optimization goal as compared to conventional approaches focused primarily on exploiting locality in a data cache. We demonstrate the importance of each set of transformations through simulation results.

1.0 Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 60% per year while memory access times improve at merely 7%. Further, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [Burger96]. Recent VLSI technology trends offer a promising solution to bridging the processor-memory gap: integrating processor logic and memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (up to 2 orders of magnitude, tens or even hundreds of gigabits per second aggregate bandwidth on a chip). Latency to on-chip logic is also reduced, down to as little as one-fourth that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

An important class of applications well-suited to PIM-based systems arise in image processing and other multimedia problems. Such applications are bandwidth limited because they perform repeated computations on streams of data; sometimes the applications have little temporal reuse [Ranganathan99]. At the same time, the applications often exhibit inherent spatial locality and both fine-grain and coarse-grain parallelism. These properties map well to PIM-based architectures. PIMs exploit spatial locality and fine-grain parallelism by accessing and operating upon multiple words of data at a time, and exploit coarse-grain parallelism by spreading independent computations throughout the memory. Thus, there is a significant opportunity for compiler technology (or clever programmers) to achieve very high performance on PIM-based systems.

In recent years, researchers have proposed many PIM-based architectures[Elliot99,Gokhale95,Kang99, Oskin98,Patterson97,Saulsbury96,Suraga96,Torrellas00], but little attention has been paid to developing compiler technology for such systems. Nevertheless, new compiler technology is needed to exploit the very different architectural features of a PIM system. On-chip memory latencies are very low. An access to the same row in the memory array as the previous access (*i.e.*, a page mode access) costs only a few cycles, and other accesses (in random mode) are 3-4 times slower, but still quite fast. Because of this lower latency, many PIM devices do not have conventional data caches, but instead rely on simple, and much more space- and power-efficient, caching mechanisms within the memory arrays themselves (for example, to exploit page mode accesses) [Elliot99,Oskin98,Saulsbury96,Zawodny98]. To exploit available on-chip bandwidth, many PIM chips also have wide data paths from memory to processing logic, and processors that can operate on several words of data in one processor cycle [Oskin98,Patterson97]. To increase on-chip bandwidth further, many PIM chips are small-scale multiprocessors, with processing logic sprinkled throughout the memory [Elliot99,Kang99,Oskin98].

In this paper, we examine code transformations to exploit potential bandwidth of a particular PIM-based system called DIVA (Data-IntensiVe Architecture), which has all of the three architectural features described in the previous paragraph (*i.e.,* caching only within the memory array, wide datapaths, multiprocessor-on-a-chip) [Hall99]. Using an image processing application as a case study, we describe how the code could be effectively transformed to tailor it to the DIVA architecture. These transformations accomplish several goals, *exploiting:* (1) fine-grain parallelism in the wide-word processing unit to maximize the computation performed on each processor cycle; (2) data reuse in the large, wide register file to avoid unnecessary memory accesses that stall the processor; and, (3) page mode accesses in the memory array to minimize the cost of each remaining memory access. We discuss the techniques that must be supported by a compiler to perform these transformations. While most of the transformations described here are well-known compiler techniques, in DIVA we require a new optimization strategy to meet a very different optimization goal as compared to conventional approaches focused primarily on exploiting locality in a data cache.

In this paper, we assume that global data and computation partitioning has been performed across the system [Anderson93], and we concentrate on how to optimize code for a single PIM processor. We describe each transformation used, and illustrate how it impacts the performance of our application. We present simulation results demonstrating the contribution of the transformations. Overall, we find that these transformations reduce the number of memory accesses by almost a factor of 350, with most of the remaining memory accesses in page mode. We also see a factor of 13 reduction in dynamic instructions executed. For this paper, we coded the application in the DIVA ISA and performed the transformations by hand. We are using this and case studies of other applications to guide the development of a compiler transformation algorithm to automatically perform these transformations, and we are using the significant analysis and code transformation infrastructure in the Stanford SUIF compiler as a basis for our implementation.

The remainder of the paper is organized into three sections and a conclusion. The next section presents an overview of the DIVA architecture. Section 3 describes the image processing code we use as a case study and the compiler transformations applied to it. Section 4 presents simulation results.

2.0 Overview of DIVA System Architecture

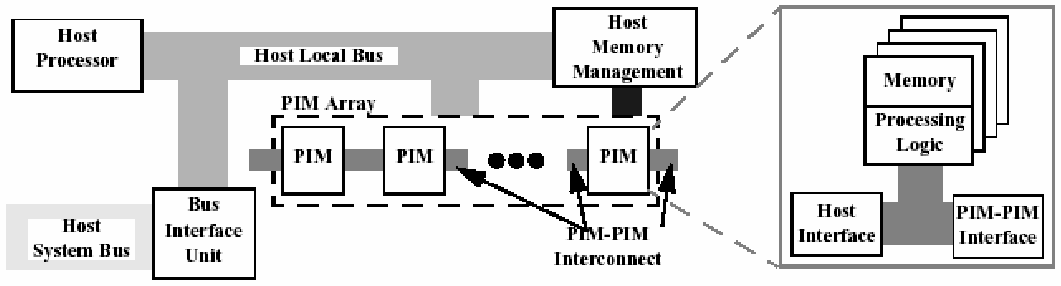


Figure 1: DIVA System Organization

In Figure 1, we show a small set of PIMs connected to a single external host through a host-memory interface; through this interface the host processor performs standard reads and writes, augmented as discussed in Section 2.2. The PIM chips communicate through separate PIM-to-PIM channels to bypass the system bus with additional memory traffic, used to spawn computation, gather results, synchronize activity, or simply access non-local data. The separate interconnect is provided because PIM-to-PIM communication requires greater bandwidth than can be achieved with a conventional memory bus. Because this paper focuses on the activity within a PIM node, we omit further description of the PIM-to-PIM interconnect, which is described in [Hall99].

2.1 PIM VLSI Component

A PIM is a VLSI memory device augmented with general and special-purpose computing hardware. A PIM may consist of multiple *nodes*, each of which are comprised of a few megabytes of memory and a node processor. The inset in Figure 1 shows a PIM with four nodes. The nodes on a chip share resources for communication with the rest of the system. As a result each chip contains a single PIM-to-PIM interface and a host interface. We anticipate that DIVA PIMs, like many other PIM chips, will be split roughly 60% memory and 40% logic (reflecting the importance of memory density).

Within a single node, shown in Figure 2, the processing logic consists of a standard scalar microprocessor including a floating-point unit and a special DIVA wide-word functional unit that performs operations on 256-bit aggregate objects stored within a row of the local memory array. The wide-word unit can be used to perform bit-level operations such as simple pattern matching, or higher-order computations such as searches, and associative and commutative reduction operations. The wide-word unit has a large register file, with 32 256-bit registers. Details on a related wide-word unit are discussed elsewhere [Brockman99].

During execution, data is transferred directly from the memory array into the register files; there is no on-chip data cache. Instead, we use the sense amps in the memory array as a small data cache, holding the full 2k-bit row selected from the previous memory access. If two consecutive accesses are within the same memory row, the second access is referred to as a *page mode access*. A page mode access is much faster than an access to an arbitrary memory row (a *random mode access*), because it does not pay the penalties for clearing the sense amps and loading a new row. In DIVA, we assume there is roughly a factor of 3 difference in latency for page mode and random mode accesses.

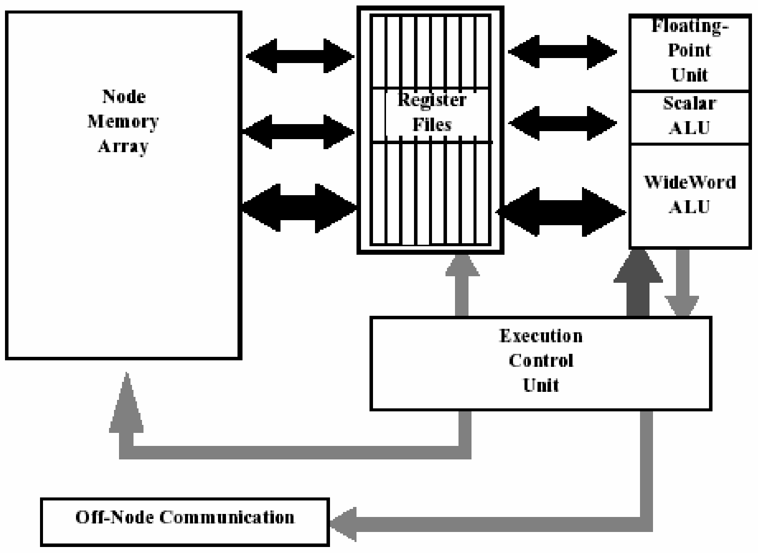


Figure 2: DIVA Node Organization

The instructions supported by the wide-word unit resemble those of multimedia ISA extensions such as MMX and Altivec, but in the case of DIVA, because the data comes directly from memory at low latency, we can expect much better performance for applications that do not make effective use of cache. The architecture also supports direct transfers of data between register files, rather than going through memory as in Altivec.

2.2 Host-Memory Interface

An underlying goal is that DIVA PIM devices can also serve as conventional memory, so that they could be used as smart-memory coprocessors in a standard system. This goal motivated a design of the PIM VLSI device to include a host interface consistent with the standard memory interface typical of commercial memories. In keeping with this goal, we would also like to package the PIMs in DIMM modules with provisions for top-plane interconnections between the memory chips to support the PIM-to-PIM communication fabric. However, unlike commercial memories, computation activities give rise to new problems: how to communicate internal exceptions and possible memory busy conditions to the host system. These issues are being addressed as part of the larger system architecture.

2.3 DIVA Memory Model

The DIVA memory model supports a globally addressable, distributed address space across the system. Coherence between the host and PIMs must be enforced in software. PIM nodes communicate using parcels, a variant of active messages, where messages are directed to objects rather than nodes. Segment registers at each PIM node support very fast on-chip address translation for local addresses; a home node provides translation if the address is non-local. Further details on the DIVA memory model can be found elsewhere [Hall99].

3.0 Code Transformations

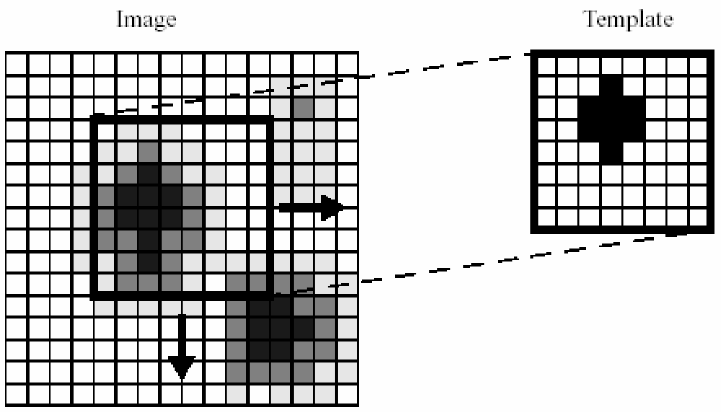
In this section, we describe code transformations for exploiting the enormous bandwidth available on PIM-based systems such as DIVA. While most of these transformations are well-known compiler techniques, PIM-based systems require a new optimization strategy to meet different optimization goals as compared to conventional approaches, which focus primarily on exploiting parallelism and locality in a data cache.

Our first optimization goal is to achieve high bandwidth utilization by exploiting fine-grain parallelism in the wide-word processing unit. Techniques for exploiting fine-grain parallelism in the wide-word unit also apply to multimedia extensions such as MMX and AltiVec, which have wide register files and allow multiple operations in a single processor cycle. Once fine-grain parallelism is exposed, the other optimization goals are to exploit data reuse in the wide-word registers to avoid unnecessary memory stalls, and to take advantage of page mode accesses in the memory array to minimize the cost of the remaining memory accesses.

To accomplish these optimization goals, we rely on several analyses and code transformations, most of which are well-known. *Parallelization analysis*, which includes data dependence analysis and array data-flow analysis, identifies loops whose iterations can be executed safely in parallel. *Reuse analysis* identifies loop iterations that access the same data (temporal reuse) or distinct data in the same cache line (spatial reuse). In addition to these analyses, several code transformations are required; the safety and profitability of the transformations are based on the above analyses. *Loop interchange* two tightly nested loops switches the inner and outer loop, and is used both: (1) to move a parallel loop to a particular position in the loop nest (innermost for fine-grain parallelism or outermost for coarser granularity); and (2) to move reuse to an innermost position. *Loop unrolling* creates multiple copies of a loop body and modifies the loop control accordingly. *Statement reordering* reorganizes statement execution while preserving data dependences. *Loop fusion* involves combining two adjacent loops with the same loop control into a single loop, used to promote reuse between the loops. *Tiling* reorders the iterations in a loop nest to bring accesses to the same data closer together in the iteration space. *Parallel reductions* are transformed versions of commutative and associative operations whose iterations can be reordered; a particular implementation of a parallel reduction is to perform independent operations on private copies of the variable and accumulate the partial results to the global copy of the variable. *Register allocation for array variables* uses data dependence analysis and loop transformations to map array variables to registers. In DIVA, we have also identified the need for a new transformation, *shifting* within a wide register between operations to exploit spatial reuse.

We are using the Stanford SUIF compiler as a basis for our implementation. With the exception of statement reordering, array register allocation, and the shift operation, implementations of the required analyses and code ransformations are already present in SUIF [Wolf92][Hall95]. These transformations must be performed in conjunction with global data and computation partitioning, which exploits coarse-grain parallelism by partitioning the computation across the PIMs [Anderson93], for which SUIF also has an implementation. Our current research involves developing a decision algorithm for using these analyses and transformations, reworking the existing decision algorithm to focus on the unique optimization goals in the DIVA architecture.

We demonstrate how to employ these transformations for PIM-based architectures with a case study from a template-matching code called SLD from Sandia National Laboratories. This code performs a correlation between an image and a series of templates to match the templates to windows within the image; it sums the image gray-scale values for pixels that are nonzero in the template. A pictorial description and a simplified, but representative loop nest is shown in Figure 3. In the figure, we show the image on the left, and the templates on the right. The two innermost loops perform a correlation between a single template and a particular window within the image; the two outer loops move the window of interest within the image.



```
for (irow = 0; irow < 32; irow ++)                              /* traverse image rows */
    for (icol = 0; icol < 32; icol ++)                          /* traverse image columns */
        for (trow= 0; trow < 32; trow ++)                       /* traverse template rows */
            for (tcol = 0; tcol < 32; tcol ++)                  /* traverse template columns*/
                if (template[trow][tcol] != 0)
                    corr[irow][icol] += image[irow+trow][icol+tcol];
```

Figure 3: Original loop nest from template-matching code

The rest of this section shows how this loop nest is transformed to better exploit features of the DIVA system. We assume that this work is done in conjunction with global data and computation partitioning, which exploits coarse-grain parallelism by partitioning the computation across the PIMs [Anderson93]. The SUIF compiler is able to partition the computation by giving different templates to different PIMs, with no inter-PIM communication.

3.1 Step 1: Fine-Grain Parallelism

.The most important opportunity for high bandwidth utilization is to take advantage of the fine-grain parallelism in the wide-word instructions. Figure 4 shows how this is achieved in our example loop nest. Because each pixel is represented by an 8-bit object, 32 pixels can be processed in a single processor cycle. The innermost loop is transformed to exploit the wide operations, consisting of loads, alignment, a pairwise logical and, and a macro that implements a reduction sum. The reduction sum is a 4-stage reduction operation that combines portions of the correlation register until the sum of all elements is computed.

```
for (irow = 0; irow < 32; irow ++)                          /* traverse image rows */
    for (icol = 0; icol < 32; icol ++)                      /* traverse image columns */
        for (trow= 0; trow < 32; trow ++) {                 /* traverse template rows */
            /* loop tcol becomes sequence of wide operations */
            wld WR11, &(image[irow+trow][icol];             /* load lower half of image row */
            wld WR12, &(image[irow+trow][icol+32];          /* load higher half of image row */
            align (WR11, WR12, icol);                       /* align to wide register WR11 */
            wld WR15, &(template[trow][0]);                 /* load template row */
            wmul WR1, WR11, WR15;                           /* select pixels according to template */
            corr[irow][icol] += reduction_sum (WR1);        /* add up selected pixels */
        }
```

Figure 4: Loop nest after transformations for fine-grain parallelism.

We use data dependence analysis and reduction recognition to identify this fine-grain parallelism. Further, we must employ reuse analysis to identify parallel loops for which there is also spatial reuse. The innermost loop must perform the same operation on adjacent (or nearby, with uniform stride) elements to be able to exploit the wide word instructions. In some cases, loop transformations such as loop interchange are needed to move the desired parallel loop, with spatial reuse, to the innermost position.

3.2 Step 2: Spatial Reuse in Large Register File

Once parallelism is exploited, our next priority is to eliminate as many accesses to memory as possible, to avoid wasting the bandwidth to memory stall cycles. Since we do not have a cache, we must reuse data within registers as much as possible. We exploit both spatial and temporal reuse in the wide registers. Spatial reuse is possible because a load into a wide register fetches several consecutive words in one transfer; for example, in the innermost loop from Figure 3, spatial reuse occurs because several arrays are accessed by columns (assuming row-major ordering).

In the `icol` loop, there is both spatial and temporal reuse. Each iteration of `icol` operates on a subrow of the image consisting of elements (`icol`, ..., `icol+31`), and therefore consecutive iterations of `icol`, say `i` and `i+1`, operate on elements (`i`, ..., `i+31`) and (`i+1`, ..., `i+32`). As illustrated in Figure 4, each iteration of `icol` accesses a new image pixel that is consecutive in memory to the last pixel accessed in the previous iteration. Also, pixels (`i+1`, ..., `i+31`), accessed in iteration `i,` are reused in iteration `i+1`. To exploit the reuse in this loop, the loop nest is transformed by interchanging loops `icol` and `trow`, making loop `icol` what is now the innermost loop. On each iteration `i` of loop `icol`, the subrow (`i`, ..., `i+31`) is brought to a wide register by shifting the data in wide registers WR11 and WR12 so that pixel `i-1` is shifted out and pixel `i+32 -1` is shifted into the last byte of WR11. Since the same wide words are loaded from memory on all iterations of loop `icol`, the memory accesses can be moved outside the loop. While this shift operation is an unusual compiler technique specifically for operations on wide data types, detecting its applicability is straightforward, involving checking the dependence distance on the loop for small, constant distances. Also, the number of accesses to the correlation matrix is reduced by exploiting the spatial reuse of `corr[irow][icol]` in loop `icol`. A correlation row is loaded into a wide register and on each iteration of loop `icol`, the correlation value `corr[icol]` is updated and stored back in the wide register. Figure 5 shows the resulting loop nest after these transformations.

```
for (irow = 0; irow < 32; irow ++)                       /* traverse image rows */
    for (trow= 0; trow < 32; trow ++)            {       /* traverse template rows */

        wld WR11, &(image[irow+trow][0]);                /* load lower half of image row */
        wld WR12, &(image[irow+trow][32]);               /* load higher half of image row */
        wld WR15, &(template[trow][0]);                  /* load template row */

        wld WR20, &(corr[irow][0]);                      /* load correlation row */

        for (icol = 0; icol < 32; icol ++)       {        /* traverse image columns */
            wmul WR1, WR11, WR15;                         /* select pixels according to template */
            WR20[icol] += reduction_sum (WR1);            /* add up selected pixels */
            shift_right (WR11, WR11, WR12);               /* shift image row by one pixel */
        }

        wst WR20,&(corr[irow][0]);                       /* store correlation row */
    }
```

Figure 5: Loop nest after spatial reuse transformations

Exploiting temporal reuse in the wide registers is also important, not only for reducing the number of memory accesses, but also for reducing potential intervening accesses that would displace the open memory row, and result in subsequent larger random-mode latencies. Temporal reuse in the wide register file is exploited in Step 4 below.

3.3 Step 3: Maximizing Page Mode Memory Accesses

The transformations for exploiting reuse result in a significant reduction in the number of memory accesses. However, since there are accesses to three distinct arrays (image, template and correlation) in the body of loop `trow`, there are intervening accesses between loads of consecutive image rows and consecutive template rows. These intervening accesses displace the current open memory row from the sense amps between reuses of the same memory row, and as a result, most of the memory accesses that reuse a memory row still suffer from higher random-mode latencies.

To exploit the lower page-mode latencies, we must reorder the loads. In the example of Figure 5, this can be accomplished by unrolling loop `trow` and grouping the memory accesses. We then fuse together the unrolled loop bodies. Figure 6 shows a simplified version of the resulting code, in which loop `trow` is unrolled by a factor of 2, for illustration purposes only. In practice, the unrolling factor depends on the size of the wide register file. In our experiments, we actually unroll the `trow` loop by 3, resulting in 4 copies of the loop body, so that the result will fit in the 32 wide registers.

Identifying potential page-mode accesses, that is, memory-row reuse, is equivalent to identifying spatial reuse at a wide word granularity. In other words, the locality analysis that identifies spatial reuse in caches can be extended to identify loop iterations that access distinct wide words in the same memory row. Once any memory-row reuse is identified, loop transformations may be applied to reduce the reuse distance. When the memory-row reuse occurs in consecutive iterations of a loop, but there are still intervening accesses on each iteration, the reuse can be exploited by unrolling the loop and grouping together the memory accesses with memory-row reuse, as in the example in Figure 6.

```
for (irow = 0; irow < 32; irow ++)                              /* traverse image rows */

    /* loop trow unrolled by 2 */
    for (trow= 0; trow < 32; trow += 2)               {         /* traverse template rows */

        /* load 2 image rows in page mode */
        wld WR11, &(image[irow+trow][0];                        /* load lower half of image row */
        wld WR12, &(image[irow+trow][32];                       /* load higher half of image row */
        wld WR13, &(image[irow+trow+1][0];                      /* load lower half of next image row */
        wld WR14, &(image[irow+trow+1][32];                     /* load higher half of next image row */

        /* load two template rows in page mode */
        wld WR15, &template[trow][0];                           /* load template row */
        wld WR16, &template[trow+1][0];                         /* load next template row */

        wld WR20, &(corr[irow][0]);                             /* load correlation row */

        for (icol = 0; icol < 32; icol ++)            {         /* traverse image columns */
            wmul WR1, WR11, WR15;                               /* select pixels according to template */
            WR20[icol] += reduction_sum (WR1);                  /* add up selected pixels */
            shift_right (WR11, WR11, WR12);                     /* shift image row by one pixel */
            wmul WR1, WR13, WR16;                               /* select pixels according to template */
            WR20[icol] += reduction_sum (WR1);                  /* add up selected pixels */
            shift_right (WR13, WR13, WR14);                     /* shift image row by one pixel */
        }

        wst WR20, &(corr[irow][0]);                             /* store correlation row */

    }
```

Figure 6: Loop nest after transformations to maximize page mode accesses

3.4 Step 4: Wide Register Allocation for Array Variables

Figure 7 shows a simplified version of the final code. In this version, we transform the code to exploit temporal reuse in wide registers, to complement the spatial reuse in Step 2. Effectively, what we are doing is performing code transformations to facilitate allocating array variables to wide registers, as is done in conventional architectures [Carr94] [Wolf92]. Previous approaches achieve this goal with some combination of tiling, unrolling and fusion; they exploit only temporal reuse, as spatial reuse only comes into play when registers hold multiple words of data. In DIVA, we must first exploit spatial reuse in the wide registers as in Steps 1 and 2, and then given the spatial reuse, also exploit temporal reuse. Here, we perform tiling to move the temporal reuse closer together in the iteration space, so that the data can fit in the limited space of the wide register file. We unroll the tiled loops so we can refer to the appropriate register explicitly in the code.

In the example, shown in Figure 7, we exploit the temporal reuse of `template[trow]` carried by loop `irow`, loops `irow` and `trow` are tiled, and the tile sizes are chosen so that a set of image rows plus a set of template rows fits in the wide register file. Furthermore, since the set of template rows (2 rows shown in the example code) is reused in the tiled loops, each template row needs to be loaded only once, before a new tile is executed. In practice, the tile size depends on the dependences and the size of the wide register file. In our experiments, we use a tile size of 2 for the `irow` loop and 4 for the `trow` loop.

```
/* loop irow tiled by irow_tsz */
for (irow = 0; irow < 32; irow += irow_tsz)                    /* traverse image rows */

    /* loop trow unrolled by 2 */
    for (trow= 0; trow < 32; trow += 2)            {           /* traverse template rows */

        /* load two template rows in page mode */
        /* template rows are reused in tiled loop irow' */
        wld WR15, &(template[trow][0]);                        /* load template row */
        wld WR16, &(template[trow+1][0]);                      /* load next template row */

        for (irow' = irow; irow' < min (irow+irow_tsz, 32); irow' ++) {

            /* load 2 image rows in page mode */
            wld WR11, &(image[irow'+trow][0]);                 /* load lower half of image row */
            wld WR12, &(image[irow'+trow][32]);                /* load higher half of image row */
            wld WR13, &(image[irow'+trow+1][0]);               /* load lower half of next image row */
            wld WR14, &(image[irow'+trow+1][32]);              /* load higher half of next image row */

            wld WR20, &(corr[irow'][0]);                       /* load correlation row */

            for (icol = 0; icol < 32; icol ++)      {          /* traverse image columns */
                wmul WR1, WR11, WR15;                          /* select pixels according to template */
                WR20[icol] += reduction_sum (WR1);             /* add up selected pixels */
                shift_right (WR11, WR11, WR12);                /* shift image row by one pixel */
                wmul WR1, WR13, WR16;                          /* select pixels according to template */
                WR20[icol] += reduction_sum (WR1);             /* add up selected pixels */
                shift_right (WR13, WR13, WR14);                /* shift next image row by one pixel */
            }

            wst WR20, &(corr[irow'][0]);                       /* store correlation row */

        }
    }
```

Figure 7: Final code, including transformations for temporal reuse in registers

4.0 Experiment

We simulated four versions of the benchmark on our DIVA simulator, and present results on the improvements due to the transformation steps from the previous section.

4.1 DSIM Simulation Environment

We have developed a system simulator called DSIM, which uses RSIM as a framework, with significant extensions [RSIM]. RSIM models shared-memory multiprocessors built with state-of-the-art processors. The DSIM host processor is taken directly from RSIM, as well as the host first and second-level caches. Our extensions include a simpler PIM processor with a WideWord ALU, a new memory system, and a new PIM-to-PIM interconnect network. We also developed application-level primitives for DIVA, such as a flush instruction, and a barrier for PIMs and host.

Table 1 summarizes the host and PIM processor parameters used in our simulations. DSIM models a host processor with out-of-order instruction execution, multiple-issue and non-blocking loads, with an architecture based on the MIPS R10000. Both L1 and L2 caches are pipelined and support multiple outstanding requests to separate cache lines. The host node is connected to the memory system via a split-transaction, 64bit-wide bus.

Each PIM node includes a PIM processor, a memory bank (which includes control and arbitration logic), and an interface to the PIM-to-PIM interconnect. The PIM processor is much simpler (and smaller) than the host processor. We extended the RSIM ISA with DIVA PIM wide instructions that operate in 256-bit wide words. For these experiments, we make the conservative assumption that the PIM processor runs at half the speed of the host system. Although the inherent speed of the logic is no slower as we are assuming the DRAM is embedded in a logic process [IBM99], we make this assumption because the wide word register accesses could impact the clock speed.

The memory system consists of the aggregation of all PIM memories, where each local memory is visible from both host and local PIM processor. DSIM models each PIM memory in detail. It maintains the current open row in the memory bank to determine the memory access time and simulates arbitration between host and PIM accesses.

Host Processor and Memory Hierarchy		**PIM Processor**	
Issue width	4	Issue width	1
Integer arithmetic units	2	Integer arithmetic units	1
Floating point units	2	Floating point units	1
Address generation units	1	Wide word units	1
L1 cache size	32K bytes	**Pim Memory**	
L1 cache hit time	1 cycle		
L2 cache size	1M bytes		
L2 cache hit time	10 cycles		
L1, L2 cache associativity	2		
Memory latency	52 cycles (page mode) 60 cycles (random mode)	Memory latency (in PIM processor cycles)	2 cycles (page mode) 6 cycles (random mode)

Table 1: Simulation Parameters

4.2 Results

In order to evaluate the benefits of the compiler transformations described in Section 3, we performed experiments using four versions of the example loop nest. The first version, Scalar, corresponds to the original loop nest of Figure 3. The second version, which we call Fine-Grain, corresponds to Figure 4, where fine-grain parallelism is exploited using wide instructions. The third version, called Spatial Reuse, exploits spatial reuse in wide registers as in Figure 5. The fourth version, called Max Page Mode +Temporal Reuse combines the transformations from Figure 6 and Figure 7 for maximizing page mode accesses and exploiting temporal reuse in the wide register file.

All four versions of the loop nest were hand-coded in the DIVA PIM ISA. We originally ran experiments using the original sequential loop nest from Figure 3, which was written in C and compiled with optimization level 4. However, since the code generated by the compiler is not tuned to the DIVA architecture, a comparison between the hand-coded transformed loops and the compiled scalar version unfairly skewed the benefits of our approach. We thus hand-coded the sequential version in order to perform a fair comparison of all versions of the loop nest. In our hand-coded version of the original loop nest, we unrolled the innermost loop by a factor of 4, so that a single 32-bit load brings 4 pixels to the register file on each loop iteration.

Table 2 shows the effect of the transformations on the number of dynamic instructions executed and on the number and type of memory accesses.

code version	# instrs	total reads	% page mode reads	total writes	% page mode writes
Scalar	395.04 M	25.17 M	33.33%	25.30 M	0%
Fine-Grain	33.47 M	3.15 M	33.33%	0.23 M	56.23%
Spatial Reuse	27.57 M	0.23 M	57.12%	0.29 M	65.97%
Max Page Mode + Temporal Reuse	29.55 M	0.11 M	77.89%	0.03 M	74.99%

Table 2: Impact of Transformations on Memory Accesses

Using Scalar as a baseline, we see that roughly 13% of the instructions are memory accesses. As compared to the baseline, we see *a factor of over 350* reduction in memory accesses in the final version, and a *factor of over 13* reduction in dynamic instructions executed. This improvement is due to several factors. Fine-Grain shows a factor of almost 15 reduction in memory accesses and a factor of over 11 reduction in dynamic instructions by exploiting the available memory bandwidth using the wide ALU and wide data path to memory. Exploiting spatial reuse in registers, particularly with the shifting operation, results in a factor of over 6.3 reduction in memory accesses in the Spatial Reuse version, as compared to Fine-Grain, but just a modest reduction in dynamic instructions executed. The number of writes actually increases we are computing partial correlation sums after interchanging the loops, which are written back to memory. The fourth version shows an additional factor of 3.7 reduction in memory accesses as a result of exploiting temporal reuse in registers, and a slight increase in the number of dynamic instructions executed due to the tiling control loop. We also see that in the final version over 74% of remaining reads and writes are now in page mode (as compared to less than 17% in the original version), which results in a lower average memory latency.

We also performed experiments comparing the execution of the entire application on the host processor against a version running on the host and multiple PIM processors. Figure 8 shows the speedups with respect to the original sequential program running on the host processor. The PIM versions were obtained by replicating the (4 Kbyte) image and assigning a subset of templates to each PIM node, with no PIM-to-PIM communication. Each PIM node computes the matches on its local templates. At the end of the PIM phase, the host collects the PIM results and computes the best match across all templates. The PIM code is hand-coded in the DIVA ISA, and it is based on the final version from Figure 7. The benefits of exploiting fine-grain parallelism and reducing memory costs result in a speedup of 2.8 for one PIM node. Combining these benefits with the coarse-grain parallelism exploited by distributing the computation among several PIM processors, we observe a speedup of 38.2 on 32 PIM nodes. Due to simulation time constraints, we used a data set size of 32 templates for the experiments shown in Figure 8. Therefore, each PIM node is assigned only one template in the 32-PIM version, and the cost of replicating the image (which is performed sequentially by the host) becomes a significant fraction of the total execution time. We expect the speedups to scale with the number of PIM nodes when using larger data set sizes, since there is no PIM-to-PIM communication.

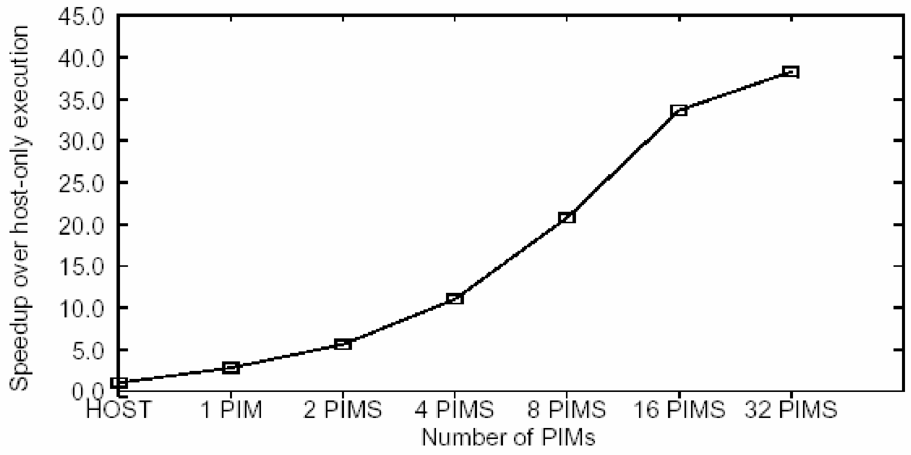


Figure 8: Speedups over host-only execution with increasing numbers of PIM nodes

5.0 Summary and Future Work

This paper presented code transformations for taking advantage of the processor-memory bandwidth in DIVA and related PIM-based architectures: exploiting fine-grain parallelism in wide word instructions, reuse in the large register file, and page mode memory accesses. We showed these techniques are highly beneficial for one image processing code; using the original hand-coded version as a baseline, we see a *factor of over 350 reduction* in memory accesses. Another nice feature of this (and many other image processing and multi-media applications) is that it can also exploit coarse-grain parallelism with (little or) no communication; each node on each PIM can execute independently. As a result, the application yields scalable parallel performance as more PIM nodes are introduced, with a speedup of 38.2 over host execution on a DIVA system with 32 PIM nodes.

We are working to automate this approach in the Stanford SUIF compiler, which we are using as the basis of the DIVA compiler. Almost all of the transformations are already implemented in the SUIF system, as well as the tests for safety and an algorithm to guide the transformations based on parallelization and reuse analysis. To automate our approach, we need to implement a few additional transformations including statement reordering, shifting and allocation of array variables to wide registers. We must also develop a new algorithm for guiding the transformations based on the requirements of the DIVA architecture.

Acknowledgements. This research was supported by the Air Force through DARPA contract F30602-98-2-0180.

References

[Anderson93] J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", In Proc. of the ACM Conference on Programing Language Design and Implementation, (PLDI'93), ACM Press, New York, June 1993.

[Babb99] J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua and S. Amarasinghe, "Parallelizing Applications Into Silicon," In Proc. of the IEEE Workshop on FPGAs for Custom Computing Machines, April, 1999.

[Brockman99] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, T. Sterling. "Microservers: A New Memory Semantics for Massively Parallel Computing", In *Proc. of the ACM International Conference on Supercomputing*, June, 1999, pp. 454-463.

[Burger96] D. Burger, J. Goodman and A. Kagi. "Memory Bandwidth Limitations of Future Microprocessors," In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA)*, May, 1996.

[Carr94] S. Carr and K. Kennedy. "Improving the Ratio of Memory Operations to Floating-Point Operations in Loops," In *ACM Transactions on Programming Languages and Systems*, Nov.,1994, 16(6);1768-1810.

[Dally92] Dally, W. J., et al, "The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanism," IEEE Micro, April 1992, pp. 23-38.

[Elliot99] D. G. Elliot, M. Stumm, W.M. Snelgrove, C. Cojocaru, R. McKenzie, "Computational RAM: Implementing Processors in Memory", IEEE Design and Test of Computers, January-March, 1999.

[Gokhale95] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*, April 1995, pp. 23-31.

[Hall95] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, M. Lam, "Detecting Coarse-Grain Parallelism Using an Interprocedural Parallelizing Compiler," In *Proceedings of Supercomputing '95*, Dec. 1995.

[Hall99] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, V. Freeh, J. Shin, J. Park, "Mapping Irregular Applications to DIVA: A Data-Intensive Architecture," In *Proc. of SC '99*, Nov., 1999.

[IBM99] IBM Microelectronics, "IBM chip advances spur "system-on-a-chip" products," www.chips.ibm.com/news/1999/sa27e/, Feb. 22, 1999.

[Kang99] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik and J. Torrellas, "FlexRAM: Toward an Advanced Intelligent Memory System," In Proceedings of the IEEE International Conference on Computer Design, Oct. 1999.

[Kogge94] P. Kogge. "The EXECUBE Approach to Massively Parallel Processing," 1994 Int. Conf. on Parallel Processing, Chicago, IL, August, 1994.

[Kogge98] P. Kogge, J.B. Brockman, V. Freeh, "Processing-In-Memory Based Systems: Performance Evaluation Considerations", Workshop on Performance Analysis and its Impact on Design, held in conjunction with ISCA '98, May 1998.

[Oskin98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. "Active Pages: A Model of Computation for Intelligent Memory". In *Proc. of the 25th International Symposium on Computer Architecture (ISCA)*, June, 1998.

[Patterson97] D. Patterson et al., "A Case for Intelligent DRAM: IRAM," *IEEE Micro*, April 1997.

[Ranganathan99] "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," In *Proc. of the International Symposium on Computer Architecture*, May, 1999.

[RSIM] http://www-ece.rice.edu/~RSIM

[Saulsbury96] A. Sauslbury, F. Pong and A. Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration, Proc. of the International Symposium on Computer Architecture, May, 1996.

[Sunaga96] T. Sunaga, P.M. Kogge, et al, "A Processor In Memory Chip for Massively Parallel Embedded Applications," IEEE J. of Solid State Circuits, Oct. 1996, pp. 1556-1559.

[Torrellas00] J. Torellas, L. Yang, and A. Nguyen. "Toward a Cost-Effective DSM Organization That Exploits Processor-Memory Integration," In *Proc. of the High Performance Computer Architecture Conference,* January, 2000.

[Wolf92] M. E. Wolf, "Improving Locality and Parallelism in Nested Loops," Phd Dissertation, Stanford University Computer Systems Laboratory, August 1992.

[Zawodny98] J. Zawodny, P. Kogge, J. Brockman, E. Johnson, "Cache-In-Memory: A Lower Power Alternative," Workshop on Power-Driven Microarchitecture, held in conjunction with Int. Symp. on Computer Arch., Barcelona, Spain, June 27-28, 1998.

The Address Translation Unit of the Data-Intensive Architecture (DIVA) System

Herming Chiueh, Jeffrey Draper, Sumit Mediratta and Jeff Sondeen
USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, CA 90292
{chiueh,draper,sumitm,sondeen}@isi.edu

Abstract

The Data-Intensive Architecture (DIVA) system incorporates Processing-In-Memory (PIM) chips as smart-memory coprocessors to a microprocessor. This architecture exploits inherent memory bandwidth both on chip and across the system. Thus, performances of pointer-based and sparse-matrix computations as well as multimedia applications are significantly enhanced.

A key feature of the DIVA architecture is the address translation mechanism, which supports virtual addressing of application code and data. Instead of prohibitive conventional page tables, DIVA provides a simplified mechanism using segments. In this paper, the design of the address translation unit is presented, and trade-offs in VLSI design including performance, area, and design modulation are also discussed.

1. Introduction

The Data-Intensive Architecture (DIVA) project is building a workstation-class system using embedded memory technology to replace the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. The goal of the project is to significantly reduce the ever-increasing processor-memory bandwidth bottleneck in conventional systems. System bandwidth limitations are thus overcome in three ways, as illustrated in Figure 1: (1) tight coupling of a single processing-in-memory (PIM) processor with an on-chip memory bank; (2) distributing multiple processor-memory nodes per PIM chip; and (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus.

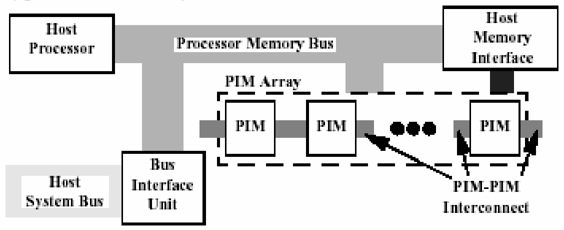


Figure 1. DIVA system architecture

This paper describes the design of an address translation unit, which is the key component that implements memory management in DIVA PIM chips. Previous literature [1] distinguished two aspects of memory management requirements from that of other PIM-based architecture.

- The PIM serves as the only memory for a standard host microprocessor, assuming the duel role of "smart memories" and conventional memory.
- DIVA targets applications that are most severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets.

As compared to system-on-chip solutions [2-3], and multiprocessors made up solely of PIM chips [4-5], DIVA's support for conventional memory access from an external host requires a dual view of memory, from host's and the PIM's perspective. A much broader range of programming paradigms are provided when compared with other PIM architectures. As a result, DIVA requires an efficient address translation mechanism and independent threads of control as the features in its memory models. A previous paper presented an overview of the DIVA project and described a memory model [6] and memory management [1] to support these requirements. This paper is focused on the design of the address translation unit and its circuit implementation. The remainder of the paper is organized as follows. Section 2 describes the mechanism of address translation in DIVA. Section 3 presents the detailed hardware design of the address translation unit (ATU). Section 4 presents a VLSI implementation and results, and Section 5 concludes the paper.

2. Address translation mechanism

The virtual address space of the host processor in the DIVA architecture can be categorized into three classifications:

- *Global memory* is composed of contiguous segments distributed across nodes, visible to applications running on the host and PIM nodes.
- *Dumb memory* is a region of a node's memory allocated as conventional pages in a host

application's virtual space and untouched by PIM node processing.

- *Local memory* is a region of a node's memory used exclusively by node routines. This rule is excepted during initialisation when the host system boot process loads node software.

A node must be able to rapidly determine if an address is located in its own memory, and if so, find the physical address. Segments are used to condense translation information. Each segment is defined by segment registers containing a base address and size.

The local memory region is partitioned into eight segments in the DIVA architecture. Like pages in a conventional system, the segment descriptors are generic in nature. It is only through system programming that the segments serve a specific purpose [1].

Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation information. In addition to the local segments, a node maintains translation information for its resident portion of the global memory, as well as for any remote data for which it is the home node.

The primary functions of the node address translation unit are to translate virtual addresses to physical addresses for those accesses that are locally resident and to provide access protection. The types of accesses generated by a DIVA PIM processor that require translation include instruction fetches and data accesses to memory or memory-mapped devices such as parcel buffers, generated by load or store instructions.

Given the simplicity of the address translation scheme discussed above, very little hardware support is needed to effect translation. A segment base address register and limit register is needed for each of the eight local segments. Also, one virtual base, limit, and physical base register are needed for each resident global segment. The DIVA architecture provides four sets of global segment registers. The address translation unit contains no direct support for home node translation, although the preferred system programming is such that the global segments resident on a node form the portion of global memory for which that node is the home node. If this is not the case, address faults invoke system software that performs the home node translation.

3. Design of ATU

The DIVA PIM processor provides 4 Gbytes of virtual address space accessible to kernel and user applications via segments that are a power of 2 in size. Segment sizes can range from 256 bytes to the maximum amount of physical memory available to a node. The maximum segment size in the initial DIVA system design is 16 Mbytes. Each virtual address generated by the PIM processor is 32 bits, and the resulting physical address generated by the address translation unit is also 32 bits.

The PIM processor address translation unit supports three main types of address translation: direct address translation, local address translation, and global address translation

Figure 2 shows the three main address translation mechanisms provided. When the address translation unit is disabled, direct address translation occurs, and the address translation unit will not generate any exceptions. In this case, the resulting physical address is identical to the virtual address. If address translation is enabled, then the scope field of the virtual address much be inspected to determine what type of translation should be used.

The scope field of the virtual address is the most significant five bits of the virtual address VA. If this 5-bit value is zero, then local translation is used. If the scope field equals binary value 00001, i.e., the virtual address falls in the range of 0x08000000 to 0x0FFFFFFF, direct translation is used to generate the physical address; however, unlike the mode where address translation is disabled, an exception can be generated in this case if access privileges are violated. By definition, the address region 0x08000000 to 0x0FFFFFFF is a supervisor–level region. Therefore, any user-level attempt to access this region while address translation is enabled will trigger an exception. Lastly, if any of the four most significant bits of the virtual address are non-zero, i.e., va[0:3] != 0, then global translation is used.

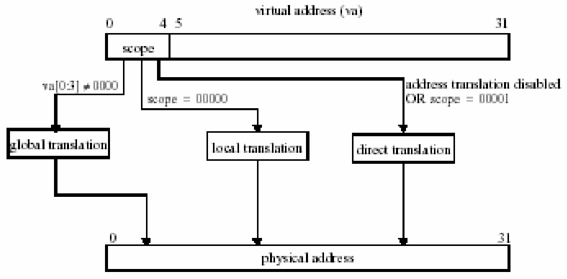


Figure 2. Address translation types

Figure 3 shows the steps involved in local address translation. The 3-bit index field of the virtual address is used to select a set of local segment registers for the translation. The segment base is simply bitwise-ORed with the zero-padded offset of the virtual address to form the physical address. The specified segment limit register is also accessed and manipulated in conjunction with the offset to determine if the virtual address is valid.

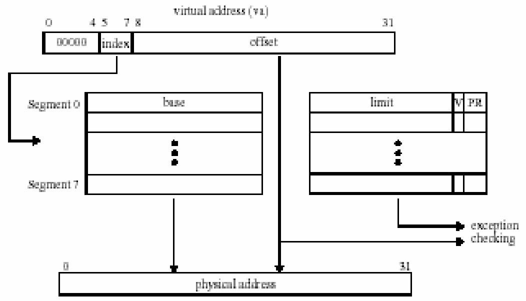


Figure 3. Local address translation

Figure 4 shows the steps involved in global address translation, which is a reverse address translation style. In this case, the address is checked to see if it is mapped locally by simply ensuring that the address is within the range specified by a valid set of the global segment base address and limit registers. The hardware does not protect against overlapping global segments. The multiple sets of global segment registers are checked concurrently to see if any one of them should be used for the translation, similar to a fully associative cache. If there is a match, the virtual address is simply translated into a physical address by a bitwise-OR of an offset with the global segment physical base register of the matching global segment. The offset is formed by using the limit register of the matching segment to mask off the appropriate part of the virtual address.

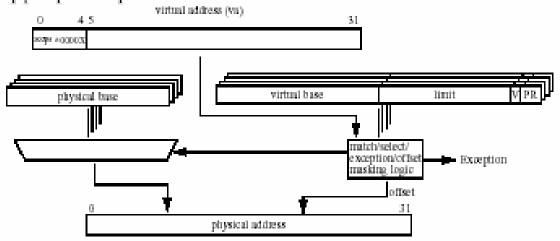


Figure 4. Global address translation

In addition to the translation of virtual addresses to physical addresses, the address translation unit provides access protection and bounds checking to ensure that the offset portion of an address is not outside the range of the segment. The 2 PR bits of a segment limit register specify the access protection mode for that segment. Table 1 shows the possible access modes and their corresponding encodings.

Table 1. Segment access modes and corresponding PR bit encodings

Encoding of PR Bits	Supervisor Privilege	User Privilege
00	**RW(read-write)**	**RW**
01	**RW**	**RO(read only)**
10	**RW**	**None**
11	**RO**	**None**

Each local segment limit register consists of a limit value, a valid bit, and the two PR bits. The first level of protection for local addresses is provided by ensuring that a valid set of segment registers is used. If the V bit of the selected local segment is not asserted, an unmapped access exception occurs. The second level of protection is provided by the PR bits. If the PIM processor mode and access type are not allowed by the PR bit setting of the selected segment, an invalid access exception occurs. The final level of protection for local addresses is provided with bounds checking. The limit value of the specified segment is used to inspect bits in the virtual address offset to ensure that the offset has not exceeded the segment size. If the segment size is exceeded, an unmapped access exception occurs. Equation (1) specifies the exception condition E for local translations.

$$E = (va_8 \wedge \overline{\text{limit}[\text{index}]_8}) \vee (va_9 \wedge \overline{\text{limit}[\text{index}]_9}) \vee \cdots \vee (va_{23} \wedge \overline{\text{limit}[\text{index}]_{23}}) \cdots\cdots\cdots\cdots (1)$$

Although the conditions for address translation exceptions for global virtual addresses are similar to that of local addresses, the mechanism is quite different due to the fully associative nature of the global segment hardware. Basically, if one of the four sets of global segment registers does not *match* an attempted global address access, an exception occurs. A successful *match* occurs when a set of segment registers is valid, the PR bit setting allows the access type being attempted, and the address range specified by the global virtual base and limit encompasses the global address of the operation. Equation (2) specifies the range match condition RM, where va is the virtual address and base is the contents of the global virtual base register.

$$\overline{RM} = (\overline{\text{limit}_0} \wedge (va_0 \oplus \text{base}_0)) \vee (\overline{\text{limit}_1} \wedge (va_1 \oplus \text{base}_1)) \vee \cdots \vee (\overline{\text{limit}_{23}} \wedge (va_{23} \oplus \text{base}_{23})) \cdots\cdots\cdots\cdots\cdots (2)$$

An unmapped access exception is triggered if there is no valid set of registers that satisfies the range match test. If there is a valid set of registers that satisfies the range match test, but the PR bits for that segment do not allow the attempted access, an invalid access exception occurs.

4. Implementation and results

Based on the design presented in Section 3, the schematic for the ATU design is presented in Figure 5. Four major components were designed and implemented with Synopsys tools: virtual to physical translation(V2P) module, controller, special purpose register files, and probe circuit.

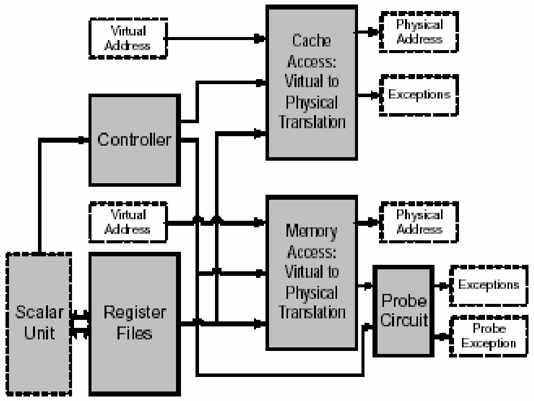


Figure 5. Schematic of ATU implementation

Design issues for these blocks are as follows:

- V2P module: This is the core circuit to implement the translation scheme specified in Section 3. The key design issue for this circuit is the translation speed. To achieve the defined specification of 5ns, several techniques in VHDL coding for synthesis were required. The result was a pure combinational logic circuit that is able to implement the translation mechanism with minimum overhead in speed and circuit area.

- Special purpose register file: This module is an interface for the PIM processor to set up the translation table. Since simultaneous translation and table set-up will never occur with proper system software, the core issue of this module's design is to reduce the circuitry area while providing a wide data bus with low propagation speed, which provides a complete static table look up for V2P module to speed up the translation.
- Controller: This module translates bus signals and memory access signals to two V2P modules, one for processor memory requests and the other for instruction cache memory requests.
- Probe Circuit: This module is used when a specific DIVA instruction is used to probe the address translation unit, allowing a user-level process to interrogate the status of a virtual address without incurring an exception if the address is not mapped.

Each module was designed and synthesized using Synposys tools; the timing and circuit size of each module was optimised using the constraints mentioned above. The circuitry of the whole address translation unit was generated using Cadence Silicon Ensemble. The design is based on TSMC 0.18µm technology with Artisan standard cells. Table 2 summarizes the results. Varying the use of different optimisations, a great difference in circuitry area is observed. After several iterations in both synthesis and layout generation, a 30% reduction in circuitry area is achieved while maintaining the same fast translation speed by ignoring the constraint of the special register file's data path, which does not affect the translation speed.

Table 2. Circuit Summary

Gate counts	7967
Power	Core: 18.93mW Total: 41.78mW
Area	500 x 450 µm
Percentage of modules' area	V2P Memory: 19.95%, V2P Cache: 19.58% Controllers: 0.03%, Probe circuitry: 0.2% Special purpose register file: 60.24%,

In Figure 6, a layout of the ATU is presented. The purpose of this layout is to form an initial estimate of the overhead of implementing an address translation mechanism in the DIVA PIM processor.

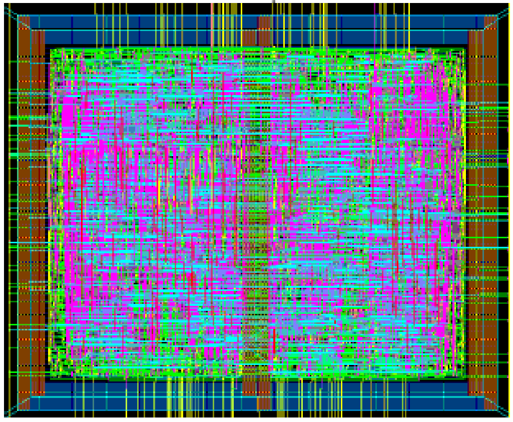


Figure 6. Layout of ATU

This circuitry occupies 500 x 450 µm, which is 2.2% of the DIVA PIM processor area [7-8]. Considering most of the circuitry (60% of standard cells is register files) is not switching during the translation, there is only a 2.9% power increase for the DIVA PIM processor (28.8mW/580mW) to support the address translation mechanism. The overall delay for the ATU is 4.76ns, which is sufficiently fast to integrate it into the DIVA PIM processor without any extra delay.

5. Conclusion

This paper has presented the design and implementation of the Address Translation Unit to be used in the DIVA PIM processor. An implementation of this design, based on TSMC 0.18µm technology, has proven to be easily integrated into the current DIVA PIM prototype. The ATU is a key component to enable DIVA's memory management design, which is essential for a user-friendly programming paradigm for PIM systems like DIVA.

Acknowledgments

This research was supported by DARPA contract F30602-98-2-0180.

References

[1] M. Hall and C. Steele, "Memory Management in PIM-Based Systems," in Proc of the Workshop on Intelligent Memory Systems, held in conjunction with Architectural Support for Programming Languages and Operating Systems, Boston, MA, 2000.

[2] D. Patterson, et al., "A case for Intelligent DRAM: IRAM," IEEE Micro, 1997.

[3] Mitsubishi, ":M32R/D Series: 32-bit RISC Processor, On-chip DRAM," www.mitsubishi-chips.com/data/datasheets/mcus/m32rdgrp.html, May 6, 1999.

[4] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin, "An Argument for Simple COMA," In Proc. of the Symposium on High-Performance Computer Architecture, 1995.

[5] P. Kogge, "The EXECUBE Approach to Massively Parallel Processing," 1994 International Conference on Parallel Processing, Chicago, IL, 1994.

[6] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, J. Shin, and J. Park, "Mapping Irregular Computations to DIVA, a Data-Intensive Architecture," in Proc. of SC '99, 1999.

[7] J. Draper, I. Kim, J. Sondeen, and S. Mediratta, "Implementation of a 32-bit RISC Scalar Processor for the Data-Intensive Architecture (DIVA) Processing-In-Memory (PIM) Chip," Submitted to International Conference on Application-Specific Systems, Architectures, and Processors (ASAP), 2002.

[8] J. Draper, C. W. Kang, and J. Sondeen, "Implementation of a 256-bit Wide Word Processor for the Data-Intensive Architecture (DIVA) Processing-In-Memory (PIM) Chip," Submitted to ESSCIRC, 2002.

The Architecture of the DIVA Processing-In-Memory Chip

Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett,
Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen,
Chang Woo Kang, Ihn Kim, Gokhan Daglikoca
USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

{draper,jchame,mhall,steele,jtb,jlacoss,granacki,jaewook,chunchen}@isi.edu, {ckang,ihnkim}@usc.edu,
giskard@computer.org

ABSTRACT

The DIVA (Data IntensiVe Architecture) system incorporates a collection of Processing-In-Memory (PIM) chips as smart-memory co-processors to a conventional microprocessor. We have recently fabricated prototype DIVA PIMs. These chips represent the first smart-memory devices designed to support virtual addressing and capable of executing multiple threads of control. In this paper, we describe the prototype PIM architecture. We emphasize three unique features of DIVA PIMs, namely, the memory interface to the host processor, the 256-bit wide datapaths for exploiting on-chip bandwidth, and the address translation unit. We present detailed simulation results on eight benchmark applications. When just a single PIM chip is used, we achieve an average speedup of 3.3X over host-only execution, due to lower memory stall times and increased fine-grain parallelism. These 1-PIM results suggest that a PIM-based architecture with many such chips yields significantly higher performance than a multiprocessor of a similar scale and at a much reduced hardware cost.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures; C.1.2 [**Computer Systems Organization**]: Processor Architectures—*Multiple Data Stream Architectures (Multiprocessors)*

General Terms

Design, Performance

Keywords

processing-in-memory, memory bandwidth, architecture

1. INTRODUCTION

A recent trend in computer architecture combines processing logic with memory in intelligent processing-in-memory (PIM) chips to address the well-known performance gap between processor and memory speeds [2, 7, 8, 12, 15, 16, 17, 20, 21, 22, 25, 27, 28, 30]. Many previous architectural solutions to the processor-memory gap such as multithreading, prefetching, and speculation, seek to reduce or tolerate memory latency, at the expense of increased memory bandwidth requirements [3]. PIMs instead dramatically improve memory bandwidth, by 10-100X over conventional DRAM systems, because internal processors can be directly connected to the memory banks. Latency to on-chip logic is also reduced, down to less than one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

For the last four years, the authors have been developing one such PIM-based system called Data IntensiVe Architecture (DIVA). The ultimate goal of the DIVA project is to design and build a prototype workstation-class system where PIMs serve as smart-memory co-processors for an otherwise conventional system. In this paper, we describe the prototype DIVA PIM chip, shown in Figure 1, which we have recently fabricated.

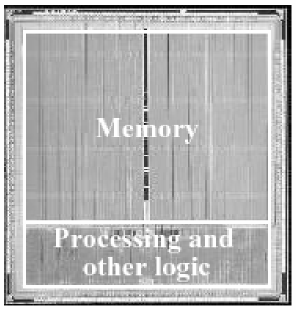


Figure 1: Microphotograph of a DIVA PIM.

DIVA targets two important classes of bandwidth-limited applications: multimedia and irregular applications, including sparse-matrix and pointer computations. Multimedia applications perform repeated computations on streams of data, often with little temporal data reuse. As processors exploit increased parallelism, multimedia applications become memory bound [23]. Performance of applications with irregular data accesses is also dominated by memory stalls since such applications usually have neither temporal nor spatial reuse of data needed to make effective use of cache [5]. DIVA accelerates both classes of applications by performing computation directly in memory, requiring novel underlying hardware structures, described in this paper. Streaming multimedia applications obtain high bandwidth to on-chip memories through a 256-bit wide datapath, while irregular applications benefit from very low latency accesses to memory. As a result, much of the traffic between the host processor and memory is eliminated.

Our experience with the DIVA PIM chip has important implications for future architectures that seek to maximize memory bandwidth. We demonstrate that simple but powerful hardware mechanisms can yield significant performance improvements on bandwidth-limited applications. Many of these hardware features, including address translation, the memory interface and memory-to-memory interconnect, are specifically oriented towards architectures such as DIVA, in which PIMs are smart-memory co-processors to a conventional host. Many other features are suitable for conventional processors and embedded systems-on-a-chip, such as the design of the WideWord unit.

In two previous papers, we presented the DIVA system architecture, memory model and simulated performance improvements due to coarse-grain parallelism in PIMs for 3 programs [9], and we described system software requirements and memory management functionality [10]. This paper focuses on the DIVA PIM device and makes the following unique contributions.

- It is the first detailed description of the DIVA PIM microarchitecture.
- It pinpoints some of the design issues that must be considered in future architectures for exploiting memory bandwidth.
- It presents simulation results demonstrating an average speedup of 3.3X on 8 programs as compared to a conventional host. The speedups are due to up to a 95% reduction in memory stall time, and, for 4 of the programs, an average speedup of 9.94X due to the WideWord unit as compared to scalar PIM execution.

The remainder of the paper is organized as follows. The next section summarizes the DIVA system architecture, to set the context for the PIM microarchitecture discussion. Section 3 describes the microarchitecture in detail. Section 4 presents a set of simulation results on eight programs. Section 5 presents the status of the DIVA project. Section 6 presents related work, and Section 7 concludes the paper.

2. SYSTEM ARCHITECTURE OVERVIEW

The DIVA system architecture was specifically designed to support a smooth migration path for application software by integrating PIMs into conventional systems as seamlessly as possible. DIVA PIMs resemble, at their interfaces, commercial DRAMs, enabling PIM memory to be accessed by host software either as smart-memory co-processors or as conventional memory. In Figure 2, we show a small set of PIMs connected to a host processor through *nearly* conventional memory control logic (see Section 3.1 for details on required modifications). A separate memory-to-memory interconnect enables communication between memories without involving the host processor.

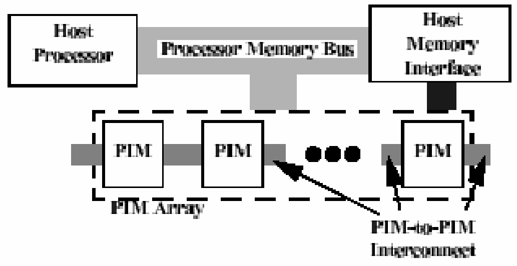


Figure 2: DIVA system architecture.

Spawning computation, gathering results, synchronizing activity, or simply accessing non-local data is accomplished via parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received [29]. Parcels are distinguished from active messages in that the destination of a parcel is an object in memory, not a specific processor.

Parcels are transmitted through a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic. This interconnect must support the dense packing requirement of memory devices and allow the addition or removal of devices from the system. For system sizes of the scale expected for DIVA (on the order of 32 PIM chips), this combination of requirements favors a one-dimensional network [14]. Future generations of DIVA-like systems that contain large numbers of PIM chips will require a more complex interconnection network and are the topic of future research.

Parcels, application code, and data contain virtual addresses. To translate these addresses without the overhead of maintaining conventional page tables at each node, we classify DIVA memory according to usage [9]: (1) *global memory* visible to the host and PIM nodes; (2) *dumb memory* allocated as conventional pages in a host application's virtual space and untouched by PIM node processing; and, (3) *local memory* used exclusively by PIM node routines. To condense translation information, rather than page tables, we use segments, each of which is defined by segment registers, as discussed in Section 3.4. In addition to local segments, a node maintains translation information for its portion of global memory. Remote addresses are translated via the concept of a home node, which is guaranteed to have the translation [26]. Thus, each node's portion of global memory includes objects for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

Memory management functionality is distributed among the host's standard operating system, augmented with support for PIMs, and run-time kernels on PIM processors. Unlike standard multiprocessor systems, the host, which has a system-level view, remains a central figure in system-level scheduling, disk I/O operations, and memory management. The PIM run-time kernel must collaborate with the host on system-level operations, such as loading PIM programs and data, memory management of PIM-visible segments, and PIM context switches between different user programs. The challenge in this collaboration is that two views of memory must be maintained. For dumb pages and for disk I/O of PIM-visible segments, the host sees memory as standard 4Kbyte pages; the PIM run-time kernel instead views PIM-visible memory as variable-sized segments [10].

3. DIVA PIM MICROARCHITECTURE

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and communication hardware. Although a PIM may consist of multiple nodes, each of which are primarily comprised of a few megabytes of memory and a node processor, Figure 3 shows a PIM with a single node, which reflects the focus of the initial research that is being conducted. Nodes on a PIM chip share a host interface and a single PIM Routing Component (PiRC). The host interface implements the JEDEC standard SDRAM protocol so that memory accesses as well as parcel activity initiated by the host appear as conventional memory accesses from the host perspective. The PiRC is responsible for both routing parcels off chip via the PIM-to-PIM interconnect and directing parcels on chip.

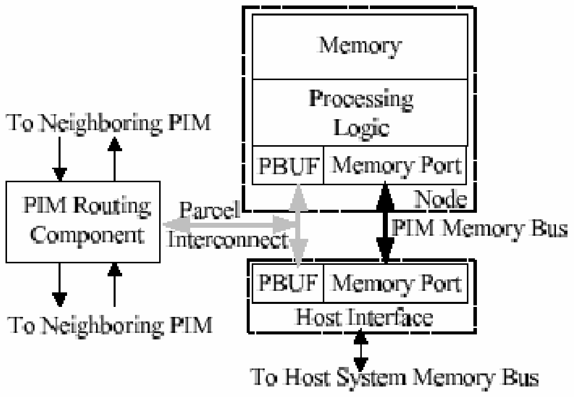


Figure 3: DIVA PIM chip architecture.

Figure 3 also shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. The host interface also contains a parcel buffer (PBUF) for parcel communication between host and PIM. Each PIM node also has a PBUF, for node-to-node parcel communication, as will be discussed in Section 3.3.

Figure 4 shows the major control and data connections within a node. The DIVA PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath that performs operations similar to those of standard 32-bit integer units, and a 256-bit WideWord datapath that performs fine-grain parallel operations on 8-, 16-, or 32-bit operands. Both datapaths execute from a single instruction stream under the direction of a single 5-stage DLX-like pipeline [11], complete with register forwarding logic to resolve data dependence hazards. This pipeline fetches instructions from a small instruction cache, which is included to minimize memory contention between instruction reads and data accesses. The instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap. The scalar datapath is a standard RISC architecture, augmented with a few DIVA-specific functions for coordinating with the wide datapath. The WideWord datapath accesses the scalar registers for addressing operations, as well as for controlling superword operations. Each datapath has its own independent general-purpose register file with 32 registers. Special instructions permit direct transfers between register files without going through memory. Although not supported in the initial DIVA prototype shown in Figure 1, floating-point extensions to the WideWord unit will be provided in future systems.

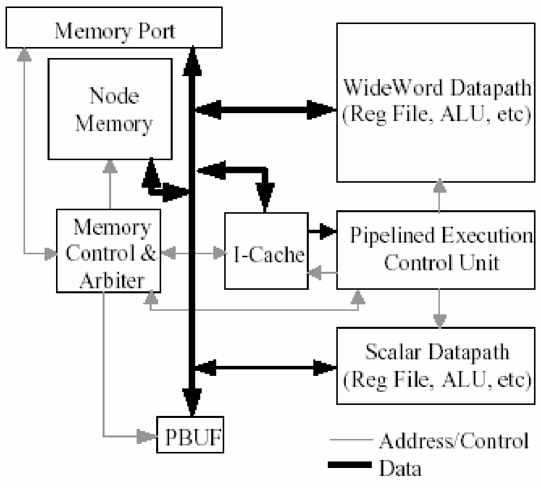

Memory Port
Node Memory
WideWord Datapath (Reg File, ALU, etc)
Memory Control & Arbiter
I-Cache
Pipelined Execution Control Unit
Scalar Datapath (Reg File, ALU, etc)
PBUF
Address/Control
Data

Figure 4: DIVA PIM node organization.

The execution control unit supports supervisor and user modes of processing and also maintains a number of special-purpose and protected registers for support of exception handling, address translation, and general OS services. Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external interrupt signal, are handled by a common mechanism. The exception handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, to maintain a flexible implementation platform. It provides an integrated mechanism for handling hardware and software exception sources.

The following sections present the DIVA PIM node in more detail and highlight some of the unique features of the DIVA microarchitecture. The first subsection focuses on the most distinguishing feature of a PIM as compared to a conventional processor, its memory unit and memory interface. Subsequently, we describe DIVA's WideWord unit, parcel interconnect and address translation mechanism.

3.1 Host Memory Interface and Memory Unit

The host interface and memory unit reflect a number of the challenges in designing a PIM that serves as a smart-memory co-processor to a conventional host. Our underlying goals were to minimize performance penalties to conventional memory accesses as viewed by the host, while maximizing the potential benefit of PIM operations. Although the original design targeted embedded DRAM, the prototype shown in Figure 1 is an SRAM-based design due to challenges in timely access to embedded DRAM fabrication lines. We first describe the resulting design implemented in this prototype chip and then present necessary considerations for a DRAM-based PIM design.

A PIM chip's host interface externally implements the JEDEC SDRAM protocol so that the PIM appears as a conventional SDRAM to the host processor. On-chip, the host interface communicates with an internal memory controller to negotiate access to the embedded memory. In essence, the host interface is a translator between the standard SDRAM protocol and an internal PIM-specific protocol. To satisfy the SDRAM timing requirement, this interface must ensure consistent timing for host memory accesses. At first glance, this may seem difficult to enforce since the PIM node processor may be accessing memory when a host memory request arrives, thereby causing the host access to incur an additional latency. However, a couple of factors allow the PIM to respond with predictable latency as required by the standard. First, the embedded SRAM macro of the prototype chip has a 3.5ns cycle time and 256-bit data bus. Secondly, the internal clock of the PIM is at least twice that of the SDRAM bus (4X for some implementations), so the addition of an arbitration cycle is negligible to the overall memory latency. Refer to Figure 5, which shows a timing diagram for a 3-cycle CAS latency SDRAM burst read operation. The

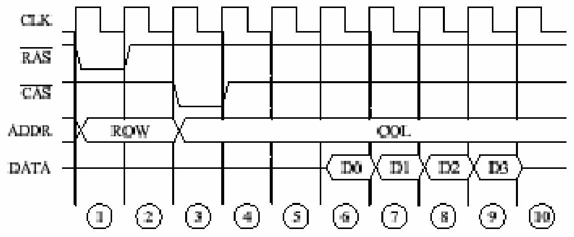


Figure 5: SDRAM burst read timing.

worst-case read latency occurs if the memory is busy satisfying a PIM processor request when the host request arrives. Even in this case, once the CAS strobe of cycle 3 in the figure has been detected, there is only a 4-PIM-cycle latency (2 SDRAM cycles) for the read request to be forwarded from the host interface to the internal node memory controller, serviced, and returned for output onto the SDRAM data bus. This allows the PIM to output the first 64-bit data word in cycle 6, satisfying the SDRAM protocol. Since all accesses to the embedded memory involve 256 bits of data, the succeeding 64-bit data words are readily available for the host interface to output them in cycles 7, 8, and 9. Similar timing applies for write accesses.

The internal node memory controller, shown in Figure 4, consists of two basic components: an *arbiter* and a *memory control unit*. The arbiter performs handshaking between requesters of memory accesses and determines priority of competing requests. Requests for accesses, *i.e.*, reads and writes, may originate from the host interface memory port, PIM processor instruction cache, and/or memory stage of the PIM processor pipeline. Arbitration priorities are formulated as follows: 1) host interface, 2) PIM processor memory stage, and 3) PIM processor instruction cache. The host interface has the highest priority since minimal latency is required for the PIM chip to appear as conventional SDRAM for host processor accesses. The arbiter communicates closely with the memory control unit, which is responsible for generating all control signals to the memory array, such as macro select, write enable, output enable, and address bits. Once a requester has been granted access, the requesting source drives the data bus and associated byte-write-enable signals for write accesses while the memory control unit drives the control signals. For read accesses, the requester simply latches data returned from the memory at the appropriate time.

For future DRAM-based implementations, the PIM chip memory system must be augmented to support refresh operations and page-mode accesses. For refresh operations, the host interface must be able to translate system memory controller refresh operations into internal refresh operations, which is a fairly straightforward exercise. To exploit page-mode accesses, the node memory controller should maintain a *current page address register*. For normal read/write accesses, the address presented with the request is compared against the contents of the current page address register, assuming that a page is currently open. If the portion of the requesting address which designates the DRAM page matches the value of the current page address register, the access is performed as a page-mode access. If the two values are unequal, a random access must be performed, which entails restoring the currently open page and strobing in the new page corresponding to the access request. Simultaneously with this access, the new page address is latched into the current page address register. Also, the current page address register is invalidated upon refresh operations, since refresh operations corrupt the values retained in the DRAM sense amps, which represent the currently open page.

Also, DRAM-based PIM implementations must carefully consider the SDRAM interface requirements. As an example, consider an implementation based on the DRAM macro provided by the IBM Cu-11 process [13]. Like the SRAM macro used in the first DIVA prototype chip, this macro supports a 256-bit data bus with byte-write-enable signals to support writes of data smaller than 256 bits, where needed. The macro page size is 2048 bits. The page-mode cycle time is 5ns, while the random-mode cycle time is 15ns.

If the system memory controller always initiates full burst requests, like the one shown in Figure 5, small modifications can be made to the internal PIM logic to satisfy the SDRAM timing requirements. As soon as a system memory controller RAS cycle is detected, the host interface should alert the internal memory controller to finish its current memory operation and remain idle in anticipation of a host request. Even with the current highest-speed SDRAM standard, 133MHz, there is a 38ns latency between the RAS cycle and when data is required for read operations for a 3-cycle CAS latency implementation. Based on the random-mode cycle times of the IBM DRAM macro mentioned above, this is adequate time for the internal node memory controller to complete its current memory cycle and respond to a host-initiated memory cycle. For systems with intelligent memory controllers that perform page-mode accesses (initiating CAS-only memory operations), the CAS latency must be configured to support the maximum PIM latency. The resulting memory latency penalty is highly system-dependent in this case.

3.2 WideWord Unit

The WideWord unit operates on 256-bit words, enabling applications to exploit fine-grain, or superword-level, parallelism and the increased processor-memory bandwidth available in a PIM node. The WideWord unit has the ability to change operand width on a per-instruction basis, enabling it to treat a WideWord operand as a packed array of objects of 8, 16, or 32 bits in size. With the exception of a few specialized instructions, this characteristic means the WideWord ALU is more generally represented as a number of parallel ALUs, where the number depends on operand size.

Besides conventional arithmetic and logic operations, the WideWord unit also supports a rich set of operations for manipulating data, including rearrangement of data within a WideWord operand, transfers between WideWord and scalar registers and packing and unpacking operations. Furthermore, the WideWord unit supports selective execution of instructions on a per-datapath basis, depending on the state of condition codes. The generality of these three features, as well as the ability to access main memory at very low latency, distinguish the DIVA WideWord capabilities from multimedia ISA extensions such as Intel SSE2 and PowerPC AltiVec, as well as subword parallelism approaches such as MAX [19]. We now discuss each of these in detail, and show examples of their use. In the examples, we use the convention that WideWord instructions and references to WideWord registers are both prepended with a 'w'.

Permutation. To rapidly align and reorganize data in WideWord registers, the WideWord unit has a permutation functional unit, which enables any 8-bit field of the source register to be moved into any 8-bit field of the destination register. A permutation is specified by a permutation vector, which contains 32 indices corresponding to the 32 8-bit subfields of a WideWord destination register, where each index selects which subfield of the source data is moved into that destination field. General permutations are specified such as **`wprm wro,wri,wrp`**, where *wrp* specifies the desired permutation vector to be applied to input *wri*, with the output in *wro*. *Wrp* is either constructed through a series of instructions or is loaded from memory. To bypass the cost of constructing or loading general permutations, commonly used permutations are instead specified such as **`wprmi wro,wri,sr`**, where *sr* is a scalar register that contains an index into a table of hard-wired permutations, such as shifts, rotates, shuffles, gathers, scatters and reductions.

Figure 6 illustrates the use of permute operations with an example of a reduction sum of the elements of an array (loaded into $wr1$). The reduction sum is performed in $log(n)$ steps, where n is the number of elements in $wr1$ (in the examples, $n = 4$, for simplicity). On each step, the first permutation swaps each even-numbered field $2i$ with its odd-numbered neighbor field $2i + 1$, $0 \leq i < n/2$, storing the result in $wr2$. Then the contents of $wr1$ and $wr2$ are added, resulting in the sum of each pair of even-/odd-numbered elements in each even-numbered field of $wr1$. Finally, all even-numbered partial sums are permuted into the lower half of $wr1$, reducing the problem size by half. After the last step, the sum of all elements is in field zero of $wr1$.

```
// sr1 refers to permutation (1,0,3,2)
// sr2 refers to permutation (0,2,1,3)
wld wr1,&array);       // wr1 = (a0,a1,a2,a3)
// step 1:
wprmi wr2,wr1,sr1;     // wr2 = (a1,a0,a3,a2)
wadd wr1,wr1,wr2;      // wr1 = (a0+a1,a0+a1,a2+a3,a2+a3)
wprmi wr1,wr1,sr2;     // wr1 = (a0+a1,a2+a3,*,*)
// step 2:
wprmi wr2,wr1,sr1;     // wr2 = (a2+a3,a0+a1,*,*)
wadd wr1,wr1,wr2;      // wr1 = (a0+a1+a2+a3,*,*,*)
```

Figure 6: Reduction sum using permutations.

Register Transfers. To enable efficient data movement between scalar and WideWord register files, the WideWord unit supports a set of transfer instructions. The transfer instructions include **`mvswr wr,sr`**, which replicates the contents of scalar register *sr* into all fields of WideWord register *wr*, **`mvsw wr,sr,i`**, which copies *sr* only to the immediate *i* field of *wr*, and **`mvws sr,wr,i`**, which copies the contents of *sr* to immediate *i* field of *wr*.

Figure 7 illustrates the use of transfer instructions in a mixed irregular-regular computation where it is advantageous to perform the regular computation in the WideWord unit and the irregular in the scalar ALU. In this example, the multiplication of $A[k] : A[k+3]$ by X can be performed in the WideWord unit, since array A is accessed with stride one. To allow the vector-scalar multiplication to be performed in parallel, X is replicated into a WideWord register. It is not profitable to perform the addition of $Y[R[k]]$ and $A[k] * X$ in the WideWord unit, since it would be necessary to pack $Y[R[k]]$ to $Y[R[k + 3]]$ into a WideWord register and check whether $R[k] : R[k+3]$ are distinct values. Nevertheless, the computation of the addresses (base address of Y plus offsets $R[k] : R[k + 3]$) can still be performed in parallel, as shown in the example. Finally, the operands are moved to scalar registers and the additions are performed in the scalar unit.

Selective execution. Selective execution is supported for most arithmetic and logic instructions, permutation instructions and some transfer instructions. Under selective execution, only the results corresponding to participating subfields are written back to the destination register specified in the instruction. Therefore, the implementation of selective execution requires that writeback enable bits be associated with each 8-bit subfield of the ALU result, which complicates the register forwarding logic somewhat. The determination of whether a subfield participates in the execution of a given instruction is derived from condition codes, two special-purpose registers, and a field in the instruction. One special-purpose register is a user-settable 32-bit *mask register*, where each bit corresponds to an 8-bit subfield of the operation. The *participation mode register* is a 5-bit register that specifies the condition for selective execution as a combination of condition codes and/or mask register. A 2-bit participation field in the instruction specifies one of four possible extents of selective execution: local participation, where a subfield participates if its local condition (as derived from the participation mode and mask register values and condition codes) is true; leftmost/rightmost participation, where only the leftmost/rightmost subfield with a condition that is true participates; and always participate, where

```
// Original loop:
// for (k = 0; k < N; k ++)
//     Y[R[k]] = Y[R[k]] + A[k] * X

// sr1 = base address of Y (&Y)
// sr2 = address of A[k] (&A[k])
// sr3 = address of X

// compute A[k]*X:A[k+3]*X in WideWord
wld wr4,sr2;             // wr4 = (A[k]:A[k+3])
ld sr4,sr3;              // sr4 = X
mvswr wr5,sr4;           // wr5 = (X,X,X,X)
wmul wr6,wr4,wr5;        // wr6 = (A[k]*X:A[k+3]*X)

// compute addresses &Y[R[k]]:&Y[R[k+3]] in WideWord
mvswr wr1,sr1;           // wr1 = (&Y,&Y,&Y,&Y)
wld wr2,&R[k];           // wr2 = (R[k]:R[k+3])
wsll wr2,wr2,2;          // convert to address offset
wadd wr3,wr1,wr2;        // wr3 = (&Y[R[k]]:&Y[R[k+3]])

// compute Y[R[k]]+A[k]*X in scalar unit
mvws sr10,wr3,0;         // sr10 = &Y[R[k]]
ld sr11,sr10;            // sr11 = Y[R[k]]
mvws sr12,wr4,0;         // sr12 = A[k]*X
add sr13,sr11,sr12;      // sr13 = Y[R[k]] + A[k]*X
st sr10,sr13;            // Y[R[k]] = Y[R[k]] + A[k]*X

// compute Y[R[k+1]]+A[k+1]*X in scalar unit
...
```

Figure 7: Irregular computation using both WideWord and scalar ALUs.

all subfields participate. Although similar designs support some type of conditional operations, the DIVA WideWord unit provides a much richer functionality through the ability to specify selective execution in almost every wide instruction and the use of global condition code information in selection decisions.

This distinction is illustrated in Figure 8. The instruction **wsubcc** subtracts X from elements of array C and sets the condition code for each 32-bit field. Then the subsequent instruction **waddlc**, where **lc** specifies local participation, performs an addition only on those fields for which the GT condition code is set.

```
// Original loop:
// for (k = 0; k < N; k ++)
//    if (C[k] > X)
//       A[k] = A[k] + B[k]

// set participation mode register (PM)
ori r2, r0, "GT"
mtspr PM, r2

wld wr1, &A;             // wr1 = (a0,a1,a2,a3)
wld wr2, &B;             // wr1 = (b0,b1,b2,b3)
wld wr3, &C;             // wr1 = (c0,c1,c2,c3)
ld r1, &X;               // r1 = X
wmvswr wr4,r1;           // wr4 = (X,X,X,X)
wsubcc wr5,wr3,wr4;      // wr5 = (c0-X,c1-X,c2-X,c3-X)
waddlc wr1, wr1, wr2;    // if (C > X) A = A+B
```

Figure 8: Selective update example.

3.3 Parcel Interconnect

Even for applications where the WideWord instructions are not applicable, the WideWord datapath is used to accelerate all parcel communication, as will be discussed here. As described earlier, the PIM Routing Component (PiRC) not only implements the PIM-to-PIM interconnect but also interacts with parcel buffers (PBUFs), the basic on-chip hardware mechanisms supporting parcels. The PBUF has a virtual as well as a physical abstraction. To the application, the PBUF locations appear as regular memory locations that are manipulated through simple loads and stores. At a physical level, the PBUF is a set of memory-mapped registers. Each PIM node contains a PBUF that serves as a port between the on-chip parcel interconnect and the node. Although a PIM node's PBUF could be implemented as special-purpose registers, a memory-mapped mechanism allows a uniform implementation for both node and host PBUF. The PBUF within the PIM chip host interface is memory-mapped into the host processor's address space to permit host and PIM parcel communication.

A parcel consists of a 96-bit header and 256-bit payload. Most of the parcel contents are written by the user program during a parcel launch; however, the system is responsible for generating some fields such as PiRC routing information, source node ID, process identifier, and interrupt status. The user program is responsible for specifying header fields that include the virtual address of the object to which the parcel is directed and a specification of the command to execute on that object. In addition, the user program specifies the 256-bit payload, which consists of arguments for the command task or other data associated with the action specified by the parcel.

Data is written to or read from the PBUF in 256-bit increments via the WideWord registers. The PBUF address space can then be viewed as a set of 256-bit registers. Besides the header and payload registers, there are also status and configuration registers. Although the payload is the only true physical 256-bit register, each register is allocated 256 bits of the address space and is aligned to the least significant bit boundary. At least two register sets are needed: one for sending and one for receiving. In addition, it is desirable to have multiple address mappings (aliases) of these sets to support different access privileges and modes, such as non-launching and launching writes to the send registers, destructive and non-destructive reads from the receive registers, and interrupt capability. The DIVA design includes several aliases to support such mechanisms.

3.4 Address Translation Hardware

The primary functions of the node address translation unit are to translate virtual addresses to physical addresses for those accesses which are locally resident and to provide access protection. The types of accesses generated by a DIVA PIM processor that require translation include instruction fetches and data accesses to memory or memory-mapped devices such as parcel buffers, generated by load or store instructions. Given the simplicity of the segment-based address translation scheme discussed in Section 2, very little hardware support is needed to effect efficient translation. The necessary descriptors for a local memory segment are a physical base address register, offset limit register, and access privilege control bits. For global memory segments, an additional virtual base address register is useful to effect efficient translation, as described below. The initial DIVA architecture provides eight sets of local segment registers and four sets of global segment registers. If an application requires a number of segments that is more than that supported by the translation hardware, the PIM run-time kernel must manage the configuration of the translation hardware to minimize address faults. Like pages in a conventional system, segments and their associated descriptors are generic in nature. It is only through system programming that a segment serves a specific purpose, such as representing user code or data segments.

To distinguish between local and global segments, we arbitrarily, but with little loss of generality, specify that the upper 5 bits of a virtual address generated by a PIM processor indicate the *scope* of the address. The value of the scope field determines what type of translation, if any, is used (see Figure 9). For local translation, bits 5 through 7 are used as an index value to select one of eight sets of local segment descriptors for translation and protection checking. The rest of the virtual address represents an offset from the segment base address. Unlike the table look-up style of local translation, for global translation it is more efficient to determine if the virtual address is contained within the span of a global segment. Thus, if the scope value indicates global translation, a fully-associative lookup is performed using the global segment descriptors. Also, as shown in the figure, a supervisor-level untranslated region that spans the exception handler addresses has been reserved. This feature is useful for kernel code to run diagnostics, such as verifying the operation of the address translation hardware without being incapacitated by related hardware errors.

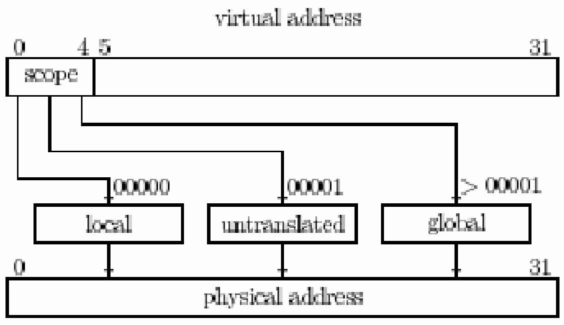


Figure 9: Address translation in DIVA PIMs.

4. EXPERIMENTAL RESULTS

4.1 Applications

To measure the performance potential of the DIVA architecture, we examine in detail eight benchmark applications, summarized in Table 1. These applications span a broad range of domains including scientific computing, databases and image processing. They exhibit both coarse-grain parallelism (which allows computation to be spread across PIMs) and, in some cases, fine-grain parallelism (which can be exploited through execution in the WideWord unit). CG, Neighborhood, Pointer, OO7 and Natural Join exhibit irregular or mixed (regular and irregular) data access patterns, resulting in high memory access overheads on conventional architectures. Cornerturn, Transitive Closure and Template Matching are dense matrix computations with regular access patterns, but memory bandwidth becomes a limiting factor in exploiting available parallelism. These three and CG rely on the WideWord unit to exploit parallelism and PIM bandwidths. Hereon, we use abbreviations for each of the program names, with a suffix -H for host and -P for PIM.

4.2 Simulation Environment and Parameters

To evaluate the DIVA architecture, we developed a system simulator called DSIM, which uses RSIM as a framework, with significant extensions [24]. RSIM is an event-driven simulator that models shared-memory multiprocessors built with state-of-the-art multiple issue, out-of-order superscalar processors. DSIM extensions include a simpler PIM processor with a WideWord unit, the DIVA memory system, the parcel communication mechanism and the PIM-to-PIM interconnect. DSIM supports the full DIVA PIM ISA.

The DSIM host processor is taken directly from RSIM, including the first and second-level caches. The host processor architecture is based on the MIPS R10000 and is configured as a four-issue processor with two integer arithmetic units, two floating-point units and one address unit. Loads are non-blocking. It has a 32Kbyte L1 and a 1Mbyte L2 cache, both two-way associative, with access times of 1 and 10 cycles, respectively. Both L1 and L2 caches are pipelined and support multiple outstanding requests.

The host is connected to the DIVA memory system via a split-transaction, 64-bit bus. The memory system consists of the aggregation of all PIM memories, where each local memory is visible from both host and local PIM processor. DSIM maintains the current open row of each memory bank to determine the memory access type (page or random mode) and simulates arbitration between host and PIM accesses, as described in Section 3.1. The memory latencies seen by the host are 52 cycles for page-mode accesses and 60 cycles for random mode, and include the bus transfer delay, the memory arbitration time and the DRAM access time (4 and 12 cycles for page and random mode, respectively). The memory latencies seen by the local PIM processor, including arbitration and DRAM access times, are 5 and 13 cycles for page and random mode accesses, respectively.

An application library supports a cache-line flush to enforce coherence between the host caches and PIM memory, as well as synchronization and communication functions. These functions are linked with the application, and their execution is simulated by DSIM in the same way as the application code. DSIM also models the parcel mechanism and the PIM-to-PIM interconnect in detail, but we omit further description since this paper focuses on 1-PIM performance.

For these experiments, we make the conservative assumption that the PIM processor runs at half the speed of the host processor. Although the inherent speed of the logic is no slower [13], we make this assumption because the subcomponents of the PIM processing logic run in lock-step, so the resulting clock speed is slower than that of superscalar schemes.

4.3 Performance Compared Against Host

Figure 10 summarizes 1-PIM performance as compared to execution on the conventional host processor. Five of the eight programs speed up significantly compared against host execution, two remain about the same, and one program is slowed down. (All programs speed up when multiple PIMs are used.) Overall, the average speedup is 3.3X. Several factors contribute to these speedups, including the lower memory stall times on the PIM nodes and the benefits of the WideWord unit in exploiting fine-grain parallelism

Program	LOC	Description	Source	Data Set Size	WideWord Usage
Template Matching (TM)	815 (C)	image correlation	Sandia	4-Kbyte image, 32 1-Kbyte templates	parallelism, selective, reuse in registers, page mode
Cornerturn (CT)	177 (C)	matrix transpose	Atlantic Aerospace	32-Mbyte matrix	parallelism, permutation
CG	857 (FORTRAN)	sparse conjugate gradient	NAS	2M double precision elements	parallelism, floating point, page mode
Transitive Closure (TC)	202 (C)	Floyd's all pairs shortest paths	Atlantic Aerospace	256 Kbytes	parallelism, selective, reuse in registers
Natural Join (NJ)	13144 (C)	relational database join	Alphatech	72 Kbytes	
Neighborhood (NH)	2098 (C)	image processing stencil	Atlantic Aerospace	500,000 bytes	
Pointer (P)	252 (C)	random walk	Atlantic Aerospace	4 Mbytes	
OO7	8000 (C++)	object-oriented database query	University of Wisconsin	888 Kbytes	

Table 1: Application description.

and taking advantage of page-mode memory accesses. The remainder of this section examines these factors in detail.

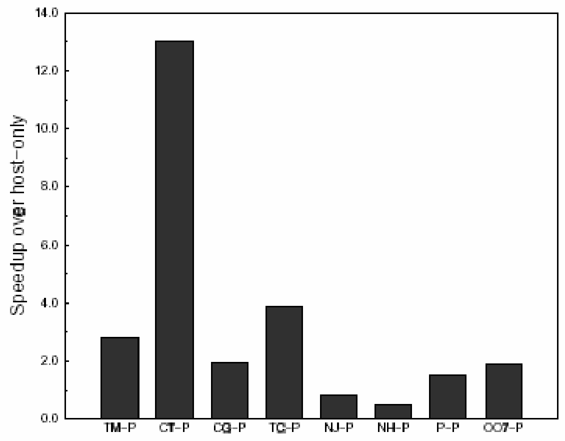


Figure 10: Speedups over host-only execution.

4.4 Reduction in Memory Stall Time

Figure 11(a) shows the memory stall times of host-only execution. PIMs reduce memory stall time in two ways: (1) lower latency to memory; and, (2) higher bandwidth to memory through wide loads and stores. (A third reduction occurs as a result of coarse-grain parallelism across the PIMs, which is not discussed in this paper.) We see from the figure that five of the eight programs spend more than 40% of their time stalled in memory accesses. DIVA achieves a reduction in memory stall time for these five programs ranging from 13.89% for Natural Join to 95% for Cornerturn, as shown in Figure 11(b).

The host version of Template Matching (TM-H) has a memory stall time of only 3% of its total execution time. The data set size of this application fits in the L2 cache, and the working set of each loop fits in the L1 cache; therefore, the data reuse exhibited by TM is effectively exploited. Even though TM-H does not suffer from large memory stall times, the 1-PIM version (TM-P) has even smaller stall times due to the high data bandwidth at the PIM node. The use of the WideWord unit for loading/storing and operating on 256-bit objects, plus the reuse of data in WideWord registers reduces the memory stall time to 20% of that of TM-H.

Cornerturn has a memory stall time of 90.17% when running on the host. This application has very little temporal reuse, since each matrix element is accessed only twice (one read and one write) during the matrix transpose. Thus primarily spatial reuse is exploited in cache, and each new cache line is only reused a few times. In the PIM version, the WideWord datapaths also exploit the available spatial reuse. Furthermore, the WideWord loads/stores and operations on 8 matrix elements at a time also reduce the number of accesses to memory. Finally, the latency seen by the PIM processor (average of 11.57 cycles, since most of the accesses are in random mode) is much lower than that suffered by the host. The combination of these factors reduce the CT-P memory stall time to 4.32% of that of CT-H.

CG also benefits from the lower memory latencies on the PIM node. Since the data set size does not fit in the host caches and the irregular access patterns cause conflict misses, CG-H spends 85.21% of its execution time stalled due to cache misses. Although most of the misses are satisfied at the L2 cache (51.32%), 46% of the stall time is due to accesses to the DRAM. On the PIM, 78% of the memory accesses are page-mode accesses, and the average latency seen by the processor is only 5.91 cycles.

Transitive Closure on the host, TC-H, spends 70% of its execution time stalled due to cache misses, with 47.14% of the misses satisfied at the L1 and 52.81% satisfied at the L2, resulting in an average miss latency of 6.23 cycles. For TC-P, the average memory latency is 5.57 cycles, due to 67% of page-mode accesses. In addition to lower memory latencies, TC-P also has a smaller number of memory accesses since the WideWord unit is used to transfer the data to/from memory and perform the computation. The use of the WideWord unit results in the added benefit of exploiting spatial reuse, since the matrix is accessed with stride one in the row dimension.

Neighborhood shows an increase in memory stall time because the data fits in cache, and thus the memory latency at the PIM is larger than that of the host. The increase in memory stall time plus the fact that the PIM processor runs at half the speed of the host result in a slowdown with respect to host-only execution.

Pointer has no spatial reuse and little temporal reuse, and since the data set size is larger than the L2 cache, P-H stalls for memory for 49.8% of its execution time, with most misses satisfied at the DRAM. P-P has roughly the same number of loads and stores, but the average latency seen by the PIM is much smaller than the memory latency suffered by the host, even though most of the PIM accesses are random-mode accesses.

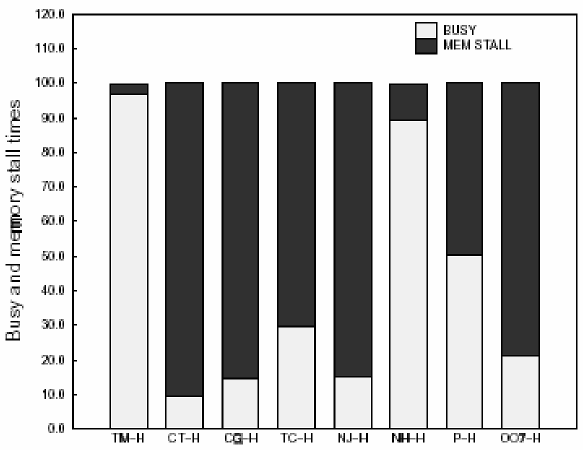
Natural Join exhibits little temporal reuse and high cache miss rates, even though the data set size fits in the L2 cache. NJ-P shows a reduction of 13.8% in memory stall times due to the lower average latency seen by the PIM processor. OO7 also has almost no temporal reuse and OO7-H suffers from a large amount of cache misses. On the PIM version the memory stall time is reduced by 62.8%, again as a result of the smaller on-chip latency.

4.5 Benefits from WideWord and Page Mode

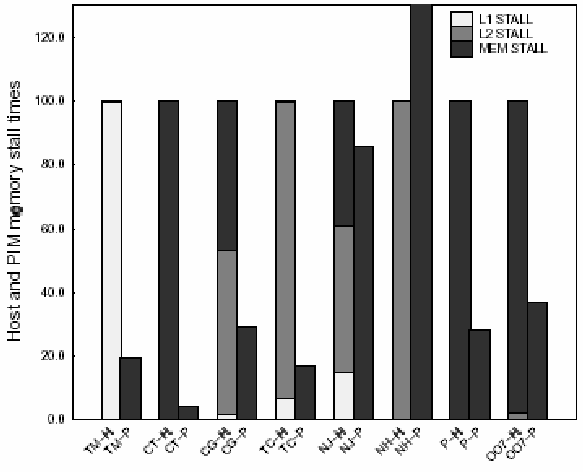
To isolate the benefit of the WideWord unit, we compare scalar versions against versions tuned to take advantage of the WideWord unit and page-mode memory accesses for the four programs that utilize the wide datapaths. These results are shown in Figure 12. Speedups are significant, ranging from 1.19X for CG up to 17.96X for TM, with an average improvement of 9.93X. The features of the instruction set that are exploited are summarized in the final column of Table 1, and described as follows.

TM computes three correlation values between an image and each of 32 templates, each correlation corresponding to a loop nest. The DIVA implementation, which is described in detail in [6], takes advantage of the inherent fine-grain parallelism by operating on 32 8-bit image pixels and 32 8-bit template elements at a time. Since a template is represented as a 32-by-32 matrix of 8-bit elements, an entire template row fits into one WideWord register. Also, since the innermost loop traverses one template row, the entire inner loop computation is transformed into a sequence of WideWord operations on one template row and 32 pixels of an image row, effectively eliminating the innermost loop. The accumulation of the pixel values is achieved by a parallel reduction sum, using permutation operations as in Figure 6, and the result of the reduction sum is added to the correlation value using selective execution as in Figure 8. To exploit temporal reuse in WideWord registers, we applied common loop transformations, particularly unroll-and-jam [4]. In addition, we exploited spatial reuse by shifting an image subrow held in a WideWord register by one pixel, to move the window of the image to be compared against the template. Further performance improvements are obtained by reordering memory accesses and grouping streaming accesses to the dense arrays to achieve page-mode memory access latencies.

The CT implementation performs a hierarchical in-place matrix transpose where the smallest submatrices, of size 8x8, are transposed in WideWord registers. Each 8x8 sub-



a) Busy and memory stall times for host-only execution.



b) Host-only and 1-PIM memory stall times.

Figure 11: Memory stall times.

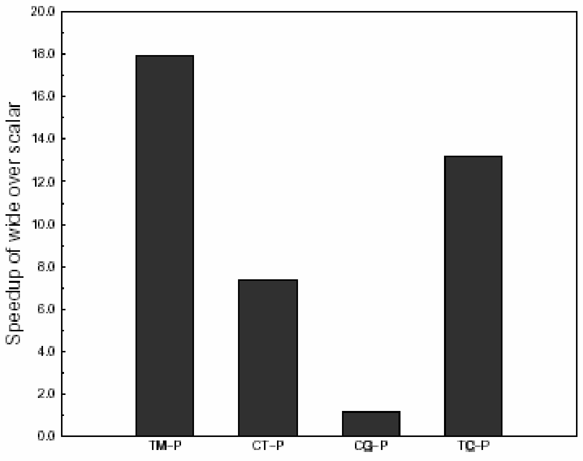


Figure 12: Speedup of WideWord vs. scalar.

matrix is loaded into the WideWord register file (an 8x8 matrix with 32-bit elements requiring 8 WideWord registers), and transposed via a sequence of permutation operations. The transposed submatrix is then stored back in memory. This implementation takes advantage of the large capacity of the WideWord register file, avoiding loads and stores to memory during the transpose of each 8x8 submatrix.

CG's key computation is a sparse matrix-vector multiply. Due to the mixed regular/irregular nature of data accesses, we only exploit fine-grain parallelism in the WideWord unit for the regular portions of the computation. The dense vector accesses are loaded into WideWord registers, and the dense vector multiplies are performed in the WideWord floating-point unit. The accumulates into the sparse matrix are performed sequentially. Selective execution is used to select the field of the WideWord operand that participates in the operation. As in TM, we also reordered memory accesses to achieve page-mode latencies on the dense arrays.

TC uses a dense matrix to represent the distance graph. It exploits fine-grain parallelism by performing WideWord arithmetic operations on eight 32-bit elements of the matrix that are held in WideWord registers. Selective execution using WideWord operation **wmrgcc** merges the contents of two WideWord registers according to condition-code bits, allowing an efficient computation of the minimum value of each pair of elements of two WideWord operands. Similar to TM, we use unroll-and-jam to obtain temporal reuse in the WideWord register file.

5. STATUS

The first DIVA PIM prototype, shown in Figure 1, is an SRAM-based single-node implementation of the DIVA PIM chip architecture and is currently in test. To minimize silicon area of this SRAM-based prototype, we used a 1-Mbyte memory macro. For comparison, a DRAM-based implementation with a 2-Mbyte macro could be fabricated in approximately half the area of the SRAM-based prototype. The current prototype chip implements all features of the DIVA PIM architecture except address translation and floating-point capabilities. A second version of a PIM chip, which not only integrates these functions but achieves a faster clock rate, is due to tape out in the second half of 2002.

The current chip was fabricated through MOSIS in TSMC 0.18μm technology, and the silicon die measures 9.8mm on a side. It contains approximately 2 million logic transistors in addition to the 53 million transistors that implement 8 Mbits of SRAM. The chip also contains 352 pads, 240 signal I/O, and is packaged in a 35mm BGA. Much of the logic was synthesized with Synopsys Design Analyzer, and the entire chip was placed and routed with Cadence Silicon Ensemble. The IP building blocks used in the chip include Artisan standard cells and register files, Virage Logic SRAM, and a NurLogic PLL clock multiplier.

The chip is currently being tested for functionality with the use of pattern generators, which apply test vectors to input pins, and logic analyzer modules, which sense the outputs. Although exhaustive testing has not yet been completed, the chip is correctly executing at 160MHz on the Cornerturn matrix transpose kernel described in Section 4, exercising all major control and datapaths within the PIM processing logic, including the WideWord permutation unit. Even in this limited test setup, the chip performs 1.28 GOPS while dissipating only 800mW. In addition to the processing logic functionality, correct operation of parcels transiting through the PiRC has also been verified.

We will soon begin integrating PIM-based DIMMs into a workstation-class development system, incorporating compiler and system software technology. For the host operating system, we have augmented Linux to include PIM-specific support, such as loading PIM code and data, booting, process management, and the memory management functions outlined in [10]. We have also developed components of the PIM run-time kernel, an augmented version of the RTEMS open-source real-time embedded operating system. We have developed a prototype compiler for the DIVA PIMs, which takes as input sequential Fortran or C code, and produces DIVA executables that exploit both the scalar and WideWord unit. We leverage the SUIF compiler, including extensions described in [18] and our own implementation of transformations described in [6], and a GCC backend for the PowerPC AltiVec. A system-level compiler is an area of future work.

6. RELATED WORK

The DIVA system architecture is focused on achieving the following four goals: (1) developing PIMs that can serve as the only memory in the system, assuming the dual roles of "smart memories" and conventional memory; (2) supporting a wide range of familiar programming paradigms, closely related to parallel computing; (3) targeting applications that are severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets; and, (4) developing a VLSI device to exploit memory and communications bandwidth in PIM-based systems while making efficient use of on-chip resources for target applications.

These four goals distinguish DIVA from other PIM-based architectures. Integration into a conventional system affords the simultaneous benefits of PIM technology and a state-of-the-art host, yielding high performance for mixed workloads. Since PIM processors are usually less sophisticated due to on-chip space constraints, systems using PIMs alone in a multiprocessor may sacrifice performance on uniprocessor computations [12, 16, 25, 27], while system-on-a-chip solutions (e.g., the IRAM [22] and the Mitsubishi M32R/D [20]) limit the application domain. DIVA's support for a broad range of familiar parallel programming paradigms, including task parallelism for irregular computations, distinguishes it from systems with restricted applicability (such as to SIMD parallelism [7, 8, 22]), as well as those requiring a novel programming methodology or compiler technology to configure logic [1], or to manage a complex memory, computation and communication hierarchy [15]. DIVA's PIM-to-PIM interconnect improves upon approaches that serialize communication through the host, which decreases bandwidth by adding traffic to the processor-memory bus [8, 21].

With respect to DIVA's WideWord unit, Table 2 compares the features described in Section 3.2 with two commercial multimedia extensions that support superword parallelism, PowerPC AltiVec and Intel SSE2, as well as a previous research design called ASAP [2]. (Most other multimedia extensions support *subword* parallelism, which performs parallel operations on subfields of a machine word.) The ASAP combines WideWord and scalar capabilities in a single unit. This approach eliminates the need for trans-

Capability	SSE2	AltiVec	ASAP	DIVA
Separate scalar, WideWord units	✓	✓	×	✓
Permutation	immediate	general	general	general, indirect
Register transfers	✓	×	n/a	✓
Selective execution	limited	limited	✓	✓

Table 2: Comparison with other superword-level parallelism approaches.

fers between register files, but with register forwarding, it can complicate the pipeline and slow down the clock rate. All other implementations have separate scalar and WideWord units and register files, and other than DIVA, only SSE2 includes transfers between register files. The absence of such capability was reported to be a performance bottleneck in the AltiVec [18]. AltiVec and ASAP support only general permutations, where permutation vectors are read from memory or constructed by instructions. Both SSE2 and DIVA can avoid these costs of deriving a permutation vector through hardwired permutation operations. In the case of SSE2, permutation operations can only be expressed through immediates, so the permutation must be known at compile time. DIVA's hardwired permutation, which is in addition to general permutation, is indirect because it references a scalar register. Hardwired indirect permutations are more powerful than immediate permutations, in that we can use nearby permutations for different iterations of a loop without requiring unrolling (*e.g.*, to do alignment). DIVA provides a detailed reference design and implementation of selective execution, related to the concept discussed in [2], that supports selective execution in almost every wide instruction. By comparison, since the AltiVec does not incorporate selective execution of arithmetic operations, to accomplish the same result as in Figure 8 on the AltiVec would require an additional instruction to commit only those fields of the result of the add for which the condition code is set.

We further consider a performance comparison with the PowerPC AltiVec 74XX. Even with a very aggressive DRAM technology, the 74XX can achieve a peak main memory bandwidth which is only one third that of the PIM DRAM. While the 74XX has better bandwidth for problems which fit into the 256KB on-chip L2 cache, for our benchmarks with high memory stall times, a single DIVA PIM processor will outperform the AltiVec despite a much smaller transistor count on a DIVA PIM. Further, since each DIVA system will include many interconnected PIM chips, the performance advantage will scale with increasing memory size for problems amenable to coarse-grain parallel computation.

7. CONCLUSION

This paper has presented a detailed description of the DIVA PIM microarchitecture. We discuss some of the issues that must be considered in future architectures for exploiting memory bandwidth, particularly the memory interface and controller, instruction set features for fine-grain parallel operations, and mechanisms for address translation. We present simulation results on eight programs, demonstrating an average speedup of 3.3X as compared to a conventional host. The speedups are due to up to 95% reduction in memory stall time, and, for four of the programs, an average speedup of 9.93X due to fine-grain parallelism in the WideWord unit as compared to scalar PIM execution. As a result of these effects, six of the programs show fairly significant speedups over host-only execution with just one PIM, even though the PIM processor is an in-order, single-issue processor running at half the speed of the host, which is an out-of-order 4-issue processor. These 1-PIM speedups suggest DIVA's potential to outperform conventional multiprocessors for certain applications, and at a much reduced hardware cost.

Acknowledgments

The authors wish to thank Peter Kogge and Jay Brockman at the University of Notre Dame for their contributions to the design of the DIVA PIM microarchitecture, and Thomas Sterling for his early contributions to the project. The authors also wish to acknowledge the work of others previously on the DIVA team at USC/ISI who contributed in some way to the work described in this paper, including Jeff Koller, Michael Gorman and Ruoming Pang. The DIVA project is sponsored by DARPA contract F30602-98-2-0180.

8. REFERENCES

[1] J. Babb et al. Parallelizing applications into silicon. In *Proceedings of the IEEE Workshop on FPGAs for Custom Computing Machines*, Apr. 1999.

[2] J. Brockman et al. Microservers: A new memory semantics for massively parallel computing. In *Proceedings of the ACM International Conference on Supercomputing*, pages 454–463, June 1999.

[3] D. Burger, J. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[5] J. Carter et al. Impulse: Building a smarter memory controller. In *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pages 70–79, Jan. 1999.

[6] J. Chame, M. Hall, and J. Shin. Code transformations for exploiting bandwidth in PIM-based systems. In *Proceedings of the ISCA Workshop on Solving the Memory Wall Problem*, June 2000.

[7] D. Elliott et al. Computational RAM: Implementing processors in memory. *IEEE Design and Test of Computers*, pages 32–41, January – March 1999.

[8] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *IEEE Computer*, pages 23–31, Apr. 1995.

[9] M. Hall et al. Mapping irregular applications to DIVA, a PIM-based Data-Intensive Architecture. In *Proceedings of Supercomputing*, Nov. 1999.

[10] M. Hall and C. Steele. Memory management in a PIM-based architecture. In *Proceedings of the ASPLOS Workshop on Intelligent Memory Systems*, Nov. 2000.

[11] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 2 edition, 1996.

[12] IBM. http://researchweb.watson.ibm.com/bluegene/.

[13] IBM Microelectronics. http://www.chips.ibm.com/ products/asics/products/edram.

[14] C. W. Kang and J. Draper. A fast, simple router for the Data-Intensive Architecture (DIVA) system. In *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, Aug. 2000.

[15] Y. Kang et al. FlexRAM: Toward an advanced intelligent memory system. In *Proceedings of the IEEE International Conference on Computer Design*, Oct. 1999.

[16] P. Kogge. The EXECUBE approach to massively parallel processing. In *Proceedings of the International Conference on Parallel Processing*, Aug. 1994.

[17] P. Kogge, T. Giambra, and H. Sasnowitz. RTAIS: An embedded parallel processor for real-time decision aiding. In *Proceedings of NAECON*, Mar. 1995.

[18] S. Larsen and S. Amarasinghe. Exploiting superword-level parallelism with multimedia instruction sets. In *Proceedings of the ACM Conference on Programming Languages Design and Implementation*, 2000.

[19] R. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, Aug. 1996.

[20] Mitsubishi. http://www.mitsubishi-chips.com/data/datasheets/mcus/m32rdgrp.html.

[21] M. Oskin, F. T. Chong, and T. Sherwood. Active pages: A model of computation for intelligent memory. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[22] D. Patterson et al. A case for intelligent DRAM: IRAM. *IEEE Micro*, Apr. 1997.

[23] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[24] Rice University. http://www-ece.rice.edu/ rsim.

[25] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[26] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *Proceedings of the Symposium on High-Performance Computer Architecture*, Dec. 1995.

[27] T. Sterling. An introduction to the Gilgamesh PIM architecture. In *Euro-Par*, pages 16–32, Aug. 2001.

[28] T. Sunaga et al. A processor in memory chip for massively parallel embedded applications. *IEEE Journal of Solid State Circuits*, pages 1556–1559, Oct. 1996.

[29] T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[30] J. Zawodny, P. Kogge, J. Brockman, and E. Johnson. Cache-in-memory: A lower power alternative. In *Proceedings of the ISCA Workshop on Power-Driven Microarchitecture*, June 1998.

Implementation of a 32-bit RISC Processor for the Data-Intensive Architecture Processing-In-Memory Chip

Jeffrey Draper, Jeff Sondeen, Sumit Mediratta, Ihn Kim
University of Southern California Information Sciences Institute
draper@isi.edu, sondeen@isi.edu, sumitm@isi.edu, ihnk@usc.edu

Abstract

The Data-Intensive Architecture (DIVA) system employs Processing-In-Memory (PIM) chips as smart-memory coprocessors to a microprocessor. This architecture exploits inherent memory bandwidth both on chip and across the system to target several classes of bandwidth-limited applications, including multimedia applications and pointer-based and sparse-matrix computations. The DIVA project is building a prototype workstation-class system using PIM chips in place of standard DRAMs to demonstrate these concepts. We have recently completed initial testing of the first version of the prototype PIM device.

A key component of this architecture is the scalar processor that coordinates all activity within a PIM node. Since such a component is present in each PIM node, we exploit parallelism to achieve significant speedups rather than relying on costly, high-performance processor design. The resulting scalar processor is then an in-order 32-bit RISC microcontroller that is extremely area-efficient. This paper details the design and implementation of this scalar processor in TSMC 0.18μm technology. In conjunction with other publications, this paper demonstrates that impressive gains can be achieved with very little "smart" logic added to memory devices.

1 Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 50–60% per year while memory access times improve at merely 5–7%. Furthermore, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [2]. A recent VLSI technology trend, embedded DRAM, offers a promising solution to bridging the processor-memory gap [9]. One application of this technology integrates logic with high-density memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (with hundreds of gigabit/second aggregate bandwidth available on a chip—up to 2 orders of magnitude over conventional DRAM systems). Latency to on-chip logic is also reduced, down to as little as one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data-Intensive Architecture (DIVA) project leverages PIM technology to replace or augment the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. System bandwidth limitations are thus overcome in three ways: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processor-memory nodes per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect that allows PIM chips to communicate without interfering with host memory bus traffic. Although suitable as a general-purpose computing platform, DIVA specifically targets two important classes of applications that are severely performance limited by the processor-memory bottlenecks in conventional systems: multimedia processing and applications with irregular data accesses. Multimedia applications tend to have little temporal reuse [12] but often exhibit spatial locality and both fine-grain and coarse-grain parallelism. DIVA PIMs exploit spatial locality and fine-grain parallelism by accessing and operating upon multiple words of data at a time and exploit coarse-grain parallelism by spreading independent computations across PIM nodes. Applications with irregular data accesses, such as sparse-matrix and pointer-based computations, perform poorly on conventional architectures because they tend to lack spatial locality and thus make poor use of caches. As a result, their execution is dominated by memory stalls [3]. DIVA accelerates such applications by eliminating much of the traffic between a host processor and memory; simple operations and dereferencing can be done mostly within PIM memories.

Performance evaluation of many applications has shown that a DIVA platform provides significant speedups. These results as well as thorough descriptions of system architecture issues have appeared in previous papers [5, 6, 7]. Also included in previous publications are comparisons to other PIM architectures as well as conventional architectures. This paper focuses on the microarchitecture design and implementation of the scalar processor, or microcontroller, that coordinates all activity on a DIVA PIM node. Due to area constraints, the design goal was a relatively simple processor with a coherent, well-designed instruction set, for which a gcc-like compiler is being adapted. The resulting scalar processor is a RISC processor that supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. Its novelty lies in the special-purpose functions it supports to interface to other crucial components of the DIVA design. The processor was fabricated as part of a DIVA prototype chip in TSMC 0.18μm technology and is currently in test. The remainder of the paper is organized as follows. Sections 2 and 3 present an overview of the DIVA system architecture and microarchitecture, to put the scalar processor design into its proper context. Section 4 describes the scalar processor microarchitecture in detail. Section 5 presents details of the fabrication and testing of the scalar processor as part of a PIM chip, and Section 6 concludes the paper.

2 System architecture overview

A driving principle of the DIVA system architecture is efficient use of PIM technology while requiring a smooth migration path for software. This principle demands integration of PIM features into conventional systems as seamlessly as possible. As a result, DIVA chips are designed to resemble commercial DRAMs, enabling PIM memory to be accessed by host software as if it were conventional memory. In Figure 1, we show a small set of PIMs connected to a single host processor through conventional memory control logic.

Spawning computation, gathering results, synchronizing activity, or simply accessing non-local data is accomplished via parcels. A parcel is closely related to an active message as it is a relatively lightweight communication mechanism containing a reference to

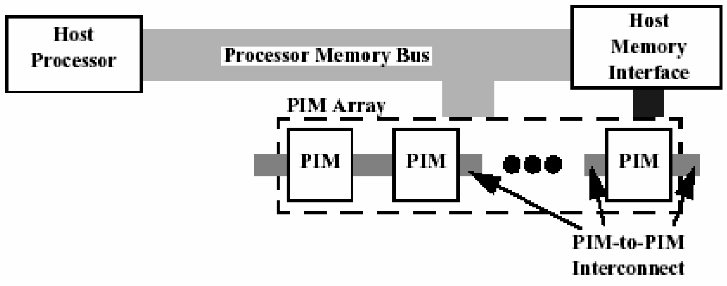
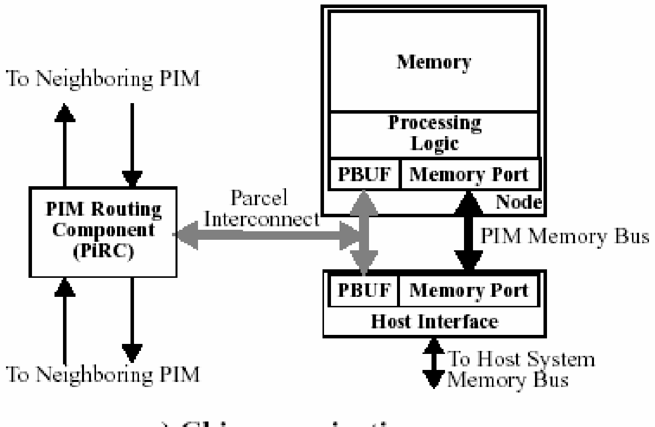


Figure 1. DIVA system architecture

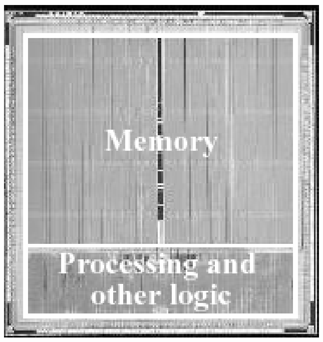
a function to be invoked when the parcel is received [14]. Parcels are distinguished from active messages in that the destination of a parcel is an object in memory, not a specific processor. From a programmer's view, parcels, together with the global address space supported in DIVA, provide a compromise between the ease of programming a shared-memory system and the architectural simplicity of pure message passing. Parcels are transmitted through a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic, as shown in Figure 1. Details of this interconnect may be found in [11], and more details of the system architecture may be found in [5, 6, 7].

3 Microarchitecture overview

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and networking/communication hardware. Although a PIM may consist of multiple nodes, each of which are primarily comprised of a few megabytes of memory and a node processor, Figure 2a shows a PIM with a single node, which reflects the focus of the initial



a) Chip organization



b) Microphotograph of die

Figure 2. DIVA PIM chip

research that is being conducted. Nodes on a PIM chip share a single PIM Routing Component (PiRC) and a host interface. The PiRC is responsible for routing parcels between on-chip parcel buffers and neighboring off-chip PiRCs. The host interface implements the

JEDEC standard SDRAM protocol [10] so that memory accesses as well as parcel activity initiated by the host appear as conventional memory accesses from the host perspective.

Figure 2a also shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. Within the host interface, a parcel buffer (PBUF) is a buffer that is memory-mapped into the host processor's address space, permitting application-level communication through parcels. Each PIM node also has a PBUF, memory-mapped into the node's local address space.

Figure 3 shows the major control and data connections within a node. The DIVA PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a 32-bit scalar datapath that performs operations similar to those of standard 32-bit integer units, and a 256-bit WideWord datapath that performs fine-grain parallel operations on 8-, 16-, or 32-bit operands. Both datapaths execute from a single instruction stream under the control of a single 5-stage DLX-like pipeline [8]. The

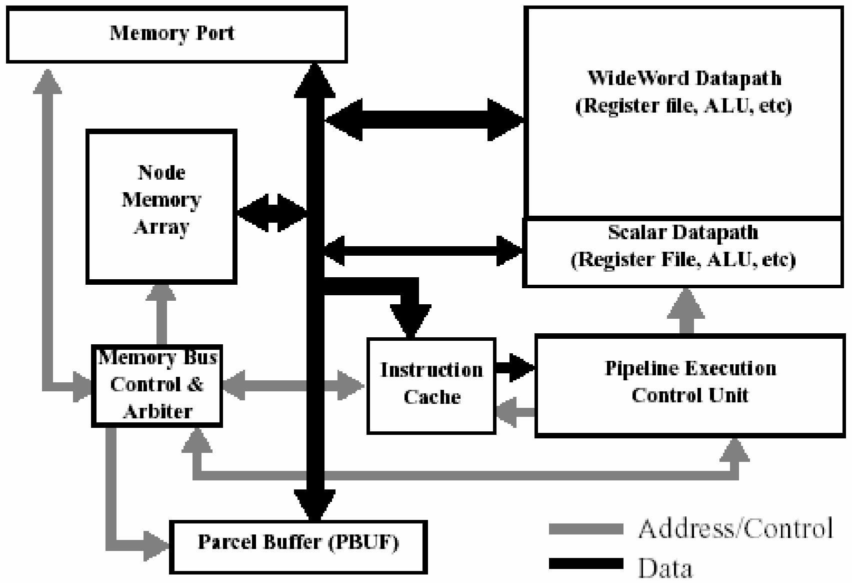

Memory Port
WideWord Datapath (Register file, ALU, etc)
Node Memory Array
Scalar Datapath (Register File, ALU, etc)
Memory Bus Control & Arbiter
Instruction Cache
Pipeline Execution Control Unit
Parcel Buffer (PBUF)
Address/Control
Data

Figure 3. DIVA PIM node architecture

instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap. Each datapath has its own independent general-purpose register file, 32 32-bit registers for the scalar datapath and 32 256-bit registers for the WideWord datapath, but special instructions permit direct transfers between datapaths without going through memory. Although not supported in the initial DIVA prototype, floating-point extensions to the WideWord unit will be provided in future systems. In addition to the execution unit and associated datapaths, each DIVA PIM node contains other essential components of note. Descriptions of these components as well as the WideWord datapath will appear in future publications.

4 Microarchitecture details of the DIVA scalar processor

The combination of the execution control unit and scalar datapath is for the most part a standard RISC processor and serves as the DIVA scalar processor, or microcontroller. It coordinates all activity within a DIVA PIM node, including SIMD-like operations in the WideWord datapath, interactions between the scalar and WideWord datapaths, and parcel communication. To avoid synchronization overhead and compiler issues associated with coprocessor designs and also design complexity associated with superscalar interlocks, the DIVA scalar processor was designed to be tightly integrated with other subcomponents, as described in the previous section. This characteristic led to a custom design rather than augmenting an off-the-shelf embedded IP core. This section describes the microarchitecture of the DIVA scalar processor by first presenting an overview of the instruction set architecture, followed by a description of the pipeline and discussion of special features.

4.1 Instruction set architecture overview

Much like the DLX architecture [8], most DIVA scalar instructions use a three-operand format to specify two source registers and a destination register, as shown in Figure 4. For these types of instructions, the opcode generally denotes a class of operations, such as arithmetic, and the function denotes a specific operation, such as add. The **C** bit indicates whether the operation performed by the instruction execution updates condition codes. In lieu of a second source register, a 16-bit immediate value may be specified. The scalar instruction set includes the typical arithmetic functions add, subtract, multiply, and divide; logical functions AND, OR, NOT, and XOR; and logical/arithmetic shift operations. In addition, there are a number of special instructions, described in Section 4.3. Load/store instructions adhere to the immediate format, where the address for the memory operation is formed by the addition of an immediate value to the contents of **rA**, which serves as a base address. The DIVA scalar processor does not support a base-plus-register addressing mode because it requires an extra read port on the register file for store operations.

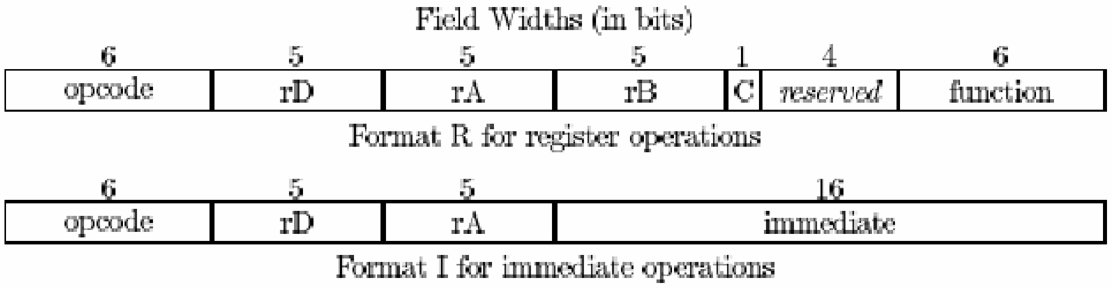


Figure 4. DIVA scalar arithmetic/logical instruction formats

Branch instructions use a different format (not shown due to page constraints). The branch target address may be PC-relative, useful for relocatable code, or calculated using a base register combined with an offset, useful with table-based branch targets. In both formats, the offset is in units of instruction words, or 4 bytes. By specifying the offset in instruction words, rather than bytes, a larger branch window results. To support function calls, the branch instruction format also includes a bit for specifying linkage, that is, whether a return instruction address should be saved in R31. The branch format also includes a

3-bit condition field to specify one of eight branch conditions: always, equal, not equal, less than, less than or equal, greater than, greater than or equal, or overflow.

4.2 Pipeline description and associated Hazards

A more detailed depiction of the pipeline execution control unit and scalar datapath are given in Figure 5. The pipeline is a standard DLX-like 5-stage pipeline [8], with the following stages: (1) instruction fetch; (2) decode and register read; (3) execute; (4) memory; and, (5) writeback. The pipeline controller contains the necessary logic to handle data, control, and structural hazards. Data hazards occur when there are read-after-write register dependences between instructions that co-exist in the pipeline. The controller and datapath contain the necessary forwarding, or bypass, logic to allow pipeline execution to proceed without stalling in most data dependence cases. The only exception to this generality involves the load instruction, where a "bubble" is inserted between the load instruction and an immediately following instruction that uses the load target register as one of its source operands. This hazard is handled with hardware interlocks, rather than exposing it, to be compatible with a previously developed compiler.

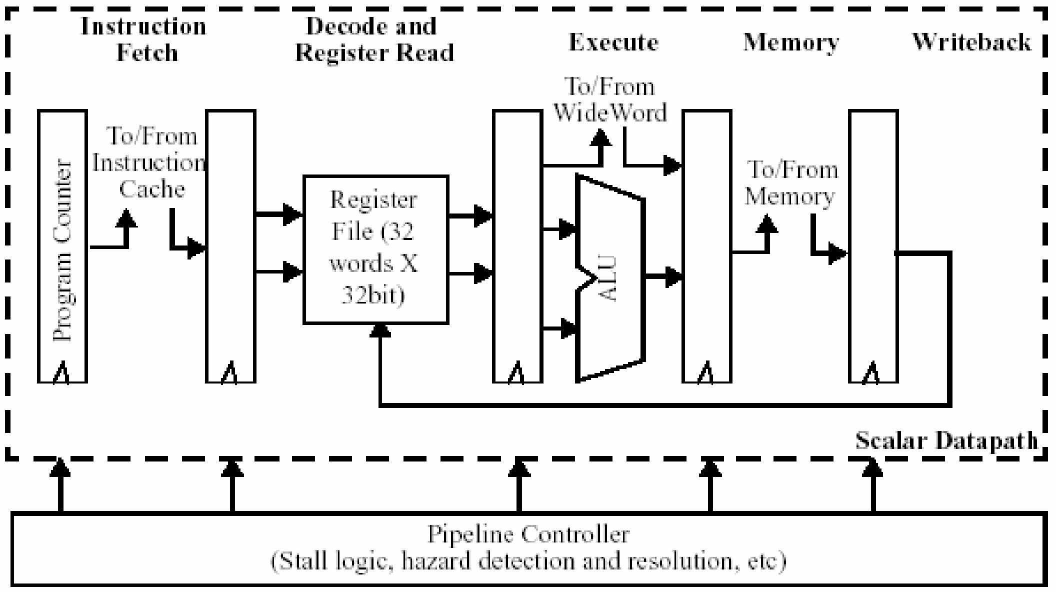


Figure 5. DIVA scalar processor pipeline description

Control hazards occur for branch instructions. Unlike the DLX architecture [8], which uses explicit comparison instructions and testing of a general-purpose register value for branching decisions, the DIVA design incorporates condition codes that may be updated by most instructions. Although a slightly more complex design, this scheme obviates the need for several comparison instructions in the instruction set and also requires one fewer instruction execution in every comparison/branch sequence. The condition codes used for branching decisions are: **EQ** - set if the result is zero, **LT** - set if the result is negative, **GT** - set if the result is positive, and **OV** - set if the operation overflows. Unlike the load data dependence hazard, which is not exposed to the compiler, the DIVA pipeline design imposes a 1-delay slot branch, so that the instruction following a branch instruction is always executed. Since branches are always resolved within the second stage of the pipeline, no stalls occur with branch instructions. The delayed branch was selected because it was compatible with a previously developed compiler.

Since the general-purpose register file contains 2 read ports and 1 write port, it may sustain two operand reads and 1 result write every clock cycle; thus, the register file design introduces no structural hazards. The only structural hazard that impacts the pipeline operation is the node memory. Pipeline stalls occur when there is an instruction cache miss. The pipeline will resume once the cache fill memory request has been satisfied. Likewise, since there is no data cache, stalls occur any time a load/store instruction reaches the memory stage of the pipeline until the memory operation is completed.

4.3 Special features

The novelty of the DIVA scalar processor lies in the special features that support DIVA-specific functions. Although by no means exhaustive, this section highlights some of the more notable capabilities.

4.3.1 Run-time kernel support

The execution control unit supports supervisor and user modes of processing and also maintains a number of special-purpose and protected registers for support of exception handling, address translation, and general OS services. Exceptions, arising from execution of node instructions, and interrupts, from other sources such as an internal timer or external component like the PBUF, are handled by a common mechanism. The exception handling scheme for DIVA has a modest hardware requirement, exporting much of the complexity to software, to maintain a flexible implementation platform. It provides an integrated mechanism for handling hardware and software exception sources and a flexible priority assignment scheme that minimizes the amount of time that exception recognition is disabled. While the hardware design allows traditional stack-based exception handlers, it also supports a non-recursive dispatching scheme that uses DIVA hardware features to allow preemption of lower-priority exception handlers.

The impact of run-time kernel support on the scalar processor design is the addition of a modest number of special-purpose and protected (or supervisor-level) registers and a non-negligible amount of complexity added to the pipeline control for entering/exiting exception handling modes cleanly. When an exception is detected by the scalar processor control unit, the logic performs a number of tasks within a single clock cycle to prepare the processor for entering an exception handler in the next clock cyle. Those tasks include:

- determining which exception to handle by prioritizing among simultaneously occurring exceptions,
- setting up shadow registers to capture critical state information, such as the processor status word register, the instruction address of the faulting instruction, the memory address if the exception is an address fault, etc,
- configuring the program counter logic to load an exception handler address on the next clock cycle, and
- setting up the processor status word register to enter supervisor mode with exception handling temporarily disabled.

Once invoked, the exception handler first stores other pieces of user state and interrogates various pieces of state hardware to determine how to proceed. Once the exception handler routine has completed, it restores user state and then executes a return-from-exception instruction, which copies the shadow register contents back into various state registers to resume processing at the point before the exception was encountered. If it is impossible to resume previous processing due to a fatal exception, the run-time kernel exception handler may choose to terminate the offending process.

4.3.2 Interaction with the WideWord datapath

There are a number of features in the scalar processor design involving communication with the WideWord datapath that greatly enhance performance. The path to/from the WideWord datapath in the execute stage of the pipeline, shown in Figure 5, facilitates the exchange of data between the scalar and WideWord datapaths without going through memory. This capability distinguishes DIVA from other architectures containing vector units, such as AltiVec [1]. This path also allows scalar register values to be used as specifiers for WideWord functions, such as indices for selecting subfields within WideWords and indices into permutation look-up tables [4]. Instead of requiring an immediate value within a WideWord instruction for specifying such indices, this register-based indexing capability enables more intelligent, efficient code design.

There are also a couple of instructions that are especially useful for enabling efficient data mining operations. ELO, encode leftmost one, and CLO, clear leftmost one, are instructions that generate a 5-bit index corresponding to the bit position of the leftmost one in a 32-bit value and clear the leftmost one in a 32-bit value, respectively. These instructions are especially useful for examining the 32-bit WideWord condition code register values, which may be transferred to scalar general-purpose registers to perform such tests. For instance, with this capability, finding and processing data items that match a specified key are accomplished in much fewer instructions than a sequence of bit masking and shifting involved in 32 bit tests, which is required with conventional processor architectures.

There are some variations of the branch/call instructions that also interact with the WideWord datapath. The **BA** (branch on all) instruction specifies that a branch is to be taken if the status of condition codes within every subfield of the WideWord datapath matches the condition specified in the BA instruction. The **BN** (branch on none) instruction specifies that a branch is to be taken if the status of condition codes within no subfield of the WideWord datapath matches the condition specified in the BN instruction. With proper code structuring around these instructions, inverse forms of these branches, such as branch on any or branch on not all, can also be effected.

4.3.3 Miscellaneous instructions

There are also several other miscellaneous instructions that add some complexity to the processor design. The probe instruction allows a user to interrogate the address translation logic to see if a global address is locally mapped. This capability allows users who wish to optimize code for performance to avoid slow, overhead-laden address translation exceptions. Also, an instruction cache invalidate instruction allows the supervisor kernel to evict user code from the cache without invalidating the entire cache and is useful in process termination cleanup procedures. Lastly, there are versions of load/store instructions that "lock" memory operations, which are useful for implementing synchronization functions, such as semaphores or barriers.

5 Implementation and testing of the DIVA scalar processor

The specification of the DIVA scalar processor required on the order of 10,000 lines of VHDL code, consisting of a mix of RTL-level behavioral and gate-level structural code. A preliminary, unoptimized stand-alone layout of the scalar processor consisted of 23,000 standard cells (approximately 200,000 transistors) and occupied 1 sq mm in 0.18μm technology. It was projected to operate at 400MHz while dissipating 80mW.

Although the scalar processor is suitable for stand-alone embedded implementations, the DIVA project employs it as part of a tightly integrated node design, as discussed in Section 3. The scalar processor VHDL specification was included as part of the DIVA PIM prototype specification, which was synthesized as a "sea of gates" using Synopsys Design Analyzer. The entire chip was placed and routed with Cadence Silicon Ensemble, and physical verification, such as DRC and LVS, was performed with Mentor Calibre. The intellectual property building blocks used in the chip include Virage Logic SRAM, a NurLogic PLL clock multiplier, and Artisan standard cells, pads, and register files.

The first DIVA PIM prototype, shown in Figure 2b, is a single-node implementation of the DIVA PIM chip architecture and is currently in test. Due to challenges in gaining access to embedded DRAM fabrication lines in a timely fashion, this first prototype is SRAM-based. This chip implements all features of the DIVA PIM architecture except address translation and floating-point capabilities. A second version of a PIM chip, which not only integrates these functions but achieves a faster clock rate, is due to tape out in the second half of 2002. The chip shown in Figure 2b was fabricated through MOSIS in TSMC 0.18μm technology, and the silicon die measures 9.8mm on a side. It contains approximately 2 million logic transistors in addition to the 53 million transistors that implement 8 Mbits of SRAM. The chip also contains 352 pads, 240 signal I/O, and is packaged in a 35mm TBGA. The chip is estimated to dissipate 2.5W at 100MHz.

The chip is being tested with the use of an HP 16702A logic analysis mainframe. Pattern generator modules apply test vectors to the inputs of the chip, and timing/state capture modules sense the outputs of the chip. The chip is currently being tested for functionality at a testbench speed of 80MHz. Although exhaustive testing has not yet been completed, the chip is running a demonstration application of matrix transpose that exercises all major control and datapaths within the scalar processor, including many of the special features highlighted in Section 4.3. Even in this limited test setup, the chip is performing 640 MOPS while dissipating only 800mW. We estimate that the scalar processor is contributing only 80mW to this power measure. Also, though at-speed testing has not been completed yet, we do not anticipate this prototype to operate much beyond 100MHz due to critical path limitations in the WideWord datapath. If implemented and optimized separately as an embedded microcontroller, we expect the scalar processor to easily operate above 500MHz.

6 Conclusion

This paper has presented the design and implementation of the scalar PIM processor used in the DIVA system, an integrated hardware and software architecture for exploiting the bandwidth of PIM-based systems. Although the core of the scalar processor design is much like a standard 32-bit RISC processor, it has a number of special features that make it well-suited to serving as a PIM node microcontroller. A working implementation of this architecture, based on TSMC 0.18μm technology, has proven the validity of the design. The resulting workstation system architecture that incorporates PIMS using this processor is projected to achieve speedups ranging from 8.8 to 38.3 over conventional workstations for a number of applications [5, 6]. These results demonstrate that by sacrificing a small amount of area for processing logic on memory chips, PIM-based systems are a viable method for combatting the memory wall problem.

Acknowledgments

The authors would like to acknowledge the support of the DIVA project team and DARPA (Contract No. F30602-98-2-0180).

References

[1] http://www.altivec.org.

[2] D. Burger, J. Goodman and A. Kagi, "Memory Bandwidth Limitations of Future Microprocessors," *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996.

[3] J.B. Carter, et al, "Impulse: Building a Smarter Memory Controller", *Proceedings of the Fifth International Symposium on High Performance Computer Architecture*, pp. 70-79, January 1999.

[4] J. Chame, M. Hall and J. Shin, "Code Transformations for Exploiting Bandwidth in PIM-Based Systems," *Proceedings of the ISCA Workshop on Solving the Memory Wall Problem*, June 2000.

[5] J. Draper, et al, "The Architecture of the DIVA Processing-in-Memory Chip", to appear at the International Conference on Supercomputing, June 2002.

[6] Mary Hall, et al, "Mapping Irregular Application to DIVA, a PIM-based Data-Intensive Architecture," *Supercomputing*, November 1999.

[7] M. Hall and C. Steele, "Memory Management in a PIM-Based Architecture," *Proceedings of the Workshop on Intelligent Memory Systems*, October 2000.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, Second Edition, 1996.

[9] S. Iyer and H. Kalter, "Embedded DRAM technology," *IEEE Spectrum*, April 1999, pp. 56-64.

[10] http://www.jedec.org.

[11] C. Kang and J. Draper, "A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System," *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, August 2000.

[12] P. Ranganathan, S. Adve and N. Jouppi, "Performance of Image and Video Processing with General-Purpose Processors and Media ISA Extensions," *Proceedings of the International Symposium on Computer Architecture*, May 1999.

[13] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin, "An Argument for Simple COMA", *Proceedings of the Symposium on High-Performance Computer Architecture*, January 1995.

[14] T. von Eicken, D. Culler, S. C. Goldstein and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992.

Implementation of a 256-bit WideWord Processor for the Data-Intensive Architecture (DIVA) Processing-In-Memory (PIM) Chip

Jeffrey Draper, Jeff Sondeen
USC Information Sciences Institute
draper@isi.edu, sondeen@isi.edu

Chang Woo Kang
University of Southern California
ckang@usc.edu

Abstract

The Data-Intensive Architecture (DIVA) system incorporates Processing-In-Memory (PIM) chips as smart-memory coprocessors to a microprocessor. This architecture exploits inherent memory bandwidth both on chip and across the system to target several classes of bandwidth-limited applications, including multimedia, pointer-based, and sparse-matrix applications. The DIVA project is building a prototype workstation-class system using PIM chips in place of standard DRAMs to demonstrate these concepts.

A key component of this architecture is the WideWord Processor, which is a 5-stage pipelined 256-bit datapath, complete with register file and ALU blocks. This component offers fine-grained data parallelism resulting in significant speedups. This paper details the design and implementation of this WideWord Processor in TSMC 0.18μm technology.

1. Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 50-60% per year while memory access times improve at merely 5-7%. Furthermore, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [3]. A recent VLSI technology trend, embedded DRAM, offers a promising solution to bridging the processor-memory gap [9]. One application of this technology integrates logic with high-density memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (with hundreds of gigabit/second aggregate bandwidth available on a chip--up to 2 orders of magnitude over conventional DRAM). Latency to on-chip logic is also reduced, down to as little as one half that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data-Intensive Architecture (DIVA) project uses PIM technology to replace or augment the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. System bandwidth limitations are thus overcome in three ways: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processor-memory "nodes" per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus. The system architecture of DIVA is focused on achieving the following four goals: (1) developing PIMs that can serve as the only memory in the system, assuming the dual roles of "smart memories" and conventional memory; (2) supporting a wide range of familiar programming paradigms, closely related to parallel computing; (3) targeting applications that are severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets; and, (4) developing a VLSI device to exploit memory and communications bandwidth in PIM-based systems while making efficient use of on-chip resources for target applications.

This paper focuses on the microarchitecture design and implementation of the WideWord Processor component of the PIM processing logic. Similar in style to vector extensions like AltiVec [1], the DIVA WideWord Processor uses a 256-bit datapath that enables significant processing speedups through the use of data parallelism. The WideWord Processor was fabricated as part of a DIVA prototype chip in TSMC 0.18μm technology and is currently in test. The remainder of the paper is organized as follows. Sections 2 and 3 present an overview of the DIVA system architecture and microarchitecture, to put the WideWord Processor design into its proper context. Section 4 describes the WideWord microarchitecture in detail. Section 5 presents details of the fabrication and testing of the WideWord Processor as part of a PIM chip, and Section 6 concludes the paper.

2. System architecture overview

A driving principle of the DIVA system architecture is efficient use of PIM technology while requiring a smooth migration path for software. This principle demands integration of PIM features into conventional systems as seamlessly as possible. As a result, DIVA PIM chips are designed to resemble commercial DRAMs, enabling PIM memory to be accessed by host software as if it were conventional memory. In Figure 1, we show a small set of PIMs connected to a single host processor through conventional memory control logic.

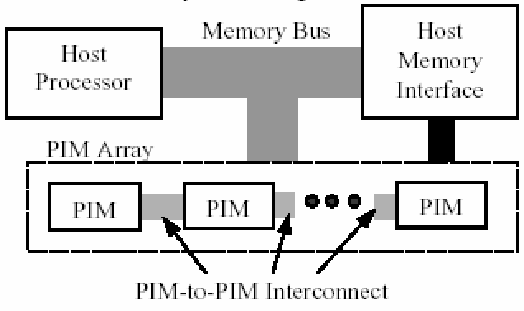


Figure 1. DIVA system architecture

Spawning computation, gathering results, synchronizing activity, or simply accessing non-local data is accomplished via parcels. A parcel is similar to an active message, as it is a relatively lightweight communication mechanism containing a reference to a function to be invoked when the parcel is received [12]. From a programmer's view, parcels, together with the global address space supported in DIVA, provide a compromise between the ease of programming a shared-memory system and the architectural simplicity of pure message passing. Parcels utilize a separate PIM-to-PIM interconnect to enable communication without interfering with host-memory traffic, as shown in Figure 1. Details of this interconnect can be found in [10], and more detail about the DIVA system architecture can be found in [2][4][6][7].

3. Microarchitecture overview

Each DIVA PIM chip is a VLSI memory device augmented with general-purpose computing and networking/communication hardware. Although a PIM may consist of multiple nodes, each of which are primarily comprised of a few megabytes of memory and a node processor, Figure 2 shows a PIM with a single node, which reflects the focus of the initial research that is being conducted. Nodes on a PIM chip share a single PIM Routing Component (PiRC) and a host interface. The PiRC is responsible for routing parcels on and off chip. The host interface implements the JEDEC standard SDRAM protocol so that memory accesses as well as parcel activity initiated by the host appear as conventional memory accesses from the host perspective.

Figure 2 also shows two interconnects that span a PIM chip for information flow between nodes, the host interface, and the PiRC. Each interconnect is distinguished by the type of information it carries. The PIM memory bus is used for conventional memory accesses from the host processor. The parcel interconnect allows parcels to transit between the host interface, the nodes, and the PiRC. Within the host interface, a parcel buffer (PBUF) is a buffer that is memory-mapped into the host processor's address space, permitting application-level communication through parcels. Each PIM node also has a PBUF, memory-mapped into the node's local address space.

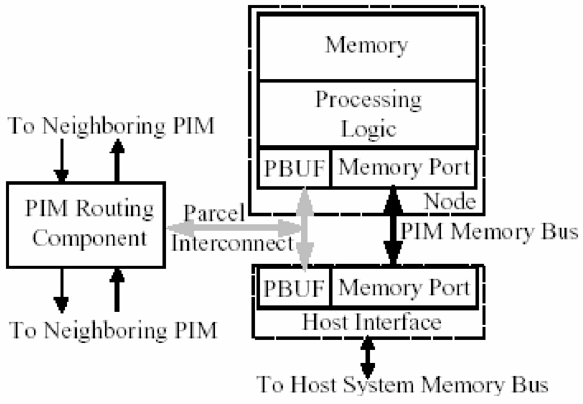


Figure 2. DIVA PIM chip organization

Figure 3 shows the major control and data connections within a node, with the 256-bit memory data bus as the centerpiece. The DIVA PIM node processing logic supports single-issue, in-order execution, with 32-bit instructions and 32-bit addresses. There are two datapaths whose actions are coordinated by a single execution control unit: a scalar datapath that performs sequential operations on 32-bit operands, and a WideWord datapath that performs fine-grain parallel operations on 256-bit operands. Both datapaths execute from a single instruction stream under the control of a single 5-stage DLX-like pipeline [8]. The instruction set has been designed so both datapaths can, for the most part, use the same opcodes and condition codes, generating a large functional overlap.

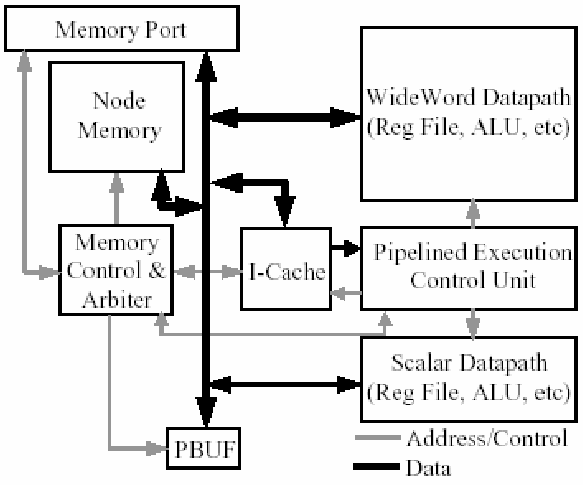

Memory Port
Node Memory
WideWord Datapath (Reg File, ALU, etc)
Memory Control & Arbiter
I-Cache
Pipelined Execution Control Unit
Scalar Datapath (Reg File, ALU, etc)
PBUF
Address/Control
Data

Figure 3. DIVA PIM node architecture

Each datapath has its own independent general-purpose register file, 32 32-bit registers for the scalar datapath and 32 256-bit registers for the WideWord datapath, but special instructions permit direct transfers between datapaths without going through memory. Although not supported in the initial DIVA prototype, floating-point extensions to the WideWord datapath will be provided in future implementations. In addition to the execution units, each DIVA PIM node contains other essential components of note. These components are described in [5].

4. Microarchitecture details of the DIVA WideWord Processor

The combination of the execution control unit and WideWord datapath is regarded as the WideWord Processor. This component enables superword-level parallelism [11] on wide words of 256 bits, similar to multimedia extensions such as MMX and AltiVec. This fine-grain parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. Selective execution, direct transfers to/from other register files, integration with communication, as well as the ability to access main memory at very low latency, distinguish the DIVA WideWord capabilities from MMX and AltiVec. This section details the microarchitecture of this component by first presenting an overview of the instruction set architecture, followed by a description of the pipeline.

4.1. Instruction set architecture

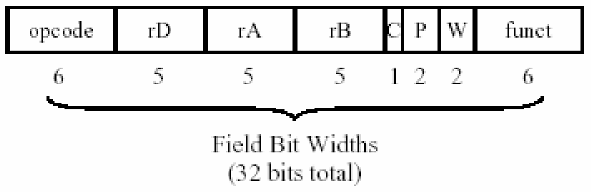


Figure 4. WideWord instruction format

As shown in Figure 4, most DIVA WideWord instructions use a three-operand format to specify two 256-bit source registers and a 256-bit destination register. The opcode generally denotes a class of operations, such as arithmetic, and the function denotes a specific operation, such as add or subtract. The **C** bit indicates whether the operation performed by the instruction execution updates condition codes. The **W** field indicates the operand width, allowing WideWord data to be treated as a packed array of objects of eight, sixteen, or thirty-two bits in size. This characteristic means the WideWord ALU can be represented as a number of variable-width parallel ALUs. The **P** field indicates the participation mode, a form of selective subfield execution that depends on the state of local and neighboring condition codes. Under selective execution, only the results corresponding to the subfields that participate in the computation are written back, or committed, to the instruction's destination register. The subfields that participate in the conditional execution of a given instruction are derived from the condition codes or a mask register, plus the instruction's 2-bit participation field. For more details, see [2].

The WideWord instruction set consists of roughly 30 instructions implementing typical arithmetic instructions like add, subtract, and multiply; logical functions like AND, OR, NOT, XOR; and logical/arithmetic shift operations. In addition, there are load/store and transfer instructions that provide for rich interactions between the scalar and WideWord datapaths.

Some special instructions include permutation, merge, and pack/unpack. The WideWord permutation network supports fast alignment and reorganization of data in wide registers. The permutation network enables any 8-bit data field of the source register to be moved into any 8-bit data field of the destination register. A permutation is specified by a permutation vector, which contains 32 indices corresponding to the 32 8-bit subfields of a WideWord destination register. A WideWord permutation instruction selects a permutation vector by either specifying an index into a small set of hard-wired commonly used permutations or a WideWord register whose contents are the desired permutation vector. The merge instruction allows a WideWord destination to be constructed from the intermixing of subfields from two source operands, where the source for each destination subfield is selected by a condition specified in the instruction. This merge instruction effects efficient sorting. The pack/unpack instructions allow the truncation/elevation of data types and are especially useful in pixel processing.

4.2. Pipeline description

The WideWord Processor pipeline is a standard DLX-like 5-stage pipeline, with the following stages: (1) instruction fetch; (2) decode and register read; (3) execute; (4) memory; and, (5) writeback. Data hazards occur when there are read-after-write register dependences between instructions that co-exist in the pipeline. The controller and datapath contain the necessary forwarding, or bypass, logic to allow pipeline execution to proceed without stalling in most data dependence cases. Register forwarding is complicated somewhat by the participation capability. Participation status must be forwarded along with each subfield to effect correct forwarding.

5. Implementation and testing of the DIVA WideWord Processor

The DIVA WideWord Processor specification required on the order of 25,000 lines of VHDL code, consisting of a mix of RTL-level behavioral and gate-level structural code. A preliminary, unoptimized stand-alone layout of the WideWord Processor used 100,000 standard cells (approximately one million transistors) and occupied 10 sq mm in 0.18µm technology, projected to operate at 300MHz while dissipating 500mW.

Although the WideWord Processor is suitable for stand-alone implementations, the DIVA project employs it as part of a tightly integrated node design, as discussed in Section 3. The WideWord Processor VHDL specification was included as part of a DIVA PIM prototype specification, which was synthesized using Synopsys Design Analyzer. The entire chip was placed and routed with Cadence Silicon Ensemble, and physical verification, such as DRC and LVS, was performed with Mentor Calibre. The intellectual property building blocks used in the chip include Virage Logic SRAM, a NurLogic PLL clock multiplier, and Artisan standard cells, pads, and register files.

The first DIVA PIM prototype, shown in Figure 5, is a single-node implementation of the DIVA PIM chip architecture. Due to challenges in gaining access to embedded DRAM fabrication lines, this first prototype is SRAM-based. This chip implements all features of the DIVA PIM architecture except address translation and floating-point capabilities. A second version of a PIM chip, which not only integrates these functions but achieves a faster clock rate, is due to tape out in the second half of 2002. The chip shown in Figure 5 was fabricated through MOSIS in TSMC 0.18µm technology, and the silicon die measures 9.8mm on a side. It contains approximately 2 million logic transistors in addition to the 53 million transistors that implement 8 Mbits of SRAM. The chip also contains 352 pads, of which 240 are signal I/O, and is packaged in a 35mm TBGA.

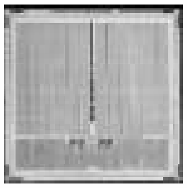


Figure 5. DIVA PIM prototype chip

The chip is being tested with the use of an HP 16702A logic analysis system. Pattern generator modules are utilized to apply test vectors to the inputs of the chip, and timing/state capture modules are used to sense the outputs of the chip. The chip is currently being tested for functionality at a testbench speed of 80MHz. Although exhaustive testing has not yet been completed, the chip is running a demonstration application of matrix transpose that exercises all major control and datapaths within the scalar processor, including the permutation network highlighted in Section 4.1. Even in this limited test setup, the chip is achieving 640MOPS and 2.56Gbytes/s memory bandwidth while dissipating only 800mW. We anticipate even greater achievements with further testing.

6. Conclusion

This paper has presented the design and implementation of the WideWord Processor used in the DIVA system, an integrated hardware and software architecture for exploiting the bandwidth of PIM-based systems. A working implementation of this design, based on TSMC 0.18µm technology, has proven the validity of the design. The workstation system that is currently being developed to use this component is projected to achieve speedups ranging from 8.8 to 38.3 over conventional workstations for a number of applications. These improvements arise mainly from three sources: decreased memory times; coarse-grain parallelism across PIMs to exploit system bandwidth; and, wide on-chip datapaths to exploit fine-grain parallelism, including especially those wide datapaths within the WideWord Processor.

Acknowledgments

This research was supported by DARPA contract F30602-98-2-0180.

References

[1] http://www.altivec.org

[2] J. Brockman, et al, "Microservers: A New Memory Semantics for Massively Parallel Computing", in *Proc. of the International Conference on Supercomputing*, June 1999, pp. 454 - 463.

[3] D. Burger, J. Goodman and A. Kagi. "Memory Bandwidth Limitations of Future Microprocessors," In *Proc. of the 23rd International Symposium on Computer Architecture*, May 1996.

[4] J. Chame, M. Hall and J. Shin, "Code Transformations for Exploiting Bandwidth in PIM-Based Systems," In *Proc. of the Workshop on Solving the Memory Wall Problem*, ISCA Workshop, June 2000.

[5] J. Draper, et al, "The Architecture of the DIVA Processing-in-Memory Chip", to appear at International Conference on Supercomputing, June 2002.

[6] M. Hall, et al, "Mapping Irregular Applications to DIVA: A Data-Intensive Architecture," In *Proc. of SC '99*, November 1999.

[7] M. Hall and C. Steele. "Memory Management in a PIM-Based Architecture," in *Proc. of the Workshop on Intelligent Memory Systems*, October 2000.

[8] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman, Second Edition, 1996.

[9] S. Iyer and H. Kalter, "Embedded DRAM Technology: Opportunities and Challenges," *IEEE Spectrum*, April 1999, pp. 56 - 64.

[10] C. Kang, J. Draper, "A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System," In *Proc. of the IEEE Midwest Symposium on Circuits and Systems*, August 2000.

[11] R. Lee, "Subword Parallelism with MAX-2," *IEEE Micro* 16(4), Aug. 1996, pp. 51 - 59.

[12] T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

Memory Management in a PIM-Based Architecture

Mary Hall and Craig Steele

University of Southern California/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292, USA
{mhall,steele}@isi.edu

Abstract. The DIVA (Data IntensiVe Architecture) system incorporates Processing-In-Memory (PIM) chips as smart-memory coprocessors to a host microprocessor. It exploits the inherently high on-chip memory bandwidth and additionally provides a separate memory-to-memory high-bandwidth interconnect across the system. By design, the DIVA system architecture targets a broad range of applications, including those with irregular data access patterns. At the same time, DIVA supports familiar programming paradigms from parallel computing and offers an evolutionary migration path for application development.

The DIVA project is constructing a demonstration system using a conventional superscalar host processor with a main memory composed of VLSI PIM chips in place of standard DRAMs. This system has a novel mix of operating-system challenges, combining aspects of conventional "dumb" memory management and both shared- and distributed-memory multiprocessor operations. This paper describes our solutions to the memory-management problems posed by this multifaceted environment.

1 Introduction

The Data IntensiVe Architecture (DIVA) project is building a workstation-class system using embedded-DRAM technology to replace the memory system of a conventional workstation with "smart memories" capable of very large amounts of processing. The goal of the project is to significantly reduce the ever-increasing processor-memory bandwidth bottleneck in conventional systems. System bandwidth limitations are thus overcome in three ways, as illustrated in Figure 1: (1) tight coupling of a single PIM processor with an on-chip memory bank; (2) distributing multiple processor-memory nodes per PIM chip; and, (3) utilizing a separate chip-to-chip interconnect, for direct communication between nodes on different chips that bypasses the host system bus.

** Appeared in "Workshop on Intelligent Memory Systems," November, 2000

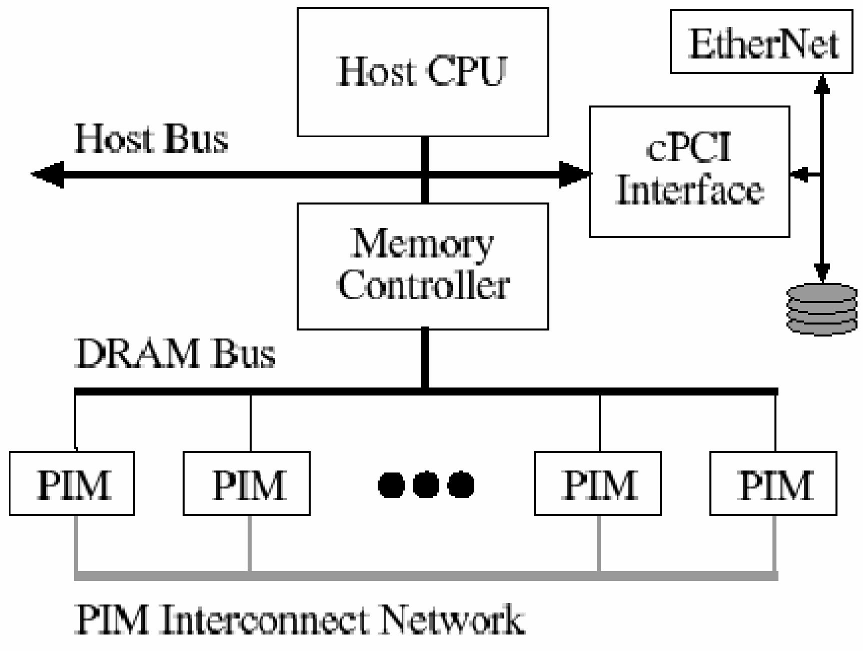


Fig. 1. DIVA System Architecture.

This paper describes memory management in DIVA. Two aspects of the DIVA project distinguish its memory management requirements from that of other PIM-based architectures.

– The PIMs serve as the only memory for a standard host microprocessor, assuming the dual roles of "smart memories" and conventional memory.
– DIVA targets applications that are most severely impacted by the processor-memory bottlenecks in conventional systems: sparse-matrix and pointer-based applications with irregular memory access patterns, and image and video applications with large working sets.

As compared to system-on-a-chip solutions [6,5], and multiprocessors made up solely of PIM chips [7,4], DIVA's support for conventional memory accesses from an external host requires a dual view of memory, from the host perspective and the PIM's perspective. Other PIM architectures address this challenge by restricting PIM functionality to SIMD execution on large streams of data, at the host's direction [1,2]. In DIVA, we support a much broader range of programming paradigms, including task-level parallelism and in-memory accesses to pointer data structures. As a result, DIVA requires a memory model that supports independent threads of control and efficient translation in memory, without necessitating host intervention.

A previous paper presents an overview of the DIVA project and describes a memory model to support these requirements [3]. This paper discusses the memory management support needed to realize this memory model.

2 Overview of Memory Model and Address Translation

The DIVA memory model attempts to satisfy a number of potentially conflicting requirements:

– scalability to tens of PIM chips;
– efficient hardware mechanisms amenable to straightforward implementation;
– an abstract machine comprehensible to both programmers and compiler writers;
– compatibility with conventional memory models and memory interfaces;
– support for virtual memory (i.e., paging to/from disk) and swapping; and,
– supporting correct functionality of both shared- and distributed-memory programming models.

The unifying concept for the DIVA memory model is communication via a global address space shared by the host processor and all the PIM node processors. Not all memory need be shared, however, so our hardware supports local PIM address spaces as well. All of the processors in our demonstration system use 32-bit addresses, but the model can be advantageously extended to future 64-bit systems.

To interpret addresses in PIM code and data, a PIM processor must support a translation mechanism. However, the space and time overhead of maintaining conventional page tables at each node is prohibitive. To simplify translation hardware, we classify DIVA memory according to usage:

– global memory is a single address space distributed across nodes, visible to applications running on the host and PIM nodes.
– dumb memory is a region of a node's memory allocated as conventional pages in a host application's virtual space and untouched by PIM node processing.
– local memory is a region of a node's memory used almost exclusively by PIM routines. Certain exceptional functions of the host operating system, such as initialization and context management, will also access this memory occasionally, requiring well-defined data-sharing conventions.

The physical memory on each PIM chip is flexibly partitioned into these three distinct uses. Dumb memory is managed exclusively by the host operating system in standard ways, with address translation handled solely by the host processor's memory-management hardware. Figure 2 depicts the two more interesting uses of PIM memory, as part of the shared global address space, or as PIM local memory. The DRAM memory associated with the global address space is physically distributed across all the PIM nodes involved in a computation. Addresses in the global virtual address space are consistent for the host and all PIM node processors, so that pointer-based data structures can be freely shared. In contrast, while the host has physical access to the PIM DRAM used as local memory, the host and PIM node processors will see it at different virtual addresses.

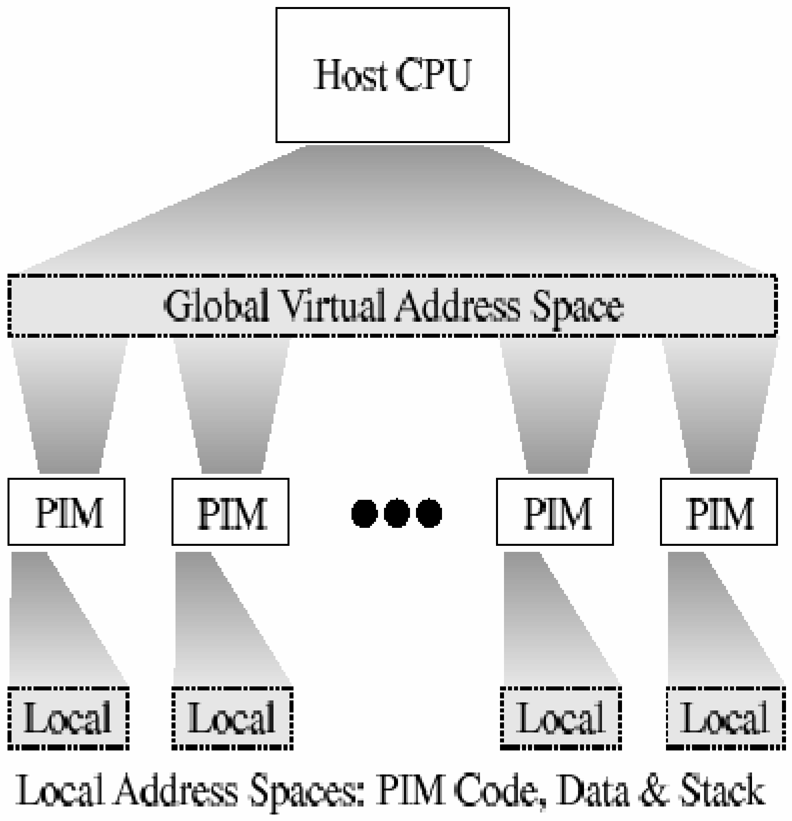

Host CPU
Global Virtual Address Space
PIM
PIM
PIM
PIM
Local
Local
Local
Local
Local Address Spaces: PIM Code, Data & Stack

Fig. 2. Shared global segments, unshared local segments.

The host processor can access PIM memory via its memory bus. To avoid saturation of this bus, PIM-to-PIM communications occur primarily by means of a distinct high-bandwidth network between PIM chips. The hardware directly supports shared-memory operations between the host and PIM memories, but PIM-to-PIM communications are implemented by network communications in the form of parcels (Section 2.3). Parcel operations are hardware assisted, but require software processing by either user- or supervisor-level code at both ends. Efficient network interface and interrupt mechanisms have been developed to support parcel functions.

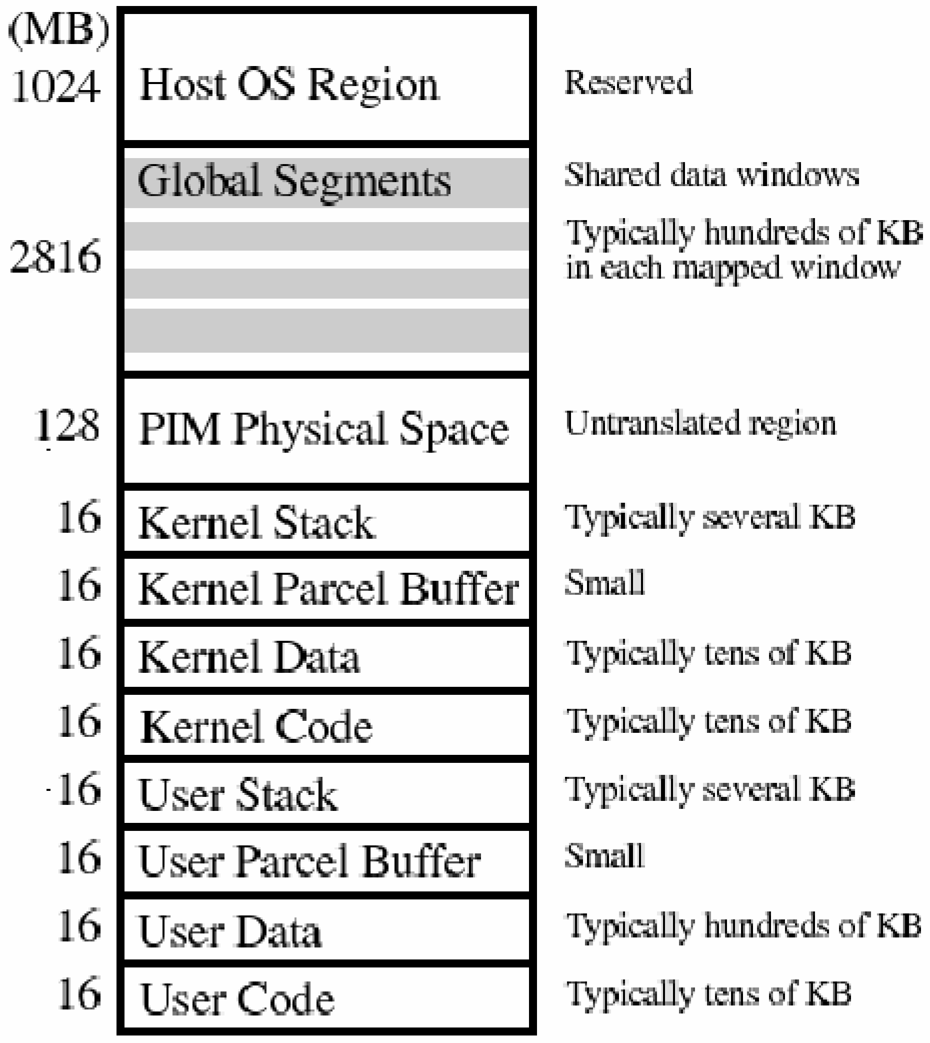


Fig. 3. PIM node processor address map.

2.1 Address Translation for Locally Mapped Data

A node must be able to rapidly determine if an address is located in its own memory, and if so, find the physical address. Each node therefore maintains translations for virtual addresses currently residing on it, including local memory and its portion of global memory. To condense translation information, we use segments, each of which is defined by segment registers containing a physical base address and limit. The segments are described by the PIM address map shown in Figure 3. The local memory region is partitioned into eight segments at fixed virtual bases, for kernel code, stack and data, user code and data/stack, and for kernel and user network-communication buffers. A small number of global segment registers are also used; since global segments must be able to map portions of a shared virtual address space much larger than the physical memory of an individual node, global segments must be represented by both a virtual and physical base address register.

Figure 3 shows the virtual address map for a PIM node processor. The virtual size of each region of the map is shown on the left; typically only a fraction of that virtual address space will be used, as noted on the right of the diagram. The lowest-addressed (bottommost) segments of the address map define the local-memory user-mode space for the current process. The next group of segments defines the local-memory supervisor-mode space for the kernel, which is the same for all processes. The kernel can also access the PIM's DRAM without address translation via the physical space region.

Each PIM has a small number of relocation registers to allow it to map portions of the shared global address space to the node's physical memory. The aggregate size of these "windows" into the shared address space is limited by the amount of physical memory available on a node. The top region of the map is unused, but reserved to conform with the host operating system's address map.

2.2 Translating Remote Addresses

Access to parts of the global address space not mapped to physical memory on the node is possible via the network, but less efficient than a mapped access. A parcel must be sent to the node which contains the physical memory to be accessed, and a response parcel received and processed, either by user or kernel code.

DIVA determines the location of remote data via a two-stage process. The virtual address of a datum is hashed by hardware to determine the "home node" of the datum [8]. The home node may or may not be the present physical location of the datum, but serves as the centralized directory and manager for it. The home node will either perform the operation itself, if the datum is resident, or forward the request to another node, if the datum resides elsewhere.

Therefore, a node must maintain translation information for only eight local segments plus a small number of segments for its portion of the global memory, as well as for any global data for which it is the home node. The major advantages of this approach are that translation may be accomplished rapidly, and translation information on each PIM scales well.

2.3 Parcels

All PIM-to-PIM network communications are performed by sending and receiving messages in the form of parcels. Parcels are an object-based variant of active messages [9], distinguished from active messages in that the destination of a parcel is an object in memory, not a specific node. From a programmer's view, parcels, together with the global address space supported in DIVA, provide a compromise between the ease of programming a shared-memory system and the architectural simplicity of pure message passing. Remote operations or accesses can be accomplished through parcel sends and receives; application programs only need specify the address of an object, and not the processor upon which the object resides.

Structurally, a parcel packet has a 256-bit payload and 96-bit header, which includes:

– the memory address of the target object, expressed in the application's address space,
– the environment id (eid) of the process in the host that is executing the application,
– a command identifying the function to be performed by the node associated with the target when the parcel arrives.

The 256-bit payload serves as arguments to the command. A parcel requiring more bits must be sent in multiple packets. The payload size matches the PIM node data bus width; streaming packets may be sent in a single bus cycle. The network interface supports both user- and supervisor-mode access to parcel sending and receiving hardware via the user and kernel parcel buffer segments. User-mode parcel processing is more efficient but less robust than kernel-mediated operations, so will typically be restricted to compiler-generated code or library routines. Error conditions cause invocation of either user- or supervisor-mode handlers.

3 Overview of Memory Management

Memory management functions are divided between two types of kernels in the DIVA system. On the host processor, the standard operating system (in DIVA, Linux) is augmented with functionality to support PIMs. On each PIM processor, there is a tiny run-time kernel that is always resident. A primary responsibility of the PIM run-time kernel is to manage parcel communication between PIMs [3]. The run-time kernel performs buffer management of incoming parcels, and directs context switches between different threads in the same user program, or between user program and kernel. The run-time kernel also performs required software intervention in response to interrupts and exceptions on the PIM processor. In addition to these autonomous functions of the PIM run-time kernel, it also must collaborate with the host on system-level operations, such as loading PIM programs and data, memory management of PIM-visible segments, and PIM context switches between different user programs (note that most host context switches will not involve the PIMs).

This division of labor is motivated by the dual goals of keeping the PIM run-time kernel as small as possible, and making only moderate changes to the standard function of the host operating system. Unlike standard multiprocessor systems, the host, which has a system-level view, remains a central figure in system-level scheduling, disk I/O operations, and memory management. The challenge in this collaboration between host and PIM system software is that there are really two views of memory that must be maintained. For dumb pages and for disk I/O of PIM-visible segments, the host sees memory as standard 4Kbyte pages; the PIM run-time kernel instead views PIM-visible memory as variable-sized segments. Reconciling these two views through different system functions is the subject of the remainder of this paper.

4 Memory Allocation: Virtual vs. Physical

The portion of memory used by the host as dumb memory is managed by the host operating system using standard allocation, paging and swapping mechanisms. The memory devoted to PIM local memory, and global shared memory, must be managed via a collaboration between host and PIMs. The most unusual aspect of this collaboration is memory allocation.

Figure 4 shows the functions associated with memory allocation, and whether they are performed by host or PIM. There are three phases to allocation: (1) host allocation of contiguous virtual address spaces for global and PIM local segments using the *Reserve* functions; (2) physical allocation of an object and binding to reserved virtual segments; and, (3) mapping of existing global objects to a global segment for sharing between PIMs. Deallocation (*GlobalFree*) frees physical memory but does not shrink the virtual-space allocation.

The standard memory allocation functions *malloc* and *free* can be used on either the host or PIMs; the meaning depends on where the functions are executed. On the host, a call to *malloc* performs a standard allocation from dumb memory. On the PIMs, it allocates memory from the PIM's local heap segment. Memory obtained from *malloc* is private to a process and unsharable.

4.1 Virtual Memory Allocation

Using a segmented approach requires that data in a segment reside in contiguous virtual addresses. For this reason, as part of the allocation process, we must reserve a contiguous chunk of the virtual address space for each segment prior to physical allocation. The virtual memory allocation is performed by the host using the *Reserve* functions for global and local segments. Because the virtual address space is quite large, these reservations should always strive to overestimate the space requirements of the segment, particularly since growing a segment beyond what was initially reserved results in very costly adjustments in virtual and physical allocations. Linux supports this reservation process by clustering free

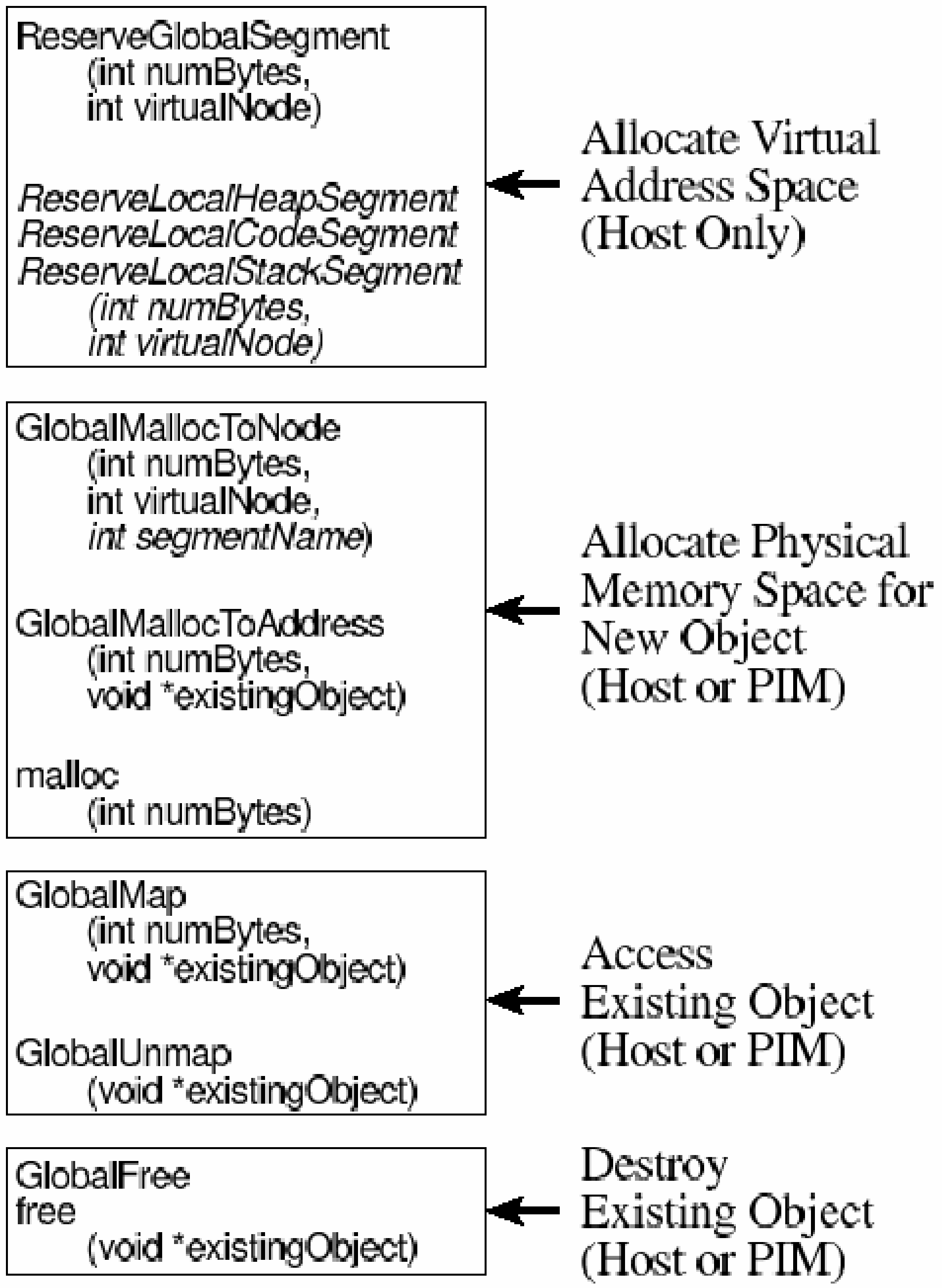


Fig. 4. Host and PIM memory management functions and steps of memory allocation.

pages in the virtual address space together; reservations select a cluster that matches the requested size.

Multiple global segments can be reserved by separate invocations to the *ReserveGlobalSegment* function. Each global segment reservation will create a new segment with a unique name (see Section 7.2). The segment name can subsequently be used optionally by allocation functions, as discussed below. Similar functions exist to allocate the virtual address space for PIM-local code, data and stack segments. These are italicized to indicate that they are optional, since standard default values often suffice.

4.2 Physical Memory Allocation

The physical allocation is performed through a collaboration between host and PIM. As part of physical allocation, the page table entries on the host are filled. On the PIM side, segment registers may be updated.

The functions shown in Figure 4 allocate a specific object to a segment. However, global and per-node local memory allocation and deallocation could swamp the host operating system with fine-grained memory allocation requests. Behind the scenes, we distribute this task using a two-level scheme where coarser-grained requests to the host are made by each PIM run-time kernel to replenish locally managed memory pools of pre-allocated global and local memory. This approach keeps the host involved in memory allocation, but still permits the PIMs to allocate memory independently as needed for managing pointer-based and other dynamic structures during PIM computation.

The DIVA programming model offers a globally addressable, distributed address space on shared data. PIM applications perform correctly when accessing non-local memory, either by communicating via the parcel mechanism, or by retrieving data in response to a more expensive address translation fault. Nevertheless, just as with distributed-shared-memory architectures, to achieve the best performance, an application must whenever possible co-locate data with the computation that accesses it. For this purpose, there are two flavors of memory allocation functions. The *GlobalMallocToNode* function associates allocated data to a specific virtual PIM home node. An optional *segmentName* argument permits this allocation to occur within a specific global segment. To allow two related objects to be collocated without requiring the virtual PIM identifier, the *GlobalMallocToAddress* function instead permits dynamic allocation of objects to the same virtual PIM node and global segment upon which another datum resides.

To simplify the programming model, *GlobalMalloc* functions performed on the PIM match the interface used on the host. Most of the time these functions will be used to allocate data from the PIM's locally resident global segments, but it is possible for a PIM to perform an allocation on a remote PIM node. The effect of such an allocation is to allocate virtual addresses from the remote node, and locally map the object to the requesting PIM's global segments and physical storage. Such an allocation can be performed to support efficient updates of the remote data prior to forwarding them to the remote node (see Section 7.3).

4.3 Mapping Existing Objects to PIM Global Segments

Like the *GlobalMalloc* to a remote node, it is sometimes desirable to temporarily map non-resident global data to facilitate sharing among PIMs. The *GlobalMap* function performs this mapping to global segment registers, and *GlobalUnMap* returns the data to its home node (see Section 7.3).

5 Paging

To perform computations requiring access to global data structures larger than the actual amount of physical memory, we support a virtual-memory "paging" mechanism for PIM-process memory. (We use the slightly inaccurate term "paging" in preference to the technically preferable term "global-segment access fault management" for brevity.) If the memory access which caused the fault references a datum resident in a PIM memory, it can be resolved without troubling the host operating system, by retrieving the accessed datum via a parcel request to its home node. On the other hand, if the home node returns a message indicating that the requested datum is not resident in the system's physical memory, the initiating PIM kernel must request paging service from the host, which is connected to the disk backing store.

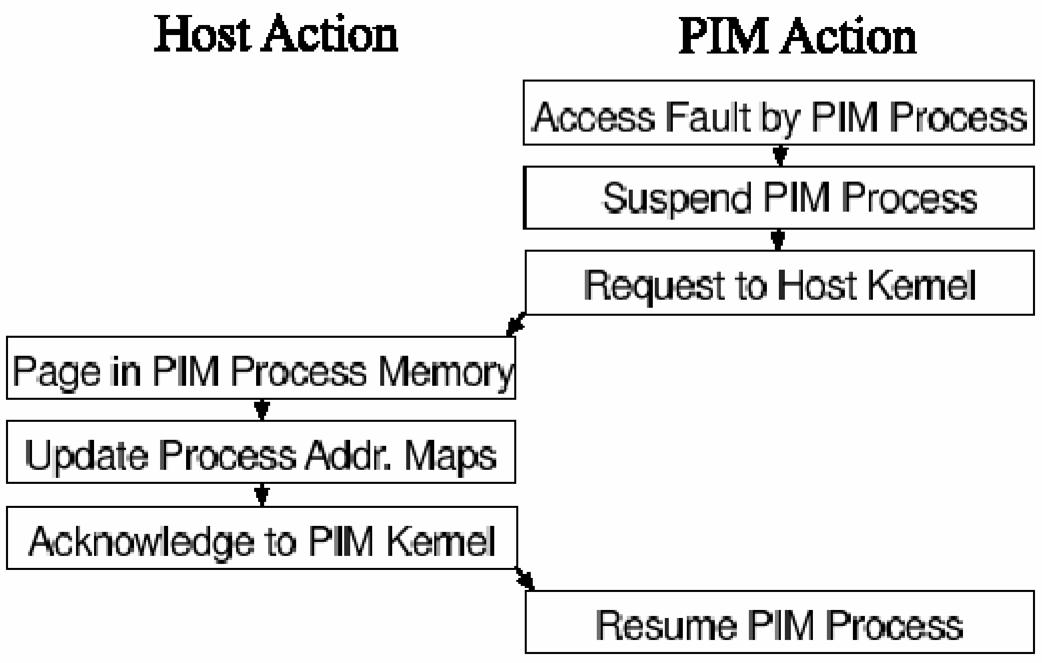

Host Action
PIM Action
Access Fault by PIM Process
Suspend PIM Process
Request to Host Kernel
Page in PIM Process Memory
Update Process Addr. Maps
Acknowledge to PIM Kernel
Resume PIM Process

Fig. 5. Paging sequence.

As shown in Figure 5, the faulting PIM process is suspended until the datum is paged in from disk, and the host kernel becomes the owner of that process context during the memory reorganization. The host pages in a section of the global virtual space containing the datum. The paged-in section is typically mapped to a distinct segment of the global space. However, if the faulting address is adjacent to an existing locally mapped segment, the segment may be extended to contain it.

After the host kernel has resolved the fault and adjusted the faulting process' PIM context mappings, it returns ownership of the context to the PIM kernel, which reloads the PIM address translation hardware when it reactivates the process.

The paging system is a useful but relatively expensive feature, best used sparingly.

6 Contexts and Swapping

A DIVA PIM node supports very efficient context switching for the most common cases, either switching between a user program and the PIM run-time kernel, or between two distinct threads within the same user program. Switching to the run-time kernel requires no change to segment registers, and requires minimal saving and restoring of register state. Switching between different threads in the same user program, such as when performing the command associated with an incoming parcel, requires modification to only two of the segment registers, but does require saving and restoring of portions of the register state. In either case, there is no need to swap memory in or out in response to a context switch.

In performing its normal job scheduling function, the host may direct the PIM node's context to change to a different user program that requires PIM functionality. In this case, a full context switch is necessary, saving all the register state as well as updating the program-specific segment registers. Further, memory may need to be swapped in or out. If the user-code physical memory is swapped out and recycled, the content of the PIM node processor's instruction cache must also be invalidated by software, since the new program's code memory may overlap with the previous program's. (Our processor, like many others, does not enforce coherency in the instruction cache hardware.) Note that at any time, the host may be executing in a different context from one or all PIMs; for most host context switches, it will not be necessary to change the PIM node context.

The host operating system is responsible for creating contexts for the PIMs, and also for updating contexts in response to major system context switches. (Lightweight PIM context switches, e.g., multithreading, do not involve the host.) To facilitate host management of contexts, during initialization the host creates a data structure, mapped to the PIM's memory, that it shares with the PIM run-time kernel. While it is possible for the host to build this data structure through a series of parcels sent to the PIM run-time kernel, for efficiency we permit the host operating system in privileged mode to write directly into portions of the PIM run-time kernel data segment. The host also updates this structure in response to a system context switch.

6.1 Contents of Context

Figure 6 graphically depicts the contents of a context. On initialization of a user program, the host performs virtual memory allocation of the segments, as discussed in Section 4.1, and writes the range of allocated virtual addresses into the context data structure in the PIM run-time kernel segment. The remainder of the context is reset to default values. As a result of physical memory allocations, the physical segment mappings are added to this structure. The remainder of the fields are filled in by the PIM run-time kernel when saving state as a result of a context switch. When this context is restored, the host updates the segment mappings as needed.

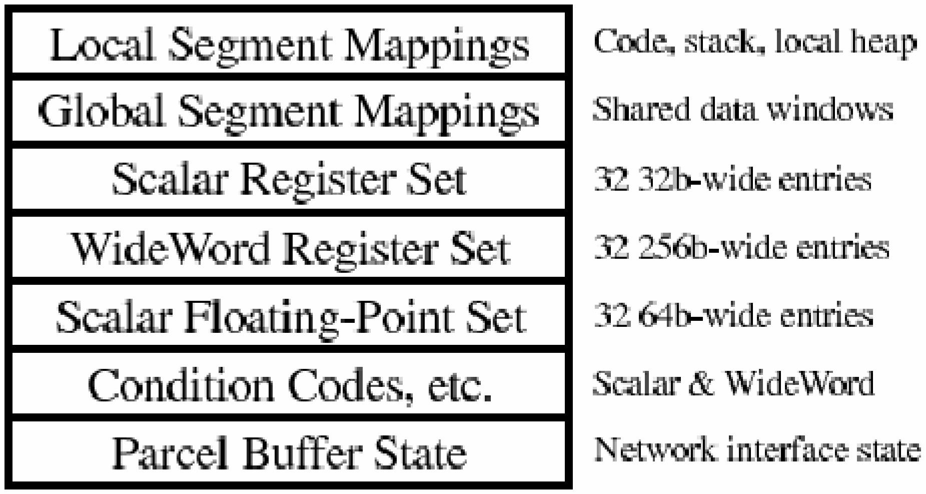


Fig. 6. Contents of context.

6.2 Swapping

Swapping is another mechanism for supporting computations with large memory requirements. Many computations can be broken up into distinct phases which need not be simultaneously active. Peak memory requirements may be reduced by swapping out inactive processes or low priority active processes. Swapping is somewhat similar to paging; the primary distinctions are that the entire context is moved to the disk backing store, freeing all the process memory, and that the host operating system, rather than the PIM kernel, initiates the swap as part of its overall scheduling function.

The sequence of actions required to effect a process swap and restore is sketched in Figure 7. As in the paging sequence, the ownership of the process context and its associated resources passes from the PIM kernel to the host operating system when the PIM process is suspended. Restoring a process context from an out-swapped state is quite similar to the initial instantiation of the process.

Swapping out a context frees most local process resources for reuse, but does not free memory used to store global segments, since they are likely to be in use by related processes on other PIM nodes. The global memory use may well exceed the local, so this is a potentially major problem for our storage reclamation capabilities. To be able to effectively manage groups of related processes, and to be able to decide when their associated global segments may be swapped out, we adopt a system of naming global segments, discussed in Section 7.2. We can thereby "gang schedule" related processes which use particular global segments and swap out their local and global resources together. We can record process references to a given global segment by explicit segment mappings and by remote parcel accesses. Statistics such as these can be recorded by each PIM node kernel and stored locally. The host operating system will only need to examine and aggregate these distributed runtime statistics in the event of a requirement for a major swapping operation, such as phase transition for a very large computation.

In general, the host must attend to global changes in the PIM-based computation, *i.e.*, scheduling functions, where resource allocation policies may be altered, and to chores which require access to external devices, such as swapping or paging to disk. We minimize the host's workload, and its potential for saturation, by requiring it to perform only those tasks which are global in their essence.

7 Local and Global segments

Section 4 described how local and global segments are allocated; here we consider how they are managed. Local segments should remain fairly small, so there is little concern that portions of them will be paged to disk during active PIM execution. Rather, we assume that most of the data read or written by PIM computations will reside in global segments.

Global segments provide a mechanism for sharing global data between host and PIM or across PIMs. Data-intensive applications will have a large amount of global data that can easily exceed the available physical memory capacity; thus, it is desirable to break up global data into multiple global segments. Global segments can be much larger than the 4Kbyte page size of the system, and there can be many more global segments associated with a user program than are mapped to the small set of global segment registers on each PIM. As a result, data required by a portion of the computation of the PIM program may be spread across multiple global segments; to avoid thrashing, care must be taken to map these segments to physical memory simultaneously during this portion of the computation. The remainder of this section describes the mechanisms for organizing and managing data in multiple or very large global segments.

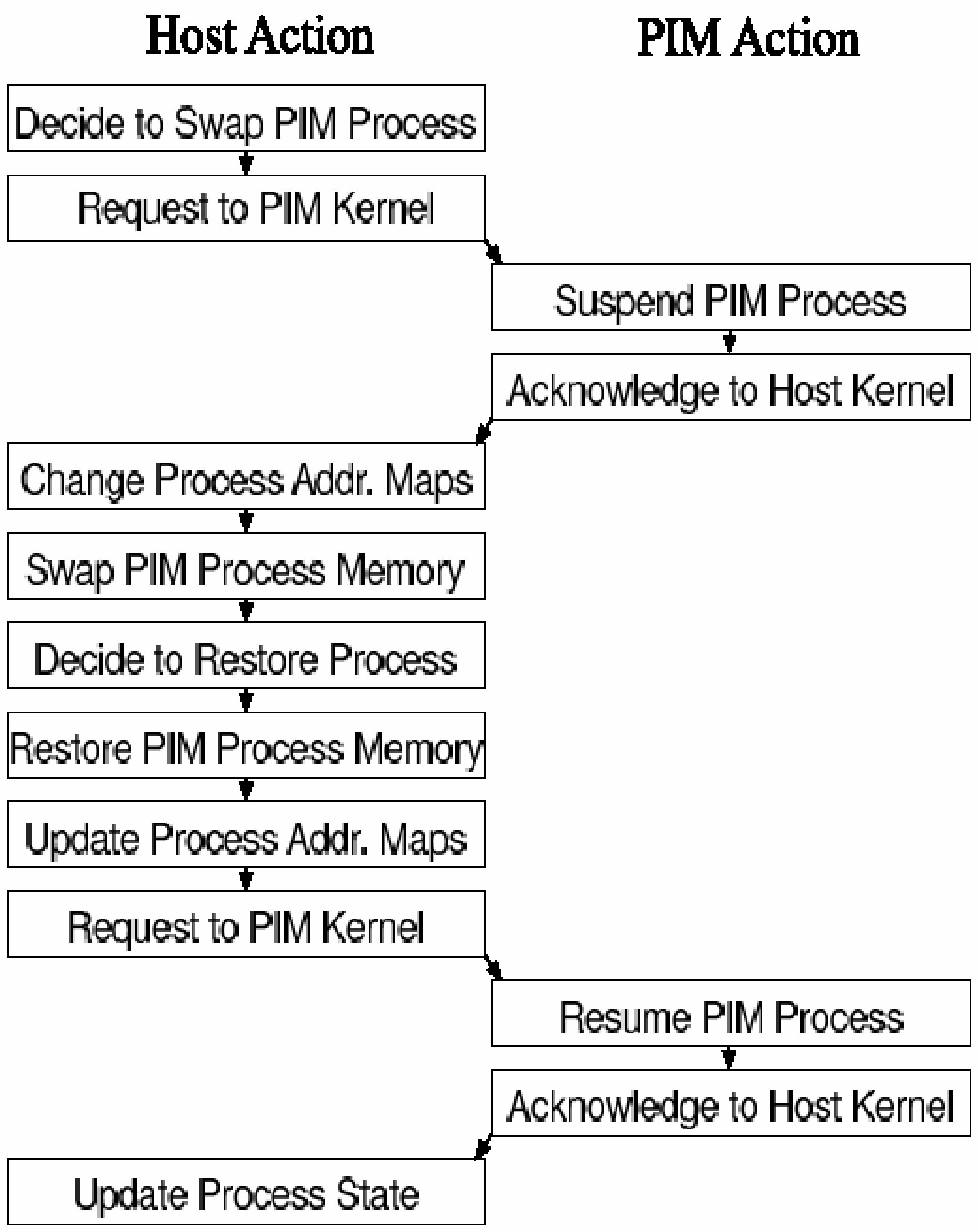

Host Action
PIM Action
Decide to Swap PIM Process
Request to PIM Kernel
Suspend PIM Process
Acknowledge to Host Kernel
Change Process Addr. Maps
Swap PIM Process Memory
Decide to Restore Process
Restore PIM Process Memory
Update Process Addr. Maps
Request to PIM Kernel
Resume PIM Process
Acknowledge to Host Kernel
Update Process State

Fig. 7. Swapping sequence.

7.1 Large Global Segments

In the absence of compiler or application-level support for defining segments, the operating system's default behavior is to create one or a small number of possibly very large global segments for a user application program. In this case, a single segment can be much larger than the available physical memory capacity, so that only a portion of the segment can reside in memory at a time.

Since each PIM has multiple global segment registers, even a single global segment can be managed as multiple segments by having distinct segment registers mapping different portions of the segment. This approach works well, for example, if an application is streaming through its data set sequentially. As it completes its accesses to data represented by one segment register, it can move on to data represented by another segment register. The operating system and PIM run-time kernel can page out the data associated with the former segment registers, and reclaim the segment registers and physical memory for a subsequent portion of the segment.

7.2 Assigning Names to Global Segments

While a single large segment can be managed effectively for streaming applications, in general, a more flexible mechanism is required for organizing data into multiple segments. For example, an application may revisit data in different phases of a computation; or, one data structure may be needed at the same time as another data structure in one phase of computation, and also required at the same time as a third data structure during a later phase of computation.

Our approach is to assign names to segments as they are being created, and permit the compiler or application program to optionally reference these segment names in memory allocation functions. For example, the effect of the allocation function *GlobalMallocToNode(int numBytes, int virtualNode, int segmentName)* is to allocate *numBytes* from the named segment *segmentName* on virtual PIM node *virtualNode*. (The effect of a *GlobalMallocToAddress* call is to perform the allocation on the same virtual PIM node and in the same global segment as that of the specified address.) By allocating two objects from the same global segment that are always used together, we can maximize the likelihood they will always be simultaneously in memory whenever they are being accessed. In cases where grouping all related data would result in too large a segment, the related data must be broken into multiple smaller segments, such that their size more manageably maps to physical memory, but at the same time, there are sufficiently few related segments so that all can simultaneously map to the small number of global segment registers on each PIM.

7.3 Sharing Global Segments across PIMs

As noted above (Section 4.2 and Section 4.3), mapping a global segment to local physical memory provides a mechanism for efficient sharing of large blocks of global data by asserting temporary ownership of a local copy of a data block that may be "homed" on another node. The home node of a datum in the global address space is a function of its virtual address, but the item may reside on another node. The home node provides a central access point for the item regardless of its actual location. In the absence of active mappings by other nodes, a datum will be (re)located to its home node.

The shared data block is created by either the host or a PIM node via the *GlobalMalloc* functions. The *GlobalMalloc* functions perform two distinct roles: allocating a block of physical memory and mapping a portion of the (previously reserved) global virtual address space to that physical memory. The *GlobalMallocToNode* function associates allocated data with a specific virtual PIM home node. However, if the function is invoked by code on a given PIM node, the initial physical memory allocation is made on that PIM node, which need not be the home node. A virtual-memory allocation request is sent to the remote home node, which records the location of that data object and returns a range of allocated virtual addresses drawn from its virtual pool. The requestor node maps that virtual address range to the physical memory it has allocated from its own physical pool. The requestor process is then free to access its instantiated data object at will. The process may terminate its use of the data object by invoking either the *GlobalUnMap* or *GlobalFree* function. Calling *GlobalFree* unmaps the object and indicates that its physical storage may be recycled and its content destroyed. Calling *GlobalUnMap* merely unmaps the object from the current process and indicates that its content should persist. The object will be relocated to its home node, where other processes may subsequently access it by calling the *GlobalMap* function. The object will be destroyed when some process, host or PIM, calls *GlobalFree* on it, or the computation terminates.

For simplicity, our sharing model supports only a single copy of the data and will block to enforce serialized access if necessary. All access control is serialized through the home node. In such a basic environment, careless use of the *GlobalMap* function can result in deadlock; this is regarded as a programming error.

The distributed-shared-memory mechanism outlined above is intended for simple block-oriented data sharing, for applications where bandwidth is a more appropriate metric than latency. More flexible and finer-grained access is available via the parcel mechanism, which may be invoked either explicitly with user-mode access to the network interface, or implicitly, by the PIM kernel in response to an access fault. Note that our remote-access model permits access to portions of existing global segments which are not mapped to physical memory on the local PIM node, at higher cost.

8 Summary and Conclusion

This paper has described the memory management requirements for DIVA, a PIM-based architecture incorporating PIMs as the only memory for a conventional host processor. Two goals of the DIVA project impose fundamentally new requirements on memory management: DIVA PIMs must perform pointer accesses within memory, and they must support both smart-memory functionality as well as conventional memory accesses. The adoption of a globally shared address space for both host and PIM nodes allows free use of pointer-based data structures. Careful partitioning of complex memory-management tasks such as paging and swapping between the host and PIM node kernel allows a single host processor to supervise many PIM nodes without overload.

Acknowledgments. The authors wish to thank the DIVA research group at University of Notre Dame, headed by Peter Kogge and Jay Brockman, for their continuing contributions to the DIVA project. This research was supported by DARPA contract F30602-98-2-0180.

References

1. D. Elliot, M. Stumm, W. Snelgrove, C. Cojocaru, and R. McKenzie. Computational RAM: Implementing processors in memory. *IEEE Design and Test of Computers*, pages 32–41, January-March 1999.
2. M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the Terasys massively parallel PIM array. *IEEE Computer*, pages 23–31, April 1995.
3. M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, J. Brockman, W. Athas, A. Srivastava, V. Freeh, J. Shin, and J. Park. Mapping irregular applications to DIVA: A data-intensive architecture. In *Proc. of SC '99*, November 1999.
4. P. Kogge. The EXECUBE approach to massively parallel processing. In *Int. Conf. on Parallel Processing*, August 1994.
5. Mitsubishi. M32R/D series: 32-bit RISC processor, on-chip DRAM. http://www.mitsubishi-chips.com/data/datasheets/mcus/m32rdgrp.html, May 1999.
6. D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A case for intelligent DRAM: IRAM. *IEEE Micro*, April 1997.
7. A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proc. of the International Symposium on Computer Architecture*, May 1996.
8. A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *Proc. of the Symposium on High-Performance Computer Architecture*, May 1995.
9. T. von Eicken, D. Culler, S. C. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

pping Irregular Applications to DIVA, a PIM-based Data-Intensive Architecture

Mary Hall, Peter Kogge*, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Jay Brockman*, Apoorv Srivastava, William Athas, Vincent Freeh*, Jaewook Shin, Joonseok Park

USC Information Sciences Institute
Marina del Rey, CA 90292

* University of Notre Dame
Notre Dame, IN 46556

Abstract

Processing-in-memory (PIM) chips that integrate processor logic into memory devices offer a new opportunity for bridging the growing gap between processor and memory speeds, especially for applications with high memory-bandwidth requirements. The Data-IntensiVe Architecture (DIVA) system combines PIM memories with one or more external host processors and a PIM-to-PIM interconnect. DIVA increases memory bandwidth through two mechanisms: (1) performing selected computation in memory, reducing the quantity of data transferred across the processor-memory interface; and (2) providing communication mechanisms called *parcels* for moving both data and computation throughout memory, further bypassing the processor-memory bus. DIVA uniquely supports acceleration of important *irregular applications*, including sparse-matrix and pointer-based computations. In this paper, we focus on several aspects of DIVA designed to effectively support such computations at very high performance levels: (1) the memory model and parcel definitions; (2) the PIM-to-PIM interconnect; and, (3) requirements for the processor-to-memory interface. We demonstrate the potential of PIM-based architectures in accelerating the performance of three irregular computations, sparse conjugate gradient, a natural-join database operation and an object-oriented database query.

1.0 Introduction

The increasing gap between processor and memory speeds is a well-known problem in computer architecture, with peak processor performance increasing at a rate of 60% per year while memory access times improve at merely 7%. To mask memory latency in current high-end computers now demands up to 25 times the number of overlapped operations required of supercomputers 30 years ago. Further, techniques designed to hide memory latency, such as multithreading and prefetching, actually increase the memory bandwidth requirements [Burger96]. Recent VLSI technology trends offer a promising solution to bridging the processor-memory gap: integrating processor logic and memory in a processing-in-memory (PIM) chip. Because PIM internal processors can be directly connected to the memory banks, the memory bandwidth is dramatically increased (up to 2 orders of magnitude, tens or even hundreds of gigabits aggregate bandwidth on a chip). Latency to on-chip logic is also reduced, down to as little as one-fourth that of a conventional memory system, because internal memory accesses avoid the delays associated with communicating off chip.

The Data-IntensiVe Architecture (DIVA) project is developing a system, from VLSI design through system architecture, systems software, compilers and applications, to take advantage of this technology for applications of growing importance to the high-performance computing community. DIVA combines PIM memory chips with one or more external host processors and a PIM-to-PIM interconnect (see Figure 1). Within a single PIM chip, we observe dramatic improvements in and bandwidth and significant reductions in memory latency. But a more important effect, and a distinguishing feature of DIVA, is the coupling of increased opportunity for concurrency with *aggregate* processor-memory bandwidth increases. Multiple memory chips can work in parallel on independent data, and perform PIM-to-PIM communication without going through the processor-memory bus.

An obvious class of applications well-suited to PIM technology is *regular* --- dense-matrix computations on large amounts of data that are "embarrassingly parallel," such as image processing. While good candidates for DIVA, such applications also perform well on conventional systems. In this domain, locality-exploiting architecture features (such as long cache lines and vector units) and compiler optimizations (such as tiling [Wolfe89]), and techniques for hiding latency (such as prefetching [Mowry92]) are effective because such applications exhibit significant data reuse, and compilers are able to predict their memory access requirements.

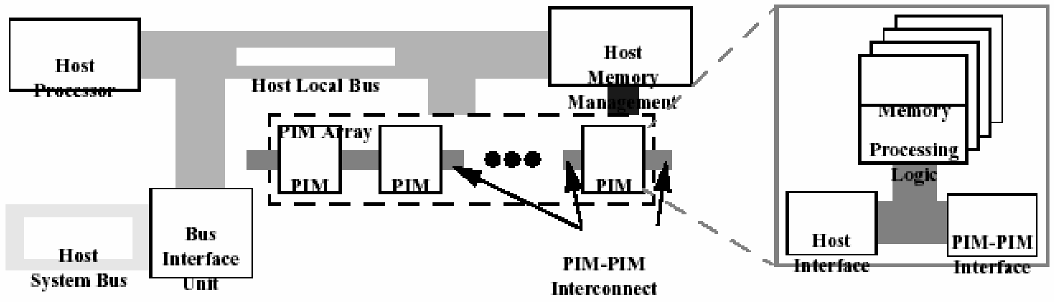


Figure 1: DIVA System Organization.

This paper argues the effectiveness of DIVA for a completely different class of applications: *irregular*, sparse-matrix and pointer-based computations with high processor-memory bandwidth requirements (*e.g.*, sparse conjugate gradient and database applications). Such applications perform poorly on conventional architectures because their control and data accesses cannot be statically predicted, and they do not make effective use of cache. As a result, their execution is dominated by waiting for memory accesses [Carter99]. DIVA can accelerate the performance of such applications by eliminating much of the memory traffic --- simple operations and dereferencing can be done *in situ* rather than laboriously moving data around the system. In addition to the reduction in memory latency for each access, there is potential for coarse-grain parallelism across multiple PIM chips. Performance improvements also result from secondary effects such as reduced host cache and TLB pollution because irregular accesses no longer need be brought into the host processor cache.

While several PIM-based architectures have been proposed in recent years, the DIVA project differs from other efforts in several ways. There are two distinct advantages to using PIMs as smart-memory coprocessors to one or more external hosts: (1) DIVA permits augmenting conventional systems in general-purpose computing environments; and, (2) applications can be gradually migrated from sequential versions that use DIVA PIMs as "dumb" memory toward fully exploiting smart-memory capabilities and parallel in-memory execution. At the same time, this co-processor model imposes fundamentally new requirements on the system software and interfaces. Supporting in-memory pointer accesses requires a new memory model, including a mechanism for address translation within memory. We also rely on the *parcel*, a mechanism for communicating computation to memory, either from a host or a PIM processor. DIVA also requires the host-to-memory interface be augmented because memory must now be able to communicate with the processor for synchronization, exceptions, to warn of high-latency events, etc.

The primary contributions of thiis paper are as follows:

- the first description of the DIVA architecture.
- the first presentation of system requirements for in-memory processing of irregular data structures.
- a detailed description of how to map applications to a PIM-based architecture, with two case studies from important irregular computations.

The remainder of the paper is organized into five main sections and a conclusion. The next section discusses background and previous work. Section 3 presents the system architecture, particularly the PIM-to-PIM interconnect. Section 4 discusses the requirements imposed on the system software and interfaces. Section 5 presents the DIVA memory model. In Section 6, we describe how a user application can be developed for DIVA, leveraging existing approaches from parallel programming. Section 7 presents three case studies of irregular computations from scientific and database computations; we present system-level simulation results to demonstrate the potential of PIM-based systems at achieving improved performance on these applications.

2.0 Background and Related Work

The concept of mixing memory and logic closer than in a CPU-Memory dichotomy is an old one. The DAPP, STARAN, CM-2, and GAPP all used many relatively small data flows positioned very close to memory arrays to implement very large SIMD machines (all with multiple data flows per chip). At least one such chip, the TERASYS [Gokhale95], was fabricated in relatively large volumes, and targeted as the main memory for one of the later Cray machines. This grew into more or less single chip systems which contained a CPU, some memory, and I/O with machines like the INMOS Transputer [Knowles91], the nCUBE [Palmer86], the J-machine [Dally92], and the SHARC (www.analogdevices.com). While these latter chips could scale to large arrays, their system architecture was a relatively conventional MPP of some form. The first DRAM-based multiple node PIM chip was EXECUBE, fabricated in 1992 and supporting a 3D binary hypercube MIMD/SIMD MPP on a single chip [Kogge94][Sunaga96]. A more recent chip is the Mitsubishi M32 R/D, where more than 2 MB of memory is tightly tied into the on-chip CPU's cache [Shimizu96].

What stopped all these designs from becoming mainstream architectures is very simple - *memory density*. Early PIM-like devices used SRAM for memory, and even with relatively primitive MOS technology, it was quite easy to put more processing power on a single chip than the on-chip data storage could feed. A rule of thumb for scientific computing is that one byte of storage for each FLOP provides a good system balance. Taking any of the previously discussed machines and computing the ratio of on-chip memory to performance (using whatever metric of performance the chip was designed for - usually not even floating point), the ratios are uniformly 0.0001 or worse. Even the EXECUBE chip had a storage to performance ratio of only 0.01. The chips were uniformly memory starved, requiring designs which included ports to off-chip memory.

This began to change around 1997, when DRAM chips with densities greater than 32 Mbits began to appear. At this density, a reasonable ratio of storage to processing can be achieved; for example, an entire video frame buffer can fit in one chip, along with logic to perform processing on it. With current CMOS projections, in a few years a single memory chip will contain more than enough memory capacity for a conventional PC. The realization that complete systems can now be placed on a single chip has led virtually every major semiconductor manufacturer to offer some form of an embedded DRAM macro that can be coupled with other predefined logic macros. At least one industrial organization has sprung up to help set standards to enable such systems [Birnbaum99].

While the technology has finally developed to the point of reasonable systems, architectures which take distinct advantage of the new capabilities have only recently come under serious study. In addition to the Mitsubishi M32 R/D, the IRAM is another system-on-a-chip embedded DRAM device with vector processing logic, designed for streaming computations [Patterson97]. Other approaches use PIM devices as the only processors in a multiprocessor architecture: a cache-coherent distributed-shared-memory system [Saulsbury96], and a large-scale distributed-memory system [Kogge96]. The Active Pages project, which is the most closely related to DIVA, associates configurable logic with each memory page to accelerate performance of an external host [Oskin98].

There are also several other architecture approaches, not based on PIM technology, designed to improve processor-memory bandwidth [Carter99][Burger97][Rixner98]. Impulse augments the memory system to perform application-specified scatter/gather operations on irregular data in the memory controller, so that contiguous data is brought into the cache [Carter99]. Imagine is a system-on-a-chip streaming architecture designed for media applications, which uses a stream programming model [Rixner98]. The DataScalar architecture is a multiprocessor system where each processor asynchronously executes the same code and broadcasts any local data to the other processors [Burger97]. DIVA is distinguished from these approaches as it supports a wide variety of parallel programming models; DIVA PIMs, with the appropriate interconnect, can be used in a scalable system with an unlimited number of chips, not just single chip solutions.

The DIVA architecture and the material presented in this paper is distinguished from these previous approaches in several ways: (1) unlike most of these other approaches, we consider an architecture where smart memory is optionally used to improve performance of a standard host processor; (2) we develop a system that can support in-memory manipulation of both regular and irregular data structures; and, (3) we consider the requirements imposed on the system architecture and system software for mapping application execution between host and memory.

3.0 Overview of DIVA System Architecture

In Figure 1, we show a small set of PIMs connected to a single external host through a host-memory interface; through this interface the host processor performs standard reads and writes, augmented as discussed in Section 3.3. The PIM chips communicate through separate PIM-to-PIM channels to bypass the system bus with additional memory traffic from parcels used to spawn computation, gather results, synchronize activity, or simply access non-local data. The separate interconnect is provided because PIM-to-PIM communication requires greater bandwidth than can be achieved with a conventional memory bus.

3.1 PIM VLSI Component

A PIM is a VLSI memory device augmented with general and special-purpose computing hardware. A PIM may consist of multiple *nodes*, each of which are comprised of a few megabytes of memory and a node processor. The inset in Figure 1 shows a PIM with four nodes. The nodes on a chip share resources for communication with the rest of the system. As a result each chip contains a single PIM Routing Co-processor (PiRC) and a host interface. We anticipate that DIVA PIMs, like many other PIM chips, will be split roughly 60% memory and 40% logic (reflecting the importance of memory density).

Within a single node, shown in Figure 2, the processing logic consists of a standard scalar microprocesor including a floating-point unit and a special DIVA functional unit called an *At-the-Sense-Amps Processor (ASAP)*. The key idea behind the ASAP is to perform *wide* operations on aggregate objects stored within a row of the local memory array. Rather than selecting a 32-bit object from the row as is done with conventional scalar processing, the ASAP unit operates on up to 256 bits in a single processor cycle. This fine-grain parallelism offers additional opportunity for exploiting the increased processor-memory bandwidth available in a PIM. The ASAP unit can be used to perform bit-level operations such as simple pattern matching, or higher-order computations such as searches, limited pointer chasing, and associative and commutative reduction operations. Details on a related wide-word unit are discussed elsewhere [Brockman99].

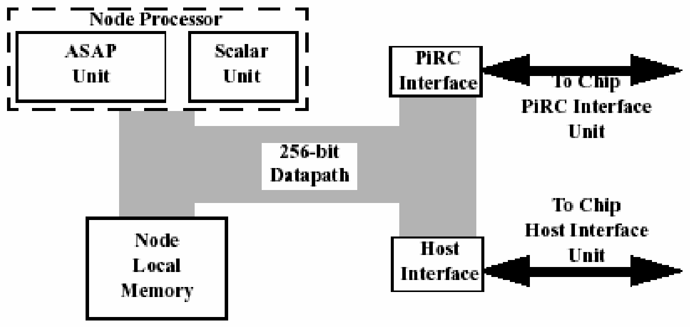


Figure 2: Processor-In-Memory Node Organization.

3.2 PIM Interconnection

We anticipate PIM chips to be physically grouped as conventional memory chips, mounted on DIMM modules, as shown in Figure 3. Bounded by host bus loading constraints, the number of PIM chips in a hosted cluster is in the range of 32 to 64 chips, depending on how many PIM chips can be packed onto a DIMM module. The PIM-to-PIM interconnect must then be amenable to the dense packing requirement of DIMM modules. Obviously, low latency and high bandwidth are also desirable properties of this interconnect. Furthermore, this network must be scalable to allow the addition or removal of modules from the system. This combination of requirements favors a one-dimensional network. Although higher-dimension networks offer lower network diameters, they are not easily scalable in all dimensions, especially in a densely packaged system. Also, the dense packing achievable with one-dimensional networks allows more data signals per channel. Hence, the slightly larger distances (in hops) of message traversals in a 32- or 64-hop one-dimensional network are compensated by shorter messages (in flits). Furthermore, router cycle times are faster in one-dimensional network routers because of simpler switching decisions.

The PIM interconnect requirements closely resemble those of interconnect in embedded scalable systems. We therefore use the interconnection network of one such system, the Package-Driven Scalable System (PDSS) [Steele97], as a model for designing the DIVA PIM interconnect. The DIVA PIM interconnect is then a point-to-point bidirectional ring using wormhole routing and the Red Rover routing algorithm [Draper96] to effect deadlock-free routing. It routes fixed-sized packets and uses source routing to achieve low latency. The interconnect is implemented by PIM Routing Co-processor (PiRC) devices - one per PIM chip.

Later generations of DIVA systems are envisioned to contain hundreds and even thousands of PIM chips. Clearly, the advantages of a flat ring topology do not extend to systems of this size. A more complex network scheme will be needed. One possibility is another level of interconnect for connecting host/PIM clusters. To provide adequate aggregate bandwidth, this higher-level interconnect will have to employ channels with greater bandwidth than those of the PIM chips. The details of these channels are beyond the scope of this paper.

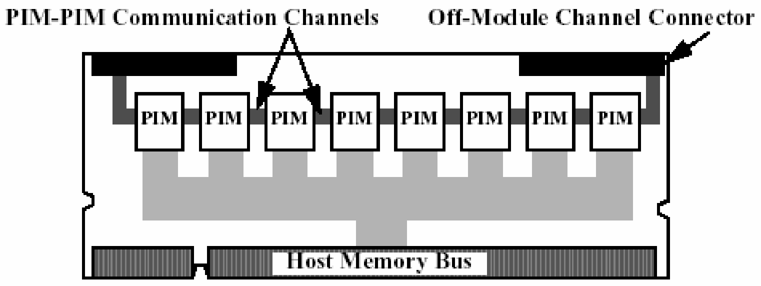


Figure 3: PIM DIMM Module Organization.

4.0 Required Mechanisms

We now present a collection of key mechanisms in DIVA.

4.1 Parcels

A *parcel* is the general mechanism for coordinating computation in memory, communicating data and performing synchronization across components of the DIVA system, a refinement of the parcel concept described previously [Brockman99]. Similar to an active message [vonEicken92], a parcel incorporates data and an encoded operation to apply to the data; a parcel is directed to a memory object, not a process or processor. A parcel has the following four fields:

- *pid:* indicates which process issued the parcel.

- *object:* the virtual address of the primary object the parcel will modify or access, used for routing the parcel.
- *command:* an integer encoding the action to be performed, which may refer to a compiled function stored on the PIM.
- *arguments:* (other than *object*), specified as virtual addresses.

An obvious requirement on parcels is small size, to prevent overloading the host-to-memory interface and PIM-to-PIM interconnect. In DIVA, we expect a single packet to consist of a header and 256 bits of payload. A parcel requiring more bits must be sent in multiple packets. A related requirement is that processing parcels must be efficient (see 3.2.1).

In addition, protection must be provided on *arguments*, *pid* and *command* fields; the protection on memory accesses cannot rely on standard host mechanisms as the parcels pass virtual rather than physical addresses to the memory. Also, the order of parcel processing must preserve sequential semantics, but parcel execution should be overlapped to exploit parallelism. To accomplish these goals, we employ optional sequence numbers on parcels when a specific ordering of processing is required.

4.2 Host-Memory Interface

In the initial DIVA prototype, an underlying assumption is that DIVA PIM devices can also serve as conventional memory, so that they can be used as smart-memory coprocessors in a standard system. For this reason, the PIM VLSI device is being designed with a host interface consistent with the standard memory interface typical of commercial memories. This enables PIMs to be packaged in the form of DIMM modules with provisions for top-plane interconnections to support the PIM-to-PIM communication fabric. However, unlike commercial memories, computation activities give rise to new problems: how to communicate internal exceptions and possible memory busy conditions to the host system. These issues are being addressed as part of the larger system architecture.

5.0 Memory Model

Systems with smart memory resemble both uniprocessors (or small SMPs) with large memory, and large, heterogeneous multiprocessors. The semantics are made precise by the DIVA memory model, developed from the following list of requirements:

- a simple virtual machine for both programmers and compiler writers;
- application-level visibility and control of data placement;
- high overall performance;
- scalability to many PIM chips, larger PIM chips, and multiprocessor hosts;
- compatibility with conventional memory models and memory interfaces;
- support for virtual memory (i.e., paging to/from disk); and,
- a host-independent PIM chip architecture.

These requirements look ahead toward future uses of PIM chips, augmenting all sorts of systems and used to accelerate all sorts of applications, both at the small and large scale.

5.1 Parcel Buffers

For high performance, applications must communicate with PIM chips without invoking the host operating system. A conventional memory interface supports this naturally, but cannot generally guarantee atomicity or ordering when caching and write buffers exist. Each PIM chip therefore has a second intelligent interface, the Parcel Buffer, which is mapped into each process as a (roughly) parcel-sized piece of SRAM. The host OS ensures each process uses a different physical address for the multiply-mapped buffer, so the interface can identify the source of each transaction. Hardware in the interface transparently manages ownership of the buffer using a wait-free protocol [Herlihy91] that can be implemented simply at the application level without supervisor state interactions; this interface hardware ensures that access patterns are grammatically correct. To communicate a parcel, a process reads or writes fields in the buffer, then performs a final read on a status field to pass the parcel to the PIM chip internals. In the rare case of corrupted accesses, a failure status is returned, and the application can retry.

5.2 Address Translation

Parcels, application code and data contain virtual addresses. For PIM processors to interpret these, they must have access to translation information, or there must be some fixed relationship between virtual and physical addresses. The latter option is simpler to implement, but was determined to be too restrictive. Each PIM thus contains translation hardware, and tables managed by the host. Any virtual page can reside on any PIM. However, the hardware is simplified by the characteristics of the system. For instance, for performance, a PIM needs to be able to rapidly determine if an address is local to its own memory bank, and find the physical address if it is. However, if the address is not local and communication is required, the additional cost of the non-local translation is negligible.

Each PIM therefore maintains translations for those virtual pages currently residing on it, plus part of a global, distributed table (similar to a home node concept as presented in [Saulsbury95]). Non-local translations are obtained by querying the distributed table, or, equivalently, submitting the virtual address in a parcel, for forwarding to the PIM where it resides. Advantages of this approach are that the translation tables on each PIM scale well; every address can be accessed in at most two parcel transmissions, and the application can optionally maintain location hints and use them to reduce this to a single parcel transmission in performance-critical cases.

5.3 PIM Memory Organization

The DRAM in the PIM subsystem is the primary storage for the DIVA system, and can be treated physically as a uniform, undifferentiated RAM. However, during operation the system uses the memory in three distinct ways, making it helpful to organize the memory on each PIM node logically into three regions according to whether it is used primarily by the host processor, primarily by the PIM processor, or significantly by both. These regions may be either physically contiguous or interspersed, and memory allocation within these regions can either be initiated by explicit system calls in the application, or undertaken at load time for all applications by the loader or start-up code. A flexible combination of static and dynamic allocation is usually most convenient for the user, but for this discussion assume explicit system calls are used.

An advantage of making this distinction is that different, optimized memory-management hardware can be used on each of the regions. As modern processor architectures demonstrate[IBMMot94], there is no conceptual problem with having multiple translation mechanisms in place, as long as they provide consistent virtual-to-physical mappings and access permissions.

Dumb Memory: Initially, the application is a normal (say Unix) process on the host. The various regions of its virtual address space (typically the user code, heap and stack and one or more kernel segments) are mapped as usual to some set of pages in DRAM, with some possibly paged out to disk. If the system memory contains both ordinary DRAM and PIM DRAM, these normal pages can be mapped into the ordinary DRAM, since they are never directly accessed by PIM processors. If all memory is PIM memory, the system can simply note that these pages are only accessed by the host, and that they need not appear in PIM-processor translation tables. A major use of dumb memory will be application code for the host CPU, which is meaningless to the PIM processors; also, many host processes will never require PIM services at all, and will remain in this configuration.

Internal Memory: If an application elects to use the PIM processing, the first step is to allocate and initialize a region of memory on each node to be used by that node for its local processing needs. These include: a small run-time kernel for parcel management, synchronization and exception handling; code for the application-level methods supported by the PIM; and storage for executing PIM programs such as buffers and stacks.

In practice, efficiency dictates whether this initialization step occurs at host boot time, application load time, during application start-up, following an explicit system call, or transparently when the first PIM operation is attempted; some combination of these initialization steps can be profitably supported. For instance, a basic PIM kernel could be installed on each node at system boot time, as could code for any widely useful PIM methods. Application load time is a good time to install application-specific method code that is used frequently; individual methods from a large system library could be loaded dynamically on demand during application execution.

User-level code on the host never accesses this internal memory during normal operation. To the host, the internal pages appear within the supervisor region, like the kernel and its associated data structures. Moreover, the host only needs to access them under exceptional conditions, *e.g.,* application loads, service requests, and errors. Access from the host is thus guaranteed to be infrequent, through trusted code with access to translation tables. On the other hand, access to internal regions by the PIM processor needs to be highly efficient and well protected, since it is used for everything from local OS code and data to execution stacks and working memory for the many light-weight user-level methods launched in response to parcels during normal operation.

One can exploit these asymmetric requirements by adopting a memory-management approach for the internal memory that is very convenient for the PIM processor, but perhaps quite unrelated to the memory-management hardware on the host. A particularly useful scheme, planned for the prototype, is to give each lightweight local context on a PIM processor eight variable-sized segments or pages of internal memory, each defined by virtual and physical base addresses, size and access permissions. By convention, these are assigned to the following:

1. Supervisor-level kernel code (shared by all contexts on the node)
2. Supervisor-level kernel data and stack (shared by all contexts on the node)
3. User-level code (shared by all contexts in the same application)
4. User-level data (shared by all contexts in the same application)
5. User stack (unique to each context)
6. Miscellaneous (possibly unique to each context)
7. Supervisor-level parcel buffer device (shared by all contexts on the node)
8. User-level parcel buffer device (shared by all contexts in the same application).

Translation of internal virtual addresses can be made extremely fast and efficient by adopting some simple conventions, e.g., high bits of all the page virtual starting addresses are the same, the next three bits specify the page number, and the size is a power of two. Then, the TLB simplifies to a look-up table, the translation information for a lightweight context fits into 256 bits, and can be switched in one clock cycle. Since PIM nodes do not access each other's internal memory, the same virtual address range can be used for internal memory on every node, making PIM contexts relocatable from one node to another.

Global Memory: The next step in setting up to use the PIM features is to allocate DRAM on each PIM for use as smart "global" storage. This can be done at run time by a series of system calls such as **`mem_alloc(pim_node, virtual_address, size)`**, which allocates a region of memory of **`size`** bytes on **`pim_node`** and maps it at **`virtual_address`**, an unmapped virtual address range within the application address space. Unlike the dumb memory, whose mapping is visible only to the host process, or the internal memory, whose mapping is visible only to the associated PIM node and the host OS, the global memory is visible to the host process and to all PIM nodes involved in the application. Although only the host and the local node where the data resides can access an element of global memory directly (i.e., by read and write instructions), pointers to global objects are meaningful to all nodes and to the host, and can be communicated freely within parcels. Once global memory has been allocated, the host process can set up any initialized data by writing to it. In practice, global memory will make up the majority of memory in a data-intensive application using the PIM features. It is important that access to this memory be efficient from both the PIM and the host, and this therefore presents the greatest implementation challenge. Address translation must be compatible with both host CPU and PIM hardware. The page size must therefore be equal to or a multiple of the hardware-supported page size of the host CPU. Also, each PIM node should ideally be able to hold in a fast TLB the translation information for all active global pages resident on it, to avoid the frequent TLB misses that would occur on an irregular application. This therefore suggests that global memory pages should be large; in the prototype, one simplifying option under consideration is a single very large global page (per application) on each node.

Parcel Buffers: The final step in invoking the PIMs is to request the host OS to allocate and map one or more virtual parcel buffers, for use in communicating parcels with the PIM system. Parcels are then sent to individual nodes to start the PIM computation. Finally, when the computation is complete, one of the PIM methods communicates this to the host, typically by setting a flag in the global memory, and the host picks up the results from the global memory.

The overall memory structure for a typical DIVA application is shown in Figure 4. In the far left column is the

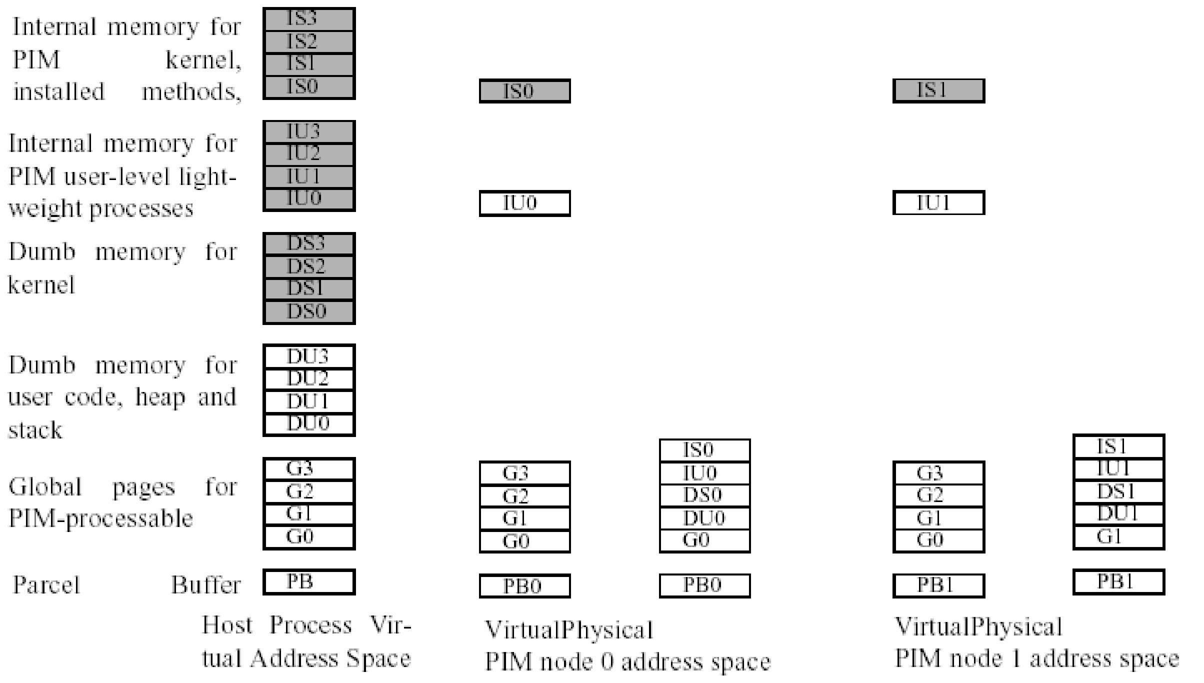


Figure 4: Decomposition of a host process address space across multiple PIM nodes. Shaded regions have supervisor-level protection.

virtual address space of a typical host application process, where each rectangle represents a page or segment from one of the memory regions. Shaded pages are accessible only while in supervisor mode. The label indicates whether the page is used for global data (G), dumb user or system pages (DU or DS) or internal user or system pages (IU or IS), as well as the PIM node (0-3) where the page currently resides. The second column shows the subset of pages visible to a method executing on PIM node 0. The third column shows the subset of pages actually resident on node 0. The last two columns show the same information for node 1. Note that global pages are visible from all nodes, while internal pages are visible only on their local nodes, where they appear at a common virtual address.

4 Coherence Management

In any system with distributed processing, the distributed information needs to be kept coherent, and a consistent model of memory access must be chosen and maintained. Conventional NUMA and COMA models are suboptimal for irregular, data-intensive applications. Specifically, in a NUMA or COMA model, a reference to remote data by a local node causes the remote data to be automatically moved or copied to the local node, where it is made available under the same virtual address as the remote version. In general, the overhead of supporting this model becomes excessive for irregular applications, where there is by definition great potential for false sharing, and little temporal locality.

The philosophy in the DIVA system is therefore to move the computation to the data, rather than move the data to the computation. At any one time, the data at a virtual address is located on exactly one PIM node, and there are no cached copies on other PIM nodes. Global pages can be moved from one node to another for load balancing purposes, but this is a heavyweight operation that should be used infrequently and explicitly managed by the operating system. Consistency of the distributed address translation table must be maintained, but since this changes relatively rarely, software coherence methods are adequate.

During normal operation, therefore, data coherence issues do not arise *between* PIMs, and there is no need for a sophisticated, hardware-supported coherence mechanism. The movement of code is a much simpler problem, since code is read-only, and can be replicated easily. Moreover, the only references to code that get passed in parcels are indirect references that index into a method table, so the translation mechanism for code references is built into the application. The result is a memory model that can be supported by fairly simple hardware in the PIM nodes, independent of the host CPU details.

The remaining coherence issue, namely between the PIM system and the host, is the most difficult. Individual cache lines may be cached by the host processor(s). The simplest solution, adopted in this prototype, is to always explicitly flush PIM-accessible data, or keep it uncached. A more transparent approach is for each PIM to track ownership of individual cache lines, and request writebacks from the CPU caches as necessary. The hardware for this on each PIM is not excessive and scales well, so this is a suitable long-term solution. However, broader issues suggest it is premature to implement in our prototype. As stated at the beginning of this section, our goal is a memory model that is independent of which processor is used as a host; the mechanism for requesting writebacks is processor specific, and usually involves the requestor driving the address bus. In a large system with many potential requestors, this introduces significant arbitration, electrical drive, and portability problems. In the long term, it would be better to develop a standard (probably network-based) memory-to-processor channel for this activity, which would find other uses in smart memory systems.

Although the explicit flushing is a burden, either to the programmer or compiler, it is not expected to degrade performance significantly. In practice, even with automated hardware, the user would probably obtain higher performance in some applications by manually flushing cache lines anyhow, to minimize the number of writeback requests.

6.0 Developing Applications in DIVA

The success of a new architecture is highly dependent on the ease in which software can be developed for it. It should be straightforward to develop correct programs, even if it is somewhat more difficult to effectively exploit the performance-enhancing features of the architecture. DIVA offers a smooth migration path for developing applications. First, the applications programmer can begin with a standard sequential program, which will run correctly with no modification by using the PIMs as standard memory. Then, either the compiler or programmer can exploit the PIMs as smart memory in portions of the application where this is deemed profitable, gradually migrating the original sequential application to make full use of the DIVA architecture.

To the applications programmer or compiler, the abstract DIVA architecture appears very similar to a distributed-shared-memory multiprocessor. The host can serve as a master to coordinate activities on the PIMs. Each node on a PIM processor acts as a worker processor waiting for work, and possibly initiating work on other

IMs through the parcel mechanism. The memory associated with a PIM node can be thought of as its local memory. The PIM node can access a datum on other memory chips through a global address space without need to know its exact location. Coherence of data shared across PIM chips is not guaranteed by the hardware and must be managed by either the compiler or programmer, similar to what is required in the Cray T3E. Also as with distributed-shared-memory multiprocessors, locality of data accesses is very important to good performance.

Because of these similarities to a distributed-shared-memory multiprocessor, most parallelizing and locality-management compiler techniques and parallel programming paradigms can be leveraged for DIVA. Applicable compilation techniques include automatic parallelization for both regular [Blume96] [Hall96] and irregular applications [Rinard97], and data and computation co-location [Anderson93]. Explicitly parallel programming languages that permit some programmer control of locality are also applicable, such as High Performance Fortran and its extensions for irregular applications, Olden [Carlisle95], and CC++[Foster95]. As discussed in Section 4.1, the parcel mechanism is really a refinement, tailored to the DIVA architecture, of active messages, which were developed for message-passing multiprocessor systems [vonEicken92].

While there are many similarities between programming for DIVA and parallel programming, there are several important differences. One additional requirement is keeping the host cache coherent with the PIM memories. As discussed in Section 5.4, this is accomplished with explicit flushing, immediately prior to sending a parcel from the host, of objects in the host cache that may be touched by the PIM computation. In keeping with the above stated goal of making correct programs easy to develop, the required flushing can be optionally automated by the compiler through analysis of the object and arguments associated with the parcel. Further, DIVA applications can exploit fine-grain parallelism using the ASAP functional unit for operations on aggregate data objects, which demands a combination of compiler technology and a user development environment for exploiting complex ASAP-oriented computations (*e.g.,* string matching). Other high-level operations such as memory management can be optimized for the PIMs to improve the locality of pointer-based computations. As an example, when building a tree data structure in parallel, each PIM can locally allocate a subtree, with the host sequentially connecting the subtrees in the upper level of the trees. Locality for each subtree is then ensured.

An important component of the DIVA project is a large software effort to develop application programmer libraries, and compiler and run-time system support. The DIVA compiler, either automatically or in response to programmer specification, partitions computation and data across host and PIMs. This partitioning requires that it must generate code that interfaces with the operating system to control data placement on the PIMs, generate code to load application-specific PIM code onto the memories, and also generate parcels in the appropriate places in the code to initiate PIM computation, communicate and synchronize. This high-level code must then pass through separate backend compilers: one for the host, for which we can use an existing native backend compiler; and one for the PIMs, which requires a DIVA PIM-specific backend that generates standard RISC as well as ASAP instructions. There are also separate run-time systems for the host and PIMs. The host run-time system performs similar functions to a standard architecture-independent parallel run-time library (e.g., Pthreads), managing threads and synchronization. The PIM run-time system is a small, DIVA-specific system, primarily for parcel processing.

As part of the software development efforts, we are currently retargeting the Stanford SUIF compiler system to DIVA, allowing us to take advantage of its wealth of compiler analyses for distributed-shared-memory machines. In addition, we are developing an extensible approach to support compiler and programmer generation of ASAP instructions that are seamlessly integrated into the PIM backend. Since DIVA is targeting irregular computations, we are also investigating a memory management library for dynamic generation and reorganization of irregular data structures.

7.0 Case Studies

To derive preliminary performance estimates for complete applications, we developed a simulator for the major system components of DIVA. The simulated architecture consists of a host processor, and a number of PIMs interconnected via a PiRC ring network. We simulate computations executing on the host and PIMs using Shade [Cmelik94]. Shade executes application programs and generates traces under the control of a user-supplied trace analyzer. We simulate parallel execution in our experiments by recording the simulated time at the beginning of a parallel section and setting the parallel execution time at the end of the concurrent execution to be the maximum value of the simulated time by each of the participating PIM nodes. The Shade-based simulator does not directly model the PiRC interconnection. To account for network latency and congestion, we generate traces of time-stamped network requests for each application, and use these traces as inputs to a network simulator [Draper96]. The throughput and contention derived by the network simulator are then used as parameters to the Shade simulator.

The PIM chips modeled in these experiments are much simpler than what was presented in Section 2. There is a single node per chip, and we only consider applications that use standard scalar integer and floating-point processing on the PIM nodes (*i.e.,* no ASAP instructions). These simplifications reduce the contention for on-chip resources, and allow us to get meaningful early results from the simple Shade-based simulation strategy. We anticipate that the multiple processing nodes per PIM chip and the ASAP functional units planned for the actual DIVA implementation will yield much better on-chip computation rates, albeit with additional costs due to contention for internal memory banks and PiRC channels.

In our simulations, each PIM node consists of a PIM processor, a 2M-byte memory bank, a host interface and a PiRC network interface. Since processor technology is optimized for speed and DRAM technology is optimized for density and yield, the PIM processing logic is expected to be slower than the host processor logic. Based on projections, we assume that the PIM processor cycle is twice the host processor cycle. The PIM node memory bank is organized as 8192 2K-bit memory rows, and the DRAM interface provides a 256-bit sub row per memory access. We assume the first access to a 2K-bit row (random-mode access) takes 2 PIM cycles, and each subsequent access to the same row (page-mode access) takes 1 PIM cycle. These parameters are based on current memory speeds [Kogge98].

The host has separate instruction and data on-chip caches, and a unified off-chip second level cache. We model a parcel issue as a sequence of writes to specific memory addresses, the last of which triggers the delivery of the parcel. Coherence between the caches and memory is enforced by software (e.g., the compiler), using an instruction to flush data from the cache. At a flush instruction, the simulator invalidates the cache line and, if the line is modified, writes it back to memory. We summarize the simulation parameters in Table 1. We now

	Cache Parameter	**Instruction L1**	**Data L1**	**Data L2**
Host Caches	***size***	32 K bytes	32 K bytes	1 M bytes
	associativity	2	2	2
	line size	64 bytes	32 bytes	32 bytes
	replacement	LRU	LRU	LRU
	write policy	write back	write back	write back
	latency (hit)	1 cycle	1 cycle	10 cycles
	latency (miss)	10 cycles	10 cycles	100 cycles
PIM Node	***processor cycle***	2 cycles[a]		
	memory size	2 M bytes		
	memory row size	256 bits		
	memory latency	1 cycle*(page mode), 4 cycles* (random mode)		
PiRC Network	***channel width***	32 bits		
	network cycle	4 cycles*		

a. Host processor cycles

Simulation Parameters used in Application Studies.

present results on three applications evaluated with this simulation methodology.

7.1 NAS Sparse Conjugate Gradient (CG)

CG implements a linear system solver using a conjugate gradient iterative method. Its main data structures are three very large arrays of floating-point double-precision values. The main computation consists of a sparse matrix-vector product (see Figure 5(a)) and accounts for about 80% of the total sequential execution time. The computation is structured as a single loop performing commutative and associative updates to array Y indexed by the values in the ROWIDX array. The sparseness of the computation is derived from the indirection of the accesses to the Y array whereas both arrays A and X are accessed using simple loop indexing functions.

To effectively map this computation to DIVA, we parallelize the execution of the sparse matrix-vector product by exploiting the commutativity and associativity of the addition operations. In this version, each PIM node has a local copy of array Y (named PRIV_Y), and performs its updates on its own private copy; after all PIMs complete their local computation, the local results are merged using a parallel reduction algorithm. The parallel reduction algorithm ensures that there is no network contention during the communication phase. However, since each PIM node has to communicate its copy of array Y to other nodes, the total amount of communication increases with the number of PIMs, as well as the number of steps of the parallel reduction. This example makes use of a lightweight run-time system and the parcel communication mechanism to generate and manage concurrency. The basic code generation strategy is for the compiler to split the computation between the host and the PIM nodes and to initiate the computation on the PIMs by sending parcel, using the *SendParcel* primitive. PIM nodes are activated by the receipt of a given parcel and proceed to execute the code associated with it. This code might in turn generate other concurrent computation on the same or on other PIM nodes. The host can enforce termination of a given computation using an explicit barrier synchronization construct (*Barrier*) or implicitly through memory. Also included in this run-time system is a *Flush* primitive that allows the compiler to maintain the consistency of the data between the host caches and the PIM nodes.

(a) Original Loop Nest.

```
DO J = 1, N
  DO K = COLSTR[J], COLSTR[J+1]-1
        Y[ROWIDX[K]] = Y[ROWIDX[K]] + A[K] * X[J]
```

(b) DIVA Host Program.

```
Flush(Y);
PartitionSize = Sizeof (ROWIDX) / NumPimNodes;
for (i=0; i<NUM_PIMNODES; i++) {
   Send_parcel (ROWIDX[I*PartitionSize], LoopBody, PartitionSize,
                 A[I*PartitionSize], PRIV_COLSTR[0,I],PRIV_X[0,I],Y);
}
Barrier();
```

(c) Code for PIM node command *LoopBody*.

```
 BarrierEnter();
for (j=1; j<=N; j++) {
   Lower = Max(PRIV_COLSTR[J], PIMID*PartitionSize);
   Upper = Min(PRIV_COLSTR[J+1]-1, (PIMID+1)*PartitionSize-1);
   for (i=Lower; i<=Upper; i++) {
          K1 = K - PIMID*PartitionSize;
          PRIV_Y[ROWIDX[K1]] = PRIV_Y[ROWIDX[K1]] + (A[K1] * PRIV_X[J])
   }
}
ParallelReduction(Y,PRIV_Y,PIMID,NUM_PIMNODES);
BarrierRelease();
```

Figure 5: CG Matrix-vector product and its mapping to DIVA.

Figure 5(b) and Figure 5(c) present the corresponding code for the DIVA architecture, which makes use of the parcel and synchronization primitives to orchestrate the computation. Figure 6 illustrates graphically the data mapping for the various arrays in this computation for a system with 4 PIM nodes.

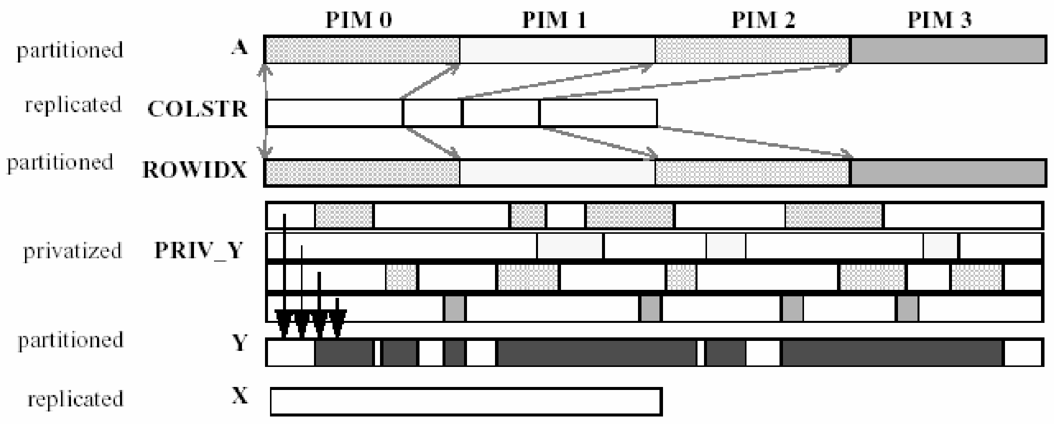
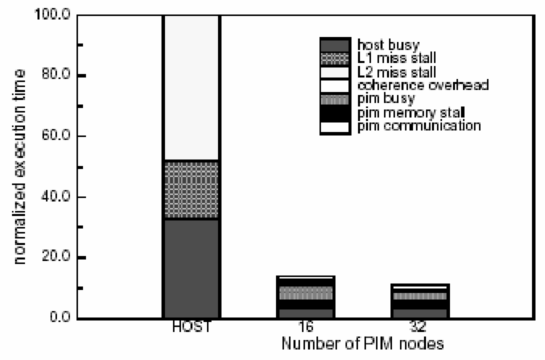
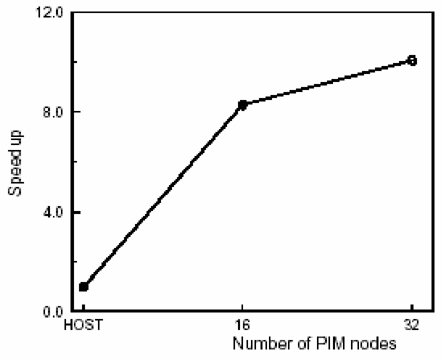


Figure 6: Data Mapping on DIVA for CG.

Figure 7 illustrates simulation results for this application. We separate original sequential execution into several components in Figure 7(a). The host busy category accounts for the time spent executing instructions. The L1 and L2 miss stall categories represent time spent waiting for memory accesses to be satisfied from either the L2 cache or main memory. In the version of the program that executes the matrix-vector product on the PIMs, we show time spent in the host and on average in one PIM, and we include additional categories (the host is idle during PIM execution, so this is an accurate reflection of overall execution time). The coherency overhead refers to time spent by the host flushing cache lines prior to execution on the PIMs. Note that additonal coherency overhead is charged as L1 and L2 cache misses in the host when PIMs are used; by flushing data from the cache prior to PIM computations, extra cache misses in the host may occur in later host computation. This cache miss effect due to flushing is not significant in the programs presented here because the irregular accesses in the PIM computations were polluting the host cache when executed on the host. Additional categories show time spent in the PIMs, including PIM-to-PIM communication overhead and time spent in local memory stalls.



(a) Execution Breakdown



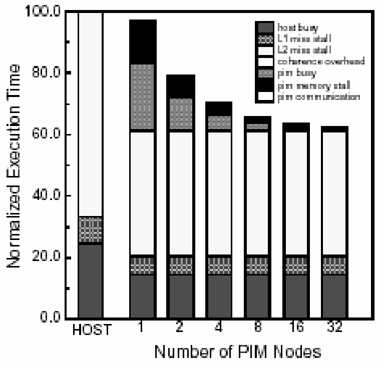
(b) Speedup

Figure 7: Execution Breakdown and Speedups for CG.

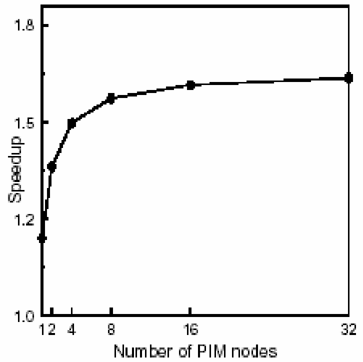
As the results in Figure 7(a) indicate, the original application suffers significantly from poor cache locality with overall L1 and L2 cache miss rates of 15% and 20%, respectively. When the matrix-vector product is executed on the PIMs, much fewer accesses to array Y are brought into the host, and the miss rates on L1 and L2 cache were reduced respectively to 10% and 7%. As Figure 7(a) shows, this contributes to a significant reduction of the application time waiting for results from memory. Figure 7(b) shows the overall application speedups for different numbers of PIM nodes as compared to the entire application executing on the host. At 16 PIMs, the application speedup is more than 8 over the original sequential execution time. While this application scales very well for up to 16 PIM nodes, the problem size we use is too small relative to the overhead of the reduction computation to scale much beyond 32 PIMs.

7.2 Hash-Based Natural Join

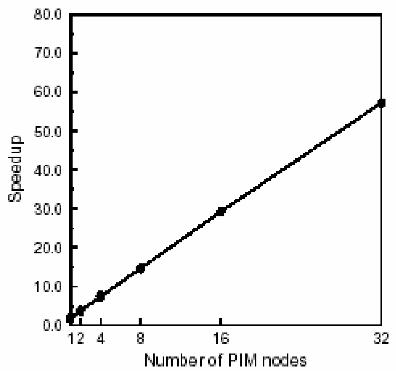
The Natural Join is a fundamental operation in relational database systems. It consists of generating all possible combinations of tuples for two relations R and S with a common attribute A. In the implementation used in these experiments, the algorithm builds a hash table for each of the relations R and S indexed by the attribute A. Then, for each hashed value in the table, the algorithm joins all tuples of the two relations that have a common value for the attribute A.



(a) Execution breakdown



(b) Overall Speedup



(c) Speedup for Join phase

Figure 8: Execution Breakdown and Speedups for Natural Join.

The strategy to map this application to DIVA is to distribute the hash table along contiguous blocks of the table entries. Each PIM node has a set of consecutive entries of the hash table and the hash-table collision lists corresponding to each of the table entries it owns. Once the host processor has constructed the distributed hash table, natural join operation proceeds by having each PIM node computing a local natural join operation. At the end, the host simply scans the partial hash tables local to each PIM node to read the results.

Figure 8 shows performance results when the first phase, constructing the local hash tables, is performed by the host, and the second phase, the join of local hash tables, executes in the PIMs. The speedups for the join phase of the computation are superlinear, as shown in Figure 8(c). These superlinear speedups result from the combined effects of the smaller memory latencies at the PIMs, as compared to the cache miss latencies suffered by the host, and the parallelism obtained by distributing the computation across PIMs. Due to Amdahl's Law, the overall speedup is limited, as the first phase, which accounts for about half the baseline execution time, is executed sequentially on the host. Even more speedup is possible from two sources, both of which we are exploring: (1) building portions of the local hash table in parallel on the PIMs and merging the results; and, (2) performing in parallel on the ASAP unit the comparison of a key from the R-tuple with that of several S-tuples with the same hash value.

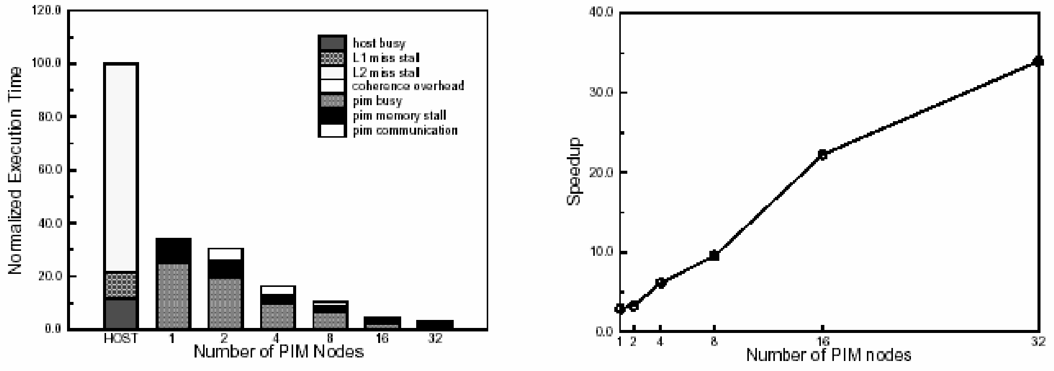
7.3 Object-Oriented Database Benchmark (007)

The 007 application implements a representative object-oriented database for CAD applications. The database schema defines several one-to-one and one-to-many relationships among database objects. These objects consist of documents, manuals and base or complex assembly components. Each complex assembly component is defined hierarchically in terms of other base or complex assemblies or base assemblies. Base assembly components are defined in terms of composite parts which in turn consist of more than one library atomic part. Each of these objects have specific attributes such as a unique identifier, creation date and other type-specific fields.

This database application was originally developed at the University of Wisconsin to study the performance of various database management systems [007]. We have ported this application to a C++ stand-alone program by implementing the dictionary and relations abstraction using hash-tables and linked lists in a total of 9,000 lines of C++ code. Our performance evaluation concentrates on a specific database query, query #6. Query #6 finds all assemblies (base or complex) B that reference (directly or transitively) a composite part with a more recent build date than B's build date. This query is implemented using set operations over the database relations and extensively uses the iteration abstraction from C++ to access successive objects in a given relation.

Besides the overall organization of the database objects in a graph data structure, the database schema also relies heavily on singly-linked and hash-table pointer-based data structures for indexing of the object in each category (documents, manual, base assemblies, etc.). The primary access pattern over the indexing structure traverses a singly-linked list or a hash-table, searching for a particular subset of objects matching a given predicate. In addition, the application also traverses the overall graph structure of the objects in the database. Such traversals perform poorly on conventional systems because they exhibit almost no temporal reuse of memory accesses, and there is little spatial locality due to the way the pointer-based data structures are created.

To take advantage of the PIM architecture, we perform two key transformations on the original application. The first transformation takes advantage of the fact that the computation accesses a set of objects; the order in which the elements of the set are accessed by the application is irrelevant, so these accesses can be performed in parallel. The second transformation restructures the code so that the PIM nodes traverse the linked data structure that represents the relations in the schema and selects the set of objects the computation needs to access. Each PIM selects a subset of the objects in the relation from its local memory only. The host then gathers the partial results and constructs a larger set. The host is responsible for any updates to the storage. Figure 9 shows the execution time breakdown and speedups for 007.



(a) Execution breakdown

(b) Speedup

Figure 9: Execution Breakdown and Speedups for 007.

The results show an impressive superlinear speedup. As the execution breakdown reveals, this result is due to the severe performance impact of the L2 miss stalls (almost 80% of the sequential computation for this query) for the case where only the host executes the computation. When the computation is partitioned across the PIM nodes, each PIM fetches data from its local memory and communicates very infrequently. The overhead of coherence is also negligible for all runs, as the query does not update the objects in the database but rather collects overall statistics. As a result, the performance scales well up to 16. For 32 PIM nodes, speedup, while still impressive, trails off a little due to the relative frequency of communication compared to computation for this data set size.

8.0 Conclusions and Future Work

This paper has described the DIVA system, an architecture incorporating PIM devices as smart memories to one or more external host processors. Other distinguishing features of DIVA include its PIM-to-PIM interconnect and explicit support for in-memory operations on irregular data structures. In this paper, we presented system-level requirements for in-memory acceleration of irregular applications. We presented three case studies, sparse conjugate gradient, natural join and an OO7 database query, to demonstrate how irregular applications can be mapped to the DIVA architecture. High-level simulation results show a speedup for all three applications, resulting from increased processor-memory bandwidth, much more effective use of cache on the host processor, lower latency accesses and parallelism.

Future descriptions of the DIVA project will include details of the PIM VLSI device, architecture studies using a high-fidelity system simulator based on RSIM, the DIVA compiler and run-time systems, and further application studies.

Acknowledgments

The authors wish to thank Thomas Sterling for his early contributions to this project. The DIVA project is sponsored by DARPA contract F30602-98-2-0180.

References

[Anderson93] J. Anderson and M. Lam, "Global Optimizations for Parallelism and Locality on Scalable Parallel Machines", In Proc. of the ACM Conference on Programing Language Design and Implementation, (PLDI'93), ACM Press, New York, June 1993.

[Birnbaum99] M. Birnbaum,and H. Sachs, "How VSIA Answers the SOC Dilemma," IEEE Computer, June, 1999, pp. 42-50.

[Brockman99] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, T. Sterling. "Microservers: A New Memory Semantics for Massively Parallel Computing", In *Proc. of the ACM International Conference on Supercomputing*, June, 1999, pp. 454-463.

[Bik97] A.J.C. Bik and H.A.G. Wijshoff, "Simple Qualitative Experiments with a Sparse Compiler". In *Proc.s of the 9th International Workshop on Languages and Compilers for Parallel Computing*, 1996. Appeared in Lecture Notes in Computer Science, No. 1239, pages 466--480, 1997.

[Blume96] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel Programming with Polaris, *IEEE Computer* 29(12), Dec. 1996, pp. 78-83.

[Burger96] D. Burger, J. Goodman and A. Kagi. "Memory Bandwidth Limitations of Future Microprocessors," In *Proc. of the 23rd International Symposium on Computer Architecture (ISCA)*, May, 1996.

[Burger97] Doug Burger, Stefanos Kaxiras, and James R. Goodman, "DataScalar Architectures", In *Proc. of the 19th International Symposium on Computer Architecture (ISCA)*, June, 1997.

[Carslile95] M. Carlisle, A. Roges and L. Hendren, "Early Experiences with Olden", In Proc. of the ACM Conference on Principles and Practice of Parallel Processin

[Carter99] J.B. Carter et. al, "Impulse: Building a Smarter Memory Controller", Fifth Int'l Symposium on High Performance Computer Architecture, pp. 70-79, January 1999.

[Cmelik94] R. Cmelik and D. Keppel. "Shade: A Fast Instruction Set Simulator for Execution Profiling", In Proc. of the ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems, June, 1994.

[Dally92] Dally, W. J., et al, "The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanism," IEEE Micro, April 1992, pp. 23-38.

[Draper96] J. Draper, "The Red Rover Algorithm for Deadlock-Free Routing on Bidirectional Rings", In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, August 1996, pp. 345-54.

[vonEicken92] T. von Eicken, D. Culler, S. C. Goldstein,and K. Schauser, "Active Messages: a Mechanism for Integrated Communication and Computation", In *Proc. of the 19th International Symposium on Computer Architecture*, May 1992.

[Foster95] I. Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[Gokhale95] M. Gokhale, B. Holmes, and K. Iobst, "Processing In Memory: the Terasys Massively Parallel PIM Array," *IEEE Computer*, April 1995, pp. 23-31.

[Hall96] M. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion and M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF Compiler," *IEEE Computer* 29(12), Dec., 1996, pp. 84-89.

[Herlihy91] M. Herlihy, "Wait-Free Synchronization," *ACM Transactions on Programming Languages and Systems* 13(1):124-129, 1991.

[IBMMot94] IBM Microelectronics and Motorola Inc., "The PowerPC Microprocessor Family: The Programming Environments," IBM Microelectronics Document MPRPPCFPE-01, Motorola Document MPCFPE/AD (9/94).

[Knowles91] S. Knowles, "Arithmetic Processor Design for the T9000 Transputer," Proc. SPIE, vol. 1566, 1991, pp 230-243.

[Kogge94] P.M. Kogge. "The EXECUBE Approach to Massively Parallel Processing," 1994 Int. Conf. on Parallel Processing, Chicago, IL, August, 1994.

[Kogge96] Kogge, Peter M., S. C. Bass, J. B. Brockman, D. Z. Chen, E, H. Sha, "Pursuing a Petaflop: Point designs for 100TF Computers Using PIM Technologies," 6th Symp. on Frontiers of Massively Parallel Computation, Annapolis, MD, Oct. 25-31, 1996.

[Kogge98] P.Kogge, J.B. Brockman, V. Freeh, "Processing-In-Memory Based Systems: Performance Evaluation Considerations", Workshop on Performance Analysis and its Impact on Design, held in conjunction with ISCA '98, May 1998.

[Mowry92] T. Mowry and M. Lam and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching", In *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 1992.

[Oskin98] Mark Oskin, Frederic T. Chong, and Timothy Sherwood. "Active Pages: A Model of Computation for Intelligent Memory". In *Proc. of the 25th International Symposium on Computer Architecture (ISCA)*, June, 1998.

[Palmer86] Palmer, J. F., "The nCube Family of Parallel Supercomputers," Proc. IEEE Int. Conf. on Computer Design, 1986, p. 107.

[Patterson97] D. Patterson et al., “A Case for Intelligent DRAM: IRAM," IEEE Micro , April 1997.

[Rinard97] M. Rinard and P. Diniz, “Commutativity Analysis: A New Analysis Technique for Parallelizing Compilers,” *ACM Transactions on Programming Languages and Systems* 19(6), Nov. 1997, pp. 942-991.

[Rixner98] S. Rixner, W. Dally, U. Kapasi, B. Kailany, A. Lopez-Lagunas, P.R. Mattson and J.D. Owens. “A Bandwidth-Efficient Architecture for Media Processing,” in Micro 31, 1998.

[Saulsbury95] A. Saulsbury, T. Wilkinson, J. Carter and A. Landin, “An Argument for Simple COMA”, In *Proc. of the Symposium on High-Performance Computer Architecture*, 1995.

[Saulsbury96] A. Sauslbury, F. Pong and A. Nowatzyk, “Missing the Memory Wall: The Case for Processor/Memory Integration, Proc. of the International Symposium on Computer Architecture, May, 1996.

[Shimizu96] Shimizu et al., “A Multimedia 32b RISC Microprocessor with 16Mb DRAM,” In *International Solid State Circuit Conference*, Feb. 1996, pp. 216-17.

[Steele97] C. S. Steele, et al, “A Bus-Efficient Low-Latency Network Interface for the PDSS Multicomputer”, *Proc. of the International Symposium on High-Performance Distributed Computing*, August 1997, pp. 213-22.

[Sunaga96] T. Sunaga, P.M. Kogge, et al, “A Processor In Memory Chip for Massively Parallel Embedded Applicatiions,” IEEE J. of Solid State Circuits, Oct. 1996, pp. 1556-1559.

[Wolfe89] M. J. Wolfe, “More Iteration Space Tiling”, In *Proc. of the IEEE Supercomputing Conference*, Nov. 1989.

A Fast, Simple Router for the Data-Intensive Architecture (DIVA) System

Chang Woo Kang and Jeffrey Draper
USC-Information Sciences Institute
4676 Admiralty Way
Marina Del Rey, CA 95292 USA

Abstract—**This paper presents a fast, simple router design for implementing the Red Rover algorithm for a bidirectional ring. This design is very suitable for the Data-Intensive Architecture (DIVA) system, a system which demonstrates the benefits of embedded DRAM technology, because of its high performance as well as simple architecture and low cost. The key attributes of this router are one clock node-to-node latency, high channel throughput, and simple hardware implementation. The router architecture employs short-cut FIFO data paths, which makes the router speed independent of the channel buffer size (in terms of flits). A prototype implementation of the router achieves a maximum channel bandwidth of 5.12 Gb/s and runs at 80 MHz using 3.3V CMOS signaling in 0.5μm technology. This high throughput and low latency were achieved without resorting to the use of complex high-speed signaling technologies.**

I. INTRODUCTION

Embedded DRAM technology is growing in popularity, as it appears to be a promising solution to the increasing gap between processor and memory speeds [6]. Integrating processor logic and memory in processing-in-memory (PIM) chips offers dramatically increased memory bandwidths (up to 2 orders of magnitude) over conventional systems. Furthermore, memory latency is also reduced because internal memory accesses avoid the delays associated with communicating off chip. The Data-Intensive Architecture (DIVA) system aims to exploit this technology by combining PIM devices with one or more external host processors and a PIM-to-PIM interconnect [9]. The DIVA system design imposes a unique set of requirements on the PIM-to-PIM interconnect. PIM chips will be physically grouped as conventional memory chips, mounted on DIMM modules. The number of PIM chips in a hosted cluster is therefore in the range of 32 to 64 chips, depending on how many PIM chips can be packed onto a DIMM module. The PIM-to-PIM interconnect must then be amenable to the dense packing requirement of DIMM modules. Low latency and high throughput are also desirable properties of this interconnect. Furthermore, this network must be scalable to allow the addition or removal of modules. This combination of requirements favors a one-dimensional network. Recently implemented routers such as SGI SPIDER [5] and the Cray T3E network router [8] are not suitable to be embedded in PIM devices because of complexity and size. The resulting PIM Routing Component (PiRC) is a one-dimensional wormhole router which implements the Red Rover routing algorithm to effect deadlock-free routing in bidirectional rings [1], [3]. The Red Rover algorithm provides a more even, symmetric distribution of message traffic among virtual channels in a bidirectional ring and therefore attains lower latencies and higher throughput than Dally's spiral algorithm [4]. Additionally, the PiRC routes fixed-size packets and uses source routing to achieve low latency. The PiRC architecture is presented in detail in Section II.. Section III. describes implementation and performance issues. Simulation scenarios for testing functionality are presented in Section IV., and concluding remarks are given in Section V..

II. ROUTER ARCHITECTURE

Because it employs the *Red Rover* algorithm, the *PIM Routing Component (PiRC)* has a very simple architecture and may be viewed as two identical routers which are time-multiplexed. One router operates on the rising transition of the clock while the other operates on the falling transition. In this manner, two virtual channels (A and B) are time-multiplexed onto each physical channel. Each virtual router contains controlling logic, consisting of an input controller, switch, and output controller, and short-cut FIFO data paths (see Figure 1). A channel *input controller* receives control signals from a sender and generates control signals for storing data into a short-cut FIFO. The *switch and output controller* determines to which output port input data should be forwarded and arbitrates fairly among contending requests for a particular output port. The handshaking protocol between sending and receiving PiRC channels, described in Section A., is also very simple and efficient.

Other factors also contribute to the simplicity of the PiRC architecture. A packet is constrained to a fixed length of ten 32-bit flits, and the phit size is the same as the flit size. All operation including receiving, switching, arbitrating, and sending is done in a half clock cycle. Thus, only one clock is needed for a flit to traverse from one node to the next in the non-blocking case. The PiRC implements wormhole routing [7] so that flits of a blocked packet remain in place in the network channels. However, each PiRC FIFO contains enough space to buffer a complete 320-bit packet. This ability simplifies the handshaking so that handshakes need only occur on packet boundaries rather than on every flit.

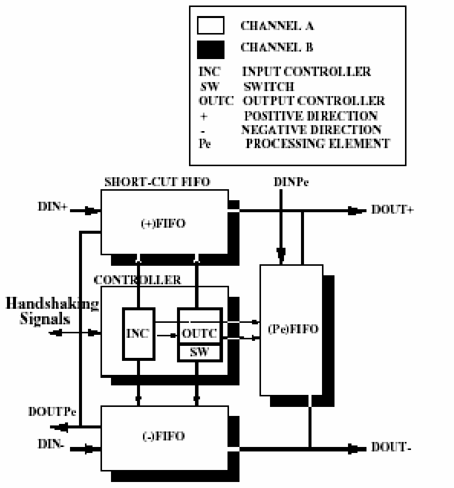


Fig. 1. PIM Routing Component (PiRC) Block Diagram

Figure 2 shows the internal interface for one virtual level of the PiRC (the other level is identical). This figure shows how the FIFOs, input controllers(INC), switches(SW), and output controllers(OUTC) interact for the positive(+), negative(-), and processing element(Pe) directions. Note especially the switching and merging combinations in the data paths. A packet entering the (+) FIFO may continue in the (+) direction or exit the network through the Pe port. Similarly, a packet entering the (-) FIFO may continue in the (-) direction or exit the network through the Pe port. Finally, a packet which is injected via the Pe FIFO may enter the network via the (+) or (-) port. These routing restrictions result in 2-way switchers and 2-way mergers at every point of contention. This artifact simplifies the router design, requiring the design of only one merge and one switch element that are then replicated as needed. The SI and RI signals are *send* and *ready* handshaking signals for the input channels, while the SO and RO signals correspond to output channels. More detail about their operation is given in the following section.

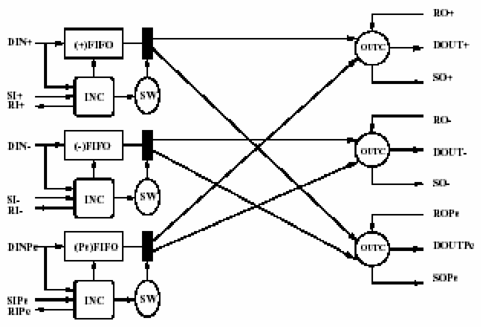


Fig. 2. PiRC Internal Interface

A. Handshaking Protocol

The handshaking protocol is really simple and efficient. First, note that the SI and RI signals of a receiving PiRC channel are connected to the SO and RO signals, respectively, of a neighboring sending PiRC channel. The receiver keeps asserting the RI signal as long as its corresponding FIFO is not full. The sender keeps sampling the corresponding RO signal at every edge of the clock and starts sending a pending message whenever the receiver is ready. By using this protocol, the sender constantly monitors the state of the receiver and does not waste time to explicitly request the status of the receiver FIFO. As depicted in Figure 3, this protocol makes it possible for the sender to make a decision to send data as soon as an asserted RO is sampled. To indicate it is sending data, the sender asserts SO, and the receiver latches DIN data into the FIFO upon sampling the corresponding asserted SI signal. The receiver then latches data on the next nine clock cycles to receive the entire packet.

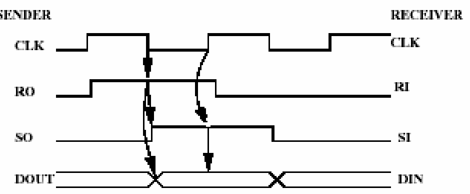


Fig. 3. Handshaking between a sender and a receiver

B. Short-Cut FIFO

In order to achieve high-speed data transmission along the physical channel, fast switching activity between channels is essential. A previous implementation of a Red Rover router, the *PDSS router* [2], specified a simple controller and complex flit buffer design and is suitable for only a small number of flits per packet. In the PDSS router, there are a large number of flit buffers that can drive the final output stage bus, as shown in Figure 4. This arrangement results in a large capacitive load. The controller is, however, very simple such that finite state machines without peripheral logic are sufficient for controlling the register-tristate buffer pairs. In contrast, the *PiRC* implements a complex controller and simple FIFOs in order to accommodate a large number of flit buffers in the channel buffer. In fact, the output stage load capacitance is independent of the number of flit buffers in a short-cut FIFO because only the top element of the FIFO is capable of driving the output stage bus. This characteristic makes the design very flexible with regard to channel size and is important as different package types impose different pin-count, and therefore channel size, constraints. With this design, every flit in the FIFO shifts toward the top of the FIFO as long as the path is not blocked. Also, incoming flits are placed in the first empty flit buffer (from the top of the FIFO). Figure 5 illustrates the cell of the FIFO, block diagram, and an example of flit movement.

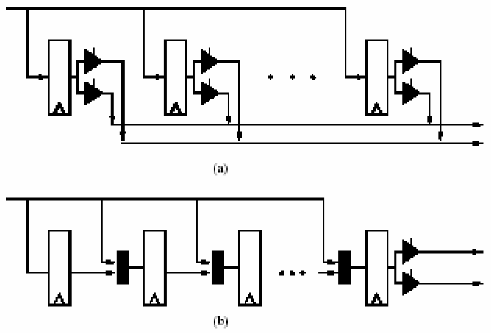


Fig. 4. Channel Buffer Design:(a) Register-Tristate Buffer in PDSS Router and (b) Short-cut FIFO in PiRC

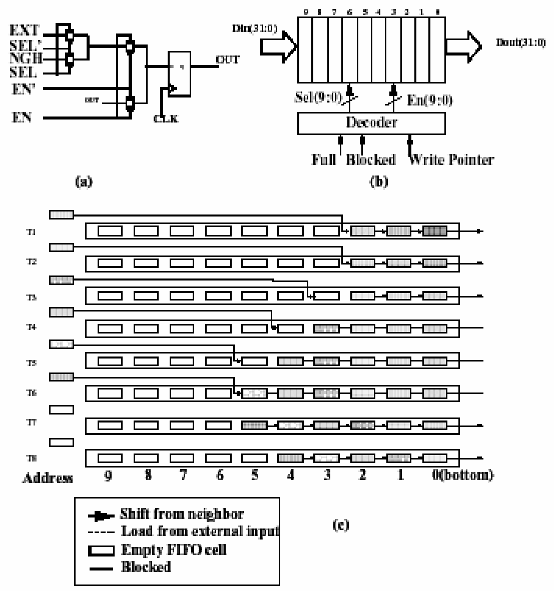


Fig. 5. Short-Cut FIFO: (a) FIFO Cell, (b) Block Diagram, and (c) Movements of Flits

The cell in (a) has two data inputs, one from the neighboring flit in the FIFO (NGH) and the other from the external input for this router channel(EXT). The decoder in (b) generates proper control signals for the FIFO based on current conditions and a write pointer indicator. (c) is an example showing the movement of flits. Until T3 there is no blocking; therefore, the flit in flit buffer 0 goes out and the other flits shift toward the top of the FIFO so that the FIFO depth is constant. New flits from the external input are loaded into flit buffer 2 (this example assumes some residual flits exist in the FIFO initially). Flits do not shift if the output path is blocked, as shown during T4, T5, and T6. However, the write pointer increments so that subsequent incoming flits begin to fill up the FIFO. When the path becomes unblocked, flits drain out as shown from T7.

In order to keep track of the header flit of a packet, the SI signal flows through a one-bit FIFO as the header flit moves. The operation of this one-bit FIFO is identical to that of the data FIFO described above. This signal becomes the output signal SO in the final output stage, indicating that the router channel is sending a header flit of a new packet.

C. Input Controller

The input controller, shown in Figure 6, is simple counter-based logic that directs the loading of flits into the short-cut FIFO. When the input controller samples an asserted SI signal, it begins latching the flits of an incoming packet. The *up/down counter* dynamically changes the write pointer value, which always points to the first empty space in the FIFO, as the router reads and writes flits. The wen (write enable) generator causes the $up/down$ counter to increment the write pointer value when SI arrives from the sender. The ren (read enable) generator is activated when the output controller starts forwarding flits from the FIFO to an output channel, and it also prevents the reading of garbage in an empty FIFO. The counter operates at both clock edges so that it can increase the write pointer at the rising edge when a new flit is written and decrease the pointer at the falling edge when a flit is read from the FIFO (these clock edges apply for A virtual channels—for B virtual channels, the opposite clock edges apply). The *full-empty detector* indicates the status of the FIFO. The RI handshaking signal is merely the inverse of the $full$ signal. As mentioned earlier, the decoder translates the write pointer value into proper control signals for the FIFO.

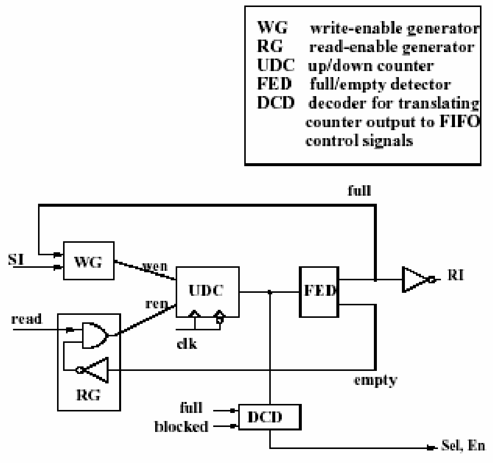


Fig. 6. Input Controller

D. Switch and Output Controller

The output controller samples RO at every clock edge so that it can send a pending packet as soon as possible. Once RO is asserted and detected by the output controller, the header flit of a pending packet and SO are sent immediately. While flits are being transmitted to the receiver, the write pointer of the sending FIFO decrements if there are no incoming flits from the neighboring PiRC. On the other hand, the pointer keeps pointing to the same flit buffer in the sending FIFO if the sending FIFO is simultaneously receiving data from its neighbor. The switch determines the direction in which a packet is to be forwarded. The first flit of a packet, the header, contains routing information for the switch. The header is unary encoded such that the number of hops a packet is to traverse is indicated by the number of 1's set in the header. The header is shifted at each hop so that this value is decremented. Therefore, the switch simply inspects the first bit of the routing header to determine which output port to request for a given packet. Using a first-come-first-served policy, the output controller arbitrates fairly between requests from two FIFOs contending for usage of the same output physical channel. If contending requests arrive in the same clock cycle to an idle output controller, an arbitrary selection is performed; however, the FIFO which is not granted access during this arbitration is guaranteed access when the current FIFO completes based on the first-come-first-served policy.

III. Implementation and Performance

The PiRC design was begun by behavioral modeling in VHDL and compiled with Synopsys. Cascade EPOCH was used for routing and placement as well as layout generation for a prototype implementation. Control blocks were synthesized, while the short-cut FIFO was generated using custom layout to achieve high density. We tested our design at the behavioral level, pre-synthesis level, and post-synthesis level with Synopsys, and transistor level with Powermill.

The resulting PiRC prototype layout is for the HP14b process available through MOSIS. This process uses 0.5μm, 3-layer metal CMOS technology. The PiRC has a die size of 2.76 mm X 2.36 mm and contains 75,276 transistors. Simple hardware based on an efficient routing algorithm allows us to achieve a clock frequency of 80MHz. The router operates on both clock edges, leading to a channel bandwidth of 5.12Gb/s. Only one clock is required for a flit to move from one node to the next, resulting in a node-to-node delay of 12.5ns. Figure 7 shows the layout of the PiRC, placed and routed with the floor plan of Figure 1. Although this prototype achieves respectable performance, we expect performance to improve significantly when we migrate to a currently available embedded DRAM process using 0.25μm or even 0.18μm technology, such as the IBM SA27-E or TSMC process.

IV. Simulation

Five critical scenarios were used to verify the PiRC design. The external PiRC connections used for simulation are shown in Figure 8. This configuration allows short-cut FIFOs to be cascaded together so that one FIFO essentially feeds another. The header flit of a packet is set in simulation to specify

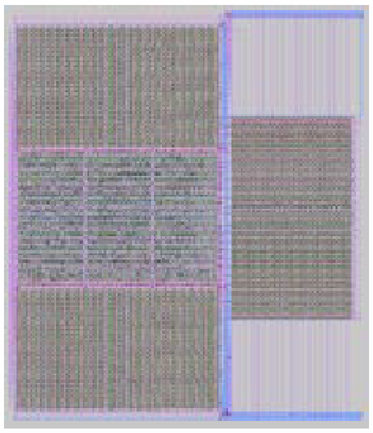


Fig. 7. Layout

whether the corresponding packet travels from the (Pe) FIFO to the (+) FIFO or the (-) FIFO. Test vectors are injected on the Tester terminals indicated in Figure 8, which essentially serve as processing element signals. The scenarios are as following:

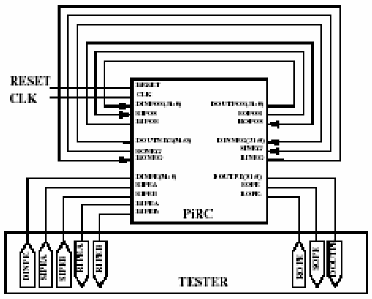


Fig. 8. Router Configuration for Testing

1. Two messages move back-to-back without blocking.
2. Two messages move back-to-back. The first message is blocked until the (+,-) FIFO is full. Consequently, the second message is blocked in the (Pe) FIFO and starts filling it. Then, the first message becomes unblocked and drains out. As soon as the first message starts moving out, the second message follows it along the path.
3. Two messages move back-to-back. The first message is blocked until the (+,-) FIFO gets half-way full, and then the first message drains out.
4. The first message is blocked until the (+,-) FIFO fills half-way, and when the first message starts draining out of the (+,-) FIFO, the second message is injected to the (Pe) FIFO from the tester. Due to the short-cut FIFO design, the second message quickly traverses the (Pe) FIFO to trail the first message.

5. Two packets in (+,-) FIFO and (Pe) FIFO request the same channel concurrently. This scenario ensures that fair arbitration is performed when resolving conflicts.

The *PiRC* performed successfully for all possible combinations of the above scenarios for two sets of virtual channels.

V. Conclusion

A fast, simple router for the Data-Intensive Architecture (DIVA) system has been presented. This device, the *PIM Routing Component (PiRC)*, implements the *Red Rover* routing algorithm to achieve high performance with minimal complexity. The PiRC has advantages of simple logic, one clock node-to-node delay, high channel throughput, and robust speed consistency, regardless of the number of flit buffers in a channel buffer. This combination of attributes makes the PiRC ideal for the DIVA system.

VI. Acknowledgments

The authors would like to acknowledge the support of DARPA (Contract No. F30602-98-2-0180).

References

[1] Jeff Draper and Fabrizio Petrini, "Routing in Bidirectional k-ary n-cubes with the Red Rover Algorithm," *Proceedings of the International conference on Parallel and Distributed Processing Techniques and Applications,* Jun 1997, pp. 1184-93.

[2] Jeff Draper, "PDSS Router Design," *Technical Report. Information Sciences Institute/USC*, 1996.

[3] Jeff Draper, "The Red Rover Algorithm for Deadlock-Free Routing on Bidirectional Rings," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications,* August 1996, pp. 345-54.

[4] William J. Dally and Charles L. Seitz, "Deadlock-Free Message Routing in Multiprocessor Interconnection Networks," *IEEE Transactions on Computers,* May 1987, pp. 547-553.

[5] Mike Galles, "Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI SPIDER Chip," *Proceedings of the Symposium on Hot Interconnects,* August 1996, pp. 141-146.

[6] Subramanian S. Lyer and Howard L. Kalter, "Embedded DRAM technology," *IEEE Spectrum,* April 1999, pp. 56-64.

[7] Lionel M. Ni and Philip K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer,* February 1993, pp. 62-76.

[8] Steven L. Scott and Gregory M. Thorson, "The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus," *Proceedings of the Symposium on Hot Interconnects,* August 1996, pp. 147-156.

[9] Mary Hall et al., "Mapping Irregular Application to DIVA, a PIM-based Data-Intensive Architecture," *Supercomputing*, 1999.

Compiler-Controlled Caching in Superword Register Files for Multimedia Extension Architectures

Jaewook Shin, Jacqueline Chame and Mary W. Hall
Information Sciences Institute
University of Southern California
{jaewook,jchame,mhall}@isi.edu

Abstract

In this paper, we describe an algorithm and implementation of locality optimizations for architectures with instruction sets such as Intel's SSE and Motorola's AltiVec that support operations on superwords, i.e., aggregate objects consisting of several machine words. We treat the large superword register file as a compiler-controlled cache, thus avoiding unnecessary memory accesses by exploiting reuse in superword registers. This research is distinguished from previous work on exploiting reuse in scalar registers because it considers not only temporal but also spatial reuse. As compared to optimizations to exploit reuse in cache, the compiler must also manage replacement, and thus, explicitly name registers in the generated code. We describe an implementation of our approach integrated with a compiler that exploits superword-level parallelism (SLP). We present a set of results derived automatically on 4 multimedia kernels and 2 scientific benchmarks. Our results show speedups ranging from 1.3 to 2.8X on the 6 programs as compared to using SLP alone, and we eliminate the majority of memory accesses.

1 Introduction

In response to the increasing importance of multimedia applications in embedded and general-purpose computing environments, many microprocessors now incorporate an expanded instruction set and architectural extensions specifically targeting multimedia requirements. The core component of such architectural extensions is a functional unit that can operate on aggregate objects, performing bit-level operations, or SIMD parallel operations on variable-sized fields in the object (*e.g.*, 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than the size of a machine word, then they are called *superwords* [20]. Examples include Motorola's AltiVec and Intel's SSE, a descendant of MMX. If the same size as the machine word, then individual fields are referred to as *subwords* [22]. A related class of architectures employ processing-in-memory (PIM) technology to exploit the high memory bandwidth when processing logic is combined on chip with large amounts of DRAM; several PIM-based architectures rely on superword parallelism to make more effective use of available memory bandwidth [2, 17, 3, 11].

While multimedia extension and related architectures have been available for some time, convenient methodologies for developing application code that targets these extensions are in their infancy. There is recent compiler research for such architectures to automatically exploit *superword-level parallelism*, performing computations or memory accesses in parallel in a single instruction issue [20, 27, 8, 10, 1].

In this paper, we recognize an additional optimization opportunity not addressed by this previous work. An important feature of all such architectures is a register file of superwords (*e.g.*, each 128 bits wide in an AltiVec), usually in addition to the scalar register file. A set of 32 such superword registers represents a not insignificant amount of storage close to the processor. Accessing data from superword registers, versus a cache or main memory, has two advantages. The most obvious advantage is lower latency of accesses; even a hit in the L1 cache has at least a 1-cycle latency. Accesses to other caches in the hierarchy or to main memory carry much higher latencies. Another advantage is the elimination of memory access instructions, thus reducing the number of instructions to be issued.

In this paper, we treat the superword register file as a small compiler-controlled cache. We develop an algorithm and a set of optimizations to exploit reuse of data in superword registers to eliminate unnecessary memory accesses, which we call *superword-level locality*. We evaluate the effectiveness of these superword-level locality (SLL) optimizations through an implementation integrated with the algorithm for exploiting superword-level parallelism (SLP) presented in [20].

Our approach is distinguished from previous work on increasing reuse in cache [9, 12, 14, 15, 16, 19, 28, 30], in that

	Original Figure 1(a)	SLP only Figure 1(b)	Scalar register reuse only Figure 1(d)	SLP and SLL Figure 1(f)
Reads	$3n^2$	$2n^2 + n^2/sws$	$n^2/2 + n$	$(n^2/2 + n)/sws$
Writes	n^2	n^2/sws	n^2	n^2/sws

Table 1. Number of array accesses under different optimization paths.

the compiler must also manage replacement, and thus, explicitly name the registers in the code. As compared to previous work on exploiting reuse in scalar registers [30, 5, 23], the compiler considers not just temporal reuse, but also spatial reuse, for both individual statements and groups of references. Further, it also considers superword parallelism in making its optimization decisions. Exploiting spatial and group reuse in superword registers requires more complex analysis as compared to exploiting temporal reuse in scalar registers, to determine which accesses map into the same superword.

The contributions of this paper are as follows:

- An algorithm for exposing opportunities for compiler-controlled caching of data in superword register files.
- A description of a set of optimizations, which in aggregate we call *superword replacement*, for exploiting superword register reuse.
- Experimental results, derived automatically, comparing performance of six benchmarks/multimedia kernels optimized for parallelism only, SLP, and optimized for both parallelism and superword-level locality. Our results show speedups ranging from 1.3 to 2.8X as compared to using SLP alone, and we eliminate the majority of memory accesses.

The remainder of the paper is organized into 5 sections. Section 2 motivates the problem and introduces terminology used in the remainder of the paper. Section 3 presents the main superword-level locality algorithm, which performs a set of transformations and an optimization search that exposes opportunities for reuse of data in superword registers. Section 4 presents optimizations to actually achieve this reuse of data in superword registers. Section 5 presents experimental results derived automatically by an implementation in the Stanford SUIF compiler. Section 6 discusses related word and Section 7 presents conclusions and future work.

2 Background and Motivation

In many cases superword-level parallelism and superword-level locality are complementary optimization goals, since achieving SLP requires each operand to be a set of words packed into a superword, which happens, with no extra cost, when an array reference with spatial reuse is loaded from memory into a superword register. Therefore, in many cases the loop that carries the most superword-level parallelism also carries the most spatial reuse, and benefits from SLL optimizations. In this paper, we achieve SLL and SLP somewhat independently, by integrating a set of SLL optimizations into an existing SLP compiler [20]. The remainder of this section motivates the SLL optimizations.

Achieving locality in superword registers differs from locality optimization for scalar registers. To exploit temporal reuse of data in scalar registers, compilers use *scalar replacement* to replace array references by accesses to temporary scalar variables, so that a separate backend register allocator will exploit reuse in registers [5]. In addition, *unroll-and-jam* is used to shorten the distances between reuse of the same array location by unrolling outer loops that carry reuse and fusing the resulting inner loops together [5]. In conventional architectures with scalar register files, spatial locality can only be obtained in caches.

In contrast, a compiler can optimize for superword-level locality in superword registers locality through a combination of unroll-and-jam and *superword replacement*. These techniques not only exploit temporal reuse of data, but also spatial reuse of nearby elements in the same superword. In fact, even partial reuse of superwords can be exploited by merging the contents of two registers containing superwords that are consecutive in memory (see Section 4.3). Thus, as is common in multimedia applications [25], streaming computations with little or no temporal reuse can still benefit from spatial locality at the superword-register level, as well as at the cache level.

While cache optimizations are beyond the scope of this paper, we observe that the SLL optimizations presented here can be applied to code that has been optimized for caches using well-known optimizations such as unimodular transformations, loop tiling and data prefetching. When combining loop tiling for caches, superword-level parallelism and superword-level locality optimizations, the tile sizes should be large enough for superword-level parallelism, and for unroll-and-jam and superword replacement to be profitable.

These points are illustrated by way of a code example, with the original code shown in Figure 1(a). This example shows three optimization paths. Figure 1(b) optimizes the code to achieve superword-level parallelism. Here, sws, an abbreviation for superword size, is the number of data ele-

```
for(i=0; i<n; i++)
   for (j=0; j<n; j++)
       a[i][j] = a[i-1][j] * b[i] + b[i+1];
```

(a) Original loop nest.

```
for(i=0; i<n; i++)
   for (j=0; j<n; j+=sws)
       a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
```

(b) After superword-level parallelism(j loop).

```
for(i=0; i<n; i+=2)
   for (j=0; j<n; j++) {
       a[i][j] = a[i-1][j] * b[i] + b[i+1];
       a[i+1][j] = a[i][j] * b[i+1] + b[i+2];
   }
```

(c) Unroll-and-jam on the example in (a)(i loop).

```
tmp1 = b[0];
for(i=0; i<n; i+=2) {
   tmp2 = b[i+1];
   tmp3 = b[i+2];
   for (j=0; j<n; j++) {
       tmp4 = a[i-1][j] * tmp1 + tmp2;
       a[i+1][j] = tmp4 * tmp2 + tmp3;
       a[i][j] = tmp4;
   }
   tmp1 = tmp3;
}
```

(d) After scalar replacement on the code in (c).

```
for(i=0; i<n; i+=2)
   for (j=0; j<n; j+= sws) {
       a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
       a[i+1][j:j+sws-1] = a[i][j:j+sws-1] * b[i+1] + b[i+2];
   }
```

(e) Unroll-and-jam on the example in (b)(i loop).

```
tmp1[0:sws-1] = b[0:sws-1];
stmp1 = tmp1[0];
stmp2 = tmp1[1];
field = 2;
for(i=0; i<n; i+=2) {
   // 'field' denotes an index into 'tmp1' for stmp3
   if(field == 0)
       tmp1[0:sws-1] = b[i+2:i+sws+1];
   stmp3 = tmp1[field];
   for (j=0; j<n; j+= sws) {
       tmp2[0:sws-1] = a[i-1][j:j+sws-1] * stmp1 + stmp2;
       a[i+1][j:j+sws-1] = tmp2[0:sws-1] * stmp2 + stmp3;
       a[i][j:j+sws-1] = tmp2[0:sws-1];
   }
   stmp1 = stmp3;
   stmp2 = tmp1[field+1];
   field = (field+2)%sws;
}
```

(f) After superword replacement on code in (e)

Figure 1. Example code.

ments that fit within a superword. For example, if a and b are 32-bit float variables, on a machine with 128-bit superwords, $sws = 4$. In Figures 1(c) and (d), we show how the original program can instead be optimized to exploit reuse in scalar registers, using unroll-and-jam and scalar replacement, respectively. In Figures 1(e) and (f), we combine these ideas, using unroll-and-jam and superword replacement, respectively, to transform the code in (b) for both superword-level parallelism and superword-level locality.

Table 1 shows how the three different optimization paths affect the number of array accesses to memory in the final code. The original code has n^2 reads and writes to array a and $2n^2$ reads to array b. Exploiting superword-level parallelism in loop j, as in Figure 1(b) reduces the number of reads and writes to array a by a factor of sws since each load or store operates on sws contiguous data items; for array b, there is no change since the array is indexed by i rather than j. If instead the code was optimized for scalar register reuse, as in Figure 1(d), we can reduce the number of array reads of a down by a factor of 2, and reads of b by a factor of n, with the number of writes remaining the same. By combining superword-level parallelism and superword-level locality as in Figure 1(f), we see that the number of reads and writes is further reduced by a factor of sws. Figure 1(f) illustrates some of the challenges in exploiting reuse in superwords. Analysis must identify not just temporal, but also spatial reuse, and for both individual statements and groups of references. The compiler also must generate the appropriate code to exploit this reuse; for example, we select scalar fields of b from the superword, since we are not parallelizing the i loop.

The remainder of this paper describes how the compiler automatically generates code such as is shown in Figure 1(f), and the performance improvements that can be obtained with this approach.

3 Superword-Level Locality Algorithm

The superword-level locality algorithm has four main steps, as summarized in the next subsection. At the heart of the algorithm is an approach for counting both memory accesses and register requirements for storing reused data, which is the subject of the subsequent subsection.

3.1 Steps of Algorithm

Step 1: Identifying Reuse. First, we identify array variables and loops carrying temporal or spatial reuse. We examine the dependence graph, looking for references that have loop-carried consistent dependences (*i.e.,* constant dependence distances) or are loop invariant with one of the loops, and so have opportunities for data reuse that can be exposed by unroll-and-jam.

Applying unroll-and-jam to a loop with a loop-variant reference creates loop-independent dependences in the unrolled loop body. In the example in Figure 1(a), there is a true dependence between references $A[i][j]$ and $A[i-1][j]$ with distance vector $\langle 1, 0\rangle$. After unroll-and-jam, a loop-independent dependence is created between $A[i][j]$ in the first statement and $A[i][j]$ in the second statement, creating a reuse opportunity. Similarly, spatial and group-temporal reuse can be exposed by unroll-and-jam when a reference has a loop-carried dependence with the loop that traverses the lowest array dimension. For loop-invariant references, unroll-and-jam generates loop-independent dependences between the copies of the reference in the unrolled loop body.

Step 2: Determining unroll factors for candidate loops. The algorithm next determines the unroll factors for each candidate loop that carries reuse and for which unroll-and-jam is legal, with the following goal.

> *Optimization Goal:* Find unroll factors $\langle X_1, X_2, ...X_n\rangle$ for loops 1 to n in a n-deep loop nest such that the number of memory accesses is minimized, subject to the constraint that the number of superword registers required does not exceed what is available.

The search algorithm uses the reuse information and the number of registers available to prune the search space, as follows. Loops that carry no reuse are not included in the search. Next, we observe that for each unrolled loop l, the amount of reuse of an array reference with reuse carried by l increases with the unroll factor X_l. Therefore reuse is a monotonic, non-decreasing function of the unroll factor for each loop, given that the unroll factor of all other loops are fixed. The algorithm uses this property to prune the search space, avoiding searching for all possible unroll factors for a given loop. It traverses the search space by varying the unroll factor of one loop while keeping the unroll factor of all other loops fixed. A binary search within a dimension can further prune the search. Also, the unroll factor of each loop, given that all other unroll factors are fixed, is limited by the number of registers available. Once the search finds an unroll factor for a given loop that exceeds the register limit, it prunes all larger unroll factors for that loop from the search space.

To guide the search towards the above optimization goal, we calculate the *superword footprint*, which represents the number of superwords accessed by the unrolled iterations of the loop nest, as a function of the unroll factor. The superword footprint can be used both to count how many registers are required to hold the accessed data, as well as how many memory accesses remain in the loop nest. Assuming that all variables are kept in registers when the superword footprint fits in the superword register file, the number of memory accesses associated with a set of references is simply the superword footprint for the references multiplied by the bounds of the loops in which they are nested after unrolling. Our method for selecting unroll factors based on required superword registers differs from related approaches oriented towards scalar registers [5], accounting for not only temporal but also spatial and group reuse. In the next subsection, we describe in detail the calculation of the superword footprint.

Step 3: Unroll-and-Jam and Superword Replacement. Once the unroll factors are decided, the loop nest is transformed and array references are replaced with accesses to superword temporaries, as discussed in Section 4.

3.2 Computing the Superword Footprint

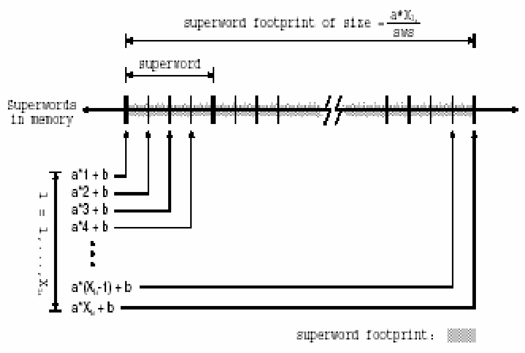
The algorithm for computing the superword footprint for a loop nest first partitions the references in the loop into groups of *uniformly generated references* [30], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term[1]. Then, for each group of references, it computes the registers needed to keep the data accessed in the unrolled loop body. Finally, the total number of registers is computed as the sum of those of each group of uniformly generated references. We first discuss how to compute the registers required for a single reference as a function of the unroll factors of each unrolled loop. Then we discuss how to compute the register requirements for a group of uniformly generated references. The registers required for such a group may be smaller than the sum of the registers required for each reference, if computed individually, since the same superword may be accessed by two or more copies of the original references when the loops are unrolled.

Our method determines the number of superword registers required to hold the data accessed by the loop references in the unrolled loops. However, extra registers may be needed to, for example, align a superword operand which is already kept in superword registers. That is, the computation may require more registers than those needed for storing the data. Therefore, we reserve some scratch registers for manipulating data and compute the number of registers needed just for storing the data accessed in the unrolled loops.

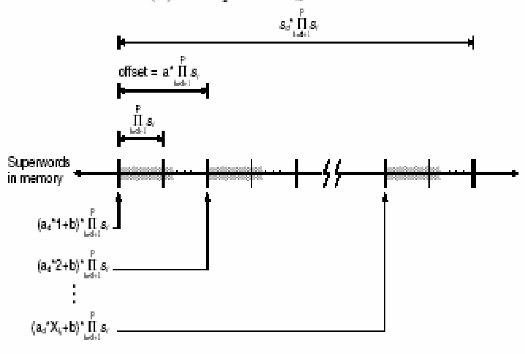
To simplify the presentation, we assume a loop nest of depth n where all array references have array subscripts that are affine functions of a single index variable (SIV subscripts)[2]. We also assume that each p-dimensional array referenced by the loop is defined as $A[s_1][s_2]\ldots[s_p]$, where s_d is the size of dimension d, $1 \le d \le p$. Dimension p is the lowest dimension of the array, *i.e.,* the dimension

[1] We assume that two or more references that access the same array but are not uniformly generated access distinct data in memory, which results in a conservative estimate of the number of registers.

[2] Our current implementation can handle affine SIV subscripts and certain affine MIV subscripts.



(a) $d = p$ and $a_d < sws$


$s_d * \prod_{l=d+1}^{p} s_i$
offset = $a * \prod_{l=d+1}^{p} s_i$
$\prod_{l=d+1}^{p} s_i$
Superwords in memory
$(a_d*1+b)*\prod_{l=d+1}^{p} s_i$
$(a_d*2+b)*\prod_{l=d+1}^{p} s_i$
$(a_d*X_{l_i}+b)*\prod_{l=d+1}^{p} s_i$

(b) $d \neq p$

Figure 2. Superword footprint of a single reference.

in which consecutive elements are in consecutive memory locations. A reference v to array A is then of the form, $A[a_1 * l_1 + b_1][a_2 * l_2 + b_2] \ldots [a_p * l_p + b_p]$. Similarly, the array subscripts of the uniformly generated references $v_1, v_2, \ldots v_m$ in dimension d are $a_d * l_d + b_1$, $a_d * l_d + b_2$, ..., $a_d * l_d + b_m$, respectively. Thus, a reference with SIV subscripts has each array dimension associated with just a single loop index variable in the nest. We also assume that the arrays are aligned to a superword in memory and that the loops are normalized.

3.2.1 Superword Footprint of a Single Reference

For each reference v with array subscripts $a_d * l_d + b$, where d is the array dimension and l_d is the loop variable appearing in subscript d, the number of registers required to keep the data referenced by v when l_d is unrolled by X_{l_d} is given by the *superword footprint* of v in l_d, or $F_{l_d}(v)$. The superword footprint consists of the superwords accessed by all copies of v resulting from unrolling.

When dimension d is the lowest array dimension ($d = p$), the superword footprint is given by Equation (1). Equation (1a) corresponds to the footprint of a loop-invariant reference. Equation (1b) corresponds to the footprint of a reference with self-spatial reuse within a superword, as illustrated in Figure 2(a), and (1c) holds when the reference has no spatial reuse.

$$F_{l_d}(v) = \begin{cases} 1 & \text{(a) if } a_d = 0 \\ \left\lceil \frac{X_{l_d} * a_d}{sws} \right\rceil & \text{(b) if } a_d < sws \\ X_{l_d} & \text{(c) if } a_d \geq sws \end{cases} \qquad (1)$$

When d is one of the higher dimensions, $1 \leq d < p$, and loop l_d is unrolled, the offset between the footprints of each copy of v is $a_d * \prod_{i=d+1}^{p} s_i$, where s_i is the size of the i^{th} array dimension, as shown in Figure 2(b). Assuming that the size of the lowest array dimension (s_p) is larger than sws, which is usually the case in practice for realistic array dimensions, each copy of v in the unrolled loop body corresponds to a separate footprint, as shown in Figure 2(b). Therefore the size of the footprint of v in l_d is the sum of the X_{l_d} disjoint footprints, and is recursively defined by Equation (2), where $F_{l_p}(v)$ is computed as in Equation (1).

$$\begin{aligned} F_{l_d}(v) &= X_{l_d} * F_{l_{d+1}}(v) \\ &= \left(\prod_{i=d}^{p-1} X_{l_i}\right) * F_{l_p}(v) \end{aligned} \qquad (2)$$

For a single reference, the number of superword registers given by Equation (1) and the number of scalar registers that would be required if the same unroll factors were used differ only when $a_d < sws$, that is, when spatial reuse can be exploited in superword registers. For a group of uniformly generated references the analysis must also consider group reuse, as discussed next.

3.2.2 Superword Footprint of a Reference Group

The number of registers required to keep a group of uniformly generated references $V = \{v_1, v_2, \ldots, v_m\}$ when loop l_d is unrolled by X_{l_d} is the superword footprint of the group, $F_{l_d}(V)$. The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The footprints of two uniformly generated references may overlap in dimension d only if they overlap in all dimensions higher than d. For example, the footprints of references $A[2i][j+2]$ and $[2i+1][j]$ do not overlap in the highest (row) dimension, since the first reference accesses the even-numbered rows of the array and the second accesses the odd-numbered rows. Therefore the footprints cannot overlap in the lowest (column) dimension. On the other hand, the footprints of $A[2i][j+2]$ and $A[2i+4][j]$ overlap in the row dimension for iterations i_1, i_2, $1 \leq i_1, i_2 \leq X_i$, such that $2i_1 = 2i_2 + 4$. For the iterations of i in which the footprints overlap in the row dimension, the footprints

$$F_{l_d}(v_1, v_2) = \begin{cases} X_{l_d} + (b_2 - b_1)/a_d & \text{(a) if } a_d > sws \text{ and } (b_2 - b_1) < a_d * X_{l_d} \text{ and} \\ & \quad (b_2 - b_1) \bmod a_d = 0 \\ \lceil (a_d * X_{l_d} + b_2 - b_1)/sws \rceil & \text{(b) if } a_d \leq sws \text{ and } (b_2 - b_1) < a_d * X_{l_d} \\ F_{l_d}(v_1) + S_{l_d}(v_2) & \text{(c) otherwise} \end{cases} \quad (3)$$

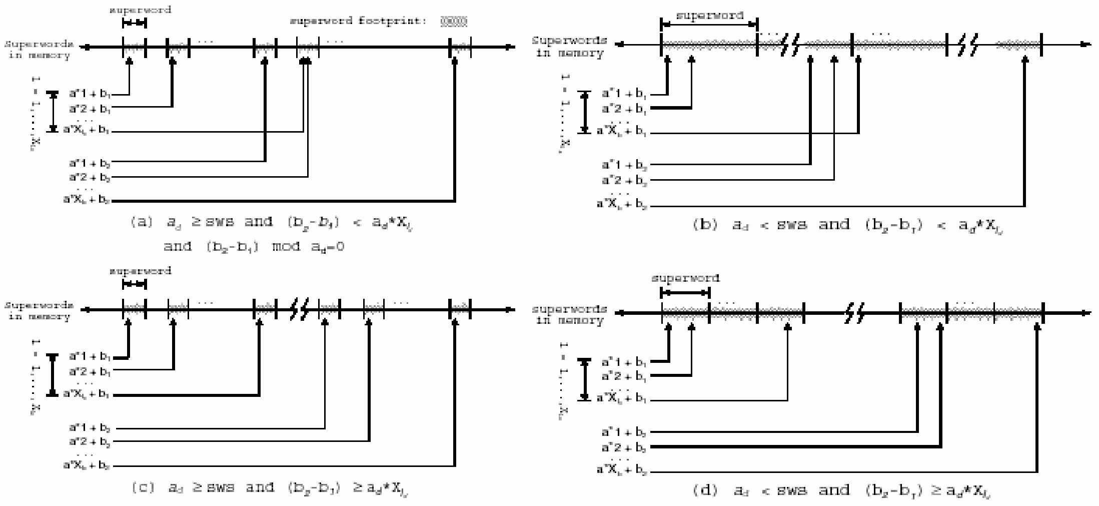

superword
superword footprint:
Superwords in memory
a*1 + b1
a*2 + b1
a*X_l + b1
a*1 + b2
a*2 + b2
a*X_l + b2
(a) $a_d \geq$ sws and $(b_2-b_1) < a_d*X_{l_d}$ and (b_2-b_1) mod a_d=0
(b) $a_d <$ sws and $(b_2-b_1) < a_d*X_{l_d}$
(c) $a_d \geq$ sws and $(b_2-b_1) \geq a_d*X_{l_d}$
(d) $a_d <$ sws and $(b_2-b_1) \geq a_d*X_{l_d}$

Figure 3. Superword footprint of a group of references.

may overlap in the column dimension if there exist iterations $j_1, j_2, 1 \leq j_1, j_2 \leq X_j$, such that $j_1 + 2 = j_2$.

The superword footprint of a group V in a set of unrolled loops is computed as follows. For each dimension d, from highest to lowest dimension, the footprint is computed assuming that the footprints of the references in the group overlap in the higher dimensions. For each dimension $d < p$, the algorithm partitions references into subsets such that each subset corresponds to a disjoint footprint in dimension d. Then, for each subset, the algorithm recursively computes the footprint in dimension $d + 1$, as we now describe.

Dimension d is the lowest dimension ($d = p$). We first compute the group footprint of two array references, and then we extend it for m references. The group footprint of two references $\{v_1, v_2\}$, with lowest dimension subscripts $a_d * l_d + b_1$ and $a_d * l_d + b_2$ such that $b_1 \leq b_2$, when loop l_d is unrolled by X_{l_d} is given by Equation (3) in Figure 3.

Equations (3a), (3b) and (3c) correspond to combinations of two basic conditions which determine the superword footprint of a pair of uniformly generated references. The first condition is whether the references have self-spatial reuse within a superword, that is, whether $a_d < sws$. The second is whether the footprints may overlap, which is the case when $(b_2 - b_1) < a_d * X_{l_d}$.

Figure 3 shows four examples of superword footprints corresponding to Equation (3). Figure 3(a) corresponds to Equation (3a), where the footprints may overlap and the group footprint is the union of the two footprints. Each of the individual footprints is a set of X_{l_d} superwords since the references have no spatial reuse. The footprints overlap if $(b_2 - b_1)$ is evenly divided by a_d and there exists an integer value $k, 1 \leq k \leq X_{l_d}$, such that $k = 1 + (b_2 - b_1)/a_d$. This equation precisely computes the overlapped footprint when the two footprints have group temporal reuse. For group spatial reuse, we conservatively approximate the footprint with Equation (3c). In Figure 3(b) the footprints of v_1 and v_2 overlap, and both references have spatial reuse within a superword. The corresponding footprint size is given by Equation (3b).

Figures 3(c) and 3(d) correspond to Equation (3c), where the footprints do not overlap and therefore the group footprint is the sum of the individual footprints. In Figure 3(c) v_1 has no self-spatial reuse and each copy of v_1 in the unrolled loop body accesses a distinct superword, and the same is true for v_2. In Figure 3(d) both v_1 and v_2 have superword spatial reuse.

The number of registers required for reference group $V = \{v_1, v_2, ..., v_m\}$ is computed by extending the equations above to more than two references. Here we describe the most interesting case (corresponding to Equation (3b)), where the footprints overlap and the references have spatial reuse. A subset group $V_i = \{v_{i_{min}}, v_{i_{min+1}}, ..., v_{i_{max}}\}$ is defined by lowest dimension subscripts $a_p * l_p + b_j$,

$i_{min} \leq j \leq i_{max}$, where the references have been sorted so that $b_{j-1} \leq b_j$. V_i has a footprint consisting of contiguous superwords if there is self-spatial reuse ($a_p \leq sws$) and possible overlap ($b_j - b_{j-1} \leq a_p * X_{l_p}$) for all j such that $i_{min} < j \leq i_{max}$. To compute the number of registers required for the entire group, the algorithm partitions V into disjoint subsets V_i as defined above, where $\forall j \quad i_{min} < j \leq i_{max}$,

$$\begin{aligned} &(b_j - b_{j-1} \leq a_p * X_{l_p}) \wedge \\ &(b_{i_{min}} = b_1 \vee b_{i_{min}} - b_{i_{min}-1} > a_p * X_{l_d}) \wedge \\ &(b_{i_{max}} = b_m \vee b_{i_{max}+1} - b_{i_{max}} > a_p * X_{l_d}) \end{aligned} \quad (4)$$

Each subset V_i corresponds to a footprint of contiguous superwords consisting of the union of the individual footprints, with size given by Equation (5).

$$\begin{aligned} F_{l_d}(V_i) &= F_{l_d}(\{v_{i_{min}}, ..., v_{i_{max}}\}) \\ &= \left\lceil \frac{a_p * X_{l_p} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil \end{aligned} \quad (5)$$

The total number of superword registers required for the references in V is then the sum of the disjoint footprints of the sets V_i, as in (6).

$$\begin{aligned} F_{l_d}(V) &= \sum_i F_{l_d}(V_i) \\ &= \sum_i \left\lceil \frac{a_p * X_{l_p} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil \end{aligned} \quad (6)$$

Dimension d is not the lowest dimension ($d \neq p$). When d is one of the higher dimensions, the superword footprint of $V = \{v_1, v_2, ..., v_m\}$ in loop l_d is again the union of the individual footprints.

From Section 3.2.1, the footprint of each reference v_i in the unrolled loop body consists of a set of X_{l_d} disjoint footprints, where each of the X_{l_d} footprints starts at superword $(a_d * l_d + b_j) * \prod_{i=d+1}^{p} s_i$, where s_i is the size of dimension i, and $1 \leq l_d \leq X_{l_d}$.

Therefore the footprints of different references in the group may overlap for some superwords, depending on the values of a_d, b_j and the unroll factor X_{l_d}. The footprints of two uniformly generated references v_1 and v_2 overlap in dimension d if there exists an integer value k such that $1 \leq k \leq X_{l_d}$ that satisfies Condition 7.

$$a_d * k + b_1 = a_d + b_2. \quad (7)$$

Furthermore, if there exists k satisfying the above condition, the footprints corresponding to the k to X_{l_d} copies of v_1 in the unrolled loop body overlap with those corresponding to the first $X_{l_d} - k + 1$ copies of v_2. The footprint of $\{v_1, v_2\}$ is then given by Equation (8).

$$\begin{aligned} F_{l_d}(v_1, v_2) &= (l_1 - 1) * F_{l_{d+1}}(v_1) \\ &+ (X_{l_d} - l_1 + 1) * F_{l_{d+1}}(v_1, v_2) \\ &+ (l_1 - 1) * F_{l_{d+1}}(v_2) \end{aligned} \quad (8)$$

To compute the size of the entire footprint of V in l_d, our algorithm partitions V into subsets $V_i = \{v_{i_{min}}, ..., v_{i_{max}}\}$ such that, for any j, $i_{min} < j \leq i_{max}$, the pair $\{v_{j-1}, v_j\}$ satisfies Condition (4). The footprint of V_i is the union of the overlapped footprints of its reference set and is computed by extending Equation (8) to more than two references.

4 Optimizations for Superword Replacement

After the appropriate unroll factors are determined by the algorithm in the previous section, the unrolled code is then optimized for superword-level parallelism. Not until after SLP are the final code transformations performed to actually exploit reuse in superword registers. In this section, we briefly describe these transformations.

4.1 Replacing Redundant Loads and Stores

Our compiler replaces redundant loads and stores from/to memory with accesses to superword temporaries. Since the code is already unrolled, it is very straightforward to recognize these opportunities. The compiler simply determines that addresses and offsets for different memory accesses fit within the same superword, and verifies that there are no intervening kills to the memory locations.

4.2 Packing in Superword Registers

As part of SLP's code generation, whenever data is packed to form superwords, this is done through memory. A data element is loaded into a scalar register from the source location and stored to the destination location. Packing through memory is in some sense motivated by the fact that many multimedia extension architectures do not support register-to-register transfers between scalar and superword register files.

In our system, we have developed an optimization we call *register packing*, shown in Figure 4, to perform this packing in the superword register file. We take advantage of two instructions that are common in multimedia extension architectures, which we call *replicate* and *shift-and-load*. *Replicate* replicates one element of a source register to all elements of a destination register. *Shift-and-load* takes two source registers. The first source register is shifted left by the amount of the third argument and the same amount is taken from the second source register to fill the destination register. Packing these operands in superword registers eliminates numerous scalar loads and stores.

```
w = *((float *)&a + 0);
x = *((float *)&b + 0);
y = *((float *)&c + 0);
z = *((float *)&d + 0);
*((float *)&p + 0) = w;
*((float *)&p + 1) = x;
*((float *)&p + 2) = y;
*((float *)&p + 3) = z;
```

(a) Packing through memory

```
temp1 = replicate(a, 0);
temp2 = replicate(b, 0);
temp3 = replicate(c, 0);
temp4 = replicate(d, 0);
p = shift_and_load(temp1, temp1, 4);
p = shift_and_load(p, temp2, 4);
p = shift_and_load(p, temp3, 4);
p = shift_and_load(p, temp4, 4);
```

(b) Packing in registers

Figure 4. Register Packing

4.3 Shifting for Partial Reuse

Spatial reuse within a superword happens when distinct loop iterations access different data in the same superword. *Partial spatial reuse* of superwords occurs when distinct loop iterations access data in consecutive superwords in memory, partially reusing the data in one or both superwords, as shown by the example in Figure 5, and illustrated graphically in Figure 5(d). In this example, as before assuming that $sws = 4$, array reference $b[i + j]$ has partial spatial reuse in loop i. For a fixed value of i and j, the data accessed in iteration $\langle i, j \rangle$ consists of the last three words of the superword accessed in iteration $\langle i - 1, j \rangle$, plus the first word of the next superword in memory. This type of reuse can be exploited by shifting the first word out of the superword, and shifting in the next word, as in Figure 5. As shown in Figure 5(c), only two superwords need to be loaded for the data accessed in the 4 copies of $b[i + j]$ in the loop body, after shifting is applied. Before shifting, $b[i + j]$ had to be loaded from memory (and aligned, for architectures that support only aligned accesses) for each of the four copies of $b[i + j]$ in the loop body.

Detecting the applicability of superword shifting is straightforward, involving checking the dependence distance on the loop for small, constant distances. Code generation is also straightforward, since multimedia extension architectures support efficient shifting and permutation mechanisms for aligning and rearranging data in superwords.

```
for (i = 0; i < n; i ++)
  for (j = 0; j < n; j ++)
    a[i][j] = b[i+j] * c[j];
```

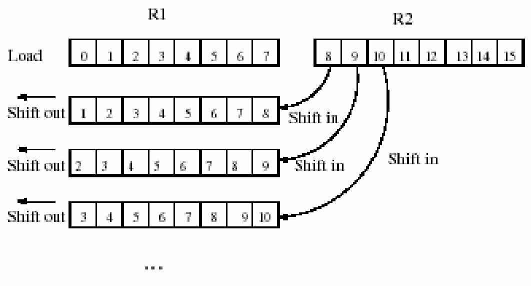
(a) Original loop nest

```
for (i = 0; i < n; i += 4)
  for (j = 0; j < n; j += 4){
    a[i][j:j+3] = b[i+j:i+j+3] * c[j:j+3];
    a[i+1][j:j+3] = b[i+j+1:i+j+4] * c[j:j+3];
    a[i+2][j:j+3] = b[i+j+2:i+j+5] * c[j:j+3];
    a[i+3][j:j+3] = b[i+j+3:i+j+6] * c[j:j+3];
  }
```

(b) After unroll-and-jam and SLP, assuming sws = 4).

```
for (i = 0; i < n; i += 4)
  for (j = 0; j < n; j += 4){
    tmp1[0:3] = b[i+j:i+j+3];
    tmp2[0:3] = b[i+j+4:i+j+7];
    a[i][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+1][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+2][j:j+3] = tmp1[0:3] * c[j:j+3];
    shift_and_load (tmp1[0:3], tmp2[0:3], 1);
    a[i+3][j:j+3] = tmp1[0:3] * c[j:j+3];
  }
```

(c) After shifting across superword registers.



(d) Graphical depiction of shifting.

Figure 5. Shifting registers for partial reuse.

5 Experimental Results

This section presents an experiment that demonstrates the dramatic performance improvements that can be derived from compiler-controlled caching in superword registers. We describe an implementation that incorporates superword register locality optimizations into an existing compiler exploiting superword-level parallelism [20]. We present a set of results on four multimedia kernels and two scientific applications, derived automatically from our implementation.

5.1 Implementation and Methodology

Figure 6 illustrates the system we have developed for this experiment, which uses the Stanford SUIF compiler as its

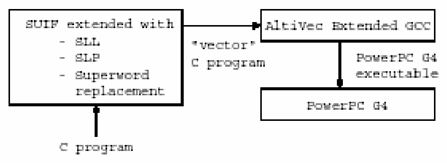


Figure 6. Implementation.

Name	Description	Data Width	Input Size
FIR	Finite impulse response filter	32-bit float	1K filter, 1M signal
VMM	Vector-matrix multiply	32-bit float	512 elements
MMM	Matrix-matrix multiply	32-bit float	1K elements
YUV	RGB to YUV conversion	16-bit integer	32K elements
SWIM	Shallow water model	32-bit float	Specfp95 reference input
TOMCATV	Mesh generation	32-bit float	Specfp95 reference input

Table 2. Benchmark programs.

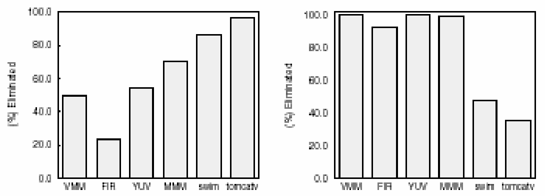
underlying infrastructure [18]. The input to the system is a C program, which is then optimized by passes in SUIF, including our Superword Locality analysis described in Section 3, followed by the Superword-Level Parallelism (SLP) optimization passes by Larsen and Amarasinghe[20], and finally, an optimization pass that performs superword replacement as described in Section 4 to steer the compiler to obtain the reuse in superword registers that the SLL algorithm determined was possible.

The output from the SUIF portion of the system is an optimized C program, augmented with special superword data types and operations. Currently, the resulting code is passed to a Gnu C backend, modified to support superword data types and operations for the PowerPC AltiVec instruction-set architecture extensions. Each superword operation corresponds, in most cases, to a single instruction in the AltiVec ISA. The role of the GCC backend includes replacing the vector operations with the corresponding AltiVec superword instruction, and allocating the vector data types to the superword registers. The resulting code is executed on a 533 MHz Macintosh PowerPC G4, which has a superword register file consisting of 32 128-bit registers.

5.2 Performance Measurements

We have applied the previously-described implementation to four of the five multimedia kernels and the two scientific programs from the Specfp95 benchmark suite for which execution time speedups were reported in Larsen and Amarasinghe, summarized in Table 2 [20]. As a first step, we verified that we could reproduce their previously reported results. For purposes of comparison, we initially followed the same methodology established in Larsen and Amarasinghe [20]: (1) we used the same programs; (2) all versions of the code were compiled on the AltiVec without optimization; and, (3) baseline measurements were derived by compiling the unparallelized code for the PowerPC G4. We are using an updated implementation of SLP from what was published, as well as a faster target machine and new releases of GCC and the Linux operating system, so there are some differences in results, but they are very minor.

Larsen and Amarasinghe were unable to use optimization on the AltiVec-extended GCC backend at the time of their study, but in the intervening time, this Motorola-supplied backend has become more robust. For the results presented in this section, we modify the methodology to perform "-O3" optimizations. To understand the overall



(a) Vector loads and stores removed. (b) Scalar loads and stores removed.

Figure 7. Reduction in dynamic memory accesses due to superword replacement.

benefits of exploiting compiler-controlled caching in superword registers, we have compared the results of the full system with those obtained when SLP is used alone. For this reason, we report results where SLP is applied to the original codes and compare these results to the full system.

We show two sets of results. First, in Figure 7(a), we show the percentage of vector loads and stores eliminated by the full system, as compared with SLP alone. Our approach eliminates over 50% of the vector loads and stores in three of the four kernels, and over 85% in SWIM and TOMCATV. We also eliminate scalar loads and stores using register packing, as described in Section 4. In Figure 7(b), we see that our approach eliminates over 90% of the scalar loads and stores in the four kernels, and over 35% in SWIM and TOMCATV.

Figure 8 shows how these reductions in instructions translates into speedups over SLP. To isolate the benefits of individual components of our system, we measure the performance of the code at several stages of the optimization process. The first bar, normalized to 1, shows the results of SLP alone. The second bar, called Unrolled+SLP, shows the results of running the first portion of the SLL algorithm, described in Section 3, which performs unroll-and-jam on the loop nest to expose opportunities for superword reuse, and following up with SLP. This bar isolates the impact of unrolling, since it is not until after SLP that this reuse is actually exploited. Also, because it is reordering the iteration space to bring reuse closer together, this version will also obtain locality benefits in the data cache. Thus, this bar provides the cache locality benefits of unroll-and-jam, which can be compared against the additional improvements from superword register locality. The third bar, Superword Replacement, provides speedup using Superword Replacement and Shifting, as described in Section 4. The final bar, entitled Register Packing, shows the additional improvement due to this technique, also described in Section 4.

Overall, we see that in combination, applications achieve speedups between 1.3 and 2.8 over SLP alone, with an average of 2.2X. Consideration of TOMCATV and SWIM shows that both programs have little temporal reuse, al-

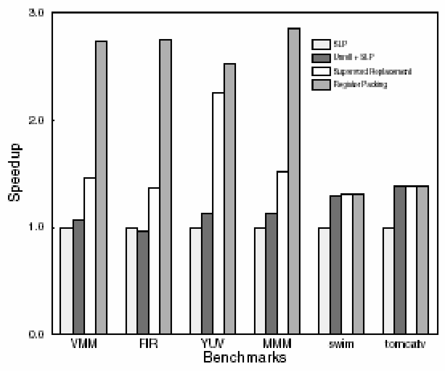


Figure 8. Speedups over SLP alone.

though there is a small amount of spatial reuse that is exploited with our approach, particularly in TOMCATV. We are obtaining a locality benefit due to unroll-and-jam. We also observe additional SLP due to iteration-space splitting, motivated by the need to create a steady-state loop where the data is aligned to a superword boundary. The four other programs show a significant improvement from superword replacement. For VMM, MMM and FIR, there are also huge gains due to register packing.

In summary, the SLL techniques presented in this paper dramatically reduce the number of memory accesses and yield significant performance improvements across these 6 programs. Thus, this paper has demonstrated the value of exploiting locality in superword registers in architectures that support superword-level parallelism such as the AltiVec.

6 Related Research

For well over a decade, a significant body of research has been devoted to code transformations to improve cache locality, most of it targeting loop nests with regular data access patterns [13, 6, 31, 32]. Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses. Of particular relevance to this paper are approaches to tiling for cache to exploit temporal and spatial reuse; the bulk of this work examines how to select tile sizes that eliminate both capacity misses and conflict misses, tuned to the problem and cache sizes [7, 9, 12, 14, 15, 16, 19, 28, 30, 26]. The key difference between our work and that of tiling for caches is that interference is not an issue in registers. Therefore, models that consider conflict misses are not appropriate. Further, our code generation strategy must explicitly manage reuse in registers.

There has been much less attention paid to tiling and other code transformations to exploit reuse in registers, where conflict misses do not occur, but registers must be explicitly named and managed. A few approaches examine mapping array variables to scalar registers [30, 5, 23]. Most closely related to ours is the work by Carr and Kennedy, which uses scalar replacement and unroll-and-jam to exploit scalar register reuse [4]. Like our approach, in deriving the unroll factors, they use a model to count the number of registers required for a potential unrolling to avoid register pressure, and they replace array accesses, which would result in memory accesses, with accesses to temporaries that will be put in registers by the backend compiler. Their search for an unroll factor is constrained by register pressure and another metric called *balance* that matches memory access time to floating point computation time. Our approach is distinguished from all these others in that the model for register requirements must take spatial locality into account, we replace array accesses with superwords rather than scalars, and we also consider the optimizations in light of superword parallelism.

There are several recent compilation systems developed for superword-level parallelism [20, 27, 8, 10, 1]. Most, including also commercial compilers [29, 24], are based on vectorization technology [27, 10]. In contrast, Larsen and Amarasinghe devised a superword-level parallelization system for multimedia extensions [20]. They point out that there are many differences between the multimedia extension architectures and vector architectures, such as short vectors, ease of mixing with scalar instructions, and need for alignment of memory accesses [21]. They argue that their algorithm for finding superword-level parallelism from a basic block instead of a loop nest is much more effective than using vectorization-based techniques. None of the above approaches exploit reuse in the superword register file.

7 Conclusion

This paper presents an algorithm for compiler-controlled caching in superword register files. The algorithm is applicable to multimedia extensions such as Intel's SSE, PowerPC's AltiVec, and also to Processor-in-memory (PIM) architectures with support for superword operations.

We implemented our approach in an existing compiler targeting superword-level parallelism. We presented experimental results, derived automatically, comparing the performance of six benchmarks/multimedia kernels optimized for parallelism only, using SLP, and optimized for both parallelism and locality. Our results show speedups ranging from 1.3 to 2.8X, and an average of 2.2X, on the 6 programs as compared to using SLP alone, and most memory accesses are removed.

The approach taken here that separates optimizations for SLL and SLP is convenient for implementation purposes, since we are building upon the work of others. Further, as there are now a few other compilers that exploit superword-level parallelism [27, 8, 10, 1], the same can be used to extend these existing systems to incorporate compiler-controlled caching in superword registers. Ideally, however, an optimizer that integrates the superword parallelism and locality techniques could be even more effective. For example, in a combined algorithm, selection of which loops to parallelize could also take superword-level locality into account. A combined algorithm is the subject of future work.

8 Acknowledgments

The authors wish to thank Samuel Larsen and Saman Amarasinghe for providing their SLP implementation. We wish to especially thank Samuel Larsen for his tremendous support. We also wish to thank all the students in our research group for providing the underlying infrastructure for this work, including Yoon-Ju Lee, Byoungro So and Rommel Dongre.

References

[1] K. Asanovic and J. Beck. T0 engineering data. UC Berkeley CS technical report UCB/CSD-97-930.

[2] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang. Evaluation of existing architectures in IRAM systems. In First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, June 1997.

[3] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[4] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[5] S. Carr and K. Kennedy. Scalar replacement in the presence of conditional control flow. *Software—Practice and Experience*, 24(1):51–77, 1994.

[6] S. Carr, K. S. McKinley, and C.-W. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 252–262, San Jose, California, October 1994.

[7] J. Chame and S. Moon. A tile selection algorithm for data locality and cache interference. In *International Conference on Supercomputing*, pages 492–499, 1999.

[8] G. Cheong and M. S. Lam. An optimizer for multimedia instruction sets. In *The Second SUIF Compiler Workshop*, Stanford University, USA, August 1997.

[9] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[10] D. J. DeVries. A vectorizing suif compiler: Implementation and performance. Master's thesis, University of Toronto, 1997.

[11] D. Elliott, M. Snelgrove, and M. Stumm. Computational RAM: a memory-SIMD hybrid and its application to DSP. In *IEEE 1992 Custom Integrated Circuit Conference*, pages 30.6.1 – 30.6.4, 1992.

[12] K. Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[13] J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 328–343, Santa Clara, California, August 1991.

[14] C. Fricker, O. Temam, and W. Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *TOPLAS*, 17(4):561–575, July 1995.

[15] S. Ghosh, M. Martonosi, and S. Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[16] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, October 1998.

[17] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.

[18] M.W. Hall, J.M. Anderson, S.P. Amarasinghe, B.R. Murphy, S. Liao, E. Bugnion, and M.S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[19] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.

[20] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC Canada, June 2000.

[21] S. Larsen, E. Witchel, and S. Amarasinghe. Increasing and detecting memory address congruence. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, September 2002.

[22] R. Lee. Subword parallelism with max2. *IEEE Micro*, 16(4):51–59, August 1996.

[23] A. Fernandez M. Jimenez, J.M. Llaberia and E. Morancho. Index set splitting to exploit data locality at the register level. Technical Report UPC-DAC-1996-49, Universitat politecnica de Catalunya, 1996.

[24] Metrowerks. *CodeWarrior version 7.0 data sheet*, 2001. http://www.metrowerks.com/pdf/mac7.pdf.

[25] P. Ranganathan, S. Adve, and N. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *International Symposium on Computer Architecture*, May 1999.

[26] G. Rivera and C. Tseng. A comparison of compiler tiling algorithms. In *the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands*, March 1999.

[27] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 2000.

[28] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *ACM International Conference on Supercomputing*, Portland, OR, November 1993.

[29] Veridian. *VAST/AltiVec Features*, June 2001. http://www.psrv.com/altivec_feat.html.

[30] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.

[31] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, June 1991.

[32] Michael J. Wolfe. More iteration space tiling. In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989.

A Compiler Algorithm for Exploiting Page-Mode Memory Access in Embedded-DRAM Devices

Jaewook Shin, Jacqueline Chame and Mary W. Hall
Information Sciences Institute
University of Southern California
{jaewook,jchame,mhall}@isi.edu

ABSTRACT

This paper presents a compiler algorithm and several optimization techniques to exploit a DRAM memory characteristic(*page mode*) automatically. A page-mode memory access exploits a form of spatial locality, where the data item is in the same row of the memory buffer as the previous access. Thus, access time is reduced because the cost of row selection is eliminated. The algorithm increases frequency of page-mode accesses by reordering data accesses, grouping together accesses to the same memory row. We implemented this algorithm and present speedup results for four multimedia kernels ranging from 1.25 to 2.19 for a Processing-In-Memory (PIM) embedded DRAM device.

1. INTRODUCTION

Memory delays are a major performance bottleneck in embedded-DRAM systems, where the memory latencies seen by the processor are dominated by the on-chip-DRAM access time. DRAM modules support an efficient *page-mode* access, where a memory access to a location currently in the DRAM open-row buffer fetches the data directly from that buffer, eliminating the cost of fetching the row from the DRAM array. Page-mode accesses, when applicable, are supported by the DRAM's memory controller. To fully exploit lower latency page-mode accesses, the user or the compiler must reorganize the computation so that accesses to a same memory row are grouped together, and there are no intervening accesses to other rows.

In the past decade, most of the research on compiler optimizations for the memory hierarchy focused on exploiting data locality in caches [4, 7, 8, 9, 10, 15, 24, 25]. Although cache optimizations and page-mode optimizations have the common goal of exploiting data reuse (in caches or in the DRAM's open row, respectively), the analysis and code transformations required are different. For example, loop tiling is used to exploit temporal reuse in caches by bringing together in time loop iterations that access the same data. The goal is to keep the data accessed in a tile in cache, and the order of the accesses within a tile is not important. On the other hand, exploiting page-mode accesses requires not only bringing together in time loop iterations that access data in a same memory row, but also grouping these data accesses together. Exposing opportunities for grouping accesses to a same array may require transformations such as unroll-and-jam, to bring accesses issued in distinct loop iterations to the body of the transformed loop, and statement reordering, to group the memory accesses.

Recent research has proposed to exploit page-mode accesses through manual code transformations [19, 17, 3]. This paper presents a compiler algorithm for exploiting page-mode automatically. Our algorithm is implemented in the SUIF compiler infrastructure [13], and it leverages well-known compiler analyses and code transformations to identify potential page-mode accesses and group these memory accesses together. The algorithm is applicable to loop-based computations in general embedded systems and it is also applicable to embedded-DRAM systems designed to exploit the large on-chip bandwidths by transferring and processing objects larger than a machine word [23, 1].

We have performed an experimental evaluation of our algorithm on a Processing-In-Memory (PIM) device that is part of the DIVA architecture [12], where the PIM processor is capable of transferring and processing 256-bit objects (superwords) in parallel. Our results show the performance improvements from exploiting page-mode accesses, and the combined benefits of page-mode accesses and other compiler optimizations targeting architectures with support for *superword-level parallelism*[1] (SLP) [16, 22]. We obtain speedups ranging for 1.25 to 2.19 for four multimedia kernels. This paper makes the following contributions:

- A new compiler algorithm for automatically exploiting page-mode memory accesses;
- An experimental evaluation of the algorithm on four data-intensive multimedia kernels;
- A discussion of practical issues that must be addressed when exploiting page-mode accesses in combination with other compiler optimizations.

This paper is organized as follows. Section 2 motivates our approach using a simple example. Section 3 introduces our algorithm for exploiting page-mode memory accesses. Section 4 presents experimental results on a set of four multimedia kernels. Section 5 addresses practical issues which are the subject of future work. Related research is discussed in Section 6 and Section 7 concludes the paper.

2. MOTIVATION

Figure 1 illustrates the benefits of page-mode accesses using a simple loop nest with two array references. Assuming that

[1]Fine grain SIMD parallelism in a register larger than a machine word

Ref.	Loop j	Loop i
A[j][i]	$m * RMLatency$	$n * m * RMLatency$
B[i]	$m * RMLatency$	$n * m * RMLatency$
Total	$2 * n * m * RMLatency$	

(a) Original

Ref.	Loop j	Loop i′
A[j][i]	$m * RMLatency$	$\frac{n}{4} * m * RMLatency$
A[j][i+1]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
A[j][i+2]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
A[j][i+3]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i]	$m * RMLatency$	$\frac{n}{4} * m * RMLatency$
B[i+1]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i+2]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
B[i+3]	$m * PMLatency$	$\frac{n}{4} * m * PMLatency$
Total	$\frac{n}{2} * m * RMLatency + \frac{3n}{2} * m * PMLatency$	

(b) After unroll-and-jam and reordering

Table 1: Memory Latency Computation

```
for(i=0;i<n;i++){
    for(j=0;j<m;j++){
        load A[j][i]
        load B[i]
            ...
    }
}
```

(a) Original

```
for(i=0;i<n;i+=4){
    for(j=0;j<m;j++){
        load A[j][i]
        load A[j][i+1]
        load A[j][i+2]
        load A[j][i+3]
        load B[i]
        load B[i+1]
        load B[i+2]
        load B[i+3]
            ...
    }
}
```

(b) After unroll-and-jam and reordering

Figure 1: Unroll-and-jam and Reordering

the sizes of arrays A and B are larger than the DRAM's open-row buffer, all array references in Figure 1(a) are in random-mode, since reference $B[i]$ displaces the DRAM row containing $A[j][i]$ from the open-row buffer and vice-versa.

For the same number of memory accesses in this loop nest, we can increase the page mode memory accesses by applying a series of code transformations, as shown in Figure 1(b). First, unroll-and-jam is used to unroll the outer i loop and fuse together the resulting inner j loop bodies. Unroll-and-jam creates opportunities for page-mode accesses by moving array references from successive loop iterations of the outer loop into the body of the transformed inner loop. Once the loop is unrolled and the copies of the loop body are fused, accesses to the same memory page in the loop body may be grouped together by reordering the memory accesses in the transformed loop body, if the reordering does not violate data dependences.

In Figure 1(b), following unroll-and-jam, where the i loop is unrolled by a factor of 4, references to the same array (A or B) in the body of the transformed loop are grouped together. This results in page-mode accesses for all references in the loop body, except leading references $A[j][i]$ and $B[i]$, which are in random mode.

Table 1 shows the total memory access cost for the code in Figures 1 (a) and (b), if we assume that latencies for random mode and page mode accesses are uniform, and that accesses are not going through cache. Assuming that random-mode latency is three times the page-mode latency as in [14], loop (a) has a total latency cost of $6 * n * m * PMLatency$, while (b) has a cost of $3*n*m*PMLatency$, a factor of 2 difference in overall memory latency.

This example shows the potential for improving performance in embedded DRAM devices through the previously-described code transformations. To expose opportunities for page-mode accesses by applying unroll-and-jam and memory access reordering, a compiler algorithm must: (1) determine the safety of these code transformations and select a loop for which unrolling is profitable; (2) select an unroll factor that increases page-mode accesses while not causing register spilling; and, (3) transform the code to reorder the memory accesses. In the next section we present our compiler algorithm for exploiting page-mode accesses, which includes the three steps above.

We have developed this algorithm in the context of a compiler for DIVA, a system-architecture that incorporates processing-in-memory embedded DRAM devices as smart-memory co-processors in an otherwise conventional system [12]. Although the proposed compiler algorithm is not specific to the requirements of the DIVA architecture, we describe the algorithm from the viewpoint of an architecture that supports *superword-level parallelism*, with an instruction set akin to multimedia extensions such as Intel's SSE and Motorola's AltiVec. Superword-level parallelism refers to performing the same operation in parallel on multiple fields of a superword, which is an aggregate object larger than a machine word. In the following algorithm description, we will refer to register width to support the notion that a machine might have different register widths for distinct objects. If a machine does not support superword-level operations, then the register width is the same as the machine word.

In previous work, we presented an algorithm for exploiting spatial and temporal locality in superword register files in a compiler that already supports superword-level parallelism [22]. In this paper, we show that with a similar approach we can also exploit spatial locality in the page of a DRAM memory array.

3. ALGORITHM

In this section we introduce a compiler algorithm for exploiting *page-mode memory accesses*. Our algorithm is applicable to loop nests with array references in the loop body, where the array subscript expressions are affine functions of the loop index variables. Only array accesses are reordered by the algorithm, since it is difficult to determine whether two scalar accesses are on the same memory page. For presentation purposes, we make some simplifying assumptions as

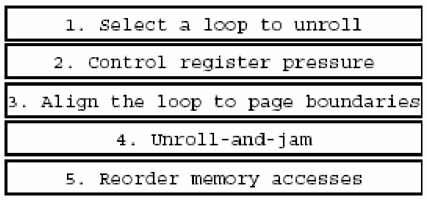

1. Select a loop to unroll
2. Control register pressure
3. Align the loop to page boundaries
4. Unroll-and-jam
5. Reorder memory accesses

Figure 2: Algorithm

follows.

1. Array objects are aligned at memory page boundaries.
2. The lowest dimension sizes of array objects are multiples of a memory page size.
3. The compiler backend does not change the memory access order generated by the algorithm.

Some of these assumptions can be removed by modifying the compiler backend (1,3) or by padding array objects (2).

The algorithm presented in this paper unrolls a single loop in a loop nest, since in practice unrolling more than one loop could create register pressure and intruction cache misses. A set of heuristics is used to select which loop to unroll and its unroll amount. These heuristics result in a fast algorithm that is effective for the benchmarks presented in Section 4.

However, unrolling multiple loops in a loop nest might expose more opportunities for page-mode accesses than when unrolling a single loop. In previous work [22] we present an algorithm for exploiting superword-level locality which uses unroll-and-jam to expose data reuse, and unrolls multiple loops in a nest. The computation of the unroll amounts requires a complex analysis to determine the exact number of superword registers needed to keep the data accessed in the loop. This complexity is due to several factors such as group reuse among copies of a reference created by unrolling (which may reuse data in superword registers) and self-spatial reuse of the original references.

A more complex algorithm for exploiting page-mode memory accesses which would consider multiple loops for unrolling is the subject of future work, and we plan to leverage our analysis and algorithm for selecting unroll amounts described in [22].

Figure 2 illustrates the steps of the algorithm, which are described in the remainder of this section. The first step selects which loop to unroll, after determining the safety of the code transformations (unroll-and-jam and statement reordering). The second steps selects an unroll factor that increases page-mode accesses while not causing register spilling. The last three steps apply the code transformations to the loop nest.

Selecting a Loop To Unroll The first step of the algorithm selects a loop to unroll, based on the number of random-mode memory accesses of the loop nest after applying unroll-and-jam. The algorithm uses data dependence information to determine the safety of unroll-and-jam and to prevent selection of unroll amounts greater than the dependence distance if inner loop dependence distances are negative.

For each loop l in the loop nest, the algorithm computes the unroll amount X_l and its corresponding number of random-mode accesses R_l, such that R_l is the smallest number of random-mode memory accesses if l is selected to be unrolled (assuming that references to a same memory page can be grouped together). Then the algorithm compares the number of random-mode accesses of each loop in the nest and selects the loop with the smallest R_l.

When computing the unroll amount X_l that minimizes R_l, the algorithm considers only references that are loop-variant with l in the lowest dimension. For a reference that is loop-variant with l in the lowest dimension, unrolling l and jamming the copies of l in the loop body creates opportunities for page-mode accesses between the copies of the original reference. On the other hand, unrolling loop l does not change the total number of random-mode accesses generated by references that are loop-variant with l in one of the higher dimensions. Loop-independent dependences can be removed by locality optimizations such as scalar replacement [2] or superword replacement [22].

For each loop l the smallest unroll amount that minimizes R_l is computed as in Equation 1.

$$X_l = \frac{\mathrm{P}}{\min_{a\in A}(T(a) * C(a,l))} \tag{1}$$

where P is the memory page size, A is the set of array references in the loop nest which are loop-variant with l in the lowest dimension, a is an array reference in A, $T(a)$ is the type size of a and $C(a,l)$ is the coefficient of the index variable l in the lowest-dimension subscript of a.

After computing the unroll amounts, the algorithm computes the corresponding number of random-mode memory accesses R_l, with the goal of selecting the loop with smallest R_l. For each loop l, the number of random-mode accesses R_l is computed as the number of distinct pages in the *memory-page footprint* of A, $F_l(A, X_l)$ (assuming that the algorithm can group together references to a same page). In previous work [22], we present the computation of the *superword footprint* of a set of array references in a loop nest, which consists of the number of distinct superwords accessed by the references, a function of the unroll amounts. The memory-page footprint can be computed in a similar way to that of the superword footprint. First, the set of references is partitioned into groups of *uniformly generated references* [25], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term[2]. Then, for each group of references, the algorithm computes the number of pages accessed in the unrolled loop body. Finally, the total number of pages is computed as the sum of those of each group of uniformly generated references.

Controlling Register Pressure After selecting a loop l to unroll, the algorithm adjusts the unroll amount of the selected loop to avoid register pressure and register spilling, which could offset the benefits of unroll-and-jam.

In a previous paper [22] we presented the computation of the number of registers required to keep the data accessed by the references in the loop nest after applying transformations for increasing locality in the superword register file. Here we present a simplification of this algorithm to provide the

[2]We assume that two or more references that access the same array but are not uniformly generated access distinct data in memory, which results in a conservative estimate of the number of memory pages.

intuition behind our approach.

We compute an upper bound of the total number of registers that can be simultaneously live by partitioning the references in the loop nest in groups of uniformly generated references and computing the superword footprint of each group.

For example, the number of registers required for a group that contains a single reference a that is variant with l is given by Equation 2, assuming $C(a, l) = 1$.

$$NR_l(\mathrm{a}) = \frac{X_l \times \mathrm{T(a)}}{\mathrm{W}} \tag{2}$$

where W is the register width in bytes (for example, $W = 4$ for a 32-bit scalar register, and $W = 16$ for a 128-bit superword register such as the AltiVec's and $T(a)$ is the type size of a in bytes. Equation 2.

The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The total number of registers required (TNR) to keep the data accessed in the loop nest is computed as the sum of the number of registers required for each group of uniformly generated references. If the total number of registers is larger than the number of registers available, the algorithm adjusts the unroll amount X_l, by dividing it by the ratio of TNR and the number of available registers $NREG$.

$$X_l \quad = \quad \left\lfloor \frac{X_l}{\left\lceil \frac{TNR}{NREG} \right\rceil} \right\rfloor \tag{3}$$

The number of available registers $NREG$ is given by number of registers in the register file minus the number of registers reserved by our algorithm for temporary storage.

Since the smallest type size is used in Equation 1, all references that have spatial reuse carried by loop l can exploit spatial reuse fully at the memory page level. Therefore, choosing a loop l that has the smallest random-mode accesses when unrolled by X_l is a reasonable choice. Dividing it evenly if too many registers are used, as in Equation 3, will result in a solution that is also aligned to a page boundary at the beginning of the loop. However, choosing a different loop can result in different register requirements. For example, if a loop is selected and then its unroll amount is reduced to half because of register pressure, there can be another loop that results in more random-mode accesses but requires fewer registers, leading to less overall random-mode accesses than the initial selection.

Aligning the Loop To Page Boundaries If the starting addresses of the memory accesses in the unrolled loop body are not aligned to a page boundary, each set of memory accesses to a same array will have one additional random-mode access per iteration. To remove these unnecessary random-mode accesses, step 4 of the algorithm splits the iteration space of the chosen loop into at most three loops (*head*, *body* and *tail*), so that the starting addresses in loop *body* are aligned to page boundaries. The body loop contains all iterations that access memory between the first and the last page boundary, with the head loop performing previous iterations starting from the lower bound of the original loop,

```
for(i=0; i<1280; i+=64){
    Load A[i+1]
    Load A[i+2]
      ...
    Load A[i+64]
    Load B[i+1]
      ...
}
```

(a) Unaligned

```
for(i=0; i<63; i++){
    Load A[i+1]
    Load B[i+1]
      ...
}
for(i=63; i<1279; i+=64){
    Load A[i+1]
    Load A[i+2]
      ...
    Load A[i+64]
    Load B[i+1]
      ...
}
for(i=1279; i<1280; i++){
    Load A[i+1]
    Load B[i+1]
      ...
}
```

(b) Aligned

Figure 3: Alignment by Iteration Space Splitting

and the tail loop computing subsequent iterations up to the upper bound of the original loop.

Figure 3(a) shows an example of an unrolled loop with misaligned memory references. Assuming that array A is aligned to a memory page, the memory accesses for one iteration of the unrolled loop span a page boundary. In (b), the iteration space of the original loop is split so that the memory accesses in the body loop start and end at page boundaries. The lower bound of the body loop and the lower bound of the tail loop are computed from the array subscript expressions and the loop bounds as follows. The earliest iteration where the most array references are aligned on a page boundary is used as the lower bound of the body loop. Let a be a representative reference to be aligned, l the loop index variable for the selected loop, and lb and ub the lower and upper bounds for l. To derive the loop bounds for the copies of the selected loop resulting from iteration space splitting, we begin with the starting address, $addr$, of the references when $l = lb$, where $addr = aligned + offset$. Here, $aligned$ refers to the largest multiple of the page size less than $addr$ and $offset$ is the offset of $addr$ within a page.

Assuming the stride of a is 1, the lower bounds of the body loop (split1) and the tail loop (split2) are computed by the following equations where P is the memory page size and $T(a)$ is the type size of a.

$$\begin{aligned} split1 &= lb + \frac{P}{T(a)} - \frac{offset \bmod P}{T(a)} \\ split2 &= ub - (ub - split1) \bmod \frac{P}{T(a)} \end{aligned}$$

The head loop is not needed if the reference is aligned, as is the case when $offset \ mod \ P = 0$. If lb is constant, $split1$ and $split2$ can be computed at compile time. Otherwise, they are computed at run time.

If the selected reference has non-unit stride, the solution is much more complex. In this case, we build a modular linear equation and choose the smallest solution [5].

Reordering Memory Accesses Finally, the reordering step hoists loads to the top of the loop body and sinks stores to the bottom. While being hoisted / sunk, the loads / stores to a same array are grouped together and sorted by their

```
for(i=32; i<N; i+=64){           for(i=32; i<N; i+=64){
    load A[i + 0] (RMA)              load A[i + 0] (RMA)
    load A[i + 32] (RMA)             load A[i + 8]
    load A[i + 8] (RMA)              load A[i + 16]
    load A[i + 40] (RMA)             load A[i + 24]
    load A[i + 16] (RMA)             load A[i + 32] (RMA)
    load A[i + 48] (RMA)             load A[i + 40]
    load A[i + 24] (RMA)             load A[i + 48]
    load A[i + 56] (RMA)             load A[i + 56]
        ...                              ...
}                                }

(a) Unsorted                     (b) Sorted
```

Figure 4: Sorting Offset Addresses

Parameters	Value	Unit
Random-mode latency	12	Cycles
Page-mode latency	4	Cycles
Page size	256	Bytes

Table 2: Simulation Parameters

offset addresses. When there are unaligned array references even after aligning the loop, sorting the offset addresses can reduce the number of random-mode accesses. Figure 4 shows an example where the page size includes 64 elements of array A. All eight memory accesses are in random mode before sorting. After sorting the offset addresses, only two random-mode accesses remain.

```
for(...){                 for(...){
    load A (RMA)              load A
    load B (RMA)              load B (RMA)
      ...                       ...
    Computation               Computation
      ...                       ...
    store A (RMA)             store B
    store B (RMA)             store A (RMA)
}                         }

(a) Before                (b) After
```

Figure 5: Grouping loads and stores

This step also groups loads and stores to the same array when possible, to exploit page mode among them. There can be page-mode accesses between loads and stores if the last load and the first store access the same page, and there are no intervening memory accesses between them. The same is true between the last store of an iteration of the innermost loop and the first load of the next iteration. Using this technique, at most 2 random-mode accesses per iteration can be eliminated. Figure 5 (a) shows an example where two array objects are read and written. Assuming all loads and stores to the same array objects access the same memory page, the loop in (a) results in four random-mode accesses whereas (b) has only two random-mode accesses per iteration.

4. EXPERIMENTS

Although our algorithm is applicable to general embedded-DRAM systems, for the experiments presented in this paper we used a compiler framework that we have built for DIVA, as previously described. The DIVA PIM device has a 256-bit datapath for executing superword operations in parallel. In addition to a conventional scalar register file, the DIVA PIM processor has 32 256-bit registers (each of which can be treated as eight 32-bit operands, sixteen 16-bit operands or 32 8-bit operands). Thus, for data allocated to superword registers, width W from Section 3 is 256, and for scalar registers W is 32. As the DIVA PIM devices contain no data cache, exploiting spatial locality in the memory pages can have significant impact on application performance.

Name	Description	Input Size
VMM	Vector-matrix multiply	64 elements
MMM	Matrix-matrix multiply	64 elements
YUV	RGB to YUV conversion	32K elements
FIR	Finite impulse response filter	256 filter, 1K signal

Table 3: Banchmark programs

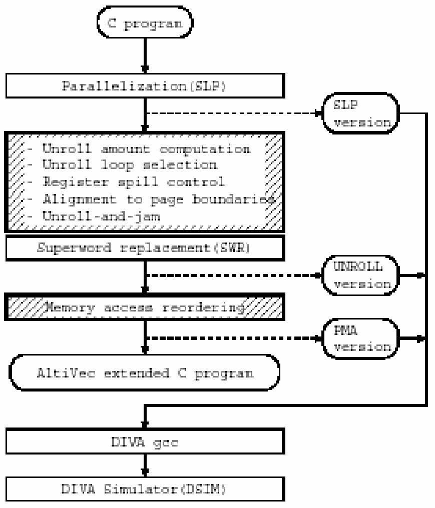


Figure 6: Experimental Flow

A prototype of the DIVA PIM chip has been fabricated recently [6], but the complete DIVA system is not available for our experiments at the time of this writing. Therefore, we used a cycle-accurate DIVA simulator(DSIM) [6], which is modified from RSIM [20]. Table 2 shows the simulation parameters for the memory system which closely match those of the IBM Cu-11 embedded DRAM macro [14]. In general there can be multiple DRAM macros and multiple open pages in a single chip, but for our experiments we assume that only one memory page is open at any given time.

We implemented the bulk of the algorithm presented in the previous section, and integrated it into the Stanford SUIF compiler. In our current implementation, we have not implemented alignment to page boundaries or combining loads and stores for page mode accesses. However, these steps of the algorithm do not affect the results for the four benchmarks used. The input to the modified SUIF compiler is a C program, and the output is an AltiVec-extended C program [18] which, in turn, is translated by a preliminary version of the DIVA gcc backend.

Table 3 shows the four kernels used to evaluate the effectiveness of the algorithm. The kernels represent data intensive applications in scientific and multimedia domains. Figure 6 shows the experimental flow. The main algorithm involves selecting unroll factors, performing unroll-and-jam and memory access reordering, and is represented by the hashed rectangles in Figure 6.

As previously stated, this algorithm is implemented as part of a compiler that exploits superword-level parallelism and locality in the superword register file. Thus, the experimental methodology also includes optimizations to exploit superword-level parallelism (SLP). Further, we exploit spatial and temporal locality in the superword register file through a combination of unroll-and-jam and *superword replacement* (SWR). Superword replacement is applied after unroll-and-jam to replace unnecessary superword memory accesses with references to superword temporaries that will then be allocated to superword registers by a backend compiler [22]. In our previous work, we selected unroll factors for unroll-and-jam that maximize reuse in superword registers; here, we use the unroll factors determined by the algorithm in Section 3, which are likely to be larger than in our previous work. In some sense, the optimizations for page mode memory accesses are complementary to exploiting SLP and locality in superword registers, and the page mode optimizations are difficult to isolate in our compiler. In fact, because the SLP and SWR optimizations reduce the number of memory accesses, we will see less benefit from the page mode optimizations than if considered in isolation.

We use as our baseline the SLP version of the code with no unrolling beyond what is required to exploit parallelization of the innermost loop. The UNROLL version includes unroll-and-jam, where the loop selected by the algorithm in Section 3 is unrolled by the chosen amount, and inner loop bodies are fused together. As compared to the baseline version, this version isolates the benefits of unroll-and-jam and superword replacement in terms of reduced memory accesses and less loop overhead, as compared to the baseline version. The PMA version reflects the performance improvements due to memory access reordering, yielding the full benefit of the optimizations for page-mode accesses.

In these experiments, we used optimization level -O1 for the DIVA gcc backend rather than a higher level of optimization. This was required to avoid reordering of memory accesses in subsequent optimization passes, which occurs at higher levels of optimization. Since reordering commonly occurs in backend optimizations, we discuss the implications of combining the page-mode optimizations with other backend compiler techniques in the next section.

For all programs but YUV, the algorithm was able to unroll the selected loop by the unroll factor determined by Equation 1. For YUV, which references six distinct arrays, this unroll factor was too large and resulted in register spilling. The algorithm reduced the unroll amount by half and the register spilling was eliminated.

We first consider how the optimizations for exploiting page-mode memory accesses impact memory stall time. In Figure 8 shows the normalized execution times broken down into processor busy time and memory stall time, derived from simulation. The UNROLL version sees a significant reduction in both processor busy time (9% to 60%) and memory stall time (25% to 71%). The primary reason for this is that superword replacement has eliminated a large number of memory accesses, which not only reduces memory stall time, but also reduce processor busy time by eliminating address calculation and instruction issue associated with the eliminated memory accesses. Further, reduction in loop control overhead also reduces processor busy time. For all programs, the PMA version further reduces memory stall

```
for(i = 0; i < 64; i++)
   for(j = 0; j < 64; j++)
      for(k = 0; k < 64; k += 8){
         load C[i][j]
         load B[i][k]
         load A[j][k]
            ...
         store C[i][j]
      }
```

(b) VMM

```
for(i = 0; i < 64; i++)
   for(j = 0; j < 64; j += 8)
      for(k = 0; k < 64; k++){
         load C[i][j]
         load A[i][k]
         load B[k][j]
            ...
         store C[i][j]
      }
```

(a) MMM

Figure 7: SLP versions of VMM and MMM

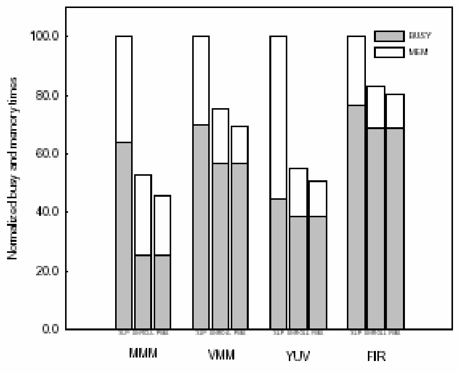


Figure 8: Normalized Execution Time

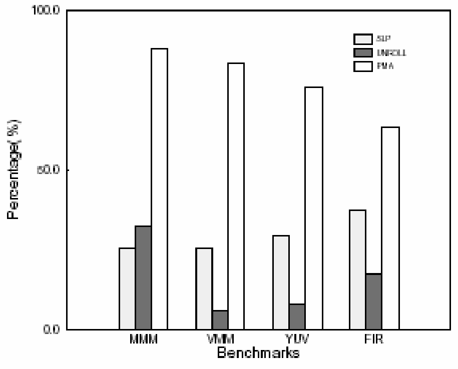


Figure 9: Percentage of Page-Mode Accesses

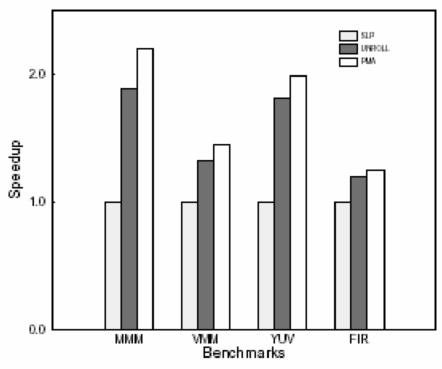


Figure 10: Speedup Breakdown

time by 21% to 33%. As compared to the UNROLL version, we have not eliminated any instructions, but rather have converted random-mode accesses to page-mode accesses.

Next we consider in Figure 9 the percentage of all memory accesses that are in page mode, with the remainder in random mode. The percentages of page-mode accesses ranges from 25% to 37% for the baseline version of the programs. We see a decrease in page-mode accesses as a percentage of memory accesses for most programs for the UNROLL version, ranging from 6% to 32%. This effect is because superword replacement has removed a large number of memory accesses, and the remainder tend to be in random mode. For example, in the VMM loop shown in Figure 7(a) after SLP, references to `C[i][j]` in the `k`-loop are loop-invariant after unrolling, and are usually removed, but were page-mode accesses in the SLP version due to the preceding store to the same location. In MMM, the page-mode percentage actually increases for the UNROLL version, as can be seen in Figure 7(b). References to `A[i][k]` are random-mode accesses, and are eliminated by superword replacement. For the PMA version, which reflects the same number of memory accesses as the UNROLL version, the percentages of page-mode accesses range from 63% to 87%.

These results show that our algorithm has been successful at increasing the percentage of page-mode accesses and reducing the memory stall time. We now see how the approach impacts the overall performance. Figure 10 shows the speedups for the SLP, UNROLL and PMA versions of Figure 6. Overall speedups as compared to the SLP baseline range from 1.25 to 2.19. Most of this speedup comes from the 1.19 to 1.89 improvement from unroll-and-jam and superword replacement, as can be seen from the UNROLL version. The speedup of the PMA version over the UNROLL version ranges from 1.04 to 1.16.

5. IMPLEMENTATION ISSUES

In this section, we consider in general terms how to incorporate this algorithm into current and future compilers. First, the compiler backend optimizations must be aware that page-mode optimizations are being performed. Otherwise, instruction reordering optimizations to increase instruction-level parallelism may undo the effect of the page-mode optimizations. A simple solution is to keep the relative order of memory operations intact when performing instruction reordering. There is an interesting tradeoff space that must be considered, since page-mode optimizations which favor memory accesses to the same page may potentially lengthen the critical path to performing computations, where multiple operands from different pages may be required. This potential problem is mitigated if there are a large number of memory units that can operate in parallel, or if there are multiple memory pages from which data can be accessed rather than the single page used in our experiments.

A second issue is how to combine this approach with cache optimizations for devices that have on-chip data caches. In cache-based architectures, the page-mode optimizations are still applicable as long as the unrolled footprint for an object exceeds the cache line size. In such a case, the spatial locality within the memory page complements spatial locality within a cache line.

6. RELATED WORK

Previous research has identified the benefits of exploiting page-mode DRAM accesses [19, 17, 3, 21, 11]. Moyer modeled memory systems analytically and developed a compiler technique called *access ordering* that reorders memory accesses to better utilize the memory system [19]. McKee et al. described a Stream Memory Controller (SMC) whose access ordering circuitry attempts to maximize memory system performance based on the device characteristics [17]. Their compiler is used to detect streams but access ordering and issue is determined by the hardware. Chame et al. manually optimized an application for a PIM-based (embedded-DRAM) system [3] by applying loop unrolling and memory access reordering to increase the number of page-mode accesses.

Panda et al. have developed a series of techniques to exploit page-mode DRAM access in high-level synthesis [21]. Their techniques include scalar variable clustering, memory access reordering, hoisting and loop transformations. While their ASIC design was able to exploit page-mode memory access, they do not describe an algorithm for automatic code generation. Grun et al. have optimized a set of benchmarks to better utilize efficient memory access modes for their IP library based Design Space Exploration [11]. However, their focus was on accurate timing models of the hardware system description.

This paper is distinguished from previous research as the design and implementation of a compiler algorithm to exploit page-mode automatically. Although the experiments are performed for a PIM-based system [12], this compiler framework is applicable to embedded-DRAM systems and can also be used as a preprocessor for high-level synthesis.

7. CONCLUSION

This paper presented a compiler algorithm for exploiting page-mode memory access in embedded-DRAM systems. Our compiler algorithm has been implemented in the Stanford SUIF compiler infrastructure and evaluated for four scientific and multimedia kernels. The speedups achieved by exploiting page-mode memory access alone range from 1.04 to 1.16 for four multimedia kernels, resulting in overall speedups ranging from 1.25 to 2.19 when combined with optimizations targeting superword-level parallelism and locality. These results show that there is a distinct benefit in exploiting page-mode memory access in embedded systems, where the DRAM access time dominates the memory latency seen by the processor. Furthermore, our results show that for embedded systems with support for superword-level parallelism [23, 1, 12], optimizations for exploiting the DRAM's page-mode accesses are complementary to optimizations for superword-level parallelism and superword-level locality.

8. REFERENCES

[1] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling. Microservers: A new memory semantics for massively parallel computing. In *ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[2] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 15(3):400–462, July 1994.

[3] Jacqueline Chame, Jaewook Shin, and Mary Hall. Compiler transformations for exploiting bandwidth in PIM-based systems. In The 27th Annual International Symposium on Computer Architecture, Workshop on Solving the Memory Wall Problem, June 11, 2000, Vancouver, British Columbia, Canada.

[4] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[5] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. McGraw Hill, 2nd edition, 1990.

[6] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steel, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the DIVA processing-in-memory chip. In *Proceedings of the 16th ACM International Conference on Supercomputing*, pages 26–37, June 2002.

[7] Karim Esseghir. Improving data locality for caches. Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[8] Christine Fricker, Olivier Temam, and William Jalby. Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply. *ACM Transactions on Programming Languages and Systems*, 17(4):561–575, July 1995.

[9] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[10] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, California, October 1998.

[11] Peter Grun, Nikil D. Dutt, and Alexandru Nicolau. Memory aware compilation through accurate timing extraction. In *Design Automation Conference*, pages 316–321, 2000.

[12] Mary Hall, Peter Kogge, Jeff Koller, Pedro Diniz, Jacqueline Chame, Jeff Draper, Jeff LaCoss, John Granacki, Apoorv Srivastava, William Athas, Jay Brockman, Vincent Freeh, Joonseok Park, and Jaewook Shin. Mapping irregular applications to DIVA, a PIM-based data-intensive architecture. In *ACM International Conference on Supercomputing*, November 1999.

[13] Mary W. Hall, Jennifer M. Anderson, Saman p. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, Edouard Bugnion, and Monica S. Lam. Maximizing multiprocessor performance with the SUIF compiler. *Computer*, 29(12):84–89, 1996.

[14] IBM. *IBM Cu-11 embedded DRAM macro datasheet*, March 2002. http://www-3.ibm.com/chips/techlib/techlib.nsf/techdocs/4CBB96F927E2D6D287256B98004E1D98/$file/Cu11_Embedded_DRAM.pdf.

[15] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimization of blocked algorithms. *ACM SIGPLAN Notices*, 26(4):63–74, 1991.

[16] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Conference on Programming Language Design and Implementation*, pages 145–156, Vancouver, BC Canada, June 2000.

[17] Sally A. McKee, William A. Wulf, James H. Aylor, Robert H. Klenke, Maximo H. Salinas, Sung I. Hong, and Dee A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, 2000.

[18] Motorola. *AltiVec Technology Programming Interface Manual*, June 1999. http://e-www.motorola.com/brdata/PDFDB/docs/ALTIVECPIM.pdf.

[19] Steven A. Moyer. *Access Ordering and Effective Memory Bandwidth*. PhD thesis, University of Virginia, 1993.

[20] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. Rsim reference manual. version 1.0. Technical Report 9705, Department of Electrical and Computer Engineering, Rice University, July 1997.

[21] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. Exploiting off-chip memory access modes in high-level synthesis. *IEEE Transactions on CAD*, February 1998.

[22] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, Charlottesville, Virginia, September 2002.

[23] Thomas Sterling. An introduction to the gilgamesh pim architecture. In Rizos Sakellariou, John Keane, John R. Gurd, and Len Freeman, editors, *Euro-Par*, volume 2150 of *Lecture Notes in Computer Science*, pages 16–32. Springer, 2001.

[24] O. Temam, E. Granston, and W. Jalby. To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts. In *ACM International Conference on Supercomputing*, Portland, OR, November 1993.

[25] Michael E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.

Journal of Instruction-Level Parallelism 5 (2003) 1-28 Submitted 10/02; published 4/03

Exploiting Superword-Level Locality in Multimedia Extension Architectures

Jaewook Shin JAEWOOK@ISI.EDU
Jacqueline Chame JCHAME@ISI.EDU
Mary W. Hall MHALL@ISI.EDU
Information Sciences Institute, University of Southern California,
Suite 1001, 4676 Admiralty Way, Marina del Rey, CA 90292-6695

Abstract

In this paper, we describe an algorithm and implementation of locality optimizations for architectures with instruction sets such as Intel's SSE and Motorola's AltiVec that support operations on superwords, i.e., aggregate objects consisting of several machine words. We treat the large superword register file as a compiler-controlled cache, thus avoiding unnecessary memory accesses by exploiting reuse in superword registers. This research is distinguished from previous work on exploiting reuse in scalar registers because it considers not only temporal but also spatial reuse. As compared to optimizations to exploit reuse in cache, the compiler must also manage replacement, and thus, explicitly name registers in the generated code. We describe an implementation of our approach integrated with a compiler that exploits superword-level parallelism (SLP). We present a set of results derived automatically on 4 multimedia kernels and 2 scientific benchmarks. Our results show speedups ranging from 1.3 to 3.1X on the 6 programs as compared to using SLP alone, and we eliminate the majority of memory accesses.

1. Introduction

In response to the increasing importance of multimedia applications in embedded and general-purpose computing environments, many microprocessors now incorporate an expanded instruction set and architectural extensions specifically targeting multimedia requirements. The core component of such architectural extensions is a functional unit that can operate on aggregate objects, performing bit-level operations, or SIMD parallel operations on variable-sized fields in the object (*e.g.,* 8, 16, 32 or 64-bit fields). If the aggregate objects are larger than the size of a machine word, then they are called *superwords* [1]. Examples include Motorola's AltiVec and Intel's SSE, a descendant of MMX. If the same size as the machine word, then individual fields are referred to as *subwords* [2]. A related class of architectures employ processing-in-memory (PIM) technology to exploit the high memory bandwidth when processing logic is combined on chip with large amounts of DRAM; several PIM-based architectures rely on superword parallelism to make more effective use of available memory bandwidth [3, 4, 5, 6].

While multimedia extension and related architectures have been available for some time, convenient methodologies for developing application code that targets these extensions are in their infancy. There is recent compiler research for such architectures to automatically exploit *superword-level parallelism,* performing computations or memory accesses in parallel in a single instruction issue [1, 7, 8, 9, 10].

In this paper, we recognize an additional optimization opportunity not addressed by this previous work. An important feature of all such architectures is a register file of superwords (*e.g.,* each 128 bits wide in an AltiVec), usually in addition to the scalar register file. A set of 32 such superword registers represents a not insignificant amount of storage close to the processor. Accessing data from superword registers, versus a cache or main memory, has two advantages. The most obvious advantage is lower latency of accesses; even a hit in the L1 cache has at least a 1-cycle latency. Accesses to other caches in the hierarchy or to main memory carry much higher latencies. Another advantage is the elimination of memory access instructions, thus reducing the number of instructions to be issued.

In this paper, we treat the superword register file as a small compiler-controlled cache. We develop an algorithm and a set of optimizations to exploit reuse of data in superword registers to eliminate unnecessary memory accesses, which we call *superword-level locality*. We evaluate the effectiveness of these superword-level locality (SLL) optimizations through an implementation integrated with the algorithm for exploiting superword-level parallelism (SLP) presented in [1].

Our approach is distinguished from previous work on increasing reuse in cache [11, 12, 13, 14, 15, 16, 17, 18], in that the compiler must also manage replacement, and thus, explicitly name the registers in the code. As compared to previous work on exploiting reuse in scalar registers [18, 19, 20], the compiler considers not just temporal reuse, but also spatial reuse, for both individual statements and groups of references. Further, it also considers superword parallelism in making its optimization decisions. Exploiting spatial and group reuse in superword registers requires more complex analysis as compared to exploiting temporal reuse in scalar registers, to determine which accesses map into the same superword.

In conjunction with exploiting SLP, the algorithm performs what we call *superword replacement*, to replace accesses to contiguous array data with superword temporaries and exploit reuse by replacing accesses to the same superword with the same temporary. Following this code transformation, a separate compilation pass will be able to allocate superword registers corresponding to the superword temporaries. To enhance the effectiveness of superword replacement, it is combined with a loop transformation called *unroll-and-jam*, whereby outer loops in a loop nest are unrolled, and the resulting duplicate inner loop bodies are fused together. Unroll-and-jam reduces the distance between reuse of the same superword, when reuse is carried by an outer loop, and brings opportunities for superword replacement into the innermost loop body of the transformed loop nest. The optimization algorithm derives appropriate unroll factors for each loop in the nest that attempt to maximize reuse while not exceeding the number of available registers.

The contributions of this paper are as follows:

- An algorithm for exposing opportunities for compiler-controlled caching of data in superword register files using unroll-and-jam. The two main components of this algorithm are a model of the number of memory accesses and registers required associated with a set of unroll factors, and a strategy for navigating the search space of possible unroll factors.

- A description of a set of code transformations, which in aggregate we call superword replacement, for exploiting superword register reuse.

- Experimental results, derived automatically, comparing performance of six benchmarks/multimedia kernels optimized for parallelism only, SLP, and optimized for both parallelism and superword-level locality. Our results show speedups ranging from 1.3 to 3.1X as compared to using SLP alone, and we eliminate the majority of memory accesses.

This paper extends an earlier description of this work in several ways [21]. We have extended the algorithm and register requirements analysis to exploit group-temporal reuse across iterations of the transformed loop nest. We have also greatly expanded the description of code generation. In the experimental results description, we have improved the results and provided a more detailed breakdown of the contributions of the different techniques.

The remainder of the paper is organized into 8 sections. Section 2 motivates the problem and introduces terminology used in the remainder of the paper. Section 3 presents an overview of the superword-level locality algorithm. Section 4 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 5 describes aspects of how the search space is navigated. Section 6 presents optimizations to actually achieve this reuse of data in superword registers. Section 7 presents experimental results derived automatically by an implementation in the Stanford SUIF compiler. Section 8 discusses related work and Section 9 presents conclusions and future work.

2. Background and Motivation

In many cases superword-level parallelism and superword-level locality are complementary optimization goals, since achieving SLP requires each operand to be a set of words packed into a superword, which happens, with no extra cost, when an array reference with spatial reuse is loaded from memory into a superword register. Therefore, in many cases the loop that carries the most superword-level parallelism also carries the most spatial reuse, and benefits from SLL optimizations. In this paper, we achieve SLL and SLP somewhat independently, by integrating a set of SLL optimizations into an existing SLP compiler [1]. The remainder of this section motivates the SLL optimizations.

Achieving locality in superword registers differs from locality optimization for scalar registers. To exploit temporal reuse of data in scalar registers, compilers use *scalar replacement* to replace array references by accesses to temporary scalar variables, so that a separate backend register allocator will exploit reuse in registers [19]. In addition, *unroll-and-jam* is used to shorten the distances between reuse of the same array location by unrolling outer loops that carry reuse and fusing the resulting inner loops together [19].

In contrast, a compiler can optimize for superword-level locality in superword registers through a combination of unroll-and-jam and *superword replacement*. These techniques not only exploit temporal reuse of data, but also spatial reuse of nearby elements in the same superword. In fact, even partial reuse of superwords can be exploited by merging the contents of two registers containing superwords that are consecutive in memory (see Section 6.4). Thus, as is common in multimedia applications [22], streaming computations with little or no temporal reuse can still benefit from spatial locality at the superword-register level, in addition to the cache level.

While cache optimizations are beyond the scope of this paper, we observe that the SLL optimizations presented here can be applied to code that has been optimized for caches using well-known optimizations such as unimodular transformations, loop tiling and data prefetching. When combining loop tiling for caches, superword-level parallelism and superword-level locality optimizations, the tile sizes should be large enough for superword-level parallelism, and for unroll-and-jam and superword replacement to be profitable.

These points are illustrated by way of a code example, with the original code shown in Figure 1(a). This example shows three optimization paths. Figure 1(d) optimizes the code to achieve

```
for(i=0; i<n; i++)
  for (j=0; j<n; j++)
     a[i][j] = a[i-1][j] * b[i] + b[i+1];
```

(a) Original loop nest.

```
for(i=0; i<n; i+=2)
  for (j=0; j<n; j++) {
     a[i][j] = a[i-1][j] * b[i] + b[i+1];
     a[i+1][j] = a[i][j] * b[i+1] + b[i+2];
  }
```

(b) Unroll-and-jam on the example in (a)(i loop).

```
tmp1 = b[0];
for(i=0; i<n; i+=2) {
  tmp2 = b[i+1];
  tmp3 = b[i+2];
  for (j=0; j<n; j++) {
     tmp4 = a[i-1][j] * tmp1 + tmp2;
     a[i+1][j] = tmp4 * tmp2 + tmp3;
     a[i][j] = tmp4;
  }
  tmp1 = tmp3;
}
```

(c) After scalar replacement on the code in (c).

```
for(i=0; i<n; i++)
  for (j=0; j<n; j+=sws)
     a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
```

(d) After superword-level parallelism(j loop).

```
for(i=0; i<n; i+=2)
  for (j=0; j<n; j+= sws) {
     a[i][j:j+sws-1] = a[i-1][j:j+sws-1] * b[i] + b[i+1];
     a[i+1][j:j+sws-1] = a[i][j:j+sws-1] * b[i+1] + b[i+2];
  }
```

(e) Unroll-and-jam on the example in (d)(i loop).

```
tmp1[0:sws-1] = b[0:sws-1];
stmp1 = tmp1[0];
stmp2 = tmp1[1];
field = 2;
for(i=0; i<n; i+=2) {
  // 'field' denotes an index into 'tmp1' for stmp3
  if(field == 0)
     tmp1[0:sws-1] = b[i+2:i+sws+1];
  stmp3 = tmp1[field];
  for (j=0; j<n; j+= sws) {
     tmp2[0:sws-1] = a[i-1][j:j+sws-1] * stmp1 + stmp2;
     a[i+1][j:j+sws-1] = tmp2[0:sws-1] * stmp2 + stmp3;
     a[i][j:j+sws-1] = tmp2[0:sws-1];
  }
  stmp1 = stmp3;
  stmp2 = tmp1[field+1];
  field = (field+2)%sws;
}
```

(f) After superword replacement on code in (e)

Figure 1: Example code.

superword-level parallelism. Here, sws, an abbreviation for superword size, is the number of data elements that fit within a superword. For example, if a and b are 32-bit float variables, on a machine with 128-bit superwords, $sws = 4$. In Figures 1(b) and (c), we show how the original program can instead be optimized to exploit reuse in scalar registers, using unroll-and-jam and scalar replacement, respectively. In Figures 1(e) and (f), we combine these ideas, using unroll-and-jam and superword replacement, respectively, to transform the code in (d) for both superword-level parallelism and superword-level locality.

Table 1 shows how the three different optimization paths affect the number of array accesses to memory in the final code. The original code has n^2 reads and writes to array a and $2n^2$ reads to array b. Exploiting superword-level parallelism in loop j, as in Figure 1(d) reduces the number of reads and writes to array a by a factor of sws since each load or store operates on sws contiguous data items; for array b, there is no change since the array is indexed by i rather than j. If instead the code was optimized for scalar register reuse, as in Figure 1(c), we can reduce the number of array reads of a down by a factor of 2, and reads of b by a factor of n, with the number of writes remaining the

	Original Figure 1(a)	Scalar register reuse only Figure 1(c)	SLP only Figure 1(d)	SLP and SLL Figure 1(f)
Reads	$3n^2$	$n^2/2 + n$	$2n^2 + n^2/sws$	$(n^2/2 + n)/sws$
Writes	n^2	n^2	n^2/sws	n^2/sws

Table 1: Number of array accesses under different optimization paths.

same. By combining superword-level parallelism and superword-level locality as in Figure 1(f), we see that the number of reads and writes is further reduced by a factor of sws. Figure 1(f) illustrates some of the challenges in exploiting reuse in superwords. Analysis must identify not just temporal, but also spatial reuse, and for both individual statements and groups of references. The compiler also must generate the appropriate code to exploit this reuse; for example, we select scalar fields of b from the superword, since we are not parallelizing the i loop.

The remainder of this paper describes how the compiler automatically generates code such as is shown in Figure 1(f), and the performance improvements that can be obtained with this approach.

3. Overview of Superword-Level Locality Algorithm

The superword-level locality algorithm has three main steps, as summarized below. Each step will be described in more detail in the three subsequent sections.

Step 1: Identifying Reuse. The first step of the algorithm is to identify both array references and loops carrying reuse. The array references carrying reuse are the ones for which superword replacement may be applicable. The loops carrying reuse are the ones to which the algorithm will consider applying unroll-and-jam.

Reuse between two distinct array references in an $n-$dimensional loop nest is determined from data dependences, in the form of *dependence vectors*, $d = \langle d_1, d_2, \ldots, d_n \rangle$[23]. A dependence vector captures the vector distance, in terms of the loop iteration space, such that the two references may map to the same memory location. Each vector element d_i may be either a constant integer, $+$ (a positive direction where the distance is not fixed), $-$ (a negative direction), or $*$ (the direction and distance are unknown). We refer to a dependence vector as being *lexicographically positive* if the first non-zero d_i is $+$ or a positive integer.

For the purposes of reuse, the relevant dependences carrying reuse are a subset, and are characterized as follows:

1. We consider only true dependences (writes followed by reads), input dependences (reads followed by reads), and output dependences (writes followed writes). Although output dependences do not capture reuse of the same data value, they suggest an opportunity for eliminating unnecessary writes back to memory. Anti-dependences (writes followed by reads) are not considered.

2. We consider only lexicographically positive dependences.

3. A dependence vector must be *consistent, i.e.,* the dependence distance in the iteration space must be constant, or it must be invariant with respect to one of the loops in the nest.

```
for(i=0; i<N; i+=4){
    vec1[0:3] = A[i:i+3];
    vec2[0:3] = A[i+8:i+11];
        ⋮
}
```

(a) Original

```
tmp[0:3] = A[i:i+3];
vec2[0:3] = A[i+4:i+7];
for(i=0; i<N; i+=4){
    vec1[0:3] = tmp[0:3];
    tmp[0:3] = vec2[0:3];
    vec2[0:3] = A[i+8:i+11];
        ⋮
}
```

(b) After exploiting reuse

Figure 2: Reuse Across Iterations

Applying unroll-and-jam to a loop i with a consistent dependence varying with respect to loop i can create loop-independent dependences in the innermost loop of the unrolled loop body. In the example in Figure 1(a), there is a true dependence between references $A[i][j]$ and $A[i-1][j]$ with distance vector $\langle 1, 0 \rangle$. After unroll-and-jam, a loop-independent dependence is created between $A[i][j]$ in the first statement and $A[i][j]$ in the second statement of the loop body, creating a reuse opportunity.

In addition to reuse between copies of a reference created by unrolling, there can be reuse across loop iterations. References with consistent dependences carried by a loop have group reuse which can be exploited by using extra registers to hold the data across iterations. As in previous work [19], our algorithm exploits reuse across iterations of the innermost loop only, because exploiting reuse carried by an outer loop could potentially require too many registers to hold the data between uses. Figure 2 shows how reuse can be exploited across iterations of the innermost loop by using one register to keep the data that is reused on every two iterations.

For loop-invariant references, unroll-and-jam generates loop-independent dependences between the copies of the reference in the unrolled loop body, since the same location is being referenced by each copy.

Step 2: Determining unroll factors for candidate loops. The algorithm next determines the unroll factors for each candidate loop that carries reuse, as previously described, and for which unroll-and-jam is legal. The optimization goal is as follows.

> *Optimization Goal:* Find unroll factors $\langle X_1, X_2, ...X_n \rangle$ for loops 1 to n in an n-deep loop nest such that the number of memory accesses is minimized, subject to the constraint that the number of superword registers required does not exceed what is available.

The algorithm determines the unroll factors $\langle X_1, X_2, ...X_n \rangle$ by searching for the combination of unroll factors that satisfies the above optimization goal. To guide the search, the algorithm calculates the total number of registers required for exploiting reuse, which is the sum of the number of superwords accessed by the references in the loop body after unroll-and-jam is applied, plus the number of registers needed for holding data across iterations of the innermost loop. Section 4 describes how the algorithm computes the total number of registers required for exploiting reuse and the resulting number of memory accesses. Section 5 describes aspects of how the search space is navigated.

Step 3: Code Transformations - Unroll-and-Jam, Superword Replacement, and Related Optimizations. Once the unroll factors are decided, unroll-and-jam is applied to the loop nest. Array references are replaced with accesses to superword temporaries. As part of code generation, our compiler performs related optimizations to reduce the number of additional memory accesses and register requirements introduced by the SLP passes. These code transformations are the topic of Section 6.

4. Computing Registers Required and Memory Accesses

This section presents the computation of the number of registers required for exploiting data reuse in superword registers and the resulting number of memory accesses, which are the parameters used to guide the search for the combination of unroll amounts to be applied to the loop nest. The next subsection describes how the algorithm computes the *superword footprint*, which represents the number of superwords accessed by the unrolled iterations of the loop nest as a function of the unroll factors. Subsection 4.2 presents the computation of the extra registers needed for reusing data across loop iterations. The total number of registers and the corresponding number of memory accesses are computed in subsection 4.3.

4.1 Computing the Superword Footprint

This section presents the computation of the superword footprint of the references V in a loop nest, $F_L(V)$, after unroll-and-jam is applied to the nest with unroll factors $\langle X_1, X_2, ..., X_n \rangle$.

The algorithm for computing the superword footprint for a loop nest first partitions the references in the loop into groups of *uniformly generated references* [18], that is, references to the same array such that, for each array dimension, the array subscripts differ only by a constant term[1]. Then, for each group of references, it computes the number of superwords accessed in the unrolled loop body. Finally, the total number of superwords is computed as the sum of those of each group of uniformly generated references.

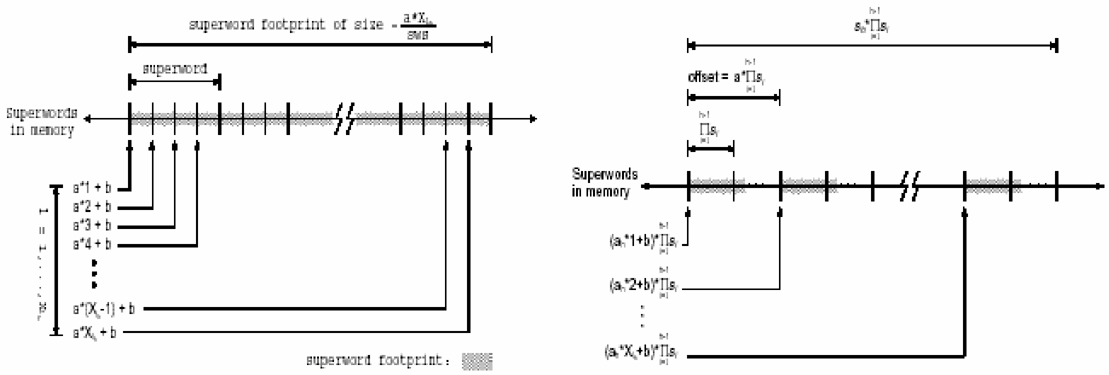
We first discuss how to compute the superword footprint of a single reference as a function of the unroll factors of each unrolled loop. Then we discuss how to compute the superword footprint of a group of uniformly generated references. The superword footprint of a group may be smaller than the sum of the individual fooptrints, since the same superword may be accessed by two or more copies of the original references when the loops are unrolled.

Our method determines the number of superword registers required to hold the data accessed by the loop references in the unrolled loops. However, extra registers may be needed to, for example, align a superword operand which is already kept in superword registers. That is, the computation may require more registers than those needed for storing the data. Therefore, we reserve some scratch registers for manipulating data and compute the number of registers needed just for storing the data accessed in the unrolled loops.

To simplify the presentation, we assume a loop nest of depth n where all array references have array subscripts that are affine functions of a single index variable (SIV subscripts)[2]. We also assume that each p-dimensional array referenced by the loop is defined as $A[s_p][s_{p-1}] \ldots [s_1]$, where s_h is

1. We assume that two or more references that access the same array, but are not uniformly generated, access distinct data in memory, which results in a conservative estimate of the number of superwords accessed by the group and of the number of registers required.

2. Our current implementation can handle affine SIV subscripts and certain affine MIV subscripts.



(a) $h = 1$ and $a_h < sws$ (b) $h \neq 1$

Figure 3: Superword footprint of a single reference.

the size of dimension h, $1 \leq h \leq p$. Dimension 1 is the lowest dimension of the array, *i.e.,* the dimension in which consecutive elements are in consecutive memory locations. A reference v to array A is then of the form $A[a_p * l_p + b_p][a_{p-1} * l_{p-1} + b_{p-1}] \ldots [a_1 * l_1 + b_1]$. Thus, a reference with SIV subscripts has each array dimension h associated with just a single loop index variable in the nest, and the loop index variable associated with h is represented as l_h. We also assume that the arrays are aligned to a superword in memory and that the loops are normalized.

4.1.1 Superword Footprint of a Single Reference

For each reference v with array subscripts $a_h * l_h + b$, where h is the array dimension and l_h is the loop index variable appearing in subscript h, the number of superwords accessed by all copies of v when l_h is unrolled by X_{l_h} is given by the *superword footprint* of v in l_h, or $Fl_h(v)$.

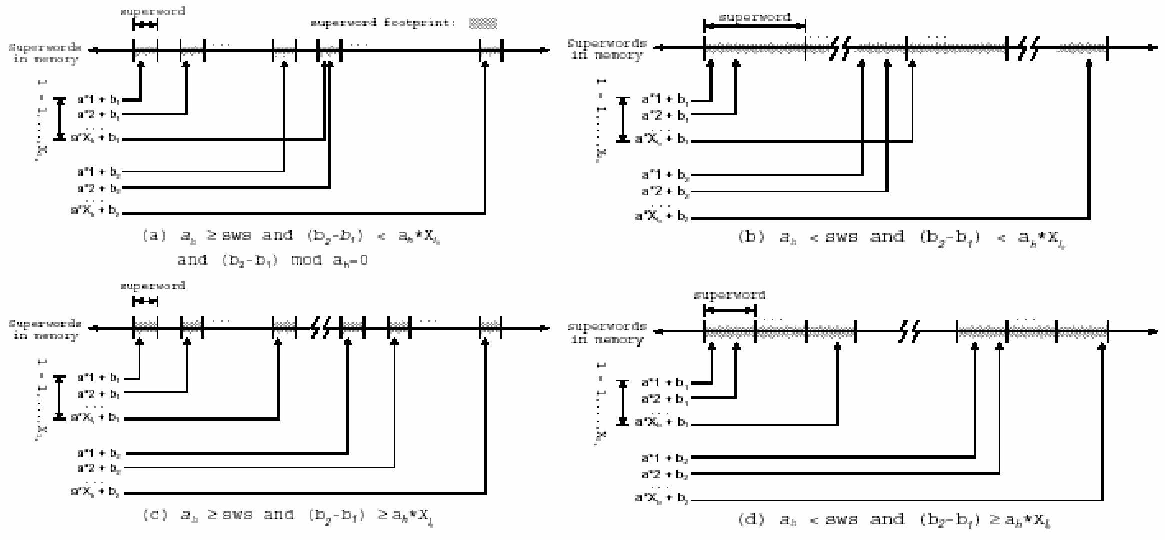
When dimension h is the lowest array dimension ($h = 1$), the superword footprint is given by Equation (1). Equation (1a) corresponds to the footprint of a loop-invariant reference. Equation (1b) corresponds to the footprint of a reference with self-spatial reuse within a superword, as illustrated in Figure 3(a), and (1c) holds when the reference has no spatial reuse.

$$Fl_h(v) = \begin{cases} 1 & \text{(a) if } a_h = 0 \\ \left\lceil \frac{X_{l_h} * a_h}{sws} \right\rceil & \text{(b) if } a_h < sws \\ X_{l_h} & \text{(c) if } a_h \geq sws \end{cases} \quad (1)$$

When h is one of the higher dimensions, $1 < h \leq p$, and loop l_h is unrolled, the offset between the footprints of each copy of v is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of the i^{th} array dimension, as shown in Figure 3(b). Assuming that the size of the lowest array dimension (s_1) is larger than sws, which is usually the case in practice for realistic array dimensions, each copy of v in the unrolled loop body corresponds to a separate footprint, as shown in Figure 3(b). Therefore the size of the footprint of v in l_h is the sum of the X_{l_h} disjoint footprints, and is recursively defined by Equation (2), where $Fl_1(v)$ is computed as in Equation (1).

$$\begin{aligned} Fl_h(v) &= X_{l_h} * Fl_{h-1}(v) \\ &= (\prod_{i=2}^{h} X_{l_i}) * Fl_1(v) \end{aligned} \quad (2)$$

$$F_{l_h}(v_1, v_2) = \begin{cases} X_{l_h} + (b_2 - b_1)/a_h & \text{(a) if } a_h \geq sws \text{ and } (b_2 - b_1) < a_h * X_{l_h} \text{ and} \\ & \quad (b_2 - b_1) \bmod a_h = 0 \\ \lceil (a_h * X_{l_h} + b_2 - b_1)/sws \rceil & \text{(b) if } a_h < sws \text{ and } (b_2 - b_1) < a_h * X_{l_h} \\ F_{l_h}(v_1) + F_{l_h}(v_2) & \text{(c) otherwise} \end{cases} \quad (3)$$



superword
superword footprint:
Superwords in memory
$l_h = 1,\ldots,X_{l_h}$
$a*1 + b_1$
$a*2 + b_1$
$a*X_{l_h} + b_1$
$a*1 + b_2$
$a*2 + b_2$
$a*X_{l_h} + b_2$
(a) $a_h \geq$ sws and $(b_2-b_1) < a_h*X_{l_h}$ and (b_2-b_1) mod a_h=0
(b) $a_h <$ sws and $(b_2-b_1) < a_h*X_{l_h}$
(c) $a_h \geq$ sws and $(b_2-b_1) \geq a_h*X_{l_h}$
(d) $a_h <$ sws and $(b_2-b_1) \geq a_h*X_{l_h}$

Figure 4: Superword footprint of a group of references.

For a single reference, the number of superword registers required to keep the superword footprint given by Equation (1) and the number of scalar registers that would be required if the same unroll factors were used differ only when $a_h < sws$, that is, when spatial reuse can be exploited in superword registers. For a group of uniformly generated references the analysis must also consider group reuse, as discussed next.

4.1.2 Superword Footprint of a Group of References

The number of superwords accessed by a group of uniformly generated references $V = \{v_1, v_2, ..., v_m\}$ when loop l_h is unrolled by X_{l_h} is the superword footprint of the group, $F_{l_h}(V)$. The superword footprint of a group consists of the union of the footprints of the individual references, as some of the reference footprints may overlap, depending on the distance between the constant terms in the array subscripts.

The footprints of two uniformly generated references may overlap in dimension h only if they overlap in all dimensions higher than h. For example, the footprints of references $A[2i][j+2]$ and $[2i+1][j]$ do not overlap in the highest (row) dimension, since the first reference accesses the even-numbered rows of the array and the second accesses the odd-numbered rows. Therefore the footprints cannot overlap in the lowest (column) dimension. On the other hand, the footprints of $A[2i][j+2]$ and $A[2i+4][j]$ overlap in the row dimension for iterations i_1, i_2, $1 \leq i_1, i_2 \leq X_i$, such that $2i_1 = 2i_2 + 4$. For the iterations of i in which the footprints overlap in the row dimension, the footprints may overlap in the column dimension if there exist iterations j_1, j_2, $1 \leq j_1, j_2 \leq X_j$, such that $j_1 + 2 = j_2$.

The superword footprint $F_L(V)$ of a group V, following unroll-and-jam, is computed as follows. First, the array dimensions with array subscripts that are a function of any of the unrolled loops are identified. Then, for each such dimension h, from highest to lowest dimension, the footprint is computed assuming that the footprints of the references in the group overlap in the higher dimensions. For each dimension $h > 1$, the algorithm partitions references into subsets such that each subset corresponds to a disjoint footprint in dimension h. Then, for each subset, the algorithm recursively computes the footprint in dimension $h - 1$, as we now describe.

Dimension h is the lowest dimension ($h = 1$). We first compute the group footprint of two array references, and then we extend it for m references. The footprint of group $V = \{v_1, v_2\}$, where references v_1 and v_2 have lowest dimension subscripts $a_h * l_h + b_1$ and $a_h * l_h + b_2$ such that $b_1 \leq b_2$, when loop l_h is unrolled by X_{l_h} is given by Equation (3) in Figure 4. Equations (3a) and (3b) apply when the two footprints overlap, that is, when $(b_2 - b_1) < a_h * X_{l_h}$, as shown in Figures 4(a) and (b). When the footprints do not overlap, the group footprint is the sum of the individual footprints, as in Equation (3c), with examples in Figures 4(c) and (d).

In Figure 4(a), the references have no self-spatial reuse, that is, $a_h \geq sws$, and each individual footprint is a set of X_{l_h} superwords. The footprints overlap if $(b_2 - b_1)$ is evenly divided by a_h and there exists an integer value k, $1 \leq k \leq X_{l_h}$, such that $k = 1 + (b_2 - b_1)/a_h$. This case corresponds to Equation (3a), which computes the group footprint precisely when the two references have group-temporal reuse. In Figure 4(b), both references have self-spatial reuse within a superword, that is, $a_h < sws$. The corresponding footprint size is given by Equation (3b). In Figure 4(c), v_1 has no self-spatial reuse and each copy of v_1 in the unrolled loop body accesses a distinct superword, and the same is true for v_2. In Figure 4(d) both v_1 and v_2 have self-spatial reuse.

The footprint of a group $V = \{v_1, v_2, ..., v_m\}$, with array subscripts $a_1 * l_1 + b_i$ such that $1 \leq i \leq m$ and $b_1 \leq b_2 \leq ... \leq b_m$, is computed by first partitioning V into subgroups with disjoint footprints in the lowest dimension, as follows. A subgroup $V_i = \{v_{i_{min}}, v_{i_{min+1}}, ..., v_{i_{max}}\}$ is defined by lowest dimension subscripts $a_1 * l_1 + b_j$, where $\forall j, \quad i_{min} < j \leq i_{max}$,

$$\begin{aligned}
&(b_{j-1} \leq b_j) \wedge \\
&(b_j - b_{j-1} < a_1 * X_{l_1}) \wedge \\
&(b_{i_{min}} = b_1 \vee b_{i_{min}} - b_{i_{min}-1} \geq a_1 * X_{l_1}) \wedge \\
&(b_{i_{max}} = b_m \vee b_{i_{max}+1} - b_{i_{max}} \geq a_1 * X_{l_1})
\end{aligned} \qquad (4)$$

Then the group footprint V is computed as the sum of the disjoint footprints of sets V_i, as in (5).

$$F_{l_h}(V) \quad = \quad \sum_i F_{l_h}(V_i) \qquad (5)$$

The footprint of each subgroup V_i is computed by extending Equation (3) to $m > 2$ references. For example, when the references in V have self-spatial reuse, as in Equation (3b) ($a_1 < sws$), each subgroup V_i has a footprint consisting of contiguous superwords, since $b_j - b_{j-1} < a_1 * X_{l_1}$ for all j such that $i_{min} < j \leq i_{max}$. The footprint of V_i consists of the union of the individual footprints, with size given by Equation (6).

$$\begin{aligned} F_{l_h}(V_i) &= F_{l_h}(\{v_{i_{min}}, ..., v_{i_{max}}\}) \\ &= \left\lceil \frac{a_1 * X_{l_1} + b_{i_{max}} - b_{i_{min}}}{sws} \right\rceil \end{aligned} \quad (6)$$

For example, if $sws = 4$ and $X = 4$, group $V = \{A[i], A[i+2], A[i+5], A[i+12], A[i+14]\}$ can be partitioned into two subgroups $V_1 = \{A[i], A[i+2], A[i+5]\}$ and $V_2 = \{A[i+12], A[i+14]\}$ with disjoint superword footprints. Since the references have self-spatial reuse, each individual footprint and the footprint of each subgroup is a set of contiguous superwords. The total number of superwords accessed by the references in V is the sum of the disjoint footprints of sets V_1 and V_2, as in (7).

$$F_{l_1}(V) \;=\; F_{l_1}(V_1) + F_{l_1}(V_2) = \left\lceil \frac{1*4+5-0}{4} \right\rceil + \left\lceil \frac{1*4+14-12}{4} \right\rceil = 5 \quad (7)$$

Dimension h is not the lowest dimension ($h \neq 1$). When h is one of the higher dimensions, the superword footprint of $V = \{v_1, v_2, ..., v_m\}$ in loop l_h is again the union of the individual footprints.

From Section 4.1.1, the footprint of each reference v_i in the unrolled loop body consists of a set of X_{l_h} disjoint footprints (each footprint corresponding to a copy of v_i created by unrolling), and the offset between each pair of consecutive footprints is $a_h * \prod_{i=1}^{h-1} s_i$, where s_i is the size of dimension i.

Therefore the footprints of different references in the group may overlap, depending on the values of a_h, b_j and the unroll factor X_{l_h}. The footprints of two uniformly generated references v_1 and v_2 overlap in dimension h if there exists an integer value k, $1 \leq k \leq X_{l_h}$ that satisfies Condition (8):

$$a_h * k + b_1 = a_h + b_2. \quad (8)$$

that is, if $(b_2 - b_1)\%a_h = 0$ and $(b_2 - b_1)/a_h + 1 \leq X_{l_h}$. Furthermore, if there exists k satisfying the above condition, the footprints of the last $X_{l_h} - k + 1$ copies of v_1 in the unrolled loop body overlap with those of the first $X_{l_h} - k + 1$ copies of v_2. The footprint of $\{v_1, v_2\}$ is then given by Equation (9).

$$F_{l_h}(v_1, v_2) \;=\; (k-1) * F_{l_{h-1}}(v_1) + (X_{l_h} - k + 1) * F_{l_{h-1}}(v_1, v_2) + (k-1) * F_{l_{h-1}}(v_2) \quad (9)$$

To compute the size of the entire footprint of V in l_h, our algorithm partitions V into subsets $V_i = \{v_{i_{min}}, ..., v_{i_{max}}\}$ such that, for any j, $i_{min} < j \leq i_{max}$, the pair $\{v_{j-1}, v_j\}$ satisfies Condition (8). The footprint of V_i is the union of the footprints of its reference set and is computed by extending Equation (9) to more than two references.

4.2 Registers for Reuse Across Iterations

In addition to superword registers for exploiting reuse in the body of the transformed loop nest, extra superword registers may be required for exploiting reuse across iterations of the innermost loop for references with group-temporal reuse carried by the innermost loop n of the transformed loop nest.

To compute the number of registers needed to exploit group-temporal reuse across iterations of loop n, the algorithm examines groups of references that have consistent dependences carried by n.[3] Assume that unroll-and-jam has been applied to outer loops in a nest. After subsequently unrolling the innermost loop, extra registers are required if the reuse distance between references prior to unrolling loop n is larger than the unroll amount, *i.e.*, if $d_n > X_n$, as in Figure 2, where $d_n = 8$ and $X_n = 4$.

Let $C = \{v_1, v_2, ...v_m\}$ be a set of references that is a subset of a uniformly generated set, and, prior to unrolling the innermost loop resulting from unroll-and-jam by X_n, each pair $\langle v_i, v_{i+1}\rangle$ in C has a consistent dependence $d^i = \langle 0, 0, ..., d^i_n\rangle$, $d^i_n > 0$. Also, assume that the array subscript of the lowest dimension of each reference v_i in C is of the form $a_i * n + b_i$, and that $b_1 \leq b_2 \leq ... \leq b_m$. Unrolling loop n generates X_n copies of each original reference v_i in the body of the transformed loop nest.

When d^i_n is a multiple of the unroll factor X_n, each pair of copies of references $\langle v_i, v_{i+1}\rangle$ will reuse data after $\frac{d^i_n}{X_n}$ iterations. When d^i_n is not a multiple of X_n, some copies of a reference will reuse data after $\left\lceil\frac{d^i_n}{X_n}\right\rceil - 1$ iterations of n, while others will have a reuse distance of $\left\lceil\frac{d^i_n}{X_n}\right\rceil$ requiring one more register per copy. Thus, each pair of copies of references $\langle v_i, v_{i+1}\rangle$ requires at most $\left\lceil\frac{d^i_n}{X_n}\right\rceil - 1$ additional superword registers to keep the data across iterations of the innermost loop.

The number of registers required to exploit reuse across iterations of n by all pairs of copies is the number of registers required for each pair times the number of registers required to keep the superword footprint of reference v_i in the transformed loop nest:

$$R_A(v_i, v_{i+1}) \quad = \quad (\left\lceil\frac{d^i_n}{X_n}\right\rceil - 1) \times F_L(v_i) \tag{10}$$

Equation (10) may overestimate the number of registers if the footprint component $(F_L(v_i))$ overestimates registers, or for certain copies of references if d^i_n is not a multiple of X_n.

The total number of registers required for exploiting reuse across iterations for set C with leading reference v_1 is given by:

$$R_A(C) \quad = \quad \sum_{1 \leq i < m}\left((\left\lceil\frac{d^i_n}{X_n}\right\rceil - 1) \times F_L(v_1)\right) \tag{11}$$

4.3 Putting It All Together

Subsections 4.1 and 4.2 describe the computation of the number of registers required to exploit reuse in the body of the innermost loop (superword footprint) and across iterations of the innermost loop, assuming that unroll-and-jam has been applied the loop nest. This section presents the computation of the total number of registers required and the total number of memory accesses in the innermost loop of the transformed loop nest, which are the metrics used to prune and guide the search for unroll factors described in Section 3.

The total number of registers required to exploit reuse is the sum of the superword footprint of the references in the innermost loop of the transformed loop nest and the number of registers needed for exploiting reuse across iterations of the same innermost loop.

The superword footprint of the references, $F_L(V)$, is computed as in subsection 4.1. The total number of extra registers required for exploiting reuse across iterations of the innermost loop is

3. Note that such references, if their lowest dimension varies with n, may also have group-spatial reuse across loop iterations. However, our algorithm focuses on exploiting group-temporal reuse across iterations, since most of the group-spatial reuse is achieved within the body of the unrolled loop.

computed as in subsection 4.2, for each set C of loop-variant references with consistent dependences carried by the innermost loop.

The total number of superword registers required is then:

$$R(V) \quad = \quad F_L(V) + \sum_C R_A(C) \tag{12}$$

The total number of memory accesses in the innermost loop of the transformed loop nest is the sum of the memory accesses of each group C of references that are variant with the innermost loop n and have consistent dependences carried by n. For each group C, the number of memory accesses is given by the superword footprint of the leading reference of the group, v_1^c:

$$M(C) = F_L(v_1^c) \tag{13}$$

The total number of memory accesses is then:

$$M(V) \quad = \quad \sum_c F_L(v_1^c) \tag{14}$$

5. Search Algorithm

As previously stated, the goal of the search algorithm is to identify the unroll factors for the loops in the loop nest such that the number of memory accesses is minimized, without exceeding available registers. Thus, we must consider an $n-$dimensional search space, where each dimension has the number of elements corresponding to the iteration count of the loop. A full global search of this search space is prohibitively expensive, especially for deep loop nests or large loop bounds. Thus, we use a number of strategies for pruning the search space.

First, we eliminate from the search loops that do not carry reuse or for which unroll-and-jam is not safe. Further, we rely on the observation that the number of registers required monotonically increases with the unroll factor of a loop, assuming that all other unroll factors are fixed. Thus, we need not search beyond the unroll factors that exceed available registers. This latter point significantly prunes the search space in that the number of registers is usually fairly small (*e.g.*, 32 superword registers on the AltiVec), so that the search is concentrated on fairly small unroll factors. These pruning strategies are used in our current implementation, and at least for the programs in this study, are quite effective at making the search practical.

Further pruning is possible by making the additional observation that for each unrolled loop l, the amount of reuse of an array reference with reuse carried by l increases with the unroll factor X_l. Therefore reuse, like the register requirement calculation, is a monotonic, non-decreasing function of the unroll factor for each loop, given that the unroll factor of all other loops is fixed. Thus, within each dimension, holding all other unroll factors constant, binary search can be used rather than searching all points. We can also increase unroll factors by amounts corresponding to the superword size without much loss of precision, rather than considering each possible unroll factor, since the register requirements increase stepwise as a function of superword size. Additional pruning techniques that take into account the hardware's capability to take advantage of the results of optimization have been used in prior work [19, 24].

Our implementation navigates the search space from innermost loop to outermost loop, for the applicable loops in the nest, varying the unroll factor of one loop while keeping the unroll factors of all other loops fixed. Within a dimension of the search space, the lowest number of memory accesses will be derived at the largest unroll factor that meets the register constraint. However, lower unroll factors may also have the same estimate of memory accesses (because reuse is monotonically non-decreasing), so we identify the lowest unroll factor with the equivalent estimate of memory accesses. Then, the implementation considers the next applicable outer loop and the applicable inner loops nested inside it, and in a particular dimension, each time it reaches the largest unroll factor that meets the register constraint, it compares the estimated number of memory accesses to the lowest estimate so far to determine if a better solution has been found. The final result of the algorithm is the unroll factors corresponding to the best solution.

As a subtle point, when unroll-and-jam is applied from outermost to innermost loop, unrolling the inner loop does not affect data access patterns or reuse distance. For this reason, inner loop unrolling is not performed in earlier work [19]. In our context, however, because of the relationship between superword-level parallelism and superword replacement, inner loop unrolling exposes opportunities for superword loads and stores and thus can impact the analysis of register requirements. Nevertheless, when reuse is exploited across iterations of the innermost loop body as described in Section 4.2, it is not necessary to unroll the innermost loop beyond the superword size to achieve the goal of considering register requirements in conjunction with superword-level parallelism. Note, however, that smaller unroll factors for the innermost loop may be selected, if an unroll-and-jam of an outer loop carries more parallelism and reuse.

Although this search should theoretically find the optimal solution, according to our optimization criteria, in fact the solution is not guaranteed to result in the fewest number of memory accesses, for a number of reasons. First, in a few cases as noted, the register requirement analysis defined in the previous section must conservatively approximate. Second, it is difficult to estimate the register requirements used to hold temporaries, so we conservatively approximate this as well. Third, there is a tradeoff between using extra registers to hold values across iterations, as discussed in Section 4.2, versus using them to actually exploit reuse within the transformed innermost loop body. In fact, in general the algorithm does not take into consideration the amount of reuse resulting from performing superword replacement on specific references; replacing some references has more impact on decreasing memory accesses than others.

This section and the previous one have described how the compiler analyzes the code to identify reuse, register requirements and the unroll factors leading towards the lowest number of memory accesses. In the next section, we describe how these analyses are used in transforming the code to achieve the desired result.

6. Code Generation

In the previous section, we showed how consideration of superwords instead of scalar variables greatly increases the complexity of determining the number of registers and memory accesses associated with exploiting reuse under different unroll amounts. In this section, we further discuss the increased complexity of code generation when performing superword replacement instead of scalar replacement. The chief source of code generation complexity is the need for superword objects to be properly *aligned*, as in the following examples.

When performing memory operations, the architecture may actually require that an access be aligned at superword boundaries. For example, the AltiVec ignores the last four bits of an address when performing a superword load or store. In such an architecture, when an access is not aligned at a superword boundary, the compiler or programmer must read/write two adjacent superwords. A series of additional instructions *packs* the two superwords for reads or *unpacks* a superword into its corresponding two superwords for writes. Even on architectures that support memory accesses not aligned at superword boundaries, such as Intel's SSE, there is a performance penalty on unaligned accesses because the hardware must perform this realignment.

To perform an arithmetic or logical operation on two superword registers, the fields of the two operands must also be aligned. For example, to add the third and fourth fields of one superword register to the first and second fields of another, one of the registers must be shifted by two fields. Consider also the following example:

```
for i = 1, n
   c[i] = a[2i] + b[i]
```

The access to a has a stride of 2, while the access to b has a unit stride. Thus, the compiler or programmer must first pack the even elements of a into a superword register before adding them to the elements of b. A third example occurs when exploiting partial reuse of a superword where data in a register must be aligned to accommodate the next operation.

In the SLP compiler, the default solution to alignment involves packing data through memory. The SLP compiler allocates superword variables by declaring them using a special `vector` type designation, which is interpreted by the backend compiler to align the beginning of the variable to a superword boundary in memory. The start of each dimension of an array of such objects should also be aligned, by padding if necessary. Under these assumptions, the SLP compiler can detect when operations are unaligned. Unaligned data is packed into an aligned superword in memory before being loaded into a superword register, and is unpacked before storing back to memory.[4]

In summary, alignment is a key consideration in code generation, and the overhead of performing alignment operations can be quite high. Further, alignment operations may require a number of additional superword registers, and in some cases, may result in additional accesses to memory not accounted for by the model in the previous section. In this section, we show how to achieve the number of registers derived by our model through a set of code transformations, presented in the order in which they are performed by our compiler. In addition to superword replacement, described in Section 6.2, we also describe how index set splitting is used to align accesses to the beginning of an iteration in Section 6.1, and how our compiler eliminates additional memory accesses resulting from packing through memory for alignment in Section 6.3. We illustrate how these transformations collaborate with each other by way of an example in Figure 5, which is a simplified `FIR` filter.

6.1 Index Set Splitting

A simple way to reduce the need for alignment operations, when applicable, is to perform index set splitting on loops. For example, in Figure 5(b), the initial access to `out[1]` refers to the

4. For architectures that support copying between scalar and superword register files, such as Intel's SSE and DIVA, this packing can be performed more efficiently through register copies.

```
1)  for (i = 1; i < 64; i++)
2)      out[i] = 0.0;
3)
4)  for (i = 256; i < 320; i++)
5)      for (j = 0; j < 256; j++)
6)          out[i-256] = out[i-256] + in[i-j] * coe[j];
```

(a) Original

```
1)  for (i = 1; i < 4; i++){
2)      out[i] = 0.0;
3)  }
4)  for (i = 4; i < 64; i++){
5)      out[i] = 0.0;
6)  }
7)  for (i = 256; i < 320; i++){
8)      for (j = 0; j < 256; j++){
9)          out[i - 256] = out[i - 256] + in[i - j] * coe[j];
10)     }
11) }
```

(b) After index set splitting

```
1)  for (i = 1; i < 4; i++){
2)      out[i] = 0.0;
3)  }
4)  for (i = 4; i < 64; i += 4){
5)      out[i + 0] = 0.0;
6)      out[i + 1] = 0.0;
7)      out[i + 2] = 0.0;
8)      out[i + 3] = 0.0;
9)  }
10) for (i = 256; i < 320; i += 8){
11)     for (j = 0; j < 256; j += 8){
12)         out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 0)] * coe[j + 0];
13)         out[i + 0 - 256] = out[i + 0 - 256] + in[i + 0 - (j + 1)] * coe[j + 1];
14)                      ⋮
15)         out[i + 7 - 256] = out[i + 7 - 256] + in[i + 7 - (j + 7)] * coe[j + 7];
16)     }
17) }
```

(c) After unroll-and-jam

Figure 5: Code Generation Example

second field of a superword, assuming `out[0]` is aligned at a superword boundary. Through index set splitting, the portion of the loop from line 4-6 will always perform aligned accesses. This transformation is always safe, and is profitable whenever it increases the number of aligned memory accesses.

We assume index set splitting is performed prior to the SLP compiler. The loop is transformed so that accesses corresponding to a particular reference in the main loop body are aligned to superword boundaries. If there are multiple references and different choices for index set splitting are needed to align specific references, we select a representative reference that, if aligned through index set splitting, will also maximize alignment for other references. The reference selected must have unit stride within the innermost loop.

Let i be the loop index variable for the innermost loop, and lb and ub are the lower and upper bounds for i. To derive the loop bounds for the copies of the innermost loop resulting from index set splitting, we begin with the starting address, $addr$, of the reference when $i = lb$, where $addr = base + \mathit{offset}$. Here, $base$ refers to the beginning of the lowest dimension of the selected array, and

```
                     ⋮
1)   flat1 = *((float *)&vec0 + 3);
2)   flat2 = *((float *)&vec1 + 0);
3)   flat3 = *((float *)&vec1 + 1);
4)   flat4 = *((float *)&vec1 + 2);
5)   *((float *)&vec2 + 0) = flat1;
6)   *((float *)&vec2 + 1) = flat2;
7)   *((float *)&vec2 + 2) = flat3;
8)   *((float *)&vec2 + 3) = flat4;
9)   vec4 = vec_add(vec3, vec2);
10)  vec_st(vec4, i * 4 + 0, (float *)&out[-63]);
11)  vec5 = vec_ld(i * 4, (float *)&out[-63]);
12)  flat5 = *((float *)&vec6 + 2);
13)  flat6 = *((float *)&vec7 + 2);
14)  *((float *)&vec8 + 0) = flat5;
15)  *((float *)&vec8 + 1) = flat6;
                     ⋮
```

(d) After SLP compilation

```
                     ⋮
1)   flat1 = *((float *)&vec0 + 3);
2)   flat2 = *((float *)&vec1 + 0);
3)   flat3 = *((float *)&vec1 + 1);
4)   flat4 = *((float *)&vec1 + 2);
5)   *((float *)&vec2 + 0) = flat1;
6)   *((float *)&vec2 + 1) = flat2;
7)   *((float *)&vec2 + 2) = flat3;
8)   *((float *)&vec2 + 3) = flat4;
9)   vec4 = vec_add(vec3, vec2);
10)  flat5 = *((float *)&vec6 + 2);
11)  flat6 = *((float *)&vec7 + 2);
12)  *((float *)&vec8 + 0) = flat5;
13)  *((float *)&vec8 + 1) = flat6;
                     ⋮
```

(e) After superword replacement

```
                     ⋮
1)   temp1 = replicate(vec0, 3);
2)   temp2 = replicate(vec1, 0);
3)   temp3 = replicate(vec1, 1);
4)   temp4 = replicate(vec1, 2);
5)   vec2 = shift_and_load(temp1, temp1, 4);
6)   vec2 = shift_and_load(vec2, temp2, 4);
7)   vec2 = shift_and_load(vec2, temp3, 4);
8)   vec2 = shift_and_load(vec2, temp4, 4);
9)   vec4 = vec_add(vec3, vec2);
10)  temp1 = replicate(vec6, 2);
12)  temp2 = replicate(vec7, 2);
11)  vec8 = shift_and_load(temp1, temp1, 4);
13)  vec8 = shift_and_load(vec8, temp2, 12);
                     ⋮
```

(f) After packing in registers

Figure 5: Code Generation Example(Continued)

offset is the offset within that dimension. (Recall that the beginning of each dimension is aligned at superword boundaries.)

The lower bound (*split*) of the main loop body is computed by the following equation.

$$split = \begin{cases} lb & \text{if } \mathit{offset} \bmod sws = 0 \\ lb + sws - (\mathit{offset} \bmod sws) & \text{if } \mathit{offset} \bmod sws \neq 0 \end{cases} \qquad (15)$$

If lb is constant, $split$ can be computed at compile time. Otherwise, it is computed at run time. In the example in Figure 5, *offset* for `out [1]` is 1, so if $sws = 4$, then $split = 4$.

6.2 Superword Replacement

Superword replacement removes redundant loads and stores of superword variables, using superword temporaries instead. We assume that this code transformation will be followed by register allocation that places these variables in registers. For example, in Figure 5(d) and (e), the store and load at statements 10 and 11 can both be eliminated, and `vec4` can be used in place of `vec5` in subsequent statements. Superword replacement is also affected by alignment, in that we detect redundant loads and stores by identifying distinct memory operations that refer to the same aligned superword, even if the addresses are not identical.

The compiler recognizes opportunities for superword replacement by determining that addresses and offsets for different memory accesses fit within the same superword, and verifies that there are no intervening kills to the memory locations. The current implementation uses *value numbering* [25] to detect such opportunities. Value numbering is a well-known compiler technique for detecting redundant computation, but it is sensitive to operand and operator ordering. To increase the success of value numbering, we first preprocess the code so that memory access operations are rewritten into a canonical form, constant folding has been applied to simplify addresses, and alignment is taken into account. As earlier stated, all memory accesses are aligned at superword boundaries, so if an unaligned address appears in a memory access, the resulting access will be aligned to the preceding superword boundary. The preprocessing performs this alignment in software so that redundant accesses will be identified by value numbering.

The current implementation of superword replacement is more restrictive than what was presented in Section 3. Value numbering operates on a basic block at a time so we cannot exploit reuse across iterations of the unrolled loop body. This is because we are performing this transformation after the SLP compiler has flattened the loop structure to gotos and labels. The dependence information used to perform the register requirement analysis cannot easily be reconstructed from such low-level code. In an implementation where SLP and SLL are more tightly integrated, it should be possible to perform superword replacement as a byproduct of the analysis in Section 3.

6.3 Packing in Superword Registers

As previously described, packing in memory is performed to align superword objects. Memory packing moves data elements from a set of locations in memory (*sources*) to a superword location (*destination*) so that the destination superword contains contiguous data, aligned to a superword boundary or to another operand. For example, in Figure 5(e), superword variables `vec0` and `vec1` are the sources and superword variable `vec2` is the destination for memory packing in lines 1-8.

Our implementation performs a transformation we call *register packing* to optimize memory packing operations. A series of memory loads and stores for scalar variables are replaced by superword operations on registers, as shown in Figure 5 (f). We identify a destination as a superword data type that is the target of a series of scalar store instructions into its fields, such as `vec2` in the example. The corresponding sources are identified by finding preceding loads of these scalar variables. If the inputs to these loads are fields of superword data types, then these superwords are the sources. In the example, `flat1` is stored into a field of `vec2`, and there is a preceding load of `flat1` that copies a field of source `vec0`. Once we find such a pattern, we verify the safety of this tranformation by guaranteeing that there are no intervening modifications or uses of either the scalar variables or destination superwords between loading the scalar variables and completion of storing into the destination. We also verify that the destination statements ultimately produce con-

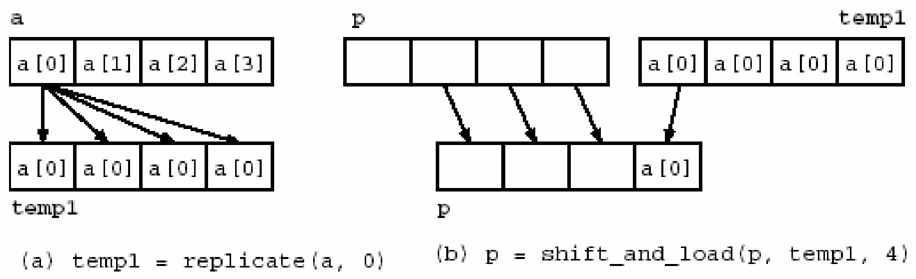


Figure 6: Operations used for packing in registers

tiguous data in the superword. We define *source* and *destination indices* as the fields in the source and destination superword variables, respectively. For example, the source index of `vec0` is 3 in line 1 of the example.

Once the compiler identifies sources and destinations, it transforms the code to replace memory accesses with operations on superword registers. The register packing transformation takes advantage of two instructions that are common in multimedia extension architectures. *Replicate* replicates one element of a source register to all elements of a temporary output register (Figure 6(a)). *Shift-and-load* takes two input registers. The first input register is a temporary, and is shifted left by the number of bytes specified by the third argument. The same number of fields is taken from the second input register, which is a temporary derived from a source superword, to fill the output temporary register (Figure 6(b)). Simply stated, we are shifting each source element into the destination superword, in order, so that the final result is a destination superword that corresponds to contiguous aligned data.

The steps of the register packing transformation are as follows.

1. We sort the destination statements in increasing order of their destination indices. We then sort the source statements to correspond to the ordering of the destination statements, so that, for example, the scalar variable associated with the first source statement is the same as the scalar variable associated with the first destination statement.

2. For each source statement, in sorted order, we generate a replicate statement whose two inputs are the source superword and the source index, and the output is a superword temporary. For example, as in Figure 5(f), we have replaced line 1 of Figure 5(e) with $temp1 = replicate(vec0, 3)$.

3. We replace each destination statement, in sorted order, with a `shift_and_load` operation. The first input is the destination superword. The second input is the temporary generated by the `replicate` of the corresponding source statement. The third argument, the shift amount, usually involves shifting by a single superword field. For the last destination field, the shift amount is the difference, in bytes, between the *sws* and the last destination field. For completely filled destination superwords, it will also be just a single field. For example, in lines 1-8 of Figure 5(e), the destination superword is completely filled, so the shift amount is always a single 4-byte field. In lines 10-13, however, only the first two fields are filled, so the shift amount of the last destination statement is a total of 12 bytes

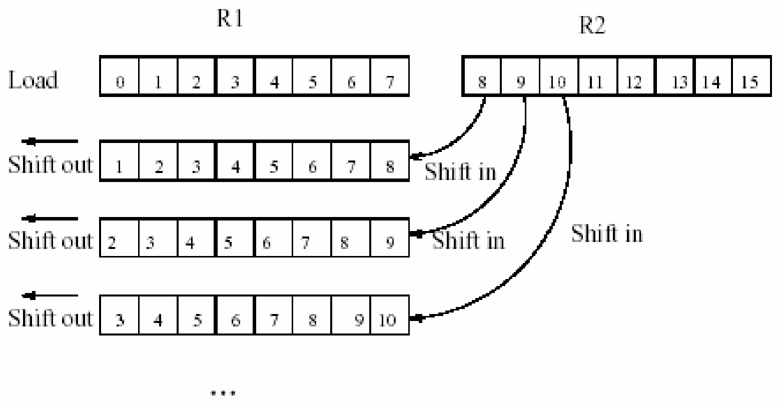


Figure 7: Shifting

4. Source statements are deleted if the scalar variables are not live beyond the corresponding destination statements.

6.4 An Example: Shifting for Partial Reuse

Spatial reuse within a superword happens when distinct loop iterations access different data in the same superword. *Partial spatial reuse* of superwords occurs when distinct loop iterations access data in consecutive superwords in memory, partially reusing the data in one or both superwords, as shown by the example in Figure 5 (a), and illustrated graphically in Figure 7. In this example, as before assuming that $sws = 4$, array reference $in[i-j]$ has partial spatial reuse in loop i. For a fixed value of i and j, the data accessed in iteration $\langle i, j\rangle$ consists of the last three words of the superword accessed in iteration $\langle i-1, j\rangle$, plus the first word of the next superword in memory. This type of reuse can be exploited by shifting the first word out of the superword, and shifting in the next word, as in Figure 7. As partially shown in Figure 5(c) and (f), only four superwords need to be loaded for the data accessed in the 64 copies of $in[i-j]$ in the loop body, after shifting is applied. Before shifting, $in[i-j]$ had to be loaded from memory (and possibly aligned) for each of the four copies of $in[i-j]$ in the loop body.

This shifting opportunity arises frequently in both signal and image processing applications, where one object is compared to a subcomponent of another object, such as the example in Figure 5(a). We detect these opportunities through the analysis described in Section 3. The optimization shown in Figure 7 falls out from the combination of unroll-and-jam, alignment operations generated by the SLP compiler, superword replacement and register packing.

7. Experimental Results

This section presents an experiment that demonstrates the dramatic performance improvements that can be derived from compiler-controlled caching in superword registers. We describe an implementation that incorporates superword register locality optimizations into an existing compiler exploiting superword-level parallelism [1]. We present a set of results on four multimedia kernels and two scientific applications, derived automatically from our implementation.

7.1 Implementation and Methodology

Figure 8 illustrates the system we have developed for this experiment, which uses the Stanford SUIF compiler as its underlying infrastructure [26]. The input to the system is a C program, which is then optimized by passes in SUIF, including our Superword Locality analysis described in Section 3, followed by the Superword-Level Parallelism (SLP) optimization passes by Larsen and Amarasinghe[1], and finally, an optimization pass that performs superword replacement as described in Section 6.2 to steer the compiler to obtain the reuse in superword registers that the SLL algorithm determined was possible.

This ordering of passes was selected primarily for implementation convenience, since we were building on the existing SLP compiler implementation. The SLP passes operate on the code at a low level, where it is difficult to reconstruct the loop structure and array access expressions. Thus, register requirement analysis and unroll-and-jam were applied prior to SLP, rather than afterward, as was suggested by the examples in Section 2. Superword replacement must follow SLP, which is the reason the components of our algorithm are performed on either side of SLP. Note that both the SLP passes and SLL employ loop unrolling, but for different reasons. The SLP compiler operates on basic blocks and unrolls the innermost loop of a loop nest to convert loop-level parallelism into basic-block parallelism. The SLL algorithm performs unroll-and-jam to expose locality in basic blocks. However, the loop that carries the most spatial locality at the superword level is often the loop that carries the most superword-level parallelism. Therefore, it is a reasonable choice to use the SLL algorithm to expose both parallelism and locality in the loop body while suppressing the unrolling originally performed by the SLP compiler.

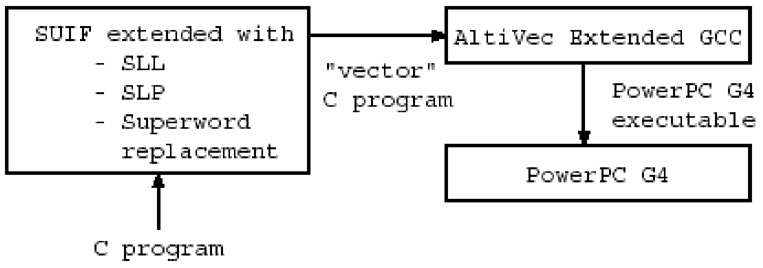


Figure 8: Implementation.

The output from the SUIF portion of the system is an optimized C program, augmented with special superword data types and operations. Currently, the resulting code is passed to a Gnu C backend, modified to support superword data types and operations for the PowerPC AltiVec instruction-set architecture extensions. Each superword operation corresponds, in most cases, to a single instruction in the AltiVec ISA. The role of the GCC backend includes replacing the vector operations with the corresponding AltiVec superword instruction, and allocating the vector data types to the superword registers. The resulting code is executed on a 533 MHz Macintosh PowerPC G4, which has a superword register file consisting of 32 128-bit registers.

7.2 Performance Measurements

We have applied the previously-described implementation to four of the five multimedia kernels and the two scientific programs from the Specfp95 benchmark suite for which execution time speedups were reported in Larsen and Amarasinghe, summarized in Table 2 [1]. As a first step, we verified

Name	Description	Data Width	Input Size
VMM	Vector-matrix multiply	32-bit float	512 elements
FIR	Finite impulse response filter	32-bit float	256 filter, 1M signal
YUV	RGB to YUV conversion	16-bit integer	32K elements
MMM	Matrix-matrix multiply	32-bit float	512 elements
swim	Shallow water model	32-bit float	Specfp95 reference input
tomcatv	Mesh generation	32-bit float	Specfp95 reference input

Table 2: Benchmark programs.

that we could reproduce their previously reported results. For purposes of comparison, we initially followed the same methodology established in Larsen and Amarasinghe [1]: (1) we used the same programs; (2) all versions of the code were compiled on the AltiVec without optimization; and, (3) baseline measurements were derived by compiling the unparallelized code for the PowerPC G4. We are using an updated implementation of SLP from what was published, as well as a faster target machine and new releases of GCC and the Linux operating system, so there are some differences in results, but they are very minor.

Larsen and Amarasinghe were unable to use optimization on the AltiVec-extended GCC backend at the time of their study, but in the intervening time, this Motorola-supplied backend has become more robust. For the results presented in this section, we modify the methodology to perform "-O3" optimizations. To understand the overall benefits of exploiting compiler-controlled caching in superword registers, we have compared the results of the full system with those obtained when SLP is used alone. For this reason, we report results where SLP is applied to the original codes and compare these results to the full system.

We show three sets of results. First, in Table 3, we show the number of vector, scalar and total memory accesses for the baseline and the full system. Our approach eliminates from 38% to 69% of the vector loads and stores in the four kernels, and over 85% in SWIM and TOMCATV. We also eliminate over 90% of the scalar loads and stores in the four kernels, and over 35% in SWIM and TOMCATV using register packing, as described in Section 6.3. When combined, more than 50% of memory accesses are eliminated.

Figure 9 shows how these reductions in instructions translates into speedups over SLP. To isolate the benefits of individual components of our system, we measure the performance of the code at several stages of the optimization process. The first bar, normalized to 1, shows the results of SLP alone. The second bar, called Unrolled+SLP, shows the results of running the first portion of the SLL algorithm, described in Section 3, which performs unroll-and-jam on the loop nest to expose opportunities for superword reuse, and following up with SLP. This bar isolates the impact of unrolling, since it is not until after SLP that this reuse is actually exploited. Also, because it is reordering the iteration space to bring reuse closer together, this version will also obtain locality benefits in the data cache. Thus, this bar provides the cache locality benefits of unroll-and-jam, which can be compared against the additional improvements from superword register locality. The third bar, Superword Replacement, provides speedup using superword replacement, as described in Section 6.2. The final bar, entitled Register Packing, shows the additional improvement due to this technique, described in Section 6.3.

Name	Mem. Acc	SLP only(baseline)	SLP+SLL+RegPack	Removed(%)
VMM	Scalar	301,989,888	0	100.00
	Vector	100,663,297	50,462,723	49.87
	Total	402,653,185	50,462,723	87.47
FIR	Scalar	1,113,940,672	82,031,104	92.64
	Vector	196,558,849	120,631,297	38.63
	Total	1,310,499,521	202,662,401	84.54
YUV	Scalar	9,400	0	100.00
	Vector	52,428,801	23,756,801	54.69
	Total	52,438,201	23,756,801	54.70
MMM	Scalar	135,267,328	525,312	99.61
	Vector	167,772,161	50,397,187	69.96
	Total	303,039,489	50,922,499	83.20
swim	Scalar	17,150,342,657	8,920,336,007	47.99
	Vector	8,495,723,139	1,200,754,698	85.87
	Total	25,646,065,796	10,121,090,705	60.54
tomcatv	Scalar	599,038,032	384,070,586	35.89
	Vector	284,631,621	9,915,592	96.51
	Total	883,669,653	393,986,178	55.41

Table 3: The number of dynamic memory accesses.

Overall, we see that in combination, applications achieve speedups between 1.3 and 3.1 over SLP alone, with an average of 2.2X. Consideration of TOMCATV and SWIM shows that both programs have little temporal reuse, although there is a small amount of spatial reuse that is exploited with our approach, particularly in TOMCATV. We are obtaining a locality benefit due to unroll-and-jam. We also observe additional SLP due to index set splitting, motivated by the need to create a steady-state loop where the data is aligned to a superword boundary. The four other programs show a significant improvement from superword replacement. For VMM, MMM and FIR, there are also huge gains due to register packing.

In Figure 10, we further explore the relationship between superword replacement and register packing. The first bar, which is normalized to 1, shows the Unrolled+SLP version (the second bar in the previous figure). The second bar is the Unrolled+SLP+SWR result from the previous figure, but this time it is normalized to Unrolled+SLP. To show the isolated benefit of register packing without superword replacement, we applied register packing to the Unroll+SLP version, obtaining the results shown in the third bar (Unroll+SLP+RP) of Figure 10. The final bar is the result of applying all of the optimizations. As might be expected from the previous figure, register packing, either in isolation or in conjunction with superword replacement, does not impact the results for YUV, swim or tomcatv. We see that for VMM and MMM, register packing yields about the same improvement when applied prior to superword replacement than afterward. Especially interesting are the results for FIR, because the speedup is much larger when superword replacement and register packing are applied together than when they are applied separately. On further investigation, we found that the Unroll+SLP+RP version suffered from register spilling. Superword replacement removes the majority of the superword variables used in the Unroll+SLP+RP version, which in turn reduces register pressure. This result is consistent with the goal of the algorithm in Section 3. We selected unroll factors based on the assumption that superword replacement would be performed.

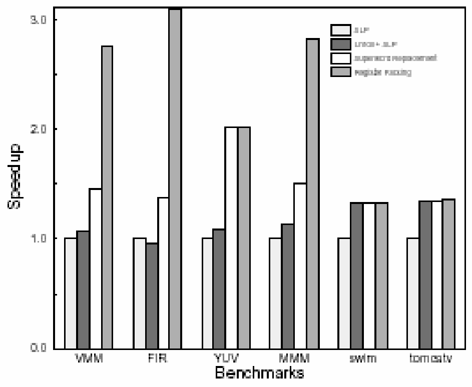


Figure 9: Speedups over SLP alone.

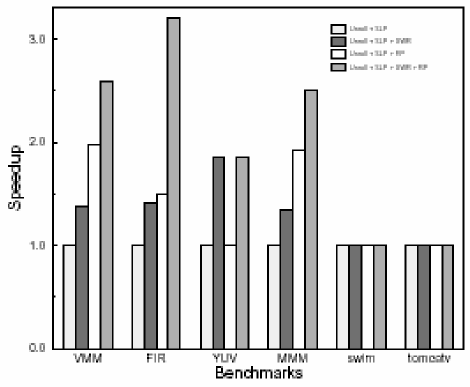


Figure 10: Impact of register packing.

Without superword replacement, there is register pressure after unrolling, and this is magnified by register packing because it introduces additional superword variables.

In summary, the SLL techniques presented in this paper dramatically reduce the number of memory accesses and yield significant performance improvements across these 6 programs. Thus, this paper has demonstrated the value of exploiting locality in superword registers in architectures that support superword-level parallelism such as the AltiVec.

8. Related Research

For well over a decade, a significant body of research has been devoted to code transformations to improve cache locality, most of it targeting loop nests with regular data access patterns [27, 28, 29, 30]. Loop optimizations for improving data locality, such as tiling, interchanging and skewing, focus on reducing cache capacity misses. Of particular relevance to this paper are approaches to tiling for cache to exploit temporal and spatial reuse; the bulk of this work examines how to select tile sizes that eliminate both capacity misses and conflict misses, tuned to the problem and cache sizes [31, 11, 12, 13, 14, 15, 16, 17, 18, 32]. The key difference between our work and that of tiling for caches is that interference is not an issue in registers. Therefore, models that consider conflict misses are not appropriate. Further, our code generation strategy must explicitly manage reuse in registers.

There has been much less attention paid to tiling and other code transformations to exploit reuse in registers, where conflict misses do not occur, but registers must be explicitly named and managed. A few approaches examine mapping array variables to scalar registers [18, 33, 20]. Most closely related to ours is the work by Carr and Kennedy, which uses scalar replacement and unroll-and-jam to exploit scalar register reuse [19]. Like our approach, in deriving the unroll factors, they use a model to count the number of registers required for a potential unrolling to avoid register pressure, and they replace array accesses, which would result in memory accesses, with accesses to temporaries that will be put in registers by the backend compiler. Their search for an unroll factor is constrained by register pressure and another metric called *balance* that matches memory access time to floating point computation time. Our approach is distinguished from all these others in that the model for register requirements must take spatial locality into account, we replace array accesses with superwords rather than scalars, and we also consider the optimizations in light of superword parallelism.

There are several recent compilation systems developed for superword-level parallelism [1, 7, 8, 9, 10]. Most, including also commercial compilers [34, 35], are based on vectorization technology [7, 9]. In contrast, Larsen and Amarasinghe devised a superword-level parallelization system for multimedia extensions [1]. They point out that there are many differences between the multimedia extension architectures and vector architectures, such as short vectors, ease of mixing with scalar instructions, and need for alignment of memory accesses [36]. They argue that their algorithm for finding superword-level parallelism from a basic block instead of a loop nest is much more effective than using vectorization-based techniques. None of the above approaches exploit reuse in the superword register file.

9. Conclusion

This paper presents an algorithm for compiler-controlled caching in superword register files. The algorithm is applicable to multimedia extensions such as Intel's SSE, PowerPC's AltiVec, and also to Processor-in-memory (PIM) architectures with support for superword operations.

We implemented our approach in an existing compiler targeting superword-level parallelism. We presented experimental results, derived automatically, comparing the performance of six benchmarks/multimedia kernels optimized for parallelism only, using SLP, and optimized for both parallelism and locality. Our results show speedups ranging from 1.3 to 3.1X, and an average of 2.2X, on the 6 programs as compared to using SLP alone, and most memory accesses are removed.

The approach taken here that separates optimizations for SLL and SLP is convenient for implementation purposes, since we are building upon the work of others. Further, as there are now a few other compilers that exploit superword-level parallelism [7, 8, 9, 10], the same can be used to extend these existing systems to incorporate compiler-controlled caching in superword registers. Ideally, however, an optimizer that integrates the superword parallelism and locality techniques could be even more effective. For example, in a combined algorithm, selection of which loops to parallelize could also take superword-level locality into account. A combined algorithm is the subject of future work.

10. Acknowledgments

The authors wish to thank Samuel Larsen and Saman Amarasinghe for providing their SLP implementation. We wish to especially thank Samuel Larsen for his tremendous support. We also wish to thank all the students in our research group for providing the underlying infrastructure for this work, including Yoon-Ju Lee, Byoungro So and Rommel Dongre.

References

[1] S. Larsen and S. Amarasinghe, "Exploiting superword level parallelism with multimedia instruction sets," in *Conference on Programming Language Design and Implementation*, (Vancouver, BC Canada), pp. 145–156, June 2000.

[2] R. Lee, "Subword parallelism with max2," *IEEE Micro*, vol. 16, pp. 51–59, Aug. 1996.

[3] N. Bowman, N. Cardwell, C. Kozyrakis, C. Romer, and H. Wang, "Evaluation of existing architectures in IRAM systems." In First Workshop on Mixing Logic and DRAM: Chips that Compute and Remember, June 1997.

[4] M. Hall, P. Kogge, J. Koller, P. Diniz, J. Chame, J. Draper, J. LaCoss, J. Granacki, A. Srivastava, W. Athas, J. Brockman, V. Freeh, J. Park, and J. Shin, "Mapping irregular applications to DIVA, a PIM-based data-intensive architecture," in *ACM International Conference on Supercomputing*, Nov. 1999.

[5] J. Brockman, P. Kogge, V. Freeh, S. Kuntz, and T. Sterling, "Microservers: A new memory semantics for massively parallel computing," in *ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[6] D. Elliott, M. Snelgrove, and M. Stumm, "Computational RAM: a memory-SIMD hybrid and its application to DSP," in *IEEE 1992 Custom Integrated Circuit Conference*, pp. 30.6.1 – 30.6.4, 1992.

[7] N. Sreraman and R. Govindarajan, "A vectorizing compiler for multimedia extensions," *International Journal of Parallel Programming*, 2000.

[8] G. Cheong and M. S. Lam, "An optimizer for multimedia instruction sets," in *The Second SUIF Compiler Workshop*, (Stanford University, USA), Aug. 1997.

[9] D. J. DeVries, "A vectorizing suif compiler: Implementation and performance," Master's thesis, University of Toronto, 1997.

[10] K. Asanovic and J. Beck, "T0 engineering data." UC Berkeley CS technical report UCB/CSD-97-930.

[11] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *The SIGPLAN '95 Conference on Programming Language Design and Implementation*, (La Jolla, CA), June 1995.

[12] K. Esseghir, "Improving data locality for caches," Master's thesis, Dept. of Computer Science, Rice University, September 1993.

[13] C. Fricker, O. Temam, and W. Jalby, "Influence of cross-interferences on blocked loops: A case study with matrix-vector multiply," *TOPLAS*, vol. 17, pp. 561–575, July 1995.

[14] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: An analytical representation of cache misses," in *Proceedings of the 1997 ACM International Conference on Supercomputing*, (Vienna, Austria), July 1997.

[15] S. Ghosh, M. Martonosi, and S. Malik, "Precise miss analysis for program transformations with caches of arbitrary associativity," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 228–239, October 1998.

[16] M. S. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimization of blocked algorithms," *ACM SIGPLAN Notices*, vol. 26, no. 4, pp. 63–74, 1991.

[17] O. Temam, E. Granston, and W. Jalby, "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts," in *ACM International Conference on Supercomputing*, (Portland, OR), Nov. 1993.

[18] M. E. Wolf, *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, 1992.

[19] S. Carr and K. Kennedy, "Improving the ratio of memory operations to floating-point operations in loops," *ACM Transactions on Programming Languages and Systems*, vol. 15(3), pp. 400–462, July 1994.

[20] A. F. M. Jimenez, J.M. Llaberia and E. Morancho, "Index set splitting to exploit data locality at the register level," Tech. Rep. UPC-DAC-1996-49, Universitat politecnica de Catalunya, 1996.

[21] J. Shin, J. Chame, and M. W. Hall, "Compiler-controlled caching in superword register files for multimedia extension," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, (Charlottesville, Virginia), September 2002.

[22] P. Ranganathan, S. Adve, and N. Jouppi, "Performance of image and video processing with general-purpose processors and media ISA extensions," in *International Symposium on Computer Architecture*, May 1999.

[23] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.

[24] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in fpga-based systems," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, (Berlin, Germany), June 2002.

[25] S. S. Muchnick, *Advanced Compiler Design and Implementation*. 340 Pine St. Sixth Floor, San Francisco, CA 94104-3205, USA: Morgan Kaufmann, 1997.

[26] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, and M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer*, vol. 29, pp. 84–89, Dec. 1996.

[27] J. Ferrante, V. Sarkar, and W. Thrash, "On estimating and enhancing cache effectiveness," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, (Santa Clara, California), pp. 328–343, August 1991.

[28] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, (San Jose, California), pp. 252–262, October 1994.

[29] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, (Toronto), pp. 30–44, June 1991.

[30] M. J. Wolfe, "More iteration space tiling," in *Proceedings of Supercomputing '89*, (Reno, Nevada), pp. 655–664, November 1989.

[31] J. Chame and S. Moon, "A tile selection algorithm for data locality and cache interference," in *International Conference on Supercomputing*, pp. 492–499, 1999.

[32] G. Rivera and C. Tseng, "A comparison of compiler tiling algorithms," in *the 8th International Conference on Compiler Construction (CC'99), Amsterdam, The Netherlands*, Mar. 1999.

[33] S. Carr and K. Kennedy, "Scalar replacement in the presence of conditional control flow," *Software—Practice and Experience*, vol. 24, no. 1, pp. 51–77, 1994.

[34] Veridian, *VAST/AltiVec Features*, June 2001. http://www.psrv.com/altivec_feat.html.

[35] Metrowerks, *CodeWarrior version 7.0 data sheet*, 2001. http://www.metrowerks.com/pdf/mac7.pdf.

[36] S. Larsen, E. Witchel, and S. Amarasinghe, "Increasing and detecting memory address congruence," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, (Charlottesville, Virginia), September 2002.