AFRL-IF-WP-TR-2002-1536

NIMBLE COMPILER ENVIRONMENT FOR AGILE HARDWARE

Volume 1

Dr. Don MacMillen

Synopsys Inc.
Advanced Technology Group
700 East Middlefield Road
Mountain View, CA 94043-4033

OCTOBER 2001

FINAL REPORT FOR 01 APRIL 1998 – 24 JULY 2001

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7334

Report Documentation Page

Report Date 01SEP2001	**Report Type** N/A	**Dates Covered (from... to)** 17OCT1997 - 28SEP2001

Title and Subtitle Nimble Compiler Environment for Agile Hardware, Volume 1	**Contract Number**
	Grant Number
	Program Element Number
Author(s) MacMillen, Don	**Project Number**
	Task Number
	Work Unit Number
Performing Organization Name(s) and Address(es) Synopsys, Inc. Advanced Technology Group 700 East Middlefield Road Mountain View, CA 94043-4033	**Performing Organization Report Number**
Sponsoring/Monitoring Agency Name(s) and Address(es) Information Directorate Air Force Research Laboratory Air Force Materiel Command Wright-Patterson AFB, OH 45433-7334	**Sponsor/Monitor's Acronym(s)**
	Sponsor/Monitor's Report Number(s)
Distribution/Availability Statement Approved for public release, distribution unlimited	
Supplementary Notes The original document contains color images.	
Abstract	
Subject Terms	
Report Classification unclassified	**Classification of this page** unclassified
Classification of Abstract unclassified	**Limitation of Abstract** SAR
Number of Pages 232	

NOTICE

USING GOVERNMENT DRAWINGS, SPECIFICATIONS, OR OTHER DATA INCLUDED IN THIS DOCUMENT FOR ANY PURPOSE OTHER THAN GOVERNMENT PROCUREMENT DOES NOT IN ANY WAY OBLIGATE THE US GOVERNMENT. THE FACT THAT THE GOVERNMENT FORMULATED OR SUPPLIED THE DRAWINGS, SPECIFICATIONS, OR OTHER DATA DOES NOT LICENSE THE HOLDER OR ANY OTHER PERSON OR CORPORATION; OR CONVEY ANY RIGHTS OR PERMISSION TO MANUFACTURE, USE, OR SELL ANY PATENTED INVENTION THAT MAY RELATE TO THEM.

THIS REPORT IS RELEASABLE TO THE NATIONAL TECHNICAL INFORMATION SERVICE (NTIS). AT NTIS, IT WILL BE AVAILABLE TO THE GENERAL PUBLIC, INCLUDING FOREIGN NATIONS.

THIS TECHNICAL REPORT HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION.

KERRY L. HILL
Project Engineer
Embedded Info Sys Engineering Branch
Information Technology Division

Alfred J. Scarpelli
ALFRED J. SCARPELLI
Team Leader
Embedded Info Sys Engineering Branch
Information Technology Division

James S. Williamson
JAMES S. WILLIAMSON, Chief
Embedded Info Sys Engineering Branch
Information Technology Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document requires its return.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS**.

1. REPORT DATE ***(DD-MM-YY)***
October 2001

2. REPORT TYPE
Final

3. DATES COVERED ***(From - To)***
04/01/1998 – 07/24/2001

4. TITLE AND SUBTITLE
NIMBLE COMPILER ENVIRONMENT FOR AGILE HARDWARE
Volume 1

5a. CONTRACT NUMBER
F33615-98-2-1317

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Dr. Don MacMillen

5d. PROJECT NUMBER
ARPI

5e. TASK NUMBER
FT

5f. WORK UNIT NUMBER
02

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Synopsys Inc.
Advanced Technology Group
700 East Middlefield Road
Mountain View, CA 94043-4033

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

INFORMATION DIRECTORATE
AIR FORCE RESEARCH LABORATORY
AIR FORCE MATERIEL COMMAND
WPAFB, OH 45433-7334

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
INFORMATION TECHNOLOGY OFFICE
3701 NORTH FAIRFAX DRIVE
ARLINGTON, VA 22209-2308

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/IFTA

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2002-1536

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
This report contains copyright material. Report contains color.

14. ABSTRACT ***(Maximum 200 Words)***

The utilization of a tightly coupled general purpose processor (GPP) and reconfigurable logic array (RLA) has demonstrated appreciable acceleration in some compute intensive applications. Such systems have been very difficult to program though and thus have not been exploited for their benefits. The problem is the lack of an appropriate design environment for system engineers like those typically found in digital signal processing (DSP) embedded system development. The Nimble Complier research project aims to develop a retargetable design compiler for these adaptable architectures that will exploit the performance gains and hardware whenever it is needed as opposed to being predesigned into hardware. Field updates or modal changes in function are simple. Embedded applications across a wide spectrum of programming language styles will be targeted for support. The resultant environment should accelerate adoption of these computing platforms by making the systems easier to develop and more robust to multiple standards. The Government benefits from this effort by enabling the compute intensive portions of military electronic systems to be smaller, cheaper, and field upgradeable.

15. SUBJECT TERMS
ACS (Adaptive Computing Systems), reconfigurable computing, field-programmable gate arrays, FPGAs, system design, co-design, programming environments, partitioning, retargetable C complier, digital signal processing, DSP

16. SECURITY CLASSIFICATION OF:

a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	**17. LIMITATION OF ABSTRACT:**	**18. NUMBER OF PAGES**
Unclassified	Unclassified	Unclassified	SAR	236

19a. NAME OF RESPONSIBLE PERSON (Monitor)
Kerry L. Hill

19b. TELEPHONE NUMBER *(Include Area Code)*
(937) 255-6548 x3604

Table of Contents

1 Overview of the Nimble Compiler Project

The *Nimble Compiler for Agile Hardware* is designed to automatically compile C language applications to embedded, agile hardware architectures. Agile hardware, or Agileware for short, is defined simply as a mixed programmable RISC processor and a reconfigurable datapath co-processor (or attached processor). A template of the Agileware architectures is shown in Figure 1. The goal of Agileware and Nimble Compiler is to accelerate an applications code over what would be achievable using only the processor itself. This is achieved by getting more **operator parallelism** and **memory bandwidth** with the configurable datapath than that normally achievable with any programmable processor.

In this section, we first provide an overview of the Agileware architecture, which is the target of the Nimble Compiler. We then describe the Nimble Compiler flow from a high level point of view.

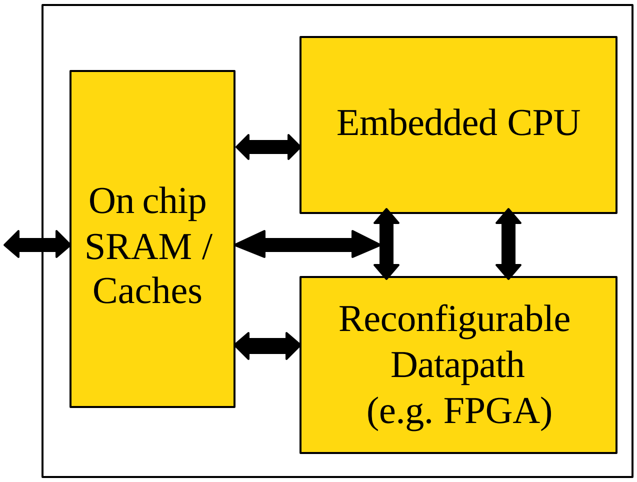


Figure 1. Agileware top-level architecture.

1.1 Introduction of Agileware Architectures

The basic architecture model defined by the term Agileware is depicted in Figure 1. Key to this model, unlike most other reconfigurable computing architectures, is the availability of the full memory hierarchy to the configurable datapath (DP) coprocessor. This takes the co-processor out of the realm of a slave, register based accelerator into a class of a full-scale processor. Additionally, option direct I/O can be handled for data reduction or generation that would not normally be possible with a programmable solution. Finally, optional local memory either inside or outside the datapath can be utilized by the compiler to assist in reducing main memory traffic. Nominally, the best performance and cost can be gained by combining all of these elements onto a single chip.

Currently, all the configurable datapath targets that the Nimble Compiler uses are defined out of fine-grain, Xilinx LUT-like FPGA structures. An off-the-shelf commercial FPGA is not designed to be a good datapath unit. This is due to the limited special long-line routing for memory access, the lack of courser grained, higher-performance operators, and the real lack of support for fast loading, compact (re)configuration bit-streams. Features like configurable operators at specific locations and configurable routing are what can still be useful and exploited. Better-designed, dedicated configurable datapaths are in development by many organizations that will overcome the limitations of FPGA's. In lieu of having these special designs, the FPGA implementation provides a test-bed to experiment with new architectural features before committing to a more custom implementation and to show that real hardware can be targeted and used to accelerate codes; even with all the limitations.

The biggest benefit of an Agileware unit is that the co-processor can be programmed to look like a VLIW (heterogeneous operator parallelism), Vector parallel (SIMD or data parallel), or fine-grain, systolic processor. Any of these forms can make use of a memory hierarchy that is also configurable to include options of cache architectures, vector queues, and localized buffers. The configurable datapath is register rich and uses direct connections between operators and registers, thus offering somewhat unlimited registers with parallel access by every operator.

1.2 The Nimble Compiler Approach

Nimble operates by looking for **inner loops** that dominate the execution of the application. Such loops, termed hardware kernels if realizable in the datapath, are retargeted to the datapath coprocessor as a dynamically loadable configuration. Such targeting can be either explicitly defined by the user or automatically detected by the compiler. In Phase 0, the compiler was dumb and attempted to put every loop it could in hardware by default, whether efficient or not. Phase 1 introduces profile-based automatic kernel selection that attempts to put only those loops in hardware that are guaranteed to provide a benefit under the conditions presented.

For embedded DSP style code, our studies reconfirm that 90+% of the code execution time is spent in a small (less than 20) number of inner loops. Thus, the user and compiler can focus on optimizing a few areas of code where the most execution time is spent. Key to this is the efficiency that can now be gained in the compilation process. With only 20 or less loops that generally tend to be 20 to 50 source lines of code, the compiler can apply normally expensive, non-polynomial algorithms in the optimization process.

So the goal of the user is to find loops implementable in the configurable datapath that will accelerate the code. Figure 2 depicts the static loop-procedure hierarchy of an example application (a simple wavelet image compression program). The loops are depicted as ovals and the procedures as rectangles. System calls are depicted as filled-in rectangles – signifying that further analysis and hierarchy is not available. The arcs represent the context that the loop or call occurs in with their usually being a single root that represents the main procedure in C. The nodes are scaled according to their relative code execution time in the original software application for a given data set. (Note: this graph is from one of the analysis tools that will be described more fully later on.) Loops that (after optimization, analysis and transformations) are feasible to accelerate in the datapath are depicted with heavier weight ovals and termed selected kernels. So the user, via the compiler, is trying to manipulate the code and compiler options to get as many loops selected as possible to accelerate the application.

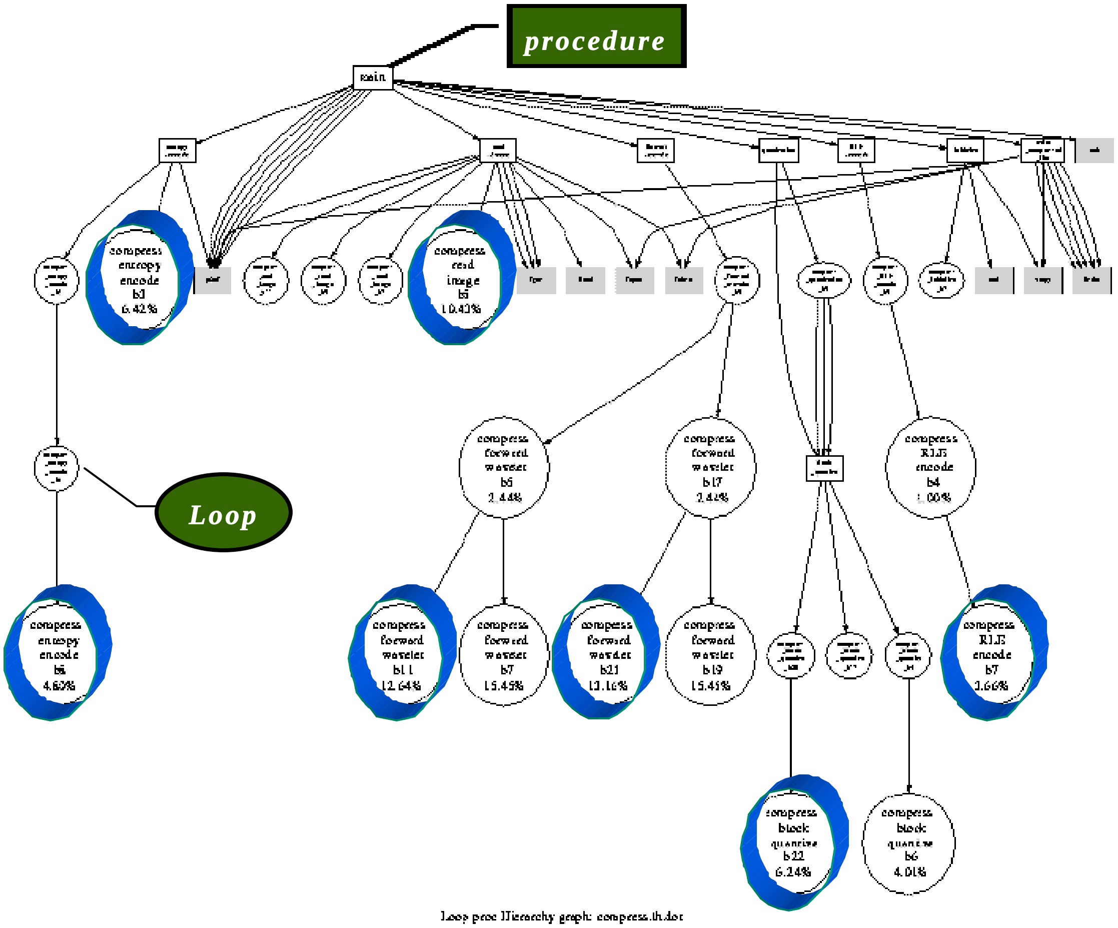

procedure
main
Loop
compress entropy encode b3 6.42%
compress read image b5 10.40%
compress forward wavelet b5 2.44%
compress forward wavelet b17 2.44%
compress RLE encode b4 1.00%
compress entropy encode b6 4.60%
compress forward wavelet b11 12.64%
compress forward wavelet b7 15.45%
compress forward wavelet b21 13.16%
compress forward wavelet b19 15.45%
compress RLE encode b7 1.66%
compress block quantize b22 6.24%
compress block quantize b6 4.01%
Loop proc Hierarchy graph: compress.lh.dot

Figure 2: **Example loop-procedure hierarchy graph with loop selection depicted.**

Unlike traditional hardware / software codesign techniques, this approach is fairly deterministic and thus achieves close to optimal results for most applications. The platform, once defined to the compiler, is fixed in the amount and performance of resources. The number of operators, number of registers, and amount of interconnect is fixed. The memory bandwidth, although optimizable with configurable elements, is also fixed. Therefore, exact determination can be made instead of gross guess-timates of hardware execution times, configuration times, and tradeoff of space for performance.

Extracting maximum **instruction level parallelism (ILP)** and deep pipelining is the goal once a loop kernel is identified. Today's quick prototype compiler performs only the most basic of transformations to get a kernel working on the co-processor. Automatic and significant operator, statement, and basic block parallelism and pipelining should eventually be achievable. This will be through the application of VLIW and Vector Parallel style data dependency analysis, profile analysis and optimization techniques on the code

Nimble does not define a strong style or programming model; not more than enforced on general application code now. Instead, it works on existing, fully ISO/ANSI compliant C standard files and tries to extract ILP and memory structures from the general specification. As in any specification, good style for optimally targeting the specific architecture is always helpful. This is especially true in trying to overcome near-term compiler limitations. But good style for this compiler is more like the intended mathematical specification of the algorithm versus an optimum, engineered and quick executing software implementation.

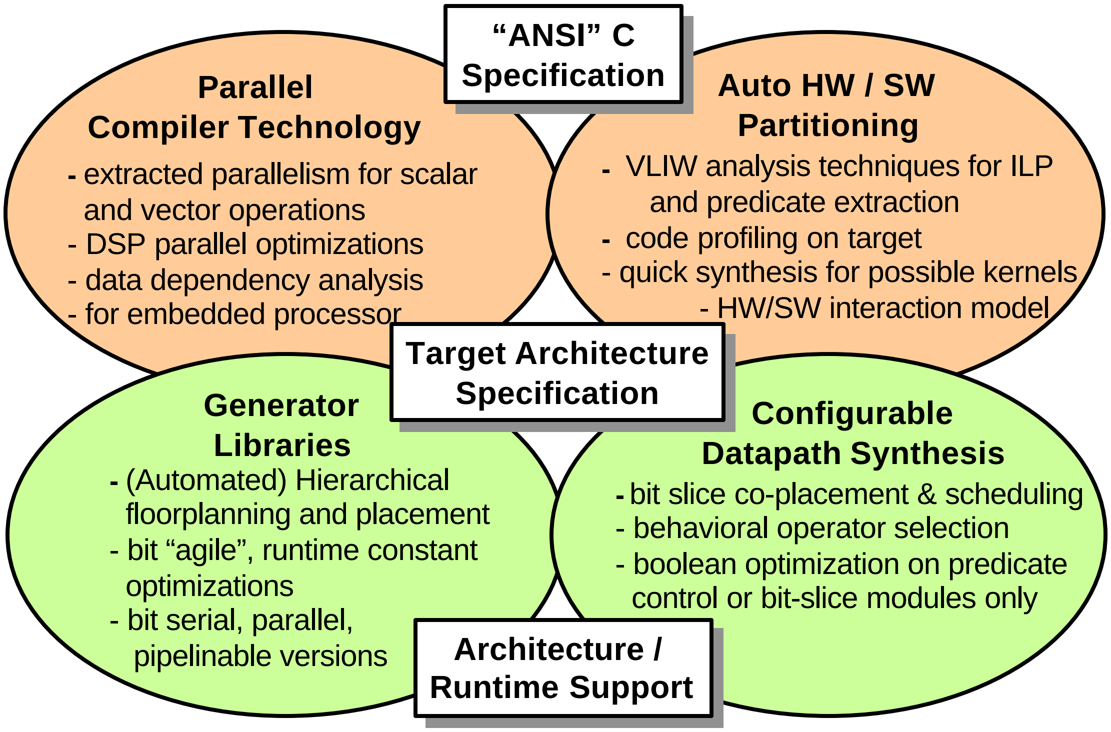

"ANSI" C Specification
Parallel Compiler Technology
- extracted parallelism for scalar and vector operations
- DSP parallel optimizations
- data dependency analysis
- for embedded processor
Auto HW / SW Partitioning
- VLIW analysis techniques for ILP and predicate extraction
- code profiling on target
- quick synthesis for possible kernels
- HW/SW interaction model
Target Architecture Specification
Generator Libraries
- (Automated) Hierarchical floorplanning and placement
- bit "agile", runtime constant optimizations
- bit serial, parallel, pipelinable versions
Configurable Datapath Synthesis
- bit slice co-placement & scheduling
- behavioral operator selection
- boolean optimization on predicate control or bit-slice modules only
Architecture / Runtime Support

Figure 3. Nimble Compiler technology.

To achieve the automatic compilation of an application written in C on the mixed hardware/software Agileware platforms, the Nimble Compile deploys an array of technologies from various research fields that did not traditionally interact with each other. Figure 3 depicts the different fields related to the Nimble effort, including:

- **Compiler technology:** Traditional compiler technology has been mainly focusing on the acceleration of software. In the Nimble project, we apply compiler technology not only to accelerate software, but also to help improve hardware implementations. At the front end of the compiler, Nimble applies compiler optimization techniques to extract maximum amount of instruction-level parallelism from the source application. Data dependence analysis is also performed. This step will facilitate the datapath synthesis process later on to achieve maximum acceleration of the kernels running in hardware.
- **Automatic hardware/software partitioning**: hardware/software partitioning is key in deciding what to execute in software and what to execute in the reconfigurable datapath to obtain best performance at the application level. This step bridges the front end and the back end of the Nimble Compiler. Hardware/software partitioning has been studied by the hardware/software co-design community, mainly targeting programmable CPUs coupled with ASICs. For the Nimble project, we have introduced novel automatic hardware/software partitioning algorithms targeting specifically mixed CPU and reconfigurable hardware. It

automatically evaluates the tradeoffs between software execution, hardware execution, and reconfiguration and selects the best kernels to be implemented in the hardware.

- **Generator libraries**: Macro generator libraries are developed to facilitate the datapath synthesis process. Generator libraries are provided to support the full set of ANSI integer operations. Additional domain-specific generations are provided to facilitate acceleration of cryptographic, multimedia, as well as general computational functions.
- **Configurable datapath synthesis**: Configurable datapath synthesis is required to map selected hardware kernels on to the reconfigurable datapath (also referred to as FPGA). The goal of the datapath compiler is to efficiently map the unscheduled data flow graph description of the datapath to an FPGA. The involved steps include performing scheduling, technology mapping, module generation, and datapath floorplanning. To leverage existing datapath compiler technology and to quickly provide a prototype solution, the U.C. Berkeley work on the Garp architecture, in particular, their datapath compiler tool Gama has been selected as the starting framework. It is modified for both the Xilinx 4K series and Virtex series FPGA's.

1.3 Top-Level Flow of the Nimble Compiler

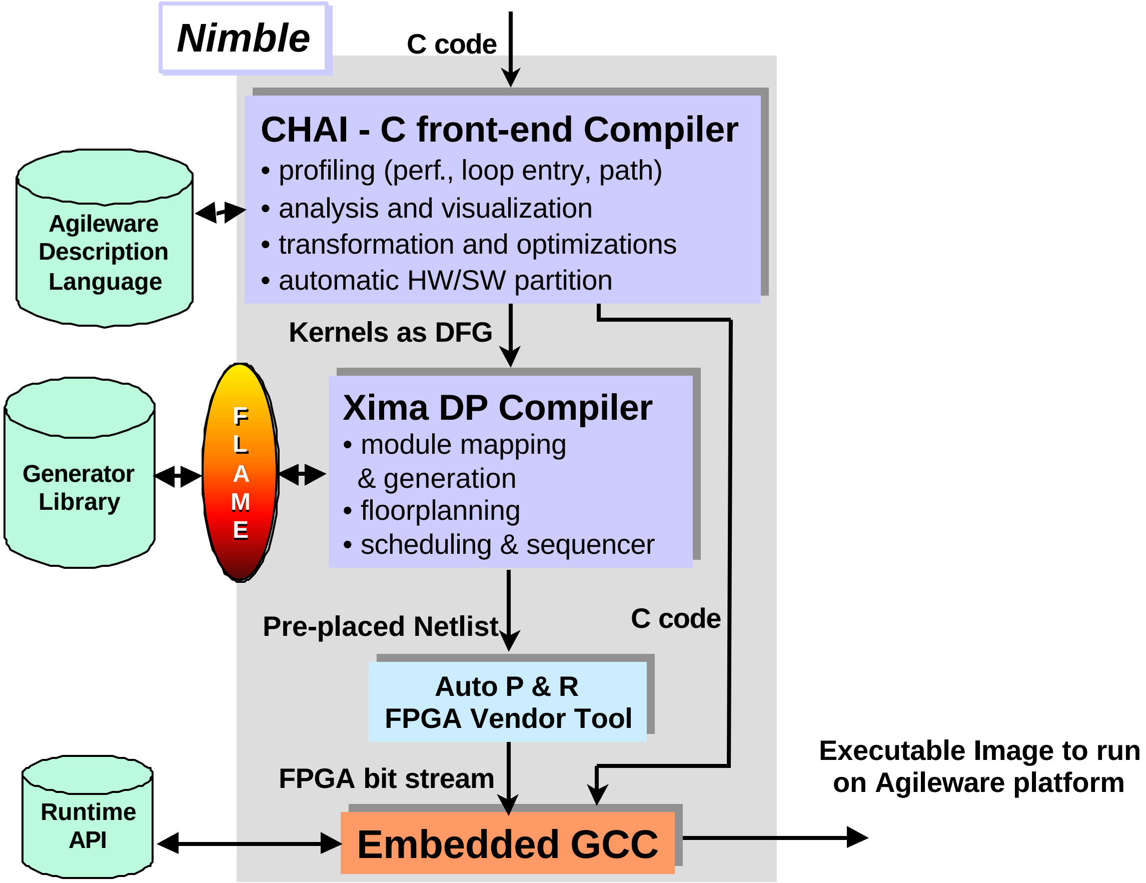

Nimble
C code
CHAI - C front-end Compiler
• profiling (perf., loop entry, path)
• analysis and visualization
• transformation and optimizations
• automatic HW/SW partition
Agileware Description Language
Kernels as DFG
Xima DP Compiler
• module mapping & generation
• floorplanning
• scheduling & sequencer
Generator Library
FLAME
Pre-placed Netlist
C code
Auto P & R FPGA Vendor Tool
FPGA bit stream
Runtime API
Embedded GCC
Executable Image to run on Agileware platform

Figure 4. Nimble Compile top-level flow.

Given the approach and technology described in the previous section, the high level flow of the Nimble Compiler is fairly apparent and is depicted in Figure 4. The compiler takes in any ANSI/ISO standard C code compatible for the target embedded processor environment and eventually puts out an executable image. For convenience, the compiler wraps up the execution of the configurable datapath APR tools along with the embedded processor C compiler environment. The compiler itself consists of two main pieces: the C Hardware and Instruction (CHAI) Front-end compiler and the Xima datapath compiler back-end.

CHAI encompasses a C compiler, instrumented profiling tool, analysis and visualization tools, optimizations, and the automatic HW/SW partitioner termed Kernel Selection (or KS). Xima consists of a module mapping and generation, floorplaning and scheduling tool. The output of Xima is a fully placed netlist implementing the original loop functionality. Once this is passed through final routing and timing verification, the configuration is linked and loaded into the data space of the C code as a compressed, initialized binary array. The original C code, optimized and altered to reflect the code in kernels, is put back out as C code by Chai for final compilation into an executable by the embedded processor C compiler. Features of the target architecture are defined in libraries that retarget the compiler to a particular implementation. These libraries consist of software, runtime libraries, datapath generator libraries, and the Agileware Description Language (ADL).

Subprogram and task level parallelism is not inherent in the C language standard. Therefore, it is not automatically detected, extracted or otherwise looked for in this compiler. Today's compiler can be used to analyze and accelerate single tasks only. Other researchers' work on task level parallelism specification and extraction could be incorporated, in general, as the topic is somewhat orthogonal to the ILP extraction looked at here. The areas in the Nimble Compiler that would be most affected by integrating the current ILP model with a new multi-tasking specification would be in estimating software task performance, determining optimum reconfigurations for multiple kernels, and understanding memory access patterns and performance issues.

1.4 What Is Included in This Report and Supporting Documents

The rest of this report along with the supporting documents explains further technical contributions and details regarding the Nimble Compiler. In our initial statement of work, the Nimble project is divided into the following six major tasks:

1. Tool architecture and environment.
2. Nimble Compiler front end and software code generator.
3. Performance and capacity driven partitioner.
4. Datapath compiler back-end.
5. Functional Generator Libraries – research and prototyping.
6. Benchmarking and demonstration.

Section 2 is the key section of this report: it revolves around the above six tasks and addresses the technical aspects of each task. Many of the technical areas are covered in separate reports in the supporting documents. For these areas, we will only include a brief description and refer the readers to the correspondent supporting documents.

Section 3 draws the final conclusions of the Nimble project..

The supporting documents are included as appendices to this report (Appendices A-M). Table 1 lists the information of each supporting document: the title, the appendix index, and the task (or tasks) in original Statement of Work it corresponds to.

Note that Appendices A—J are included in Vol I of the Final Report, which is available for public release. Appendices K—M are inlcuded in Vol II, which requires government only distribution through DTIC.

Table 1. List of supporting documents included as appendices.

Appendix	Report full title	Task number in SOW
A	Overview of the ACEV Environment	Task 1
B	A Comprehensive Prototyping-Platform for Hardware-Software Co-Design	Task 1
C	Porting RTEMS to the ACE Hardware Platform	Task 1
D	Profiling Tools and Result Viewers for the Nimble Compiler Project	Tasks 2 & 3
E	Efficient Pipelining of Nested Loops: Unroll-and-Squash, an Innovative compiler Optimization Technique used in the Nimble Compiler	Task 2.
F	The Hardware-Software Partitioning Approach of the Nimble Compiler	Task 3
G	Xima - The Nimble Datapath Compiler	Task 4
H	Domain Generator Tutorial for the Nimble Compiler Project	Task 5
I	Specification for FPGA Macro Generators for the Nimble Compiler Project	Task 5
J	Final Technical Deliverables Status	Tasks 1---6
Vol II K	Agileware Description Language for the Nimble Compiler Environment for Agile Hardware	Task 1
Vol II L	Nimble Compiler Language Manual and Style Guide	Tasks 1 & 2
Vol II M	Final Benchmark Report for The Nimble Compiler Environment for Agile Hardware	Task 6

2 Nimble Technologies

Through the Nimble Project, Synopsys Inc. and its partners aim to demonstrate a retargetable design environment for Adaptive Computing Systems comprised of a processor tightly coupled with a configurable logic array (referred to as Agile hardware, or Agileware in short). In our initial statement of work, the Nimble project is divided into the following six major development areas:

1. Tool architecture and environment
2. Nimble Compiler Front-End and SW Code Generator
3. Performance and Capacity Drive Partitioner
4. Datapath Compiler back-end
5. Functional Generator Library Research and Prototyping
6. Benchmarking and demonstration.

In this section, we will discuss our technical accomplishments in each of the above areas.

2.1 Task 1: Tool Architecture and Environment

We have finished the development of the complete Nimble Compiler Environment and targeted it to multiple Agileware platforms including real hardware. Refer to Section 1 for an overview of the Nimble Compiler Environment, the compilation flow, and the technological areas. For Task 1, we have the following technical accomplishments:

- **Nimble infrastructure**: Developed the complete Nimble environment, including the fully functional, automatic Nimble Compiler, and the target platforms. (ACEV and ACEII environment). The overview of Nimble Compiler environments was discussed in Section 1. The subsequent subsections in Section 2 will provide more insight into the technical aspects of the Nimble compiler.

- **Nimble target platforms**: Section 2 focuses on the tool aspect of the Nimble compiler. We have included two supporting documents **"Overview of the ACEV Environment**" **(Appendix A)** and **"A comprehensive prototyping platform for hardware-software co-design" (Appendix B)** that address one of the Nimble target platforms – ACEV. The ACEV platform uses a real-time operating system (RTOS) named RTEMS. As suggested by it title, **"Porting RTEMS to the ACE hardware platform" (Appendix C)** documents how we have ported the RTEMS OS to the ACEV and ACEII platforms.

- Information regarding the Garp architecture can be found in the publication by Callahan et.al [3].

- Function Generator Libraries: Function generator libraries have been developed to facilitate the datapath synthesis process. Generator libraries are provided to support the full set of ANSI integer operations. Addition domain-specific generations are provided to facilitate acceleration of cryptographic, multimedia, as well as general computational functions. The specification of the base generators and the domain specific generators is included in document **"Specification for FPGA macro generators for the Nimble Compiler Project"**

(Appendix I). The Nimble compiler supports the computing models with generator libraries and provides vendor neutral APIs to allow easy retageting of the generators. We have dedicated a separate task to generators libraries research and prototyping (Task 5). Section 2.5 will explain this area in greater detail.

- **Nimble language guide**: The **"Nimble compiler language manual and style guide"** is included as **Appendix L** in Vol II of this report.
- **Agileware description language (ADL) and document**: We have introduced Agileware Description Language, which defines the required elements and parameters of the architecture (such as the type of processor being used and the size of the reconfigurable array, etc.), as well as the available optional elements such as instruction and date caches. ADL allows parameterized and tailored compilation using Nimble compiler to various Agileware architectures. Technical information regarding ADL can be found in **Appendix K (in Vol II of this report) "Agileware description language for the Nimble compiler encironment for agile hardware".**

2.2 Task 2: Nimble Compiler Front-End and SW Code Generator

In this section, we focus on the C front-end and optimizer of the Nimble Compiler.

The Nimble Compiler front-end takes the C programs as input and performs compiler optimization, analysis, and hardware/software partitioning to extract hardware kernels to be implemented in the reconfigurable datapath, and to generate software code that executes in the microprocessor. The front-end is built upon the SUIF Compiler framework [9]. In order to achieve its goal, the front-end compiler performs the following functionality:

1. **SUIF front-end preprocessing and optimizations**: C programs are parsed and translated into SUIF intermediate representations (IR). All subsequent steps will be performed on the SUIF IR.
2. **Hardware kernel identification**: The Nimble front-end analyzes the source application and identifies kernels for potential hardware implementation.
3. **Profiling, analysis, and visualization**: Profiling is performed to obtain program execution paths and frequencies, loop iterations, performance estimation at various levels, and loop traces etc. The profiling results can be used to drive compiler optimizations and hardware-software partition. They also provide feedback to the designers and can be viewed via our visualization tools.
4. **Compiler optimizations**. They are applied to improve both the hardware feasibility and the performance of kernels.
5. **Hardware-software partitioning**: We also refer to hardware/software partitioning as (hardware) kernel selection. This step bridges the front end and the back end of the Nimble Compiler, and decides what portions of the application execute in hardware and what stay in hardware.

Steps 1—4 of the Nimble front-end fall into the domain of Task 2. Step 1 is a trivial process and we will not further discuss it. Step 5 is dedicated a separate task--Task 3 and will be addressed in Section 2.3. We now look at steps 2-4 in greater detail.

2.2.1 Front-End Step 2: Hardware Kernel Identification

In this step we first extract all kernels and then analyze the hardware feasibility of these kernels.

Control flow graph, loop (kernel), and basic block extraction: The extraction process establishes the internal data structures needed by the Nimble Compiler. For each function in the original C programs, a control flow graph (CFG) is constructed. Figure 5 shows an example CFG. Each node in the CFG corresponds to a basic block (BB), and is labeled by a number. Edges represent the control flows of the basic blocks. A back edge (colored in pink in Figure 5) in a CFG implies a loop. Since loops are the main targets of hardware implementation, it is important for the Nimble front-end to identify them and label them as potential hardware candidates.

Hardware-feasibility analysis: This step identifies what loops in the source programs are suitable for hardware implementation. While Nimble supports all standard ANSI C features, there are some features and constructs are not supported in the hardware and need to be implemented in the microprocessor. The non-hardware acceleratable features include:

- Subprogram calls that are not inlined or inlineable (such as recursive procedures).
- Inner loop entries when considering the outer loop for hardware.
- Float/double type arithmetic and relational operators (non-integral types).
- Bit field specifiers in structures.
- Variable multipliers; divide and modulus (not available in the GARP target, inefficient in the ACEv, not available except constant multiplier in the ACE4k).

Feasibility analysis is first done for each basic block in a loop (kernel). In the example of Figure 5, the feasible basic blocks are labeled in green. A kernel can be fully implemented in hardware if all of its basic blocks are feasible. If it contains infeasible basic blocks, it can still be partially implemented in hardware as long as it contains feasible paths from loop entry to loop exit. When the program needs to execute an infeasible path, an **exceptional exit** is created so that the program can exit the hardware and continue the execution of the infeasible path in the software.

The above process of the identifying and extracting hardware kernels is thoroughly discussed in the supporting document **“The Nimble Compiler Language Manual and Style Guide” (Appendix L in Vol II)** and will not be repeated here.

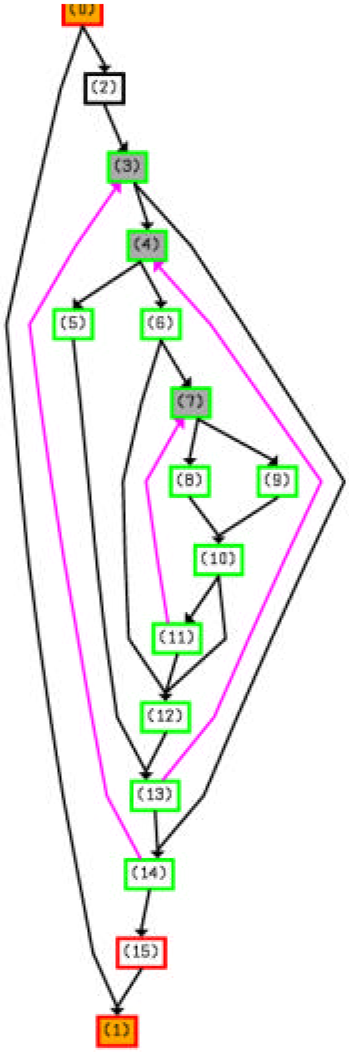


Figure 5. An example control flow graph (CFG).

2.2.2 Front-End Step 3: Profiling, Analysis and Visualization

Nimble performs automatic compiler transformations to improve application performance, followed by automatic hardware-software partitioning of an application onto the target platform. In order to decide what transformations to perform and what kernels to implement in hardware in order to achieve maximum performance, accurate performance profiling and analysis is a must. It is also needed as an important feedback to the users, to help analyze performance of different part of the application, identify performance bottleneck, and see if the final design satisfy the performance requirements. We have designed a complete solution to use in the Nimble flow. Our profiling, analysis and visualization tools (referred to as the profiling framework) provide the capability of modeling the overall software, hardware and reconfiguration time in one comprehensive framework.

The Nimble profiling framework has the following characteristics:

- It can perform profiling at different granularities: application level, functional level, loop level and basic block level.
- It can be used to estimate software performance only, but more importantly it can be used to model the overall application timing which is composed by software time, hardware time, hardware/software interface time, and (re)configuration time.
- It supports both estimation-based and measurement-based performance analysis techniques, since they can be required at different point of the Nimble compile flow. For example, the estimation-based approach is used during design space exploration of the hardware/software partitioner, and the measurement-based one is used after Nimble generates the final design to verify the satisfaction of performance constraints.
- It integrates performance, path, and trace profiling techniques for fast yet accurate results.
- It performs source-level code instrumentation during profiling and is independent of the target platform. Therefore, it can be easily retargeted to multiple Agileware platforms. Even though the profiling technique is developed to use with the Nimble Compiler, it can be either used alone purely as an analysis tool, or integrated into other system-level design flows.

The profiling framework consists of the following tools:

- **Meter:** performance measurement on target platform.
- **HALT path profiling**: path and frequency profiling tool.
- **Loop entry trace profiling (LEP)**: records and compresses loop entry trace to infer configuration frequencies.
- **Interesting loop detection (ILD):** analysis and reporting tool for performance data.
- **Loop-procedure hierarchy graph (LH)**: visualization of the loop-procedure hierarchy graph.
- **S2vcg:** visualization of control flow graph and basic blocks.

Figure 6 shows the flow of the whole Nimble profiling framework, including both the estimation-based (left dashed box) and the measurement-based (right dashed box) flows. Supporting document **“Profiling Tools and Result Viewers for the Nimble Compiler Project” (Appendix D)** describes the details and usage of all the Nimble profiling tools. In this report, we will describe some technical rationales for Meter, LEP and HALT. Please refer to Appendix D for details of other profiling tools.

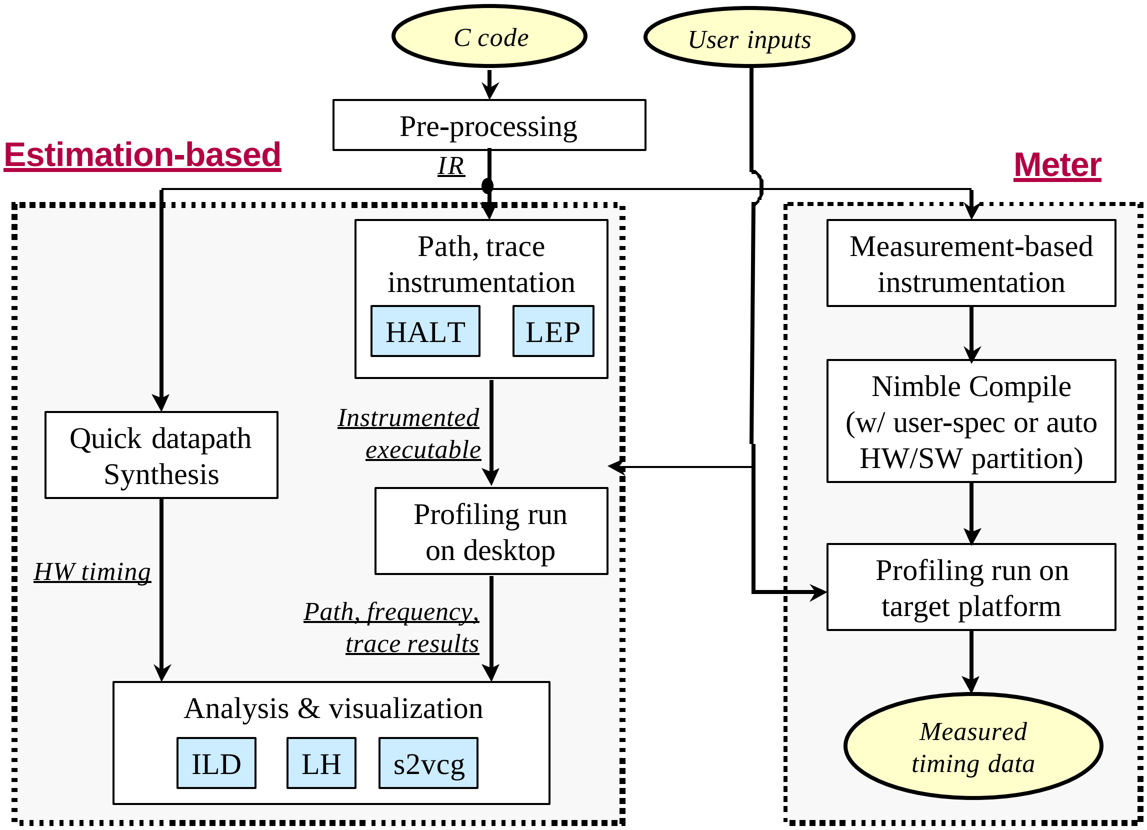


Figure 6. The Nimble profiling framework.

A. Meter: measurement-based performance profiling

The measurement-based approach inserts probing points to a program to capture the actual clock while the program is running. Since the overhead of each inserted clock-capture code can be very small (several to ten of clock cycles) compared to application execution time, this approach can yield accurate results when it is used to measure application level performance. It is also possible to use it to measure procedure, loop level performance. However, the result accuracy will not be ideal because inserting too many probing points in critical part of the code can significantly alter program execution behavior by affecting compiler optimizations, register allocation at compile time, and at run time, altering the pipeline and cache behaviors.

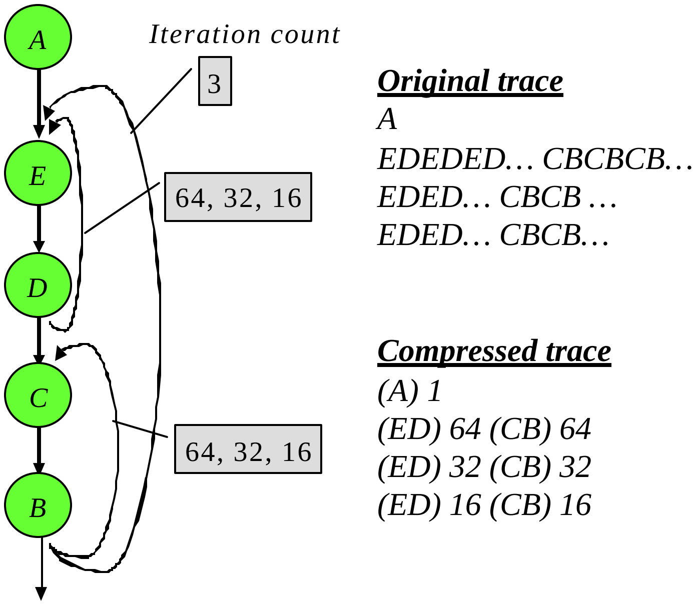
There are limitations to the use of the measurement-based technique by a design automation tool such as Nimble due to the size of the large design space. For the same source application, Nimble needs to explore many different hardware/software partitioning possibilities, different hardware and software implementations of the same kernel. Measurement is only for one design implementation. It is impossible to measure the performance of all these variations. Therefore, we only use this technique to verify the performance constraints for the final design and to validate the results of the estimation-based approach.

Inside the dashed box on the right hand side in Figure 6 is the Meter flow. C programs are translated into an intermediate representation (IR), the IR is then inserted with certain probing points based on user specifications. The tool provides a set of flexible options to allow users to select what to measure (application, any combinations of loops etc). The instrumented IR is then translated back to C and compiled for the target platform using the Nimble Compile flow, with the hardware/software partition that the user is interested in measuring (could be user specified or use the automatic Nimble selection). The executable then runs on the target platform with user-provided inputs and captures timing information for the instrumented points.

B. LEP: loop entry-trace profiling

When a hardware loop is entered for the first time, it needs to be configured into the FPGA. If it is entered again before being overwritten by another loop, it does not require reconfiguration. To compute configuration cost, we need to know the exact runtime sequence of all hardware candidate loops (i.e. the entries to these loops). Loop entry trace profiling (LEP) identifies and instruments loop entries to generate a trace. Because the trace can be potentially huge, (encoding 4 frames using standard MPEG2 generates ~200M bytes of loop entry trace,) LEP incorporates an online compression scheme to encode the trace. Loop trace compression not only saves storage space, but more importantly, the compact representation allows fast traversing of the trace in later steps of the algorithm.

The example in Figure 7 illustrates a high-level control flow of a code segment. Nodes A—E are inner-most loops. E and D, and C and B are nested in two more layers of loops. The original trace is compared with the compressed trace, in which repeated patterns are extracted and the numbers of repeats are recorded. For the MPEG-2 encoding example, the trace size is reduced to several Kbytes after compression.



Original trace

A
EDEDED… CBCBCB…
EDED… CBCB …
EDED… CBCB…

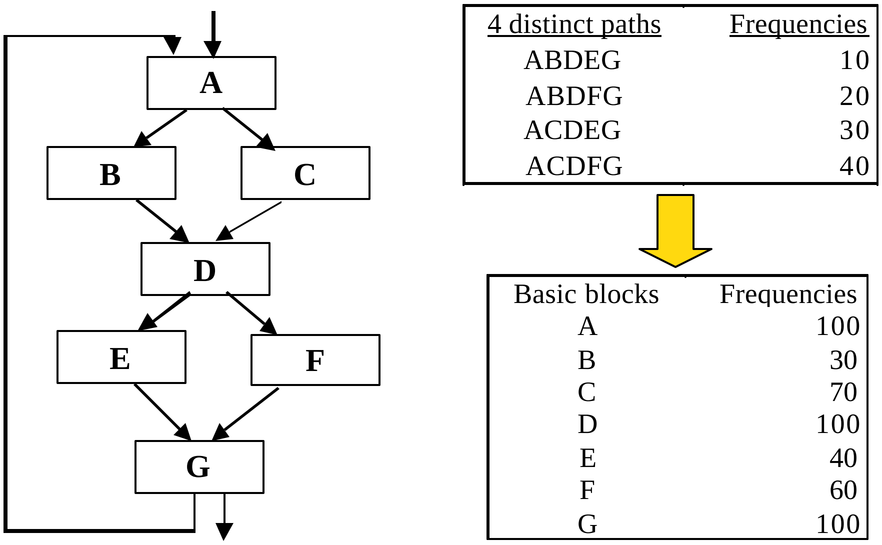
Compressed trace

(A) 1
(ED) 64 (CB) 64
(ED) 32 (CB) 32
(ED) 16 (CB) 16

Figure 7. Loop entry trace example.

C. HALT Path profiling

Program path information is needed to calculate basic block frequency and loop iteration counts. We used the HALT (the Harvard Atom-Like Tools) [8] techniques developed by Young and Smith to instrument branches in the program, and thus infer paths. We only record path up to the feedback edge (loop edge) as detected by our analysis. Figure 8 shows the control flow graph of a loop example. Nodes A-G are basic blocks inside the loop. It is obvious that there are four distinct paths inside the loop (without counting the loop feedback edge). Instrumentation is done to record the frequencies of these paths. Suppose the results of the path frequencies are as shown in Figure 8. The basic block frequencies can then be inferred from the path frequencies by summing the counts of all the appearances of each basic block.

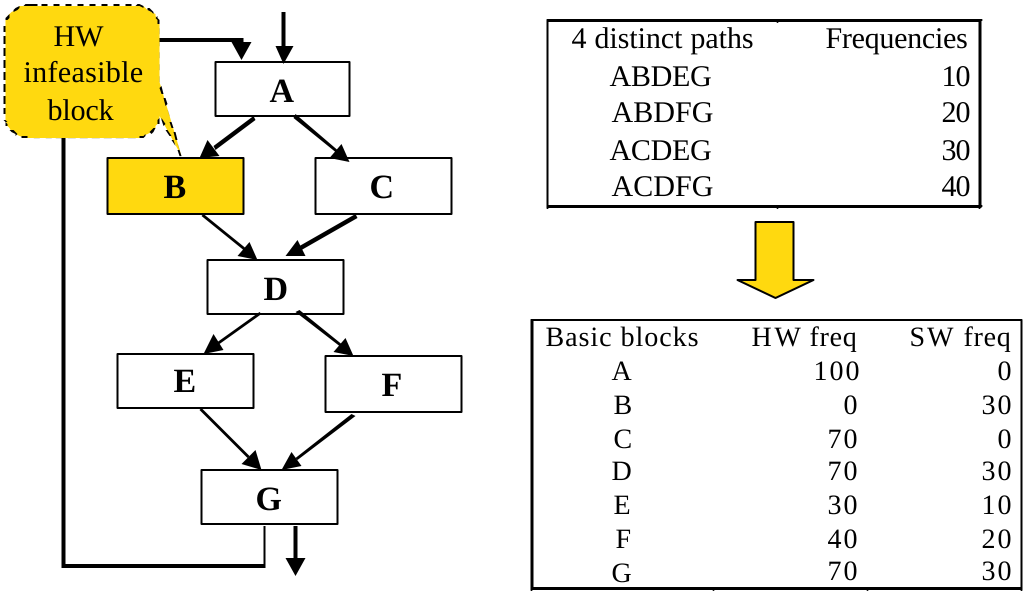


4 distinct paths	Frequencies
ABDEG	10
ABDFG	20
ACDEG	30
ACDFG	40

Basic blocks	Frequencies
A	100
B	30
C	70
D	100
E	40
F	60
G	100

Figure 8. Control flow graph of a loop example, and the correspondent path profiling results.

Path information is not only required for estimating total execution frequencies of basic blocks, but also used, when in mixed hardware / software execution, to estimate a basic block's hardware and software frequencies respectively. Figure 9 shows the same loop as in Figure 8. Suppose the implementation is to keep basic block B in software, because it is not feasible for hardware (say it contains a printf), and the rest of the basic blocks are implemented in hardware. When the program enters the loop, it will first enter hardware. But, if it takes a path containing block B (say in paths BDEG and BDFG), then the program exits hardware into software and finishes the rest of the loop iteration in software to avoid the overhead of excessive context switching. Hardware and software frequencies for the basic blocks are calculated as shown in Figure 9. Path information is essential in the design space exploration process since we need to evaluate multiple hardware/software partitions for each loop.



4 distinct paths	Frequencies
ABDEG	10
ABDFG	20
ACDEG	30
ACDFG	40

Basic blocks	HW freq	SW freq
A	100	0
B	0	30
C	70	0
D	70	30
E	30	10
F	40	20
G	70	30

Figure 9. Hardware and software frequencies inferred from path profiling results.

2.2.3 Front-end Step 4: Compiler optimizations

Nimble applies compiler optimizations to improve both the hardware feasibility and the performance of candidate kernels. We have implemented conventional compiler transformation techniques as well as developed novel techniques such as unroll-and-squash. **"Nimble Compiler Language Manual and Style Guide" (Appendix L in Vol II)** describes how to use compiler transformation in the Nimble environment. Here, we list some of the most useful transformations we have implemented.

- **Procedure inlining:** Procedure inlining enables more kernels to become feasible for hardware implementation. We have developed an inlining tool (a SUIF pass) that can not only support user manually specified inlining, but also can automatically detect what function call instances to inline in order to achieve maximum hardware feasibility. The Nimble inliner is run as a preprocessing pass before profiling, since it may dramatically alter the program and change the profiling outcome.
- **Strength reduction**: Strength reduction replaces some operations with reduced strength operations of equal functionality where possible. Since shifts are free (in terms of both time and area) in many reconfigurable datapath (for example, the Xilinx 4K and Virtex FPGAs), it is possible to replace `i*c` where `i` is an induction variable and `c` is a power-of-2 constant with a shift operation.
- **Scalarization:** Scalarization moves array index computation and array memory references out of loops, and therefore, can significantly improve loop performance. We developed a scalarization pass based on the SUIF dependence analysis. It is shown that this automatic scalarization pass can produce result quality close to that of the manual scalarization by a designer.

- **Loop unrolling**: Loop unrolling lets us directly exploit additional capacity on the FPGA by increasing potential instruction level parallelism. The scheduler in our datapath compiler will utilize any available operator parallelism that is exposed by unrolling.
- **Pipelining:** Pipelining is superior to loop unrolling for at least two reasons. First, there are no inter-iteration boundaries. Unrolling still loses all available overlap every few iterations. Second, pipelining has the potential of much higher gate utilization. A pipeline with *k* stages has as much asymptotic parallelism as *k* unrolled loops but without using nearly as many extra CLBs. The Nimble Pipelining technique has demonstrated up to 16x speedup on simple contrived loop, 4x speedup on real kernel extracted from Versatility benchmark, and 2x overall speedup at application level for Versatility.
- **Unroll-and-squash**: We developed a novel method for mapping nested loops into hardware and pipelining them efficiently, named unroll-and-squash. The technique achieves fine-grain parallelism even on strong intra- and inter-iteration data-dependent inner loops and, by economically sharing resources, improves performance at the expense of a small amount of additional area. Unroll and squash is implemented within the Nimble Compiler environment and its performance was evaluated on several signal-processing and cryptography benchmarks. The method achieves up to 2X increase in the area efficiency compared to the best known optimization techniques. Unroll and squash is extensive discussed in this supporting document **"Efficient Pipelining of Nested Loops: Unroll-and-Squash, an Innovative Compiler Optimization Technique used in the Nimble Compiler" (Appendix E).** Some results of applying this transformation to some benchmarks are included in **"Final benchmark report for the Nimble compiler environment for agile hardware" (Appendix M in Vol II).**

2.3 Task 3: Performance and Capacity Drive Partitioner

A key component of the Nimble environment is the automatic hardware/software partitioner that performs fine-grained partitioning (at loop and basic-block levels) of an application to execute on the combined CPU and reconfigurable datapath. The hardware/software partitioner considers the tradeoffs between performance and capacity (area) and optimizes the global application performance, including the software and hardware execution times, communication time and datapath reconfiguration time. Extensive benchmarking on real applications shows that our partitioning approach is effective in rapidly finding close to optimal solutions.

Research efforts in co-design mainly dealt with the conventional embedded hardware/software architectures containing ASICs. However, the partitioning problem for architectures containing reconfigurable FPGAs has a different requirement: it demands a two-dimensional partitioning strategy, in both **spatial** and **temporal** domains, while the conventional partitioning involves only spatial partitioning. Here, spatial partitioning refers to physical implementation of different functionality within different areas of the hardware resource. For dynamically reconfigurable architectures, besides spatial partitioning, the partitioning algorithm needs to perform temporal partitioning, meaning that the FPGA can be reconfigured at various phases of the program execution to implement different functionality.

The Nimble hardware/software partitioning approach focuses on the temporal partitioning aspect. The input to the algorithm is a set of candidate loops for hardware (termed **kernels**) that have been extracted from the source application. Each loop has a software version and one or more hardware versions that represent different delay and area tradeoffs. The partitioning algorithm selects which loops to implement in the FPGA, and which hardware version of each loop should be used to achieve the highest application-level performance. This algorithm fulfills all the requirements posted in the initial statement of work and beyond:

- The partitioner is driven by performance and capacity. Partitioning must be guided by various forms of profiling information to accurately assess the tradeoffs between hardware and software implementations, and configurations. We not only developed the comprehensive Nimble profiling framework (see Section 2.2.2), but also built interfaces to seamlessly integrate the profiling framework and its results to be used by the partitioning algorithm. The profiling tools have been explained in Section 2.2.2 and Appendix D.
- The partitioning algorithm is fully parameterizable. Agileware Description Language (ADL) (see Appendix K in Vol II) is used to describe important parameters and characteristics of the target platform. The partitioner takes the ADL specification as input and generates customized partition for the particular application running on the target platform.
- The partitioner allows user-specified partition and still finds optimal (or near optimal) partitions for the rest of the program under the user specification. This capability can be used to incrementally evaluate various partitions before the user settles on the final result. Since the partitioner is closely coupled with the Nimble profiling and reporting tools, it is very easy for the user to compare different partitions in terms of quality of result.

- Our partitioning algorithm effectively captures the dynamic reconfiguration costs. This is difficult as the number of reconfigurations for one kernel depends on which other kernels may go into the hardware. This is one of the first efforts to efficiently model dynamic reconfiguration cost in automatic hardware/software partitioning.
- The algorithm integrates compiler optimizations and hardware design space exploration into the hardware/software partitioning process. Compiler optimizations are applied to kernels before partitioning to generate multiple hardware design choices for each kernel. The algorithm then integrates all the hardware choices into its design space exploration process.

We published a paper describing the partitioning algorithm at *the 37th Design Automation Conference* in June 2000 [1]. Please refer to the paper (**titled "The hardware-software partitioning approach of the Nimble compiler",** included as Appendix F) for technical details and experimental results of the algorithm.

2.4 Task 4: Datapath Compiler Back-end

For a prototype environment for the Nimble Compiler, this project initially focused on providing a compiler for the reconfigurable TSI-Telsys ACE II board. The board includes a MicroSparc, two Xilinx XC4085XL devices, and dedicated SRAM and DRAM. As part of the Garp compiler developed at U.C. Berkeley is the datapath compiler named "Gama" which performs the module mapping, scheduling, and placement of the datapath portions of programs to the reconfigurable array of the Garp architecture. A restricted version, gamax, providing limited technology mapping support for the Xilinx 4000 series FPGAs, was developed in the initial phase of the project. This work leveraged much of the technology from the Garp datapath compiler.

The current Xilinx datapath compiler known as *Xima*, further improved upon *gamax* by providing full technology mapping for integer ANSI C and for some domain-specific functions, with an extensible and isolated generator library. It also included target support for the Virtex 1000 series parts, on a ADM-XRC daughter card mounted on TSI-Telsys ACE I boards. Within the context of the Nimble Compiler, the XC4085XL / ACE II combination is referred to as the "ace" target, and the Virtex 1000 / ACE I combination is referred to as the "acev" target.

The input to *Xima* is an unscheduled dataflow graph (in a format known as "AFL"), consisting of processing nodes and data (or control) communication edges. The output consists of:

- A ".ro" file, containing symbolic and other information on the datapath rows.
- A ".xnf" file, which is the datapath netlist.
- ".edn" files for each operator module required by the datapath.

The base Xima is enhanced with functional generator libraries (see Section 2.5, Task 5), and provides quick synthesis results to the hardware/software partitioning tool.

The document **"Xima – The Nimble Datapath Compiler" (Appendix G)** describes the Xima datapath architecture framework and other technical details.

2.5 Task 5: Functional Generator Library Research and Prototyping

The Functional Generator Library task was mainly done at Lockheed Martin ATL with collaboration from Synopsys Inc.

Macro functional generator libraries are developed to facilitate the datapath synthesis process. Generator libraries are provided to support the full set of ANSI integer operations. Additional domain-specific generators are provided to facilitate acceleration of cryptographic, multimedia, as well as general computational functions. Table 2 lists all generators for the intrinsic integer operations. Table 3 lists the generators for domain-specific operations.

Table 2. Functional generators supporting intrinsic operations.

Operator	Description
add	Binary Adder *(+)*
comp	Compare function *(<, <=, >, >=, ==, !=)*
div	Binary Divider *(/)*
logic	Up to 4-input logic specified by table *(&, |, ~, ^, &&, ||, !, ^^, ?)*
mul	Multiplier *(*)*
mux	Two-to-one registered multiplexor *(live variables)*
neg	Negate unary operator *(-)*
reg	Register *(for memory address, load, store, inputs, patches)*
rem	Remainder *(%)*
shift	Shifter with variable shift count *(<<, >>)*
sub	Subtractor *(-)*

Nimble has shown signification performance gain with domain-specific generators. For example, in the DES encryption application, an impressive speedup over software-only solutions of up to 400x was achieved using several domain-specific generators.

We have included two supporting documents regarding generator libraries along with this report. One document **"Domain Generator Tutorial for the Nimble Compiler Project" (Appendix H)** is a tutorial that documents the process of creating a new domain generator, integrating it, and using it within the Nimble Compiler framework. The ability to create and add custom domain generators to the Nimble Compiler allows the user provides a quick way to develop a digital circuit and then test its functionality by calling it out in a C program. Especially critical portions of a program can be custom implemented in hardware, and then simply invoked by a function call. The convention for function name is to append "nimble_" to the beginning of the generator name. The document **"Specification for FPGA Macro Generators for the Nimble Compiler Project" (Appendix I)** describes how to use all the generators implemented in Nimble (see Table 2 and Table 3), and for each generator, a complete specification including command-line generation, generator instantiation, operation supported, implementation, interface, and any special options.

Table 3. Domain-specific functional generators.

Operator	Description
abs	Fixed point absolute value: *(a>0) ? a : -a*
bytesel	Friendly endian byte select: *a >> ((3-byte&3)*8)&255*
fir	FIR filter
parcnt	A 32-bit counter which skips over parity (LSb) bits of each byte
permute	General bit permute, set, and clear
ram	CLB-based RAM (not in external memory)
rom	CLB-based ROM
sbox	The DES encryption sbox computation with “P” permute
sjg	The skipjack encryption “G” function

2.6 Task 6: Benchmarking and Demonstrations (application development using tools)

The Nimble benchmarking process is crucial for evaluating both Agileware platforms and the Nimble Compiler itself. It helps us to understand how our target applications— i.e., embedded DSP applications — behave on the Agile Hardware architectures that are the target of compilation, and how the Nimble Compiler handles a wide range of codes. More specifically, we wanted to achieve the following objectives:

- To show that the compiler and the architecture can support a wide range of general purpose, off-the-shelf applications and not be limited to one or more particular algorithm sets. There are two keywords here: general-purpose, and off-the-shelf. To demonstrate Nimble’s general utility, we need to collect benchmarks from various application areas.
- To demonstrate that the Nimble Compiler provides a better compilation model and faster compilation speed than existing VLIW/DSP processors. While conventional tools for hardware synthesis take hours or even days to generate results, the Nimble compiler aims to emulate a more software-like compilation process: The user performs incremental code revisions and the compiler generates executables in minutes instead of hours or days.
- To verify that using Nimble for Agileware architectures can achieve higher performance than existing solutions, such as general-purpose CPUs or DSPs.
- Last, but not least, to baseline the major milestones in our 3-year development efforts and to show the improvement gained through different development phases.

It is important for our project to not only benchmark the Nimble compiler itself, but also to benchmark the targeted Agileware platforms against other comparable solutions. To achieve the above goals, we have selected a representative set of applications to form the Nimble benchmark suite. We have run these applications on Agileware platforms, using executables automatically compiled using the Nimble Compiler, or hand-optimized for the datapath for comparison purposes. We have also run some of the applications on other commercially available DSPs such as the C6x processor from TI, and compared the results on these platforms to those on Agileware.

To demonstrate the Nimble Compiler's capability of working with a wide range of off-the-shelf applications, we have built the benchmark suite by selectively using applications from different sources, including:

- The ACS benchmark suite, developed by Honeywell Technology Center. This suite consists of a wide range of applications, from CAD algorithms to image processing.
- Lockheed Martin ATL INFOSEC cryptography benchmarks. The main focus here is state-of-art cryptography algorithms.
- DSPStone benchmark suite from Aachen University, Germany.
- UCLA MediaBench suite.
- SPEC'95 benchmarks.

The above benchmark suites consist of a large number of applications in various application fields, including image processing, DSP, CAD algorithms, cryptography, games etc.

The results of our benchmarking efforts are documented in an extensive report: **"Final benchmark report of the Nimble compiler environment for agile hardware" (Appendix M in Vol II)**. At past reviews and PI meetings, we have given live demonstrations of some of the above benchmarks on real hardware (ACEII and ACEV platforms).

In the Final Benchmark Report, we presented our benchmarking results and analysis for 18 applications in various domains. For each application, we presented detailed analysis both at the application and at the kernel level. Various metrics such as performance and area are obtained through both profiling (estimation based) and actual measurements. We made comparisons among different implementations include the software only ones and mixed hardware and software ones. We also compared across different platforms including both Agileware platforms such as Garp and ACEV, as well as a competitive commercial platform C6X.

During benchmarking, all the applications were required to pass the desktop step before we even included them as part of the suite. Table 4 summarizes the status of these benchmarks on the two Agileware platforms. We were able to use Nimble to profile, synthesize and compile all these applications on both Garp and ACEV. However, for some application, we failed to obtain the run time data on Garp or ACEV. On Garp, because we were using a simulator to obtain run time data, if an application has a very long run time, it may not finish simulation in days or even weeks. This is noted in the table as "too slow". On ACEV, there is a very limited amount of memory available (about 4M) and some programs' memory requirement exceeds this amount. This is noted in the table as "requires too much memory".

Refer to Appendix M in Vol II for complete benchmark results on these 18 applications.

Table 4. Nimble benchmarking status.

Benchmark name	Garp			ACEV		
	Profiling, synthesis & compile	SW-only run	HW/SW (auto) run	Profiling, synthesis, & compile	SW-only run	HW/SW (manual) run
Versatility	Ok	Ok	Ok	Ok	Ok	Ok
CFAR	Ok	Too slow		Requires too much memory		
Skipjack	Ok	Ok	Ok	Ok	Ok	Ok
DES encrypt	Ok	Ok	Ok	Ok	Ok	Ok
TwoFish	Ok	Too slow		Ok	Ok	No HW kernel
DSP Stone ADPCM	Ok	Ok	Ok	Ok	Ok	Ok
MediaBench ADPCM	Ok	Ok	Ok	Ok	Ok	Ok
G721 encode	Ok	Too slow		Ok	Ok	Ok
G721 decode	Ok	Too slow		Ok	Ok	Ok
MPEG2 encoder (revised)	Ok	Too slow		Requires too much memory		
MPEG2 decoder	Ok	Too slow		Ok	Ok	No kernel with speedup
JPEG encoder	Ok	Too slow		Ok	Ok	No kernel with speedup
JPEG decoder	Ok	Too slow		Ok	Ok	No kernel with speedup
GSM	Ok	Too slow		Requires too much memory		
UNEPIC	Ok	Too slow		Requires too much memory		
PEGWIT	Ok	Too slow		Requires too much memory		
Spec’95 GO	Ok	Too slow		Ok	Ok	No kernel with speedup

Note:

The word “Ok” means we have data for the corresponding running mode. Comments other than “Ok” explain why we did not have data for the corresponding running mode.

3 Project Conclusions

Through extensive benchmarking, we applied the Nimble Compiler to a wide array of applications and successfully targeted real hardware (ACEV platforms). We draw the following conclusions for the Nimble Project:

1. The Nimble compiler provides a viable framework for compiling a wide range of general-purpose applications written in the high-level language C automatically onto Agileware platforms. Nimble supports the full standard ANSI C, not a subset or superset of the C language. In terms of compilations, Nimble can be applied not just on one or several particular types of codes, but the general-purpose applications, even though some application types such as multimedia and cryptography may have higher performance gain potential on Nimble than others.

2. Nimble allows fast and retargetable compilations. Nimble uses a compilation model is similar to that of the software, which allows incremental code revisions and finishes compilation to mixed hardware and software in minutes.

3. Nimble framework integrates sophisticated profiling mechanisms into the compilation process which enables the compiler to make intelligent decisions (such as hardware kernel selection) automatically.

4. Using the Nimble compiler on Agileware platforms demonstrated great performance potentials. For most applications, we were able to obtain higher performance than software-only solutions. Some achieved better results than state-of-the-art DSP processor C6X. For applications like DES Key Search, using domain-specific generators, Nimble generated tremendous speedup (up to 400X over software solution) still with fast, automatic, and software-like compilation.

Because of the unavailability of a tailored commercial Agileware platform, much of Nimble's potential for generating high performance is hard to demonstrate. Even though ACEV is a real platform, its design has many flaws (as we have discussed earlier) and it is not an ideal implementation. However, being a research prototype, Nimble provides a valuable platform for doing further research and development work in reconfigurable computing.

4 References

[1] Yanbing Li et.al, "Hardware-software partitioning of embedded reconfigurable architectures", in Proceedings, *37th Design Automation Conference (DAC)*, June 2000.

[2] T. J. Callahan and J. Wawrzynek, "Instruction level parallelism for reconfigurable computing," *Proc. 8th Intl. Workshop on Field-Programmable Logic and Applications*, September 1998.

[3] T. J. Callahan, John R. Hauser, and John Wawrzynek, ``The Garp Architecture and C Compiler", *IEEE Computer*, April 2000.

[4] D. Petkov, et.al, "Efficient pipelining of nested loops: unroll-and-squash", submitted for publication.

[5] S. Kumar et.al, "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future Directions," in proceedings, FPGA 2000 - Eighth International Symposium on Field-Programmable Gate Arrays, February 10-11, 2000, Monterey, California.

[6] TSI Telsys, "ACE2 Card Manual", 1998.

[7] Richard Goering, "Compiler project marks Synopsys' step into post-ASIC world", *EE Times*, August 28, 2000.

[8] Cliff Young, "The Harvard Atom-line Tool (HALT) Manusl", http://www.eecs.harvard.edu/hube/software/v112/halt2.html/.

[9] The SUIF Compiler Group, http://suif.stanford.edu/.

Appendix A. Overview of the ACEV Environment

1 Introduction

The ACEV platform combines a conventional processor and a reconfigurable component to provide a complete environment independent of the host computer. On one hand, these features make it suitable as an prototyping environment for hardware-software Co-design. On the other hand, though, it can be considered as an emulator for a *hybrid* processor that combines both fixed and reconfigurable components to flexibly adapt to the applications.

This document is intended to familiarize potential users with the fundamentals of the platform and point to more detailed documentation for further reference. With the exception of [14], all of the documents cited in the bibliography are provided in soft-copy form (PDF files).

2 System Architecture

2.1 Hardware

The ACEV hardware combines two key components: The ACEcard, a PCI expansion card manufactured by TSI-TelSys[1] supplies the host interface, the conventional processor (called Embedded Processor Subsystem, EPS) and its main memory [2]. An ADM-XRC PMC card [6] attaches to this board as a daughter-card and provides the FPGA-based reconfigurable processing unit (RPU). Both PCI and i960-like busses are used and interconnected using Bus Interface Units (BIU). Figure 1 shows the combined architecture.

For purposes of the ACEV hardware, only a subset of the entire hardware available on both cards comes into play. For example, the ACEcard only acts as a carrier for the

[1] ... which became Lavalogic and was recently acquired by Xilinx.

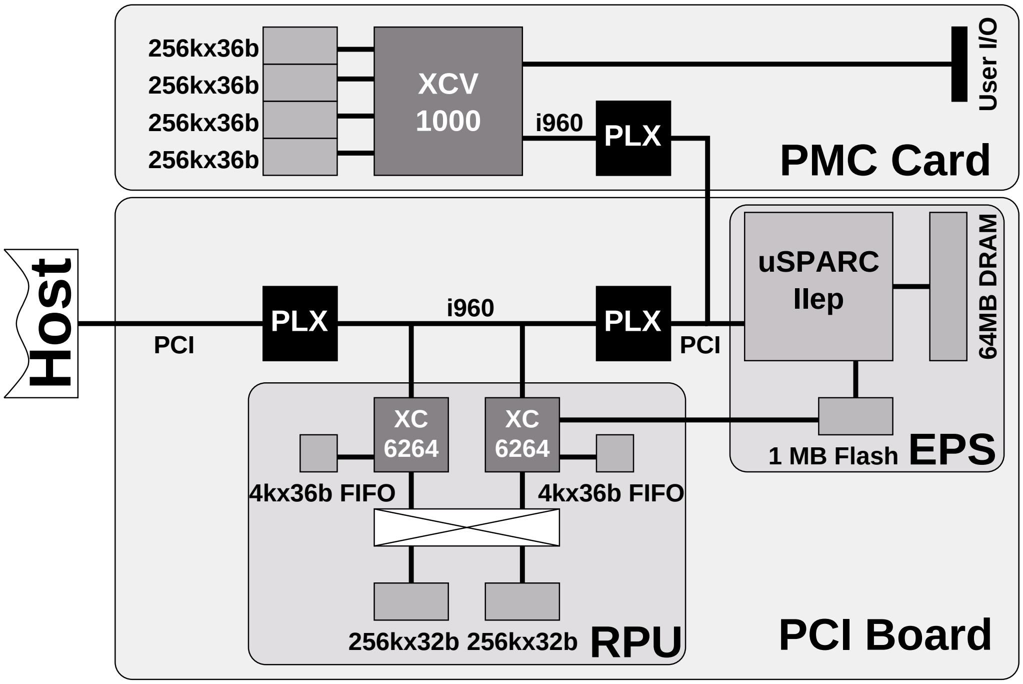


Figure 1: ACEcard combined with ADM-XRC daughter-board

ADM-XRC and provides a SUN microSPARC-IIep processor as embedded CPU. The reconfigurable resources present on the ACEcard (two Xilinx XC6264 FPGAs with associated memories) are ignored in favor of the more recent Xilinx XCV1000 FPGA supplied by the ADM-XRC card.

Thus, the ACEV hardware can be simplified down to the structure shown in Figure 2

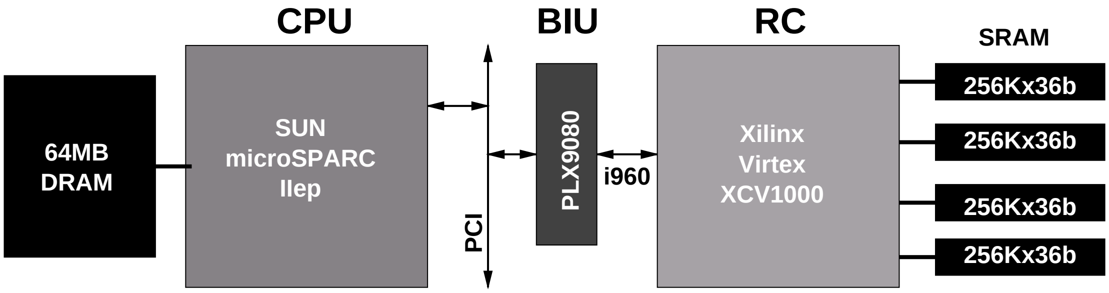


Figure 2: Simplified view of ACEV hardware

The key components are the

EPS : consisting of the microSPARC-IIep processor [5] and the associated 64MB of DRAM.

RPU : consisting of a Xilinx XCV1000BG560-3 Virtex-type FPGA [8] coupled to 4MB of No-Bus-Turnaround memory organized as four independent banks of 256kx36b GSI GS880Z36T-100 [9] chips.

BIU : a PLX9080 PCI I/O Accelerator [10] that has the FPGA attached to a simple-to-implement, yet fast i960-like bus (clock programmable from 25MHz-40MHz, separate 32b data and 24b address lines). The BIU bidirectionally translates this protocol to the standard 33MHz 32b PCI bus that is available on the EPS.

Minor components include firmware in Flash memory both on the EPS and RPU as well as a programmable clock generator for the RPU.

2.2 Software

Hardware is only one half of a complete system. On the software side, we have three major interacting modules.

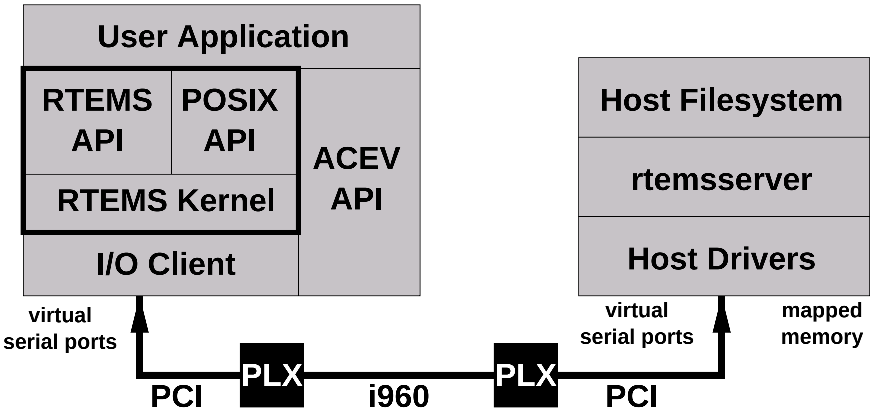


Figure 3: ACEV Software Architecture

Figure 3 shows the relationship between the different components described below.

2.2.1 ACE Firmware and Host Drivers

The boot firmware of the ACEcard (held in 1MB of Flash memory) provides basic services such as initial hardware setup and essential exception handling (e.g., processing the register window overflow/underflow traps common to the SPARC architecture [14]).

In addition, it also interacts with the host-side device drivers to allow the host-computer access to the EPS hardware using device files in the host file-system.

- The DRAM space of the EPS is accessed via the host's `/dev/ace0dram` device. Reading or writing to this file on the host reads or writes the data from/to the DRAM. The file offset (starting at 0, changeable using the C library `fseek()` and `lseek()` functions) determines the target address in DRAM. E.g., to write 1KB of data from the binary file `test.bin` to the EPS DRAM from address 16384 upwards, the command

```
dd bs=1k count=1 seek=16 <test.bin  >/dev/ace0dram
```

could be issued. Conversely, the command

```
dd bs=1k count=2 skip=8 </dev/ace0dram >data.bin
```

reads 2KB of data starting at DRAM address 8192 into the file `data.bin` on the host. These transfers use PCI block transfers and are reasonably fast for larger blocks.

- The firmware and the host drivers cooperate to simulate four bidirectional virtual serial ports (VSP). One of these is reserved for the system, two are user-definable (but see Section 2.2.2), and the final one is hard-allocated to the GDB remote debugging protocol (Section 8.2). On the host side, the user-defined ports appear as the device files `/dev/ace0ttya` and `/dev/ace0ttyb`. Communication is established using a standard terminal program such as the Unix `tip` command. E.g., the command

  ```
  acehost$ tip /dev/ace0ttya
  connected
  ... interaction ...
  ~.
  acehost$
  ```

  connects the terminal program to `/dev/ace0ttya` and sends/receives single characters to/to the EPS. When the sequence '`~.`' is typed on the host, the terminal program exits.

  Since the VSPs can also be manipulated by software, they can also act as low-bandwidth communication channels between host and EPS programs (the latencies involved for the single-character transfers are considerable). Additionally, a write to a VSP is the only way for the EPS to send an asynchronous signal to the host (e.g., when an EPS interrupt occurred).

2.2.2 RTEMS

The Real-Time Executive for Multi-processor Systems (RTEMS) [12] [13] sits on top of the firmware and drivers and provides higher-level services. Among these are:

- A standard C library including conventional I/O (`printf()` and friends), math, and memory management functions.
- Pre-emptive multi-threading and associated synchronization primitives (semaphores).
- Interrupt handling and real-time clocks.
- A flexible device-driver model to attach custom hardware.

Even more useful, however, is the capability of RTEMS combined with a server program running on the host (`rtemsserver`), to allow the EPS transparent access to files and devices in the host file system. E.g., an EPS application can execute a call to `fopen("/etc/passwd", "r")` to read the host's password file. This capability also extends to the standard input/output and error streams. For example, assuming an application-wrapper (see Section 7) named `filter.run` on the host contains an EPS program, that program can be transparently integrated into pipes on the host:

```
acehost$ cat /etc/passwd | filter.run | sort
```

These services rely on the VSP `/dev/ace0ttyb` to signal the host for EPS I/O requests. Thus, that port is no longer available for user applications. Details of the port of RTEMS to the ACE platform are covered in [11].

2.2.3 ACEV API

The ACEV API deals with accessing the configurable hardware. It maps in the FPGA into an EPS-accessible memory region and also contains utility routines for downloading configurations, interrupt handling, and addressing. This API is described in greater detail in Section 6.

3 Embedded Processor System

The SUN microSPARC-IIep RISC processor implements the SPARC V8 instruction set [14]. In addition to the CPU core, it also contains an MMU, a DRAM controller and a PCI bus interface (PCIC). Since RTEMS does not require virtual memory, the MMU is not used in the current setup.

Both the DRAM controller and the PCIC, though, play important roles in the system. The processor architecture is described in [5].

Many of the more complex functions have been abstracted in RTEMS and the ACEV API. The following items, however, *do* affect EPS applications and need to be considered explictly:

- Only 16MB-64KB of the EPS DRAM is visible to the RPU. While this window of 'blessed' memory can be moved in 16MB increments over the entire DRAM space, this limitation will affect the placement of RPU-accessible data in EPS memory. In the default RTEMS configuration, the window spans the area from address 0 to 16MB-64KB. This space holds the executable code of both the user application and RTEMS (the `.text` segment), initialized static data structures (`.data` segment), uninitialized static data (`.bss` segment), the C library heap (managed by `malloc()` and friends), and the stack(s). Above this space, to the limit of physical memory, the RTEMS heap is located. Thus, an application can determine on a per-allocation basis whether a dynamic data structure should be placed in the precious 'blessed' memory or whether it can remain in the copious conventional memory. See Section 5.4 for more details.
- For higher performance, both data and instruction caches are enabled on the EPS. Due to (possibly hardware-related) problems with the microSPARC-IIep PCI cache coherency mechanisms, all external *writes* to the DRAM (either from the host or by the FPGA in master mode) will not become completely visible to the CPU until the data cache is flushed using the RTEMS Kernel call `dcacheFlush()`. Note that parts of newly written data *may* show up in DRAM, but the completeness of the transfers is not guaranteed until the flush. See Section 6.5 for a description of the times when such a flush occurs automatically. Alternatively, you could define non-cacheable areas in the MMU *if* these external write accesses occur only in a limited memory region.

4 Reconfigurable Processing Unit

The Xilinx Virtex XCV1000BG560-4 FPGA on the Alpha Data ADM-XRC card acts as the reconfigurable processing unit on the ACEV platform. The EPS and the RPU have very different views of their hardware environment.

4.1 EPS View of the RPU

The EPS sees the RPU as two memory ranges: The 8MB S0 space holds the FPGA-internal register and might allow access to the ADM-XRC local memories. Its decoding is entirely dependent on the user logic configured into the FPGA. The 8MB S1 space holds board-level control registers and an additional bank of Flash memory (unused in the current version of the ACEV platform). Further details on these memory spaces are given in [6], Section 4.

Both of these ranges are mapped into the EPS memory space. Their start addresses and lengths in that space can be retrieved using the acev_get_s0 and acev_get_s1 functions of the ACEV API (see Section 6). In general, because the control register programming of the S1 space is hidden inside the ACEV API, only the S0 space will be used by applications. Once the base addresses have been determined, simple reads and writes using pointers in the address range will exchange data with the RPU.

```
...
// be sure do mark the base pointer as volatile!
volatile int *s0base;

// get base address (ignore length)
s0base = acev_get_s0(NULL);

// now write two words of data to the FPGA
s0base[0] = 0x12345678;
s0base[1] = 0xdeadbeef;

// read a word from the FPGA
printf("output is %08x\n", s0base[2]);
...
```

Using the function acev_irq_handler(), the EPS can register a handler for interrupts caused by the RPU. Such a handler routine should not perform I/O (since another I/O operation might already be in progress) and *must* clear the cause of the interrupt in the FPGA (e.g., by writing a certain register in the user logic to acknowledge the interrupt).

The EPS configures the RPU hardware function using another set of ACEV API routines (`acev_load_config` and `acev_load_file`) as atomic operations. The intricacies of the FPGA configuration protocol remain hidden to the user programs.

For synchronization between EPS code and RPU operation, a semaphore-based mechanism that is also amenable to multithreaded EPS programs is used in the ACEV API : Assuming the RPU signals completion by firing an interrupt, an interrupt handler must be registered that calls the function `acev_mark_done` once the operation is complete. In the main exeuctiob flow of the user program, `acev_mark_busy()` is called when the RPU is allocated to a thread and the first RPU accesses, e.g., loading parameters, occur. Once the user code has started the RPU, it can either continue to execute, or sleep and wait for RPU completion using the `acev_wait()` call. In that case, other threads will be scheduled by RTEMS until the sleeping thread is awakened by the RPU interrupt signal.

Finally, the RPU clock is also programmable by the EPS. While it runs at a default of 28.5MHz, it can be varied between 25MHz and 40MHz using the ACEV function call `acev_set_clock()`. With certain limitations, this range can also be exceeded (see [6], Section 5.1 and 5.2).

4.2 RPU View of the EPS

The FPGA can access EPS DRAM independently of the processor to store data to or retrieve data from the DRAM. This scenario is called *master* mode (in contrast to the previously described *slave* mode where all transfers are initiated by the EPS).

All communication from the FPGA to the EPS first passes through a bus using an i960-like protocol. It consists of a 32b data bus, a 24b address bus (22 actual address lines and 4 one-hot byte-enable signals), a write indicator, plus various strobes and ready signals. For details, see [6], Section 5.4, and [10], Section 5.3, Table 5-6. Note the slightly different terminology. E.g., **LREADYIL** vs. **READYi#**. The following discussion will use the terms defined in [6].

4.2.1 Slave Transfers initiated by the EPS

A slave read (see [10], p.121, Diagram 8-10) from the FPGA initiated by the EPS begins with an address phase, indicated by an asserted address strobe signal **LADSL**, at which time the read/write signal and the bytes enables must also be sampled by the FPGA. Once the FPGA drives valid data onto the data bus (after a variable number of cycles!), it asserts the **LREADYIL** signal. If the EPS was only interested in a single datum, it will have asserted **LBLASTL** to signal the end of the transfer after the address phase.

A slave write initiated by the EPS (see [10], p. 122, Diagram 8-11) to the FPGA also begins with the address phase. However, the data to be written to the FPGA will *always* be valid in time for the next positive clock edge after **LADSL** has been sampled. When the FPGA has accepted the write-data, it should assert **LREADYIL**. As before, **LBLASTL** will be de-asserted after the address phase to indicate a single transfer.

Slave transfers are described in more detail in [10], Section 3.6.2. This also discusses the burst transfers as shown in Diagrams 8-12 to 8-21 of that source.

4.2.2 Master Transfers initiated by the RPU

The memory range accessible in this manner is a block 16MB-64KB in length, leading to valid local address bus addresses ranging from 0x000000 to 0xfeffff. This window of ‘blessed’ (=FPGA-accessible) DRAM memory can be moved within the EPS DRAM space in 16MB increments using the ACEV API function `_acev_set_masterbase`. By default, the window will begin at DRAM address 0. Master mode accesses can be accomplished in the following manner:

1. Request master access on ADM-XRC local bus by asserting FHOLD.
2. Wait until master access acknowledged by asserted FHOLDA.

3. Set-up address, read/write, byte enable, and (optionally) write data signals on the local bus.
4. End address phase (for single accesses)
5. Wait until read data arrives/write data was accepted when LREADYOL becomes accepted.
6. Relinquish master mode access by de-asserting FHOLD.

The detailed timing of this operation is shown in [10], p. 143-4, Diagrams 8-32 and -33 for single cycle reads and writes, in Diagrams 8-34 and -35 for burst accesses. Note the latencies involved in this operation: A master single cycle read from the RPU to the EPS DRAM has been observed to take 46-47 cycles, a single cycle write in the reverse direction takes 10 cycles.

Note that master writes to DRAM are buffered by the BIU ([10], Section 3.6.1). The only guaranteed way to *force* these FIFOs to be written to DRAM is to have the Virtex write an additional 32 words of data (possibly to a dummy address in DRAM). This becomes critical when IRQs are used by the Virtex to indicate the completion of an operation: Since an IRQ by itself does not force writing the FIFO to DRAM, it may “bypass” the FIFO and indicate readiness to the EPS even though the result data itself has not been written yet.

4.2.3 Local RPU Memory

In addition to the EPS DRAM, the FPGA has access to 4MB of local ZBT SRAM memory. This memory is organized as four independent 36b x 256K banks. When using it, be aware of the following issues:

- The chips operate synchronously to a common clock. Note that this is not restricted to the system clock. Other combinations (e.g., double the system clock) are possible.
- Since the chips use Zero-Bus Turnaround technology, no intervening clock cycles are required when changing between read and write operation.
- Address and read/write control signals are sampled at the start of a transaction.

- Write data is required and read data will arrive on the RAM pins one or two cycles *after* the address/control signals were set. The latency is programmable to trade-off latency with clock speed:

Flow-through mode :	One cycle of latency, min. clock period 15ns.
Pipelined mode :	Two cycles of latency, min. clock period 10ns

  Be sure to note that the delay cycles also apply to write data!

- The chips are capable of burst mode operation: After a start address has been loaded into the chip (by setting the ADV signal to 0), up to four words from consecutive addresses can be read without requiring a new external address to be present (hold ADV high during the burst).

4.2.4 Interrupts

For asynchronous signaling, the FPGA may send an interrupt to the microSPARC, for which a handler can be registered using the ACEV API. The FPGA fires the interrupt by holding the `LINTIL` output signal at a low logic level. The user design itself should have a provision to clear this interrupt (e.g., by writing to an FPGA register address in S0 space). It is not recommended to deactivate the interrupt by calling the ACEV API enabling/disabling routines. These are intended only for the initial set-up and final tear-down of IRQ management, not for a per-IRQ usage.

5 Memory Map

A simplified memory map of the ACEV platform is shown in Figure 4. On the left side of the memory range, the addresses of the respective memory region are shown. On the right side, its main use is described. Note that most of the addresses are variable: They can be retrieved at run-time by reading out global variables or calling dedicated functions.

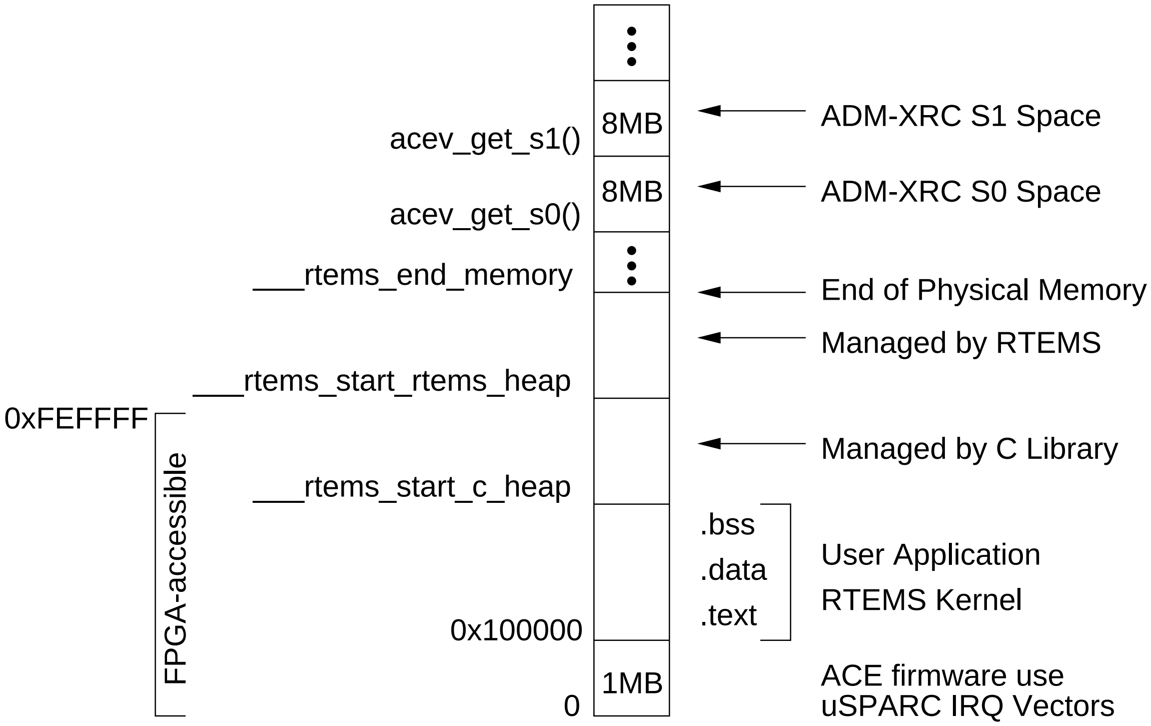


Figure 4: Simplified ACEV Memory Map

5.1 Low Memory

The bottom 1MB of the memory range holds the microSPARC interrupt vectors and is used on power-up by the TSI-provided ACE firmware.

5.2 Application Code and Data

User programs are loaded above 0x100000 (1MB). Generally, these programs will consist of a binary image holding both the RTEMS kernel and the actual user application. In this image, `.text` is the section for executable code, `.data` holds constant-initialized data, while `.bss` holds uninitialized data (which will be zeroed on system boot).

5.3 C Library Heap and Stack Space

Above these regions, the heap managed by the C library functions (`malloc()` and friends) begins. Note that the stack for each thread will also be allocated from this region. The startup-thread (in which `main()` executes has a stack of 128KB by default. If an application requires more stack space, the simplest solution is to start its main function in another thread. This can be achieved using the RTEMS function `rtems_task_create` (see [12], Section 4.4.1), which accepts the size of the thread's stack space as an input parameter. The initial task can then sleep until the new main task completes.

5.4 FPGA-Accessible Memory

The default RTEMS/ACEV configuration *guarantees* that all of the structures listed above (low memory, application code and data, C library heap, and all stacks) lie within the 16MB-64KB window of 'blessed' FPGA-accessible memory (also see Section 3) starting at address 0. By doing so, all microSPARC pointers within that region (to global or local variables or dynamically allocated memory) can also be passed to the RPU for access. Conversely, the size of this region will always be limited to 16MB-64KB *regardless* of the amount of physical memory actually present on the ACE-card. If more memory is available, it will be managed as part of the RTEMS workspace (also known as the RTEMS heap, see next section). It is possible to move the blessed window in 16MB increments using an ACEV API function (see Section 6.2.4) into this higher memory region. However, it will then be the user's responsibility to ensure that all pointers passed to the RPU are expressed as *offsets* originating from the actual physical start address of the 'blessed' window.

5.5 RTEMS Workspace

All physical memory above 16MB-16KB (above the default 'blessed' window) is allocated to the RTEMS workspace. The OS allocates its internal data structures (e.g., task control blocks, queues, etc.) from this region. Additionally, however, a user program also request an allocation from the workspace. This can be used for large data structures that will not need to be accessed by the RPU (e.g., profiling logs, large I/O buffers, etc.). The API describes next handles access to the workspace:

Include the following headers in your source code:

```
#include <rtems.h>
#include <rtems/score/wkspace.h>
#include <rtems/score/wkspace.inl>
```

This requires that you have added a

```
-I $(RTEMSACE)/sparc-rtems/include/rtems-ace2
```

to the compile command in your Makefile. We assume here that **RTEMSACE** is an environment variable that points to your base RTEMS/ACE install directory (the one containing **bin/**, **lib/**, **sparc-rtems/**, etc.).

Workspace memory is then allocated using the call

```
void *_Workspace_Allocate(unsigned32 size);
```

This function will return NULL if the request cannot be fulfilled. Blocks are released by

```
boolean _Workspace_Free(void *block_to_free);
```

This will return TRUE on success and FALSE if the block could not be freed.

Note that these routines use rather simplistic allocation algorithms. If you have many requests to make, just allocate one large block using these facilities and manage it yourself by handing the block to the

RTEMS Partition Manager : to manage fixed-size requests

RTEMS Region Manager : to manage variable-size requests

See the [12], Chapters 12 and 13, for further info on these services.

6 ACEV API

The ACEV API is a set of routines that hides the hardware-specific details ACEV hardware behind a small and easy to use interface. It deals with the following issues:

Initialization : Attach the ADM-XRC card to the local PCI bus on the ACEcard.

Addressing : Retrieves the addresses of various memory ranges.

FPGA Configuration : Load hardware designs into the Virtex FPGA. Note that the interface supports compressed configurations created using the `bit2o` tool (see Section 8.1).

Clock Programming : Set the ADM-XRC local bus clock to a given frequency.

IRQ Handling : Set-up and register a handler for Virtex IRQs.

Synchronization : Manage the FPGA as a shareable resource between different (or within the same) threads.

The following sections will describe the functionality of the specific calls. All of the prototypes, constants, and data structures are defined in the include file `acevapi.h`.

Many routines return negative status codes on failure. As shown in Table 1, four ranges are defined for these codes.

See `acevapi.h` for the macros `LZOERR` and `RTEMSERR` that describe how the various actual error codes (occurring in subroutines) are mapped to this common range.

6.1 Initialization

During initialization, the ACEV API maps the address ranges on the ADM-XRC card into the local PCI address space and

Range	Description
$c > 0$	No error (user defined value)
$c = 0$	No error
ACEV_RTEMS_ERR $< c <$ ACEV_LZO_ERR	Decompression error
$c <$ ACEV_RTEMS_ERR	RTEMS error

Table 1: Status code ranges

then programs the microSPARC PCI controller to map these PCI addresses into its own address range.

6.1.1 acev_init

Interface :
```
int
acev_init()
```

Function : Initialize the ACEV API and map an attached ADM-XRC daughter-board into the EPS address space. Install a default handler for interrupts that just prints a message and then exits. Finally, enable interrupts from the FPGA to the microSPARC. This function **must** be called before any other ACEV API calls are used.

Diagnostics : ACEV_FPGA_INV – No ADM-XRC daughter-board found
RTEMS – Could not create synchronization semaphore.

Source Code : `acevinit.c`

6.2 Addressing

To flexibly allow future expansions, many of the address ranges described in Section 5 are not hard-coded to fixed addresses, but are allowed to remain variable. For the most commonly used of these ranges, the current start addresses and extents can be retrieved using ACEV API calls.

6.2.1 acev_get_s0

Interface :
```
volatile void *
acev_get_s0(size_t *length)
```

Function : Return the start address of the ADM-XRC S0 space (see Section 4.1 and [6], Section 4). Optionally, the length of S0 in bytes may be retrieved by passing in a pointer to a `size_t` variable. If this is not desired, `NULL` may be passed in as a pointer.

Diagnostics : None.

Source Code : `acevinit.c`

6.2.2 acev_get_s1

Interface : `volatile void *`
`acev_get_s1(size_t *length)`

Function : Return the start address of the ADM-XRC S1 space (see Section 4.1 and [6], Section 4). Optionally, the length of S1 in bytes may be retrieved by passing in a pointer to a `size_t` variable. If this is not desired, `NULL` may be passed in as a pointer.

Diagnostics : None.

Source Code : `acevinit.c`

6.2.3 acev_get_masterbase

Interface : `volatile void *`
`acev_get_masterbase(size_t *length)`

Function : Return the start address of the ‘blessed’ memory range in EPS DRAM which is also accessible by the FPGA (see also Sections 3 and 4.2.2). Optionally, the length of this region in bytes may be retrieved by passing in a pointer to a `size_t` variable. If this is not desired, `NULL` may be passed in as a pointer.

Diagnostics : None.

Source Code : `acevinit.c`

6.2.4 _acev_set_masterbase

Interface : `void`
`_acev_set_masterbase(void *base)`

Function : Set the start address of the ‘blessed’ memory range in EPS DRAM which is also accessible by the FPGA (see also Sections 3 and 4.2.2). Note that this window may currently only be moved in 16MB increments. Be sure that it lies entirely in physical memory: FPGA accesses to non-existent addresses may hang the entire system (including the host!). **Caution:** All default address mappings in the ACEV tools assume that the window is located at address 0. Your local variables, data segment, and the C heap will no longer be accessible to the FPGA if you move the window to a different address!.

Diagnostics : None.

Source Code : `acevinit.c`

6.3 FPGA Configuration

The ACEV API hides the intricacies of the FPGA configuration process (which may require delay loops at certain point to ensure protocol compliance). To the user, the API offers two simple calls that differ only in the source of the configuration data to load.

One call accepts the path to a Xilinx `.bit` file on the host. The `.bit` format is the default output format of the standard Xilinx design tools and can thus be easily created. For larger chips, however, `.bit` files quickly become unwieldy: The size of a Virtex XCV1000 configuration is 765968 bytes (plus some header data in the `.bit` file). While one could embed these files into a user program (e.g., as large static arrays), the required space (especially when multiple configurations are considered) becomes unacceptable. The same applies to the compile time of the resultant C code commonly generated from the `.bit` files.

As an alternative, the ACEV API allows the use of compressed configurations that are converted directly into linkable ELF object files. In this manner, the size of each configuration may be reduced tremendously: While it is dependent on the complexity of the design actually contained in the `.bit` file, simpler designs can compress down to a few KBs. By directly writing ELF files, the lengthy compile phase can also be omitted. Compression and ELF conversion is handled as a single operation using the `bit2o` program (described in Section 8.1). During this step, a symbolic name for the compressed configuration is defined that is used at run-time to refer to the data.

6.3.1 acev_load_file

Interface : `int acev_load_file(char *bitfilename)`

Function : Load the `.bit` file named by the path `bitfilename`. Note that this path refers to a file on the host. Relative paths describe a file relative to the current directory of the `rtemsserver` program executing on the host (see Section 8.2)

Diagnostics : `ACEV_BITLOAD_FAIL` – Could not open `.bit` file.
`ACEV_BITFILE_INV` – Not a standard `.bit` file.
`ACEV_BITFPGA_INV` – `.bit` unsuitable for FPGA part.
`ACEV_DECOMP_FAIL` – Bitstream has invalid length.
`ACEV_CONFIG_FAIL` – Configuration download failed.

Source Code : `acevload.c`

6.3.2 acev_load_config

Interface : `int acev_load_config(acev_ELF_bitstream *configdata)`

Function : Load the compressed configuration pointed to by `configdata`. In general, you will use the symbolic name passed to `bit2o` during the conversion process as `configdata` here.

Diagnostics : Decompression errors – LZO package failed.
`ACEV_BITOBJ_INV` – Not a valid bitstream object.
`ACEV_DECOMP_FAIL` – Bitstream has invalid length.
`ACEV_CONFIG_FAIL` – Configuration download failed.

Source Code : `acevload.c`

6.4 Clock Programming

The VCLK programmable clock generator on the ADM-XRC card (see [6], Section 5.2) may be programmed to a frequency between 25MHz and 40MHz. The lower limit is due to the inability of the Virtex DLLs to lock at lower frequencies. The upper limit is imposed by the maximum PLX 9080 [10] local bus frequency. Currently, the ACEV API supports only a single clock. The optional ADM MCLK may become supported at a later time. The default VCLK frequency is 28.5 MHz.

6.4.1 acev_set_clock

Interface :	`int`
	`acev_set_clock(double freq)`

Function :	Set the ADM-XRC system clock (local bus and FPGA clock) to the frequency in Hz given by `freq`.
Diagnostics :	`ACEV_FREQ_INV` – Frequency outside valid range. `ACEV_CLOCK_ERR` – Clock programming failed.
Source Code :	`acevclock.c`

6.5 Interrupt Handling

Using the ACEV API, the user program may register a special handler routine to react to interrupts caused by the FPGA (Section 4.2.4). The default handler installed by the initial `acev_init` invocation just prints a message and then exits the program.

Additional management functions are available in the API for enabling and disabling interrupts in general. As another side effect of `acev_init`, interrupts will be enabled. The following strategy is recommended for managing these interrupts:

- Do *not* disable interrupts unless absolutely required by the rest of the software environment (e.g., when hard real time constraints have to be obeyed). In general, it is more beneficial to react to all interrupts, even if only using the default handler: Since an unexpected interrupt often points to deeper problems within a given application, IRQs should be kept observable in this manner.

- The interrupt handler routine is responsible for turning-off the *cause* of the interrupt on the FPGA. The nature of the mechanism itself is application-specific. Examples include writing to or reading from a specific S0 address on the FPGA. In this manner, the FPGA receives an acknowledge from the microSPARC and can now de-assert the interrupt line.

- This explicit deactivation of interrupts at the source is different and preferable to the alternative of relying on the API routines to just *mask* an IRQ after its arrival. Since these calls have to set registers at multiple locations in the interrupt path to mask/unmask interrupts, their execution can take considerably longer than the explicit deactivation by just a single S0 access.

The ACEV API will always flush the data cache *before* calling the user-registered handler. This ensures that the handler code will see all master-mode writes the FPGA might have performed to DRAM.

6.5.1 acev_irq_enable

Interface : `void`
`acev_irq_enable()`

Function : Enable interrupts from the ADM-XRC to the microSPARC and clear existing interrupt indicators in the various registers. Note that this *cannot* clear the cause of an interrupt on the FPGA. This function is called automatically in `acev_init`.

Diagnostics : None.

Source Code : `acevinit.c`

6.5.2 acev_irq_disable

Interface : `void`
`acev_irq_disable()`

Function : Disable interrupts from the ADM-XRC to the microSPARC and clear existing interrupt indicators in the various registers. Note that this *cannot* clear the cause of an interrupt on the FPGA.

Diagnostics : None.

Source Code : `acevinit.c`

6.5.3 acev_irq_handler

Interface : `void`
`acev_irq_handler(acev_irq_fn ifn, acev_irq_fn *old)`

Function : Register a new function `ifn` as a handler for ADM-XRC interrupts. This function must have a prototype of the form **`void handler()`**. Optionally, the current IRQ handler function can be retrieved by passing in an appropriate pointer in `old`. `NULL` may be passed in here to just overwrite the handler with the new one. Note: RTEMS I/O operations (`printf()` etc.) are not guaranteed to execute correctly when called from an interrupt routine and should be avoided!

Diagnostics : None.

Source Code : `acevinit.c`

6.6 Synchronization

The ACEV API offers three functions dealing with synchronizing the hardware and software operations. They can be used within a thread as well as to coordinate access to the FPGA from multiple RTEMS threads.

6.6.1 acev_mark_busy

Interface : `int`
`acev_mark_busy()`

Function : Request access to the FPGA for this thread. This should be called before the first write to or read from the FPGA is executed.

Diagnostics : ACEV_BUSY – FPGA is already in use.
RTEMS error – Problems with the semaphore.

Source Code : `acevinit.c`

6.6.2 acev_mark_done

Interface : `int`
`acev_mark_done()`

Function : Relinquish access to the FPGA and wake threads waiting for the completion of an FPGA operation. This may be called from an IRQ handler to restart threads that sleep using the `acev_wait` call.

Diagnostics : RTEMS error – Problems with the semaphore.

Source Code : `acevinit.c`

6.6.3 acev_wait

Interface :
```
int
acev_wait()
```

Function : Wait for an FPGA operation to complete. An operation is deemed complete once `acev_mark_done` has been called (possibly in the IRQ handler task, reacting to an interrupt by the FPGA). RTEMS will continue to schedule other threads (if any exist) while waiting, a sleeping thread will thus be prevented from hogging the entire system (as would occur with a polling 'busy wait' loop).

Diagnostics : RTEMS error – Problems with the semaphore.

Source Code : `acevinit.c`

7 Design Flow

Due to the hybrid nature of the ACEV platform, the design flow includes both hardware and software elements. Figure 5 shows the major steps.

7.1 Software

The software branch is shown on the left side. Most of it consists of a standard compile-assemble-link cycle, but with the following extensions:

- Since the ACEcard firmware is not able to load ELF-format executables (which are created by the linker), the ELF executable files have to be converted into a simple binary format using the `objcopy` tool of the GNU binutils.

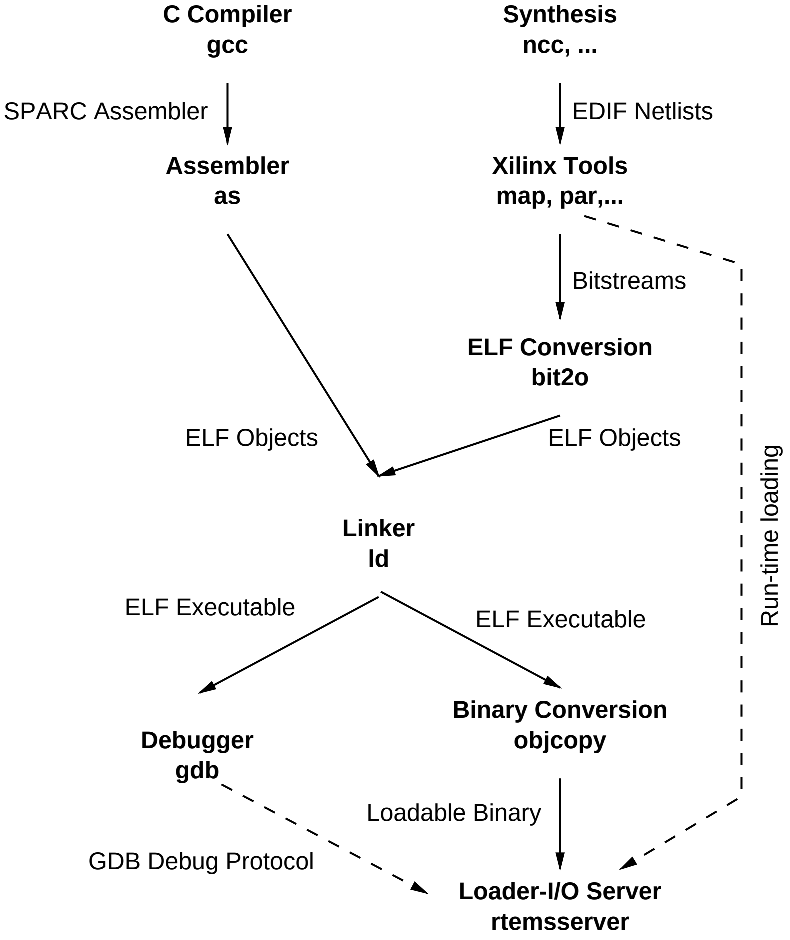


Figure 5: ACEV Design Flow

- However, these binary files no longer contain the symbol information required for debugging. Thus, the original ELF executable is loaded into GDB using the `sym` command to access the original (pre-binary conversion) symbol information.
- Since the ACEV program no longer runs on the host, but on the ACEcard EPS, GDB has to be put in remote debugging mode using the

  ```
  target remote /dev/ace0debug
  ```

  command.

7.2 Hardware

The right hand side of Figure 5 shows the hardware part of the ACEV flow. Hardware will commonly be entered in an HDL such as Verilog or VHDL and processed using logic synthesis. Sometimes, an even higher-level description such as C might be compiled. In both cases, the synthesis tool / compiler produces an EDIF netlist that is fed into the Xilinx M-series design implementation tools (`ngdbuild`, `map`, `par`, `bitgen`, use versions M2.1iSP6 or M3.1iSP1 or later). The resultant bitstream in `.bit` format may then be loaded into the FPGA at run-time (useful for debugging, see the description of `acev_load_file` in Section 6.3.1) or be processed further (recommended for production work). In that case, the `.bit` file is compressed and turned into linkable ELF-object file that will become part of the final binary during linking. These compressed and linked configurations are loadable using the function `acev_load_config` (Section 6.3.2).

7.3 Run-Time

The binary-format executable is then loaded using the host-side `rtemsserver` program (see also [11] and Section 8.2), which acts both as a loader as well as the server for I/O requests (access to the host file system, stdin, stdout, stderr, etc.). Using suitable options, `rtemsserver` can also load the program in debug mode and allows GDB (running on the host) to stop the binary before its first instruction. From that point on, the program can be source-level debugged using conventional techniques using single-stepping, break points, data displays etc.

8 Tools

The design flow described in Section 7 uses some custom tools that are described briefly below.

8.1 bit2o

This program converts Xilinx bit-streams in `.bit` format into the directly linkable ELF object format. Furthermore, it also compresses them using the LZO compression algorithm ([15]). The command line for bit2o looks like:

```
bit2o [-v] infile.bit outfile.o symbol_name
```

`infile.bit` is the input `.bit` file, `outfile.o` will hold the generated ELF object code, and `symbol_name` will be the name by which this bit-stream can be referred to in an `acev_load_config` call (see Section 6.3.2). The optional '-v' switch provides more verbose messages.

8.2 rtemsserver

The RTEMS program loader and host I/O server is described in detail in [11]. For reference, a brief description of the major options is shown in Figure 6.

For ease of use on the ACEV platform, you should set the environment variable `ACE2PREFIX` to `ace` (otherwise, the `-Xace` switch becomes mandatory). With the possible exception of the 'debug mode' switch, no other command options should be required. A sample call to run the program assumed to be named `test.bin` (which should have been converted from the ELF executable to binary executable using `objcopy`) on the ACEV might look like:

```
rtemsserver test.bin
```

For debugging, run '**`rtemsserver -d test.bin`**' in one window to load and stop the program on the EPS, and the type

```
host$ gdb
(gdb) map test.exe
(gdb) target remote /dev/ace0debug
```

```
rtemsserver [-c<cardnum>] [-d] [-i] [-v] [-P<path>] [-X<prefix>] [-e<entry>] <file>
```

-c<cardnum>	Selects card numbered <cardnum>. Default=0 Also set using variable ACE2CARDNUM.
-d	Set debug mode for GDB: breakpoint at start of program.
-i	Ignore following switches and pass them to program.
-v	Verbose mode.
-P<path>	Set local current directory on the host to <path>.
-X<type>	Select card type. Valid=ace,aceII. Default=aceII. Also set using variable ACE2PREFIX.
-l	Do not create lock files for device.
-e<entry>	Set program entry point address. Default=0x00100000.
<file>	The program binary to run on the uSPARC.

Figure 6: `rtemsserver` command options

in another. The first command loads the debug symbols from the original ELF executable (assumed to be named `test.exe`, not processed by `objcopy`), the second command attaches GDB to the stopped program on the EPS. From here on, you can use the standard GDB commands. Note: Since your program is already *running* on the EPS, use the 'c' (continue) command to resume execution. The 'r' (run) command will not work!

8.3 LPWB

`LPWB` (the acronym stands for Local Programmers Workbench) is a simple debugging tool which runs locally on the EPS. It can examine and modify memory (and thus also the S0 space holding the mapped FPGA registers).

LPWB commands refer to a previously set base-address. This base address is set by just typing it on the command line. LPWB will respond by printing the current contents of that memory location. Just pressing Enter here will return you to the main prompt and leave the value unchanged. A valid hex number will be interpreted as a write request to the previously set memory address, overwriting the old value. Both at the main prompt and at the value prompt (just described), commands may be entered. E.g., 'l' will

```
[ace] (˜/Work/Projects/ACE/LPWB) 1 % ./lpwb.run
Local PWB for RTEMS/ACE v1.3 by EIS/TSI/Synopsys (Aug  4 2000 15:24:06)
ADM-XRC attached: PLX@30100800 S0@31000000 S1@31800000
LPWB> v
Enter name of .bit file to load:
../acev/irq-V1000.bit
Configuration complete.
LPWB> 31000000                    <--- this is the S0 start shown above
31000000> 12345678 04000000       <--- write the countdown start value
LPWB> p                           <--- show the value counting down
31000000> 018b2195 018b192a 018b14b9 018b108f  ..!....*...9....
31000010> 014e0bd8 014e073d 014e0324 014dff21  .N.X.N.=.N.$.M.!
31000020> 0110f8cc 0110f417 0110eff7 0110ebdf  ..xL..t...ow..k_
31000030> 00d3eb91 00d3e6fc 00d3e302 00d3df09  .Sk..Sf|.Sc..S_.
31000040> 0096de60 0096d9d9 0096d5d2 0096d1c2  ..ˆ‘..YY..UR..QB
31000050> 0059d1ac 0059cd25 0059c922 0059c525  .YQ,.YM%.YI".YE%
*** Unhandled interrupt by ADM FPGA, exiting ... ***
[ace] (˜/Work/Projects/ACE/LPWB) 2 %
```

Figure 7: Sample LPWB session

disassemble the next 16 instructions starting at the base address. Longer memory dumps can be printed using the ’p’ command, which shows the next 64 words. The ’v’ command can be used to load a Xilinx `.bit` file into the Virtex FPGA. The ’r’ command reconfigures the FPGA into a safe default configuration that just maps all of the FPGA-local memory into S0 space. At the start of LPWB, the current address ranges for S0 and S1 are displayed.

A sample LPWB session is shown in Figure 7. The sample FPGA design (see Section 10) loaded just maps a single 32b register into S0 space. On start-up, that register contains the value 0x12345678. A write to the register will result in a countdown that fires an IRQ on reaching zero. Since LPWB only uses the default IRQ handler, it will exit after receiving the interrupt.

9 Sample Makefile

Figure 8 shows a sample Makefile that might be used for an RTEMS/ACEV application. Note the different stages:

1. The hardware design is translated from `.bit` format

into the linkable ELF object format and compressed using the `bit2o` tool.

2. The main program is compiled and linked with the hardware to form an ELF executable program (extension `.exe`).
3. The ELF executable is translated into a binary program for loading onto the EPS (extension `.bin`).
4. The binary program is wrapped to allow automatic loading from the command line using the shell interpreter '`#!`' mechanism (extension `.run`).
5. The resulting program is then loaded onto the EPS and connected to the I/O server by simply running the command `acevtest.run` from the host shell prompt.

```
# the root of the RTEMS/ACEV install tree
# (containing bin, sparc-rtems etc.)
RTEMSACE=/acs/ace/rtems

# the target program and the included target hardware
PROG=acevtest
HARDWARE=irq-V1000.o

# for running the tools
RTEMSBIN=$(RTEMSACE)/bin

# the RTEMS/ACE2 IO server
SERVER=$(RTEMSBIN)/rtemsserver

# the compile-flow tools
CC=$(RTEMSBIN)/sparc-rtems-gcc
LD=$(RTEMSBIN)/sparc-rtems-gcc
COPY=$(RTEMSBIN)/sparc-rtems-objcopy
BIT2O=$(RTEMSBIN)/bit2o

# the compile options
CCOPTS= -fasm -specs bsp_specs -qrtems -O3 -I include \
        -I $(RTEMSACE)/sparc-rtems/include/rtems-ace2
LDOPTS= -fasm -specs bsp_specs -qrtems -O3 -Xlinker -Map -Xlinker acevtest.map
COPYOPTS=-O binary

# make rules
default: all

# build the ELF-format executable
$(PROG).exe:  $(PROG).o $(HARDWARE)
        $(LD) -o $(PROG).exe $(PROG).o $(HARDWARE) $(LDOPTS) -lm
.c.o:
        $(CC) -c $(CCOPTS) $<

# convert the .bit FPGA design into ELF-linkable format
$(HARDWARE).o: $(HARDWARE).bit
        $(BIT2O) $(HARDWARE).bit $(HARDWARE).o $(HARDWARE)

# convert the ELF-executable into straight binary format
$(PROG).bin: $(PROG).exe
        $(COPY) $(COPYOPTS) $(PROG).exe $(PROG).bin

# wrap binary into format for direct running from the command line
all: $(PROG).bin
        echo "#!$(SERVER)" | cat  - $(PROG).bin > $(PROG).run
        chmod a+x $(PROG).run
        @echo "OK, ACE2 RTEMS executable for $(PROG) built. Run it using \n\t./$(PROG).run"

# clean up
clean:
        rm $(PROG).exe $(PROG).bin $(PROG).run $(PROG).o
```

Figure 8: Sample RTEMS/ACEV Makefile

10 Sample Hardware

This section shows the simple IRQ-firing hardware that was already demonstrated in the description of LPWB (Section 8.3). Note that the placement of I/O pins occurs using an extra Xilinx user constraint file (extension `.ucf`). A sample of such a file is shown in Section 11. See citeadm for a complete description of all pins.

```
//
//  irq.v
//
//  Example program for ADMXRC / any Virtex
//
//  set up a register in S0 space that holds the magic number 0x12345678
//  on startup. A write to the register will start a countdown from the
//  value written. On reaching zero, an interrupt will be fire to the EPS
//  which can be turned off by any write to the register
//
//  Andreas Koch <koch@eis.cs.tu-bs.de>, June 2000
//

module irq(
  LCLKA,     // system clock

  LRESETOL,  // system reset

  LWRITE,    // write indicator
  LADSL,     // address strobe (starts address phase)
  LBLASTL,   // end-of-access indicator
  LBTERML,   // terminate access (unused)
  LD,        // local 32b data bus
  LA,        // local 24b address bus
  LREADYIL,  // transaction complete acknowledge
  LBEL,      // bus enables

  LINTIL     // interrupt request signal

);

parameter LOGIC0 = 1’b0;
parameter LOGIC1 = 1’b1;

input         LCLKA;
input         LRESETOL;
input         LWRITE;
input         LADSL;
input         LBLASTL;
output        LBTERML;
inout [31:0]  LD;
input [23:2]  LA; // last 2b of address are in byte enable lines

output        LREADYIL;
input [3:0]   LBEL;

output        LINTIL;

wire          CLK;

// flip polarity for better readability
```

```
wire          RESET = ˜LRESETOL;
wire          ADS = ˜LADSL;
wire          BLAST = ˜LBLASTL;
wire  [3:0]   LBE = ˜LBEL[3:0];
wire          READ = ˜LWRITE;

// status register: 1 on ‘FPGA has been addressed’
reg           ADDRESSED;

reg [31:0] data;          // down counter
reg        irqon;         // current IRQ status
reg        fire;          // fire the IRQ on counter=0?

//
// the interrupt status
//
assign  LINTIL = !irqon;

//
//  During read, output ‘data’ onto local bus, otherwise float.
//  We write our two status registers as MSBs.
//
assign  LD = (ADDRESSED & READ ) ?
              data
              : 32’hzzzzzzzz;

//
//  Only drive READY when addressed, otherwise float because
//  the control logic on the XRC also drives READY.
//
assign  LREADYIL = ADDRESSED ? 1’b0 : 1’bz;

//
//  Never assert BTERM but drive it high
//  (BTERM can act as READY in burst operation)
//
assign  LBTERML = 1’b1;

//
//  Insert the STARTUP module to issue reset globally
//
STARTUP_VIRTEX ChipStartup( .GSR( RESET ) );

//
//  Define the system clocks, derived from local bus clock
//
IBUFG IBUFG_LCLK( .I(LCLKA), .O( LCLKA_I) );
//
//  Un-comment the following to use DLL’s
//  You may need to turn on the Verilog preprocessor
//
```

```
// ‘define USEDLL

//
//  Using the DLL will minimise skew and improve performance
//  Ensure Service Pack 4 is used to compile this though
//
‘ifdef USEDLL
CLKDLL CLKDLL_LCLK(
        .CLKIN( LCLKA_I),
        .CLK0( LCLK0 ),
        .CLKFB( CLK ),
        .RST( LOGIC0 ),
        .LOCKED( LCLK_LOCKED )
        );

BUFG  BUFG_LCLK( .I( LCLK0 ), .O( CLK ) );


‘else

BUFG  BUFG_LCLK( .I( LCLKA_I ), .O( CLK ) );

‘endif

//
//  Decode the MSB of address to get the FPGA space
//  Bursting is allowed with this decode
//  BLAST disables operation
//
always @ (posedge CLK or posedge RESET )
begin
  if( RESET )
    // we start up as ‘non-addressed’
    ADDRESSED <= 1’b0;
  else
  begin
    if( ADS )
      // a new address phase begins ...
      if( LA[23] == 1’b0 )
        // assume we are addressed in the entire lower 4MB range
        ADDRESSED <= 1’b1;
      else
        // upper 4MB must be someone else ...
        ADDRESSED <= 1’b0;

    if( BLAST & ADDRESSED )
      // we were addressed, but this was the end of the transaction
      // so become non-addressed again
      ADDRESSED <= 1’b0;
  end
end
```

```

//
// interrupt and downcounter logic
//

  always @(posedge CLK or posedge RESET) begin
    if (RESET) begin
      // start off
      fire  <= 0;             // unarmed, wait for first write
      irqon <= 0;             // no interrupt requested
      data  <= 32’h12345678;  // magic number for debugging
    end else if ( ADDRESSED & LWRITE) begin
      // someone has written data to us
      if (irqon) begin
        // the IRQ was already on, just turn it off now
        irqon <= 0;
      end else begin
        // we had now previous IRQ, arm countdown
        fire <= 1;
      end
      // read data from local bus into register
      data <= LD;
    end else if (data == 0 && fire) begin
      // we are armed and our counter is zero -> fire IRQ
      irqon <= 1;
      fire <= 0;
    end else if (fire) begin
      // we are armed, run countdown
      data <= data - 1;
    end
  end

endmodule
```

11 Sample Pin Assignments

The following Xilinx user constraint file shows how to assign physical pins to the ports used in the top-level hardware modules.

```
###########################################################
#                                                         #
#   CLOCK PINS FOR Local Bus and SSRAM’s                  #
#                                                         #
###########################################################
NET "LCLKA"   LOC = "A17";
###########################################################

###########################################################
#                                                         #
#   Local Bus Address, Data and Control                   #
#                                                         #
###########################################################
NET "LD[31]" LOC = "H31";
NET "LD[30]" LOC = "K29";
NET "LD[29]" LOC = "H32";
NET "LD[28]" LOC = "J31";
NET "LD[27]" LOC = "K30";
NET "LD[26]" LOC = "H33";
NET "LD[25]" LOC = "L29";
NET "LD[24]" LOC = "K31";
NET "LD[23]" LOC = "L30";
NET "LD[22]" LOC = "J33";
NET "LD[21]" LOC = "M29";
NET "LD[20]" LOC = "L31";
NET "LD[19]" LOC = "M30";
NET "LD[18]" LOC = "L32";
NET "LD[17]" LOC = "M31";
NET "LD[16]" LOC = "L33";
NET "LD[15]" LOC = "N30";
NET "LD[14]" LOC = "N31";
NET "LD[13]" LOC = "M32";
NET "LD[12]" LOC = "P29";
NET "LD[11]" LOC = "P30";
NET "LD[10]" LOC = "P31";
NET "LD[9]"  LOC = "P32";
NET "LD[8]"  LOC = "R29";
NET "LD[7]"  LOC = "R30";
NET "LD[6]"  LOC = "R31";
NET "LD[5]"  LOC = "R33";
NET "LD[4]"  LOC = "T31";
NET "LD[3]"  LOC = "T29";
NET "LD[2]"  LOC = "T30";
NET "LD[1]"  LOC = "T32";
NET "LD[0]"  LOC = "U31";
NET "LA[23]" LOC = "D28";
#NET "LA[22]" LOC = "C30";
#NET "LA[21]" LOC = "D29";
#NET "LA[20]" LOC = "E28";
#NET "LA[19]" LOC = "D30";
#NET "LA[18]" LOC = "F29";
#NET "LA[17]" LOC = "D31";
#NET "LA[16]" LOC = "F30";
#NET "LA[15]" LOC = "C33";
#NET "LA[14]" LOC = "G29";
#NET "LA[13]" LOC = "E31";
#NET "LA[12]" LOC = "D32";
#NET "LA[11]" LOC = "G30";
#NET "LA[10]" LOC = "F31";
#NET "LA[9]"  LOC = "H29";
#NET "LA[8]"  LOC = "E32";
#NET "LA[7]"  LOC = "E33";
#NET "LA[6]"  LOC = "G31";
#NET "LA[5]"  LOC = "J29";
#NET "LA[4]"  LOC = "F33";
#NET "LA[3]"  LOC = "G32";
#NET "LA[2]"  LOC = "J30";

#NET "LBEL[3]" LOC = "D27";
#NET "LBEL[2]" LOC = "B30";
#NET "LBEL[1]" LOC = "C29";
#NET "LBEL[0]" LOC = "AL18";

NET "LRESETOL" LOC = "AM18";
NET "LADSL"    LOC = "C28";
NET "LWRITE"   LOC = "E26";
NET "LBLASTL"  LOC = "C27";
NET "LBTERML"  LOC = "E25";
NET "LREADYIL" LOC = "A28";
NET "LINTIL"   LOC = "D25";

##########################################################
#                                                        #
#   Pin performance attributes                           #
#                                                        #
##########################################################

NET "LD[*" FAST;
#NET "LA[*" FAST;
NET "LA[23]" FAST;
NET "LREADYIL" FAST;
```

12 Troubleshooting

12.1 Host VSP device busy

Under some circumstances, abnormal termination of an EPS program leads to the stall of host programs that were using VSPs to connect to the ESP. Examples include `tip` connected to `/dev/ace0ttya` for watching trace messages, or GDB used on `/dev/ace0debug` for single stepping a program.

These programs apparently cannot be killed (even using the '`-9`' option) since they are held in a blocking read from the VSPs without anything alive at the other side which could actually produce the data. The solution is to load a small program into the EPS (using TSI's `acedownload` program) that simply generates output (1024 bytes seem to work well) on all four VSPs. The hanging host programs will then awaken and notice that they have been killed in the meantime. Such a VSP-blasting EPS program might be included in your distribution as the file `smack.bin`.

References

[1] Koch, A., "A Comprehensive Prototyping-Platform for Hardware-Software Codesign", *Proc. IEEE Workshop on Rapid Systems Prototyping (RSP)*, Paris, 06/2000

[2] TSI TelSys Inc., "ACEcard User's Manual", *vendor documentation*, 02/1998. 1

[3] TSI TelSys Inc., "ACEcard Hardware Designer's Manual", *vendor documentation*, 02/1998

[4] TSI TelSys Inc., "ACEcard Software Developer's Manual", *vendor documentation*, 02/1998

[5] Sun Microelectronics, "microSPARC-IIep User's Manual", *device manual*, 04/1997 1, 3

[6] Alpha Data Parallel Systems, "ADM-XRC PCI Mezzanine Card User Guide", *vendor documentation*, 1999 1, 4.1, 4.1, 4.2, 4.2, 6.2.1, 6.2.2, 6.4

[7] Alpha Data Parallel Systems, "ADM-XRC Bus Master Application Note", *application note AN-XRC02*, 1999

[8] Xilinx, "Virtex 2.5V Field Programmable Gate Arrays", *device datasheet DS003*, 1/2000 1

[9] GSI Technology, “GS880Z36T 8Mb Pipelined and Flow Through”, *device datasheet*, 1/2000 1

[10] PLX Technology, “PCI 9080 Data Book”, *device datasheet 9080-DB-015*, 9/1998 1, 4.2, 4.2.1, 4.2.1, 4.2.1, 4.2.2, 4.2.2, 6.4

[11] Rock, M., “Porting RTEMS to the ACE Platform”, *Tech. Univ. Braunschweig student project documentation*, 07/2000 2.2.2, 7.3, 8.2

[12] On-Line Applications Research Corporation, “RTEMS C User’s Guide, *software manual*, 10/1998 2.2.2, 5.3, 5.5

[13] On-Line Applications Research Corporation, “RTEMS Posix User’s Guide, *software manual*, 10/1998 2.2.2

[14] Weaver, D., Germond, T., “The SPARC Architecture Manual”, Prentice-Hall, 11/1993 1, 2.2.1, 3

[15] `http://wildsau.idv.uni-linz.ac.at/mfx/lzo.html` 8.1

Appendix B. A Comprehensive Prototyping-Platform for Hardware-Software Co-Design

A Comprehensive Prototyping-Platform for Hardware-Software Codesign

Andreas Koch
Tech. Univ. Braunschweig (E.I.S.), Gaußstr. 11, D-38106 Braunschweig, Germany
koch@eis.cs.tu-bs.de

Abstract

We present a flexible, yet cost-effective prototyping platform for hybrid hardware/software systems. Our approach is based on combining off-the-shelf hardware components with custom software to arrive at an encompassing solution. We address the hybrid nature by tightly coupling a conventional processor with configurable logic on a single PCI expansion card.

1. Introduction

One of the main difficulties in building and evaluating the hybrid solutions created by hardware/software codesign methods is their *systemic* nature: The designer is no longer faced with designing, implementing, and testing a single chip or a single program, but must consider the interplay between numerous interdependent hardware and software components. Simulating such a system is often not feasible because either the required simulation models are not available or the complexity of the resulting encompassing simulation model is so high that the simulation run-times themselves are no longer practical.

In many cases, the required level of detail can only be observed by actually prototyping a sufficiently large part of the system. Due to the this step being on the critical path of a product introduction, techniques for completing this phase as quickly as possible become crucial. While past technology generations could be easily tested using breadboard assemblies, this is no longer practical with current large systems. With the advent of Field-Programmable Gate Arrays (FPGAs), the current generation of prototyping systems [1] [2] is able to emulate circuits of up to 20 Mgates at a cost of $0.55 to $0.89 per gate [3].

While these emulators allow the rapid prototyping of very large systems, they are economically infeasible for smaller design teams and do not address the problem of efficiently executing the interplay between hard- and software (they lack conventional on-board processors). An approach that is far less costly, but that would still allow the seamless prototyping of such a hybrid system, is quite desirable.

2. Solution

Our solution for these requirements leverages state-of-the-art FPGA technology to reach the gate capacities needed for practical testing. This dispenses with the need of partitioning a larger circuit across a sea of smaller FPGAs and the resultant increase in complexity and speed. Thus, we can achieve logic capacities in the 1-3 Mgate range for $0.008 to $0.01 per gate.

To cover the software angle, the prototyping platform must contain a sufficiently powerful conventional microprocessor that is tightly coupled to the reconfigurable logic. In order to easily implement software on the system, code running on this CPU must have access to a full set of OS resources (e.g., C library, memory management, etc.), but must be unencumbered by OS constraints that would hinder access to the hardware (convoluted driver models, high interrupt latencies).

3. Hardware Architecture

Instead of custom designing an architecture fulfilling these requirements (as we did before, e.g., in [4] [5]), the hardware of our current prototyping environment is composed by combining two components off-the-shelf (COTS), an approach that makes it applicable to a wide variety of prototyping scenarios. Each of the separate parts adds features critical for arriving at an encompassing solution. Figure 1 shows a schematic of the major hardware features.

3.1. ACE2card

The Lavalogic (formerly known as TSI TelSys, Inc.) *ACE2*card (shown in the lower half of Figure 1) was initially designed to act as a key component in satellite communications equipment. To allow reuse of the same hardware when dealing with different communications protocols, the card offers sufficient configurable gate capacity to accommodate the timing-critical portions of a variety of protocols.

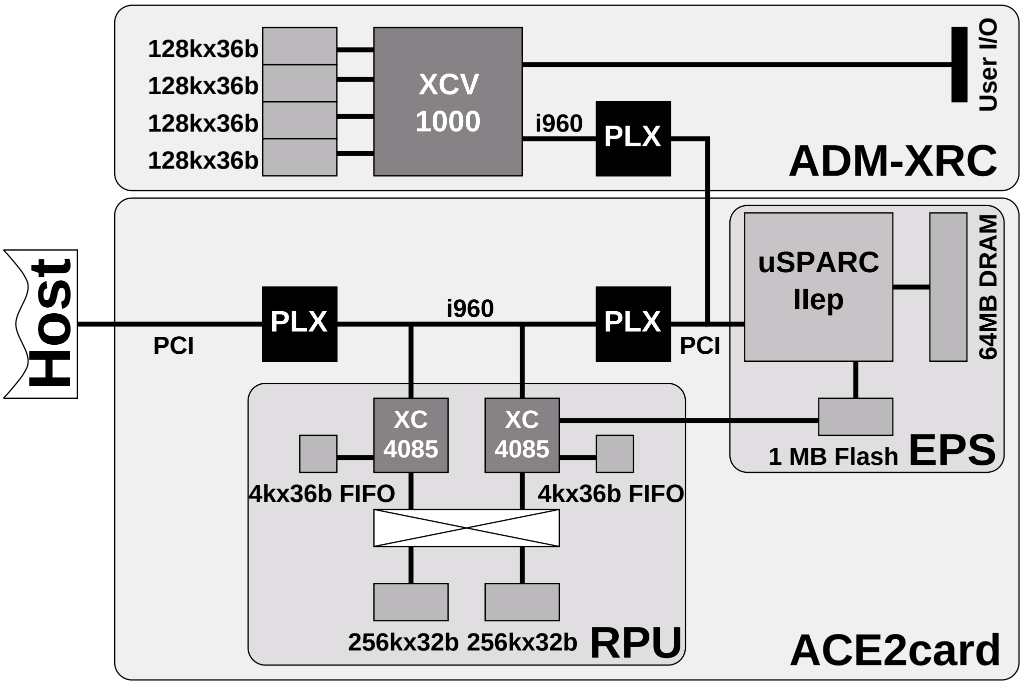


Figure 1: Hardware architecture

The major distinguishing feature between the *ACE2*card and other FPGA-supporting platforms, e.g., [6] [7], is the on-board presence of a conventional RISC processor. This tight integration allows the realistic prototyping of hybrid hardware/software solutions, unencumbered by the overhead of relying on the host computer for software execution.

One of the main components of the *ACE2*card is a conventional computer, called the Embedded Processor Subsystem (EPS). It its is based on a SUN microSPARC-IIep processor [8], an implementation of the SPARC V8 specification [9]. This RISC runs at 100 MHz and has access to 64 MB of EDO DRAM and 1 MB of user-programmable Flash memory (holding the boot firmware). Additionally, the chip also provides a 33 MHz 32-bit PCI master interface (including interrupt management), an implementation of the SPARC reference MMU, and various real-time timers and counters.

The PCI bus is used to communicate with a PMC [10] expansion connector and the on-board reconfigurable processing unit (RPU). On the *ACE2*card, this consists of two Xilinx XC4085XL [11] FPGAs having a capacity of up to 170,000 gates. Each of the FPGAs has access to a dedicated 256k x 32b bank of SRAM memory. Using a fast crossbar, the banks can be switched between the FPGAs on a per-cycle basis. Also available are four 4k x 9b FIFOs per FPGA to use as temporary buffers, e.g., in stream-based computation.

In order to simplify the user logic, the RPU and the EPS do not communicate directly over the PCI bus (complex protocol, multiplexed data and address lines, fixed clock speed). Instead, a PLX 9080 [12] PCI I/O accelerator bidirectionally converts the PCI bus to/from a non-multiplexed 32b bus similar to the one used on the Intel i960 processor. While this bus has a much simpler protocol (considerably easier to implement in user logic), it still achieves PCI performance levels. As an additional plus, the i960 bus (also called “local bus”) runs asynchronously to the PCI bus at any speed from 500 kHz to 33 MHz, thus matching the design-dependent FPGA clock-speeds to the fixed PCI clock.

The *ACE2*card attaches to the host as a full-length PCI card. The host PCI bus is converted to the local bus using another PLX 9080. By appropriately configuring the PLX registers, transparent access from the host is possible not only the RPU, but also to the EPS.

While the *ACE2*card in itself already provides a very useful platform for evaluating HW/SW codesigns (especially due to the tightly coupled processor), and is in fact already being used as target for an experimental fully automatic HW/SW compile-flow [13], it has limitations in its original form: The XC4085XL FPGAs are no longer close to the state-of-the-art in FPGA architecture. They are limited with regard to sheer logic capacity (current FPGAs reach up to 3.2 Mgates [14]) as well as to configuration speed and partial configuration ability. The last two features are important when using FPGAs as flexible compute engines. For this application, the lack of busmastering capabilities for the FPGAs on the *ACE2*card is also annoying. E.g., in the experimental compile flow, data has to be copied explicitly to the RPU SRAMs before it can be processed by the FPGAs. A more homogeneous memory model allowing the RPU direct access to the entire memory space would be desirable instead.

3.2. ADM-XRC

The Alphadata ADM-XRC card is a daughtercard following the PCI Mezzanine Card (PMC) standard [10]. It is shown in the upper half of Figure 1 and uses the *ACE2*card as a motherboard.

In contrast to the *ACE2*card, the ADM-XRC concentrates on providing a state-of-the art Xilinx Virtex or VirtexE FPGA connected to four fast memory banks. Our current configuration uses an XCV1000 FPGA with a capacity of up to 1 million gates accessing four 128k x 36b banks zero-bus latency (sometimes also called “zero-bus turnaround”) SRAMs.

As with the RPU on the *ACE2*card, the ADM-XRC relies on a PLX 9080 to convert the PCI bus to the simpler i960 bus. Now, however, the FPGA has full master access to the bus and can transparently access data residing, e.g., in the EPS DRAM. As before, this arrangement also enables the asynchronous operation of the variable-clock speed FPGA from the PCI bus. For further extension or debug connections, the ADM-XRC also offers 34 pins of user-programmable IO in the form factor of a SCSI-2 connector.

For integration with the *ACE2*card, we had to develop code (running on the EPS) that attached the ADM-XRC to the PCI bus and mapped its memory regions into the microSPARC-IIep address space.

By combining both hardware components in this fashion, we obtain an off-the-shelf platform having the strengths of the *ACE2*card (embedded processor, easy access from host), and use the ADM-XRC to compensate its weaknesses (large logic capacity, state-of-the-art FPGA, homogeneous memory model).

4. Software Architecture

Hardware is only one part of a prototyping *system*, software (while often neglected) forms the other half. When assembling our prototyping environment, this was the area that required the most effort to arrive at a usable, tightly integrated solution.

For the *ACE2*card, the vendor offers host-side drivers (Solaris and Windows NT) that map the various devices (memories, PLXs, FPGAs) into the host filesystem. E.g., the EPS DRAM can be accessed using an `open()` system call, `write()` and `read()` calls will then exchange data between the DRAM and the host.

However, for applications actually executing on the EPS, the support was far more rudimentary: Their only means of communication used the PLX9080 mailbox registers to simulate four “virtual serial ports”, which are then also mapped to host devices. While useful for debugging, these high latency and low bandwidth channels are unsuitable for any practical I/O needs. No support was included for such critical operations as FPGA access and interrupt processing on the EPS. Furthermore, not even the basic C library functions (memory management, math, signal handling, etc.) were available. These restrictions severely hindered the porting of conventional C code to the EPS.

Since the ADM-XRC is intended as a general purpose extension to all PMC-compatible environments, it did not include any *ACE2*card-specific software. Only small C fragments illustrating the transformation of Virtex bitstreams into a downloadable form and the actual configuration sequence were provided.

Our approach to removing these limitations is described in the next sections and illustrated in Figure 2.

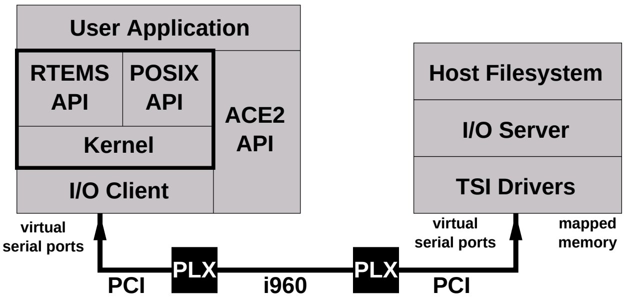


Figure 2: Software architecture

4.1. RTEMS

As a first step in presenting EPS applications with a more familiar and complete run-time environment, we ported the RTEMS operating system [15] to the EPS. RTEMS is a preemptive multi-threading real-time operating system freely available for a wide variety of boards and processors. It is sufficiently lightweight (e.g., no virtual memory, efficient direct hardware access, very short interrupt handling latencies) that it remains suitable for small embedded system. RTEMS furthermore includes a flexible model for I/O drivers and a POSIX-compliant standard C library.

The port to the EPS was facilitated by the fact that processor-level support for the SPARC V7 architecture was already in the RTEMS 4.0.0 code base. At the low level of this base port, we had to add EPS specifics such as memory-management, cache control, interrupt handling, and real-time clock access. For testing, we mapped the RTEMS console to the *ACE2*card virtual serial ports.

At this stage, it was now possible to execute conventional C programs on the EPS. I/O, was limited to interaction on the virtual serial port, though.

4.2. Host I/O Access

While the limited I/O capabilities just described might be sufficient for small embedded systems, our aim of using the *ACE2*card/ADM-XRC combination as a target for automatic HW/SW compilation requires a higher degree of host integration. Specifically, many of the applications need transparent read/write access to files residing on the host filesystem.

While ad-hoc methods of transferring this data could be used, we implemented a reusable mechanism providing full access to host files and devices. It relies on a custom RTEMS driver that forwards all I/O operations on non-local devices to a server program running on the host.

This communication occurs by setting up a parameter block in the EPS DRAM and sending an I/O request to the host using one of the virtual serial ports. The I/O server is awakened and then uses the TSI host-side device driver to retrieve the parameter block from the mapped-in EPS DRAM. Next, the I/O operation is actually performed and any read data transferred back to the EPS through the shared memory.

In this manner, an application running on the EPS can access all data on the host (even devices, network mounted volumes, pipes, etc.). Furthermore, since all three of the standard I/O streams are also routed using this mechanism, it is even possible to transparently pipe data from a host application through the EPS and back to another host application without any user intervention.

4.3. Hardware API

Instead of simply reading and writing hardcoded memory locations for access to the FPGAs, a dedicated set of routines provides these operations in an easy-to-use and portable manner.

Among the operations supported are the decompression and fast loading of configuration bitstreams, the retrieval of address mappings for the FPGAs and their associated memories, and the locking and synchronization of FPGA-based computation with RTEMS threads. Additionally, the package also encapsulates board specifics such as interrupt handling and programming the variable clock for each of the i960 busses. All of these functions can operate regardless of whether the target FPGA is in the RPU or on the ADM-XRC.

4.4. Tools

Since the microSPARC-IIep of the EPS is fully compatible with other SPARC V8-based computers (e.g., the SUN SparcStation5), existing tool chains can be used to target the EPS with only slight adjustments. In our case, we are relying on the GNU suite of C compiler, assembler, and linker for the main flow. The resulting executables are then transformed into a binary format suitable for downloading to the *ACE2*card using the GNU binutils package. Our standard compile flow wraps the binary in an envelope that automatically starts the I/O server on the host, performs the download, starts the application, and establishes contact with the I/O client on the *ACE2*card. Thus, running a program on the EPS is accomplished transparently by simply typing its name on the host command line.

For debugging, one of the virtual serial ports on the EPS is used by the *ACE2*card firmware to implement the GNU GDB remote debugging protocol. In this manner, EPS applications can be comfortably debugged from the host using GDB and enhancements like DDD.

For hardware creation, we employ conventional logic synthesis tools starting from Verilog HDL or an experimental compiler translating C into a hybrid HW/SW application. In both cases, the Xilinx M2 EDA tools are used as a back-end for creating the bit-stream files, which are then compressed and converted into ELF object files that are directly linkable into the EPS application. This approach is preferable over the standard solution of converting the bit-stream files into a C program (hex dump) which is then compiled and assembled before linking: For a current medium-capacity FPGA such as the Virtex 1000, bit-streams are 770KB in length. The resulting C file using the conventional approach would thus have a length of ca. 3.8MB. Especially when multiple configurations need to be integrated in this fashion, the compile and assembly times become unacceptable. Compare this with our way of compression and ELF generation: A 770KB bit-stream is turned into a 12KB linkable ELF object in a fraction of a second. This bit-stream can be decompressed back into a format suitable for downloading in less than 100ms on the EPS.

5. Conclusion

In this work we described an approach to obtaining a very flexible platform for prototyping hybrid hardware/software systems. We achieve our aims of tight hardware/software integration, large hardware capacity, and ease-of-use through the combination of two off-the-shelf hardware products with the addition of a powerful yet lightweight software layer. The system has proven very successful both as a conventional prototyping environment as well as the target for an automatic hardware/software compilation system.

All of the custom-developed software (e.g., PCI configuration code, RTEMS port, I/O system, hardware API, and tools) is available on request from the author.

References

[1] Cadence Design Systems, Inc., “CoBALT”, `http://www.quickturn.com/products/cobalt.htm`, 1999

[2] Aptix Corp., “System Explorer MP4”, `http://www.aptix.com`, 1999

[3] Goering, R., “ Quickturn boosts emulation to 20M gates”, EE Times, No. 1036, 23 Nov 1998

[4] Koch, A., Golze, U., “A Universal Co-Processor for Workstations” in *More FPGAs*, ed. by Moore, W., Luk,W., Oxford 1994, pp. 317-328

[5] Koch, A., Golze, U., “Practical Experiences with the SPARXIL Co-Processor”, *Proc. Asilomar Conference on Signals, Systems, and Computers*, 11/1997

[6] Virtual Computer Corp., “H.O.T. II Development System”, `http://www.vcc.com`, 1998

[7] Annapolis Micro Systems, Inc., “WILDSTAR High-Speed DSP Boards”, `http://www.annapmicro.com`, 1999

[8] Sun Microelectronics, “microSPARC-IIep User’s Manual”, `http://www.sun.com/sparc`, 1997

[9] Weaver, D.L., Germond, T., “The SPARC Architecture Manual, Version 8”, Prentice-Hall, 1992

[10] IEEE, “Draft Standard Physical and Environmental Layers for PCI Mezzanine Cards: PMC”, IEEE Standard P1386.1, 1995

[11] Xilinx, Inc., “The Programmable Logic Data Book”, `http://www.xilinx.com`, 1998

[12] PLX Technology, “PCI 9080 Data Book”, `http://www.plxtech.com`, 1998

[13] Harr, R., “The Nimble Compiler Environment for Agile Hardware”, *Proc. ACS PI Meeting*, *`http://www.dyncorp-is.com/darpa/meeting/acs98apr/Synopsys\%20for\%20WWW.ppt`*, Napa Valley (CA) 1998

[14] Xilinx, Inc., “Virtex-E 1.8V Field-Programmable Gate Arrays”, `http://www.xilinx.com`, 1999

[15] On-Line Applications Research Corporation, “RTEMS - Real-Time Executive for Multiprocessor System”, `http://www.rtems.com`, 1998

Appendix C. Porting RTEMS to the ACE Hardware Platform

Chapter 1

Introduction

This work describes the port of the RTEMS RTOS to the ACE hardware. Also described is the file I/O layer which was added to RTEMS in order to make the resulting system more usable. RTEMS/ACE is a real time operating system (RTOS) which is in use in research of adaptive computer systems (ACS) at the Technische Universität Braunschweig, Dept E.I.S.

1.1 Adaptive Computer Systems

Adaptive computer systems are systems which also carry user programmable hardware. That means that, while a program is running, the hardware can be reconfigured to provide the best environment for the program. The goal is to make the hardware run the computation intensive parts of the program, such as inner loops or vector operations.

Hardware has an immense advantage over conventional software because the configurable logic does not limit the amount of parallel execution in the same way conventional processors limit the parallel issue of instructions. When a current high-end CPU is doing vector operations, the amount of parallel operations is limited by the amount of execution units which can execute the machine instructions. Many processor manufacturers try to overcome that limitation by providing special execution hardware which provides basic SIMD capabilities. MMX and Altivec may serve as examples here.

However, when the software can utilize user configurable logic, the amount of parallelism can be greatly increased since there is no limit to the amount of parallel execution units other than the number of available gates and a limited bandwidth to memory.

1.2 What Is An 'Embedded Operating System'

RTEMS is a small real time operating system designed for the field of embedded control. Historically, the process of using software in embedded systems started with the first TI pocket calculators which had a small microprocessor that did the calculations by software algorithms rather than hardware. This saved a lot of logic space and, by reducing power consumption, achieved a longer battery life.

These systems were a combination of hardware and software, which together made up a usable device. When the hardware improved later, it was possible to have one piece of hardware solve many tasks by executing different software. This made it economically worthwile to spend more effort in developing hardware. This paid off since the more powerful hardware could manage an even wider range of tasks; it was reusable.

By abstracting the application completely from the real hardware, it was possible to bootstrap the application on a different computer system, such as a Unix workstation or a Windows PC, as long as the same interface was used. This greatly simplifies the process of development because a real debugger can be used on the application. The turnaround times are also greatly reduced because of more powerful machines used for development. Another reason for the use of cross development is that most the target hardware is unable to run a decent compiler system [1]As the service demands of the applications were increasing, the hardware abstraction layer soon became a full featured operating system.

The requirements of an embedded application are different from those of a desktop application. For example, permanent storage facilities such as disk drivers are not needed in a cell phone, virtual memory or memory protection are superfluous in a car motor control. The embedded operating systems evolved differently from the desktop operating systems. RTEMS is a multitasking real time operating system for embedded control that was chosen to be ported to the ACE card.

[1] Compiling the software for a car motor management on the car motor management is not a Good Thing.

Chapter 2

The ACE Hardware

The ACE cards are suitable as platforms for research on adaptive computing. Therefore, they hold not only a dedicated CPU and a local memory subsystem but also programmable logic circuits. Figure 2.1 shows the architecture of the ACE hardware.

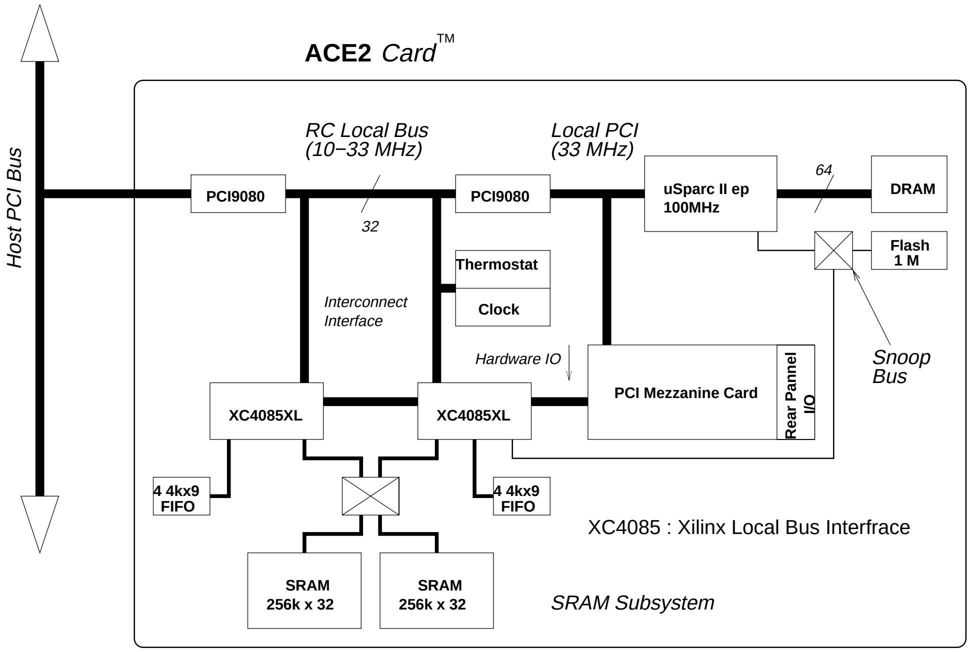


Figure 2.1: The ACE hardware

As shown in figure 2.1, the ACE2 cards carry two Xilinx XC 4085XL FPGAs. The predecessor of the ACE2, the ACE card had two XC6264. Both cards will collectively be called ACE in this document.

For a fast memory access the FPGAs are connected to two dedicated memory banks of 256kx32b each via a crossbar-like switch. This enables a quick swapping of the individual banks between FPGAs One of the FPGAs also has a snoop bus connected to 1 MB of flash memory that contains the boot code for the microSparcIIep CPU which is the controller of the whole card. On the ACE card there is a PCI Mezzanine Card (PMC) slot which is connected to the same FPGA as the bus snoop logic. To operate correctly, the ACE cards need a host computer system. That means they need to be plugged into a PCI slot of a host computer that can provide services the not implemented. This requires support from the host computer's OS. The Solaris version comes with a kernel driver module that maps the memory banks of the ACE to device files in `/dev/`. This module also supplies device files for the control registers and virtual serial communication lines provided by the ACE firmware.

2.1 The ACE Firmware

The on board microSparcIIep CPU boots from flash ROM supplied for that purpose. After setup it waits for commands coming from the host computer. A program running on the host transfers the binary image into the microSparcIIep CPU dedicated memory and tells the firmware at which address to start execution. This entry point defaults to 0x100000 because the lower megabyte of memory is used by the firmware for internal data and stack space. The memory area marked as Heap in figure 2.2 is unused and can be allocated by memory management code such as `malloc()` (if the application actually uses it).

Should the application return control to the firmware for some reason, maybe because it has finished its execution, the firmware resets the ACE card and waits for the next application to be started. The firmware supplies a complete set of interrupt vectors including the very important window-underflow and window-overflow interrupts of the SPARC CPU. Also included is a GDB remote debugger hook that enables the GDB debugger to source level debug a currently running application including memory dumps and register values.

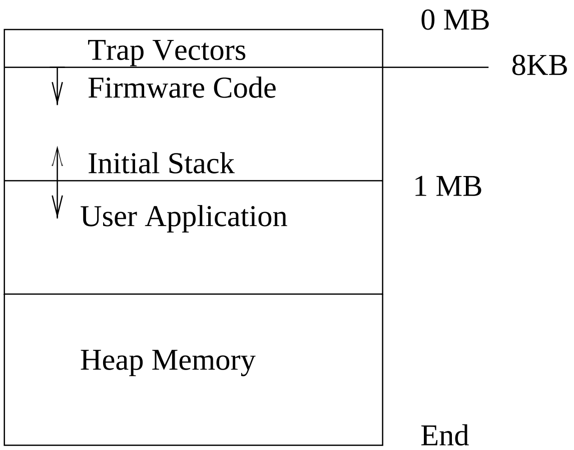


Figure 2.2: The ACE Memory Map

2.2 Communication

Since the ACE card is not a complete computer on its own, a way to exchange information with other devices is required. The obvious link between the ACE and the rest of the world is the PCI connector which connects to the host computer's bus system. This connection has to carry the data that needs to be transfered to the ACE and back to the host. This data includes the FPGA configuration and the machine code for the microSparcIIep CPU. Results produced by successful program runs on the ACE need to go back to the host system for visualization or further processing. The two main methods to transfer data between the ACE and the host are shared memory and Virtual Serial Ports (VSPs).

2.2.1 Virtual Serial Ports

One way to pass information is to use the Virtual Serial Ports (VSPs) which are implemented by the PCI part of the ACE card. There are four serial ports available [1] which can be used for transfers. The ports provide real full duplex channels with interrupt signals to the host computer. On a host system running Solaris, these ports can be accessed by device files which are used like serial interfaces. On the ACE, the programming of these ports is a bit

[1] The channel with number one is used implicitly by the remote debugger, so using it and debugging at the same time will not work.

more complicated, but a reference source exists in the TSI firmware archive (`vsp.c`). The VSP lines can transport data in both directions, but only at very low rates with high communication latencies. Also, the ports available on the ACE/ACE2 hardware are known to be buggy. (For details please refer to section 2.4).

2.2.2 Memory Mapped Communication

All memory banks on the ACE card are mapped into the address space the PCI slot acquires during the host booting process. By memory read/write accesses to the correct address range inside this PCI space, all memory locations on the ACE card are accessible from the host. This includes not only the DRAM memory dedicated to the microSparcIIep CPU, but also the local memory blocks which are connected to the FPGAs.

In this way, the program images are transfered to the CPU memory and the hardware configuration to the FPGAs. Running a real Input/Output (I/O) protocol relying only on memory mapped data structures is inefficient due to the lack of interrupts or other synchronizing methods: It would involve polling of some kind, which would drastically reduce the performance of one of the peers. For larger and less frequent transfers, this method is good because of the big speed advantage over the one-byte-at-a-time protocols, such as VSPs (see 2.2.1). The drawback of this protocol is that it is only really usable by the host computer because accessing the host memory from the ACE card is not possible in a trivial way. Even if it was possible, the hosts operating system might (very likely) implement some sort of virtual memory and memory protection which requires memory mapping by a Memory Management Unit (MMU). Currently, a MMU is part of the processors used as CPUs in todays workstations. The user processes have no way to tell what physical addresses they work on nor if their memory space is 'in-core' [2], paged out, or even whether it is linear at all [3].

[2] 'in-core' means that the memory page's content is currently placed in main (physical) memory. Pages not 'in-core' have been written to secondary storage by the OS to free the physical memory page which can then be assigned to a different process.

[3] For more information on modern computer memory mapping please refer to [1] or [5]

2.3 ACE Support Programs

Along with the ACE hardware comes a set of tool programs. These programs are delivered in source form for the purpose of documentation and to serve as a basis for other programs which are to utilize the ACE cards. These tools are

- `aceiiclock/aceclock`
  The local bus clock on the ACE card can be set with this tool within a range from 10 to 33 MHz.

- `aceiidownload/acedownload`
  This tool is supplied to load program image files into the memory of the microSparcIIep CPU on the ACE card . It loads the file to the DRAM and then starts ACE execution.

- `aceiifilter/acefilter`
  This tool can be used to strip the header from a bit stream file that is to be written into one of the XILINX FPGAs. It creates a raw output file that then can be uploaded with the `aceiixilinx` tool program. The bitstreams can be produced with any design software that supports the appropriate chip type.

- `aceiiflash/aceflash`
  The ACE card has a 1MB flash memory for the firmware code. If a change to the firmware is needed, a new image can be compiled. The write operation to the flash memory is then performed with this tool. Using it overwrites the firmware which is currently being active. Use with extreme care.

- `aceiireset/acereset`
  If a program currently running on the ACE card is not terminating as expected, the `aceiireset` tool can be used to reset the ACE card . The memory image will not be disturbed so a post-mortem debugging of the program should be possible.

- `aceiitemp/acetemp`
  Under heavy load and in a tight enclosure, heat is the foe of any hardware. To keep an eye on the current hardware temperature, the `aceiitemp` utility can be used. Please be aware that the program displays the temperature in Fahrenheit.

2.4 Known Bugs

The TSI provided VSP driver for Solaris is known to have bugs. Trying to use the Unix system call `ioctl()` on the file handle that represents the serial connection to set a timeout is not possible. This can drastically disturb the uptime of the host computer because the only way to interrupt the read from a dead line is to issue a reboot. This may be acceptable for a single user desktop system, but is unacceptable for a multi-user system in a scientific environment.

Chapter 3

RTEMS, a Real Time Operating System (RTOS)

3.1 RTEMS Overview

A typical RTEMS OS and application consists of functional blocks which deal with certain situations. The block structure comes from the modularization of the system on the source tree.

3.2 Architecture

As shown in Figure 3.1, the application and the OS form one binary. The OS is statically linked to the application to form one unit. The Hardware Abstraction Layer (HAL), shown in italics, will be discussed in greater detail in Section 3.3.

An complete RTEMS application consists of several parts :

- Application Code
  The application code written by the ”user” of the system. It contains the actual algorithms which are needed to solve the problem. The code in the application layer is not part of the operating system. However, it is possible that some code which is used to extend the operating system is placed in the same source files.

- OS Interface Layer
  This is the layer which presents the interface to the application. The

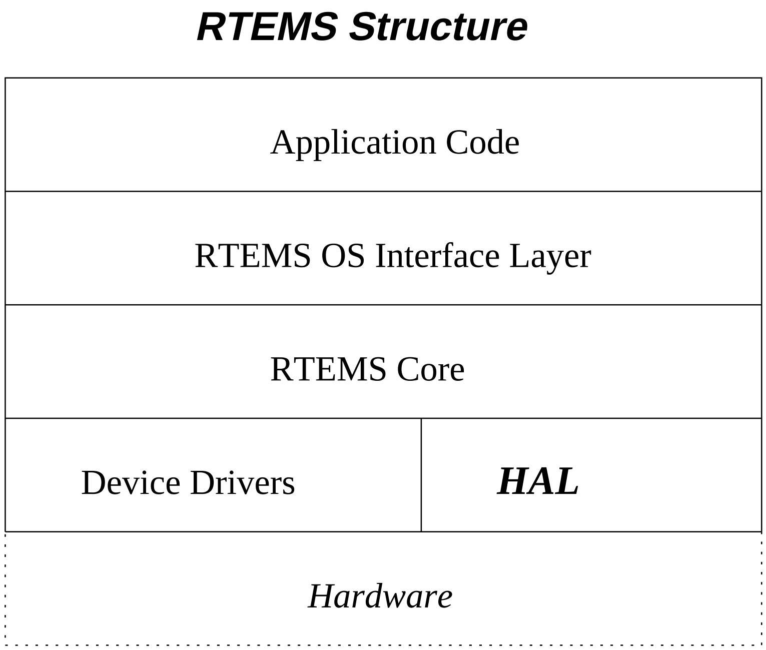


Figure 3.1: The RTEMS Structure

interface layer itself deals with different parts of the core and drivers. This simplifies the application as the programmer does not need to know details beyond the interface definition. Parts of the OS that are not part of the interface can change from one release to another, and making assumptions about them breaks the application should a new release be needed.

- RTEMS Core
  The core part contains the scheduler, the resource managers and some internal parts of the operating system. Calling the core directly is possible, but most of the time unnecessary and dangerous. Access to the core data structures should be done by calls to the interface layer.

- Device Drivers
  Device drivers are parts which are dependent on some kind of hardware, but not the actual processor. Their code interfaces with hardware and might have special requirements at runtime, such as having all interrupts disabled during its execution. Calls to the device drivers can be done by the core, the interface layer, or in rare cases, by the application.

- *HAL*

By separating all the parts of the operating system that are actual hardware dependent into a so called HAL, porting to a new platform is greatly simplified. The HAL is described in more detail in Section 3.3 and Figure 3.2 of this document.

- Hardware
  Every modern and portable application should not require knowledge about the actual hardware (CPU, memory sizes/location, interrupt controllers, . . .) it is executing on. Well written software only accesses the hardware by using the operating system. Since modern operating systems are portable also, they access the hardware with a dedicated Hardware Abstraction Layer. The only parts of RTEMS which come into contact with the hardware are the device drivers and the HAL. ”Hardware ” in Figure 3.1 does *not* mean the hardware part of an adaptive computing application.

3.3 HAL

Like all software that is intended to be portable, RTEMS has a Hardware Abstraction Layer (HAL). Its purpose is to keep all hardware dependent parts of the software contained in a small set of source files. This forces accesses to the hardware to use an interface that can be defined once and then controlled by the source maintainer. In this way all software that uses this interface can work with a different HAL without being changed, as long as the new HAL implements the same interface along with the same semantics.
The RTEMS HAL consists of several parts which are arranged as described in figure 3.2. The HAL is separated in two major parts which are independent from each other. Both parts have a well defined interface and have to be implemented for a port to a new platform.

- CPU Package
  The CPU Package contains all the CPU dependent code of the HAL. It contains the interrupt code and also contains the code for the taskswitching routines. It implements the interface the rest of RTEMS needs to access CPU dependent services in a CPU independent way [1]. This makes it easier to write core code that works on all processors. Only really weird hardware architectures need fixes in the non-CPU package parts of the operating system.

[1] The higher layers of RTEMS refer to hardware dependent parts by function calls and by macros. This abstraction has to be provided by the packages.

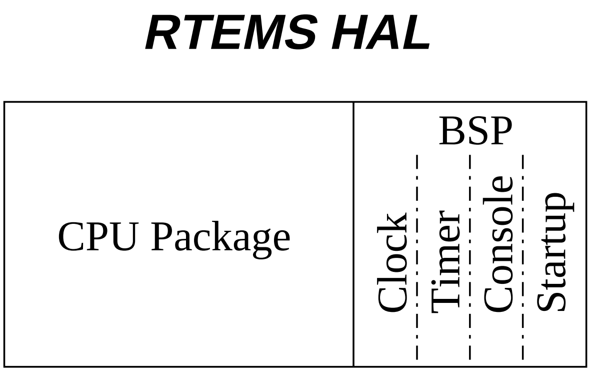


Figure 3.2: The RTEMS HAL

- BSP
  All hardware dependent parts of the HAL are organized in the Board Support Package (BSP). It consists of the modules *clock*, *timer*, *console* and *startup*.

  - clock
    The clock module contains the code to setup the clock and install the timer controlled interrupt that is needed for the pre-emptive multitasking to work. The main entry points are `Clock_initialize` and `Clock_exit`. The entry point `Clock_control` is used to trigger the interrupt code and to replace the interrupt vector code with a new routine to be called. The (private) routine `Clock_isr` may do other bookkeeping if desired, but has to call `rtems_clock_tick` to notify the RTEMS Kernel of the interrupt.
  - timer
    To support a long running timer that can be used for benchmarking the module `timer` initializes or simulates a long running timer. Simulation of a timer may be needed if the hardware does not supply a sufficiently large interval timer. The main entry points are `Timer_initialize` and `Read_timer`. The `Timer_initialize` code is called from the RTEMS core if timing is required by the application. The following calls to `Read_timer` should then return timing information. Since this module is hardware dependent, no unit is specified in which the time is measured. The total time can be calculated from the clock frequency of the timer, which can vary from port to port.

- console
  The purpose of this module is to provide I/O functionality for the `stdin` and `stdout` available to the RTEMS application. After the initial call to `console_initialize` the console should be up and working. The important entry points are `console_read` and `console_write`. As the names suggest, these functions should do I/O to the console. The actual mechanism used for the I/O is not specified, so this module is placed in the hardware-dependent part.

- startup
  Hardware dependent parts of the startup process are placed in the `startup` module. This module consists of the following source files on a normal RTEMS installation:

  - `bspstart.c`
    Main entry points in this file are `bsp_pretasking_hook()` and `bsp_start()`. The `bsp_pretasking_hook()` routine is responsible for initializing any data structures which are C-library extensions to the default BSP. The prime candidate here is the `libc`, whose memory allocator needs to be initialized. This routine is called before drivers are initialized. To perform important initializations before the kernel gets started the routine `bsp_start` is used. Its main purpose is to setup the memory ranges and initialize the workspace of RTEMS so that the internal RTEMS memory allocator can be used by the kernel itself. This routine needs to divide the memory space that is unoccupied by the application binary between the RTEMS memory allocator and the libc memory allocator (`malloc()`, . . .).
  - `bspclean.c`
    In case the embedded application terminates, some special actions may need to be performed. While initializing special hardware is done in the `bspstart.c` source, the corresponding shutdown actions occur in `bspclean.c`. The main routine here is `bsp_cleanup`.
  - `setvec.c`
    Main entry point here is the routine `set_vector`, which usually is a wrapper to the CPU dependent code. This wrapper is used to be able to introduce board dependent code into the interrupt handling while keeping the code itself CPU independent also.
  - `startup.S`
    This is the main entry point into the entire binary. It is usually a very small piece of code that is dependent on both; the CPU

used and the board. It depends on the board simply because it must ensure that the resulting object code ends up in the correct place so that the routine `boot_card` gets executed. How this is done depends on the processor in use. On a SPARC, like on most RISC processors, a dedicated ’Power On Trap’ is called. This trap code is the entry into the system initialization process. Since no other setup is done, one of the first things this code has to do is creating a stack segment.

3.4 Startup Process

Figure 3.3 is a simplified diagram of the startup process that a typical RTEMS application performs. The syntax of the diagram was inspired by UML [2] [3]. The topmost line lists the RTEMS functional blocks which contain the part listed below. Program flow from one module to another is documented by an arrow from the source of the call to the target module. The oblongs describe the lifetime of an object, in this case the stack frame. Figure 3.3 does not only describe the RTEMS startup process but also includes some details about the RTEMS/ACE layout which is described in more detail in section 4. The normal RTEMS does not distinguish between User Application and User Interface; the User Interface is part of RTEMS/ACE and hides some differences between RTEMS and other OSes from the User Application.

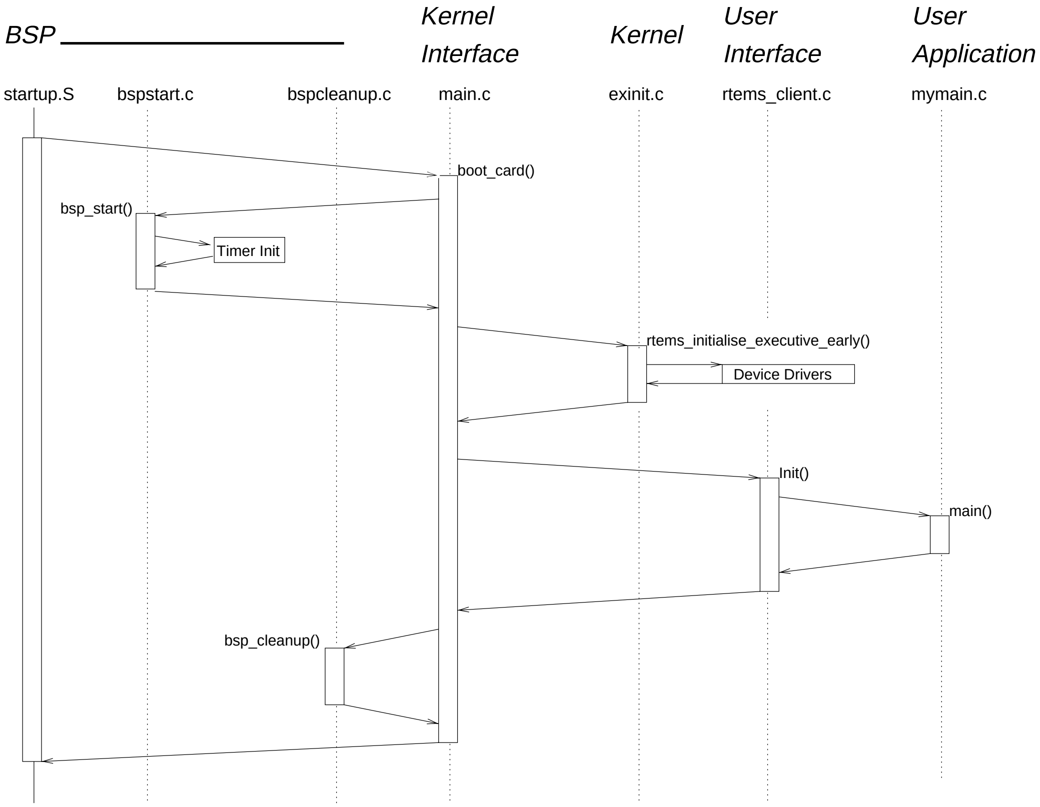


Figure 3.3: The Startup Process

The startup process depends on the CPU used. How the first executable instruction is passed to the processor depends on the processor type and the board hardware. Usually, the first entry point is placed in some small assembler source that does basic hardware setup. This setup includes fundamental things like providing a stack pointer for the programming environment but also more complex code that performs setup operations on the memory interface. After the hardware is set up and the software environment is initialized so that normal C code can be executed the flow of control is passed to the RTEMS kernel by calling `boot_card`.

The routine `boot_card` belongs to the portable part of RTEMS. More hardware dependent setups are performed by calling the board-specific routine `bsp_start`. This routine is part of the BSP and has the duty to initialize any hardware that is part of the complete system and which is needed to be in working order when the higher levels of the driver system are about to be initialized. A good example for this kind of hardware is a serial connection hardware which can be useful to printout debugging messages during the testing period. As an example, the initialization of the multitasking timer is part of figure 3.3.

After initialization has finished, flow of control is returned to `bsp_start`. The `bsp_start` then calls `rtems_initialize_executive_early` to initialize the system. This routine initializes the various parts and handlers of RTEMS. The initialization of the device drivers is picked as an example in figure 3.3 [2]. After all initializations have been performed, multitasking is started. RTEMS defaults to the name `Init()` for the main routine in the user applications. A user may decide to add additional threads to his application. The system will continue to multi-task until the first user process (`main()`) terminates, then RTEMS will begin shutdown operations. These involve killing of all remaining tasks and calling `atexit()` functions. The cleanup process then continues by calling `bsp_cleanup` to shutdown the hardware which was initialized by `bsp_start` in the very beginning. Flow of control is then returned to the system startup stub that passed control to `boot_card`, normally executing some sort of `halt` or `stop` machine instruction to shut the CPU down. This process is dependent on the hardware being used and the task that hardware has to perform. For example, when shutting down services of some hardware which is difficult to reach, such as satellite systems, a reboot might be used here instead of a simple halt [3].

[2] The placement of the device drivers may be misleading, they are **not** part of the user space; the user application only supplies a table where the needed devices are recorded and only those are initialized.

[3] Pressing the reset button on a satellite system can prove difficult.

Chapter 4

RTEMS On The ACE Platform

The main task of this work was to port an operating system to the ACE and ACE2 cards. For this, a real time operating system which is described earlier in this study was chosen: RTEMS. This decision was based on some requirements and how each candidate matched them. We considered the following aspects as important:

4.1 Requirements

- Source
  RTEMS is available in source form and ready to handle programs written in C/C++ or ADA. Since most programs which will have to work on the ACE are written in C, this fact is important. Having RTEMS in C also means that the interface is compatible.

- Fitness
  RTEMS is easy to compile and install since it is written in C. The tools used for this are widely used and have proven stable and efficient. This would not be the case for some esoteric language that is only ”spoken” on one system [1]. The distribution contains configuration scripts for the setup of RTEMS as for the generation and installation of a cross compile system for the selected hardware. This one is based on the GNU compiler system which has proven to be adaptable to many kinds of processors and which is a robust system for use in production work.

[1] 'Smalltalk' may be an example here.

- Price
  RTEMS can be obtained free of charge after registering as a user of RTEMS from the URL of On-Line Applications Research Corporation.

- Portability
  Since RTEMS is targeted at embedded control, the HAL is strictly separated and quite small. It is split between the HAL for the current board and another one for the processor hosted on the board.

- Scale
  RTEMS is able to deal with multitasking applications, can use multiple CPUs and has a clean way to add more drivers to the system without the need to modify vital parts of the operating system. It does not support virtual memory or memory protection, but these are not needed. Because the FPGAs which are placed on the board perform memory accesses on their own. Since they do not know about the current mapping of the main memory there would be considerable difficulties in letting them perform direct memory accesses in virtual address space.

- low overhead
  RTEMS has no separated user and system mode and no time consuming switching between environments. The interrupt latency and the task switch time are short.

4.2 RTEMS, pro and contra

RTEMS does not support the loading of program binaries under OS control, it is statically linked with them. This stems from its roots in the area of embedded control where the operating system and the user application are one binary which is written to ROM space. There are some reasons against and for this one-binary approach to be used on the ACE cards . Following is a brief discussion with more reasoning whether an issue is important or if it implies a limitation that can be ignored.

- pro:
  - Versions
    All executables are statically linked with their operating system. So, after being tested, a change to the operating system does not concern the application. If it was working before, it will continue to do so.

  - Space
    There exists only one namespace in the binary which contains both operating system and user application. Many support functions are present only once because the link editor will only include one symbol of a name into a binary. This saves memory space on the limited memory of the ACE Card. Also, only code parts actually used should be added to the final binary by the link editor. This reduces memory usage in the running application.
  - Hardware
    The RTEMS is intended to serve as platform for programs which use the FPGAs on the ACE cards. Since these are atomic resources which are not easily shareable, a system service would be necessary to provide fast and easy access to them. Because speed is a critical point here, an abstraction layer could have easily absorbed all the performance increase. It would also have forced the programmers to use a very different way of dealing with them which would consume more runtime. By not allowing for multiple applications running concurrently while still allowing multitasking inside the application the resource can only have one owner. The program does not need to query the current use permissions and is therefore faster.
  - Complexity
    Because the application is linked to the operating system and the final binary is relocated to a fixed address in the memory space of the microSparcIIep , there is no need for some kind of application loader. Resource tracking is also unneeded since all applications are ended by a complete shutdown of the operating system. No allocation can persist beyond that.

- contra:
  - Versions
    Due to the statically linked nature, enhancements in the operating system area of an application are only possible by recompiling or relinking the binary.
  - Space
    All binaries on the storage media of the host system have the operating system linked to them and so it consumes storage capacity every time it is used.
  - Multiuser
    Since a computer can only run one kernel at one time, it is not pos-

sible to run multiple applications on one ACE Card. This can limit the productivity of a complete system.

To sum it up, one can say that the current situation is a good tradeoff between the pro- and contra arguments. If it were possible to increase the performance or usability of the ACE Card by using host resources, this was done almost every time. This is justified since host resources are more abundant and easier upgradeable than client resources.

Chapter 5

Porting RTEMS to ACE

The porting process of RTEMS to the ACE hardware is described in the following sections. All filenames are set in the `typewriter` font and are relative to the install directory.

5.1 How to port

The porting process of RTEMS is straightforward. After obtaining the source code and extracting it, some subdirectories are created below the directory which was chosen as the root location. The directory named `rtems-4.0.0/src/c/lib/libbsp/` contains subdirectories which hold the so called Board Support Packages (BSPs). A good start in porting is to create a new one in the correct CPU directory. Since the ACE cards use the microSparcIIep CPU as main processor, the correct place for the RTEMS/ACE BSP is
`rtems-4.0.0/src/c/lib/libbsp/sparc/ACE2`. Inside this directory some more directories need to be created for the `clock`, `console`, `include`, etc. A directory listing of the ACE2 directory looks like this:

```
Makefile     bsp_specs  console  startsis  timer  wrapup
Makefile.in  clock      include  startup   tools
```

- **Makefile** is generated from
- **Makefile.in** during the build process. Makefile.in contains the definitions which files and directories are present and vital for the build.

- **bsp_specs** contains definitions for the setup. The best result can be obtained by modifying an already existing bsp_specs file from another target.
- **console** is a directory that contains the hardware dependent code for the RTEMS console device.
- **startsis** is a directory containing the very low assembler startup code.
- **timer** is also a directory, it contains the hardware dependent driver code for the RTEMS timing facilities.
- **tools** is an empty directory for the ACE port. It may contain tools for other porting endeavors, however.
- **clock** is a directory containing the hardware dependent code for the RTEMS clock device.
- **include** is a directory that contains the include files which define the hardware. Most notable is the `bsp.h` file that defines several constants for the rest of RTEMS.
- **startup** contains the files `bspstart.c` and `bspclean.c` along with some tool sources (`setvec.c, spurious.c`) and the file `linkcmd` which is the linker command file used in building the final binary.
- `wrapup` is the directory that contains the final makefile for the BSP. This makefile has to create the final BSP package that can be used in the link phase.

The specific microSparcIIep CPU is not directly supported by RTEMS, but the general SPARC architecture is. The SPARC CPU package already present can then be reused, reducing the port to the BSP package which interfaces with the actual hardware. The file `bsp_specs` contains the gcc specs for building RTEMS applications.

Another file which needs to be created is `rtems-4.0.0/make/custom/ACE2.cfg`. It contains the configuration definitions for the rest of RTEMS. Here can be selected if, e.g., support for network sockets is to be added or if ADA support needs to be compiled in.

All in all the porting process is straightforward and the documentation found at the OAR web page is sufficient to do the port. The hardware drivers, however, need documentation which is not available from OAR but from the hardware manufacturer in question.

5.2 Modifications to the main source

During the porting process, some quirks came up which needed small changes to the main source tree of RTEMS. They were needed because some intermediate versions were not compiled with optimization enabled in the setup file. Other modifications were needed because the later versions were compiled with a new version of the GNU compiler which is a bit stricter syntax wise in some situations. The changes were:

- `rtems-4.0.0/c/src/exec/score/inline/isr.inl`
  This is an include file for C, despite the nonstandard filename extension. It contains some code patterns and functions which should be in-lined. If compilation without optimization is selected, this produces linker errors because of multiple defined symbols. A ”static” was added to fix this.

- `rtems-4.0.0/c/src/exec/score/cpu/sparc/sparc.h`
  Here some constructs were added to make the microSparcIIep known here at compile time. If not done, some source codes will complain about missing definitions.

- `rtems-4.0.0/c/src/exec/score/src/wkspace.c`
  The code has been changed to output the dots and asterisks during the clearing of the main memory. Every character represents one megabyte of memory to clear. The ”...”s are printed at the beginning of the memory clear loop and after each megabyte is cleared, the corresponding ”.” is overwritten with a ”*” as a progress indicator.

- `rtems-4.0.0/c/src/lib/libbsp/shared/main.c`
  The default entry point into a RTEMS application is called `Init` and not `main`. One goal in porting RTEMS to the ACE was to make normal C programs compile and work without change, but the symbol `main` was already used in RTEMS. To solve this dependency, the RTEMS routine has been renamed to `Main`. Now the problem with GNU C++ on some target platforms is that the main function calls all the global constructors by calling ”__main”, which is now happening at a different place. So, if a device driver which is part of the application relies on an object being initialized, this assumption is now broken. The device driver initialization code is called before the ”main” function had a chance to call the constructor list.

  The consequence from this is that no C++ device driver is allowed.

- `rtems-4.0.0/c/src/exec/score/cpu/sparc/cpu_asm.s`
  Code has been added to stop the timer during a call to the context switch. This was done to provide better accuracy to the benchmarking timer.

- `rtems-4.0.0/configure`
  This file needs modifications because it checks for a cross compiler in an inelegant way. It simply tries to compile a file using the cross compiler and then starts it, assuming that the host operating system will refuse to load and start it in case of a cross compiler. This is not true for a SPARC-equipped host. Here, this binary gets executed and drops core because of illegal instructions, ending the configuration procedure with a core dump.

5.3 Memory Map

The RTEMS memory map on the ACE hardware is divided into the firmware part and the RTEMS part of ownership. The lowest megabyte is under control of the ACE firmware. Only the interrupt vectors are modified by RTEMS. Some free memory at 512K is touched by the code that copies the command line arguments over to RTEMS.

The binary program is loaded starting with in the first location of the second megabyte of the main memory by the linker setup script. That leads to the first machine instruction being relocated to the address 1048576. The linker collects all machine code together. Following the code area are DATA- and BSS sections.

The RAM space beyond the last BSS address is used for dynamic memory allocations. Both, RTEMS and the C library have their own memory allocator code, so the memory space needs to be divided between the two allocators.

The ACE RTEMS port sports a memory size auto detect routine which can find out how much memory is installed and makes the correct division between RTEMS space and user space: Beginning with 12 megabytes, the code tries to perform a write to the next memory word. Depending on the memory controller decode logic in use, this word will not be written or will be placed in memory at location zero when the memory size is reached. For speed purpose this test is done in steps of 4 megabytes each. Now, if the end of memory is crossed, the control word will not be written to that location and a read from the same place will return a different value, often 0xffffffff. If the memory decoding logic treats the memory as a loop, the write will modify memory location null, which will also be detected. In both cases, the maximum memory size will be detected correctly.

Figure 5.1 describes the current memory layout which is enforced by the limitation of the so called ”blessed” window. That window comes from the fact that the FPGAs on the ACE currently cannot access memory beyond the 16M boundary because they only have 24 address bits from the bus. The last 64 KB memory area from that window is also reserved because it contains the register sets of the PMC PCI controller (See Figure 2.1 for details) [1].

These limitations force the memory map to be rather rigid instead of being able to distribute memory dynamically. Dividing of the memory between RTEMS and the C library is necessary because both, RTEMS and the libc

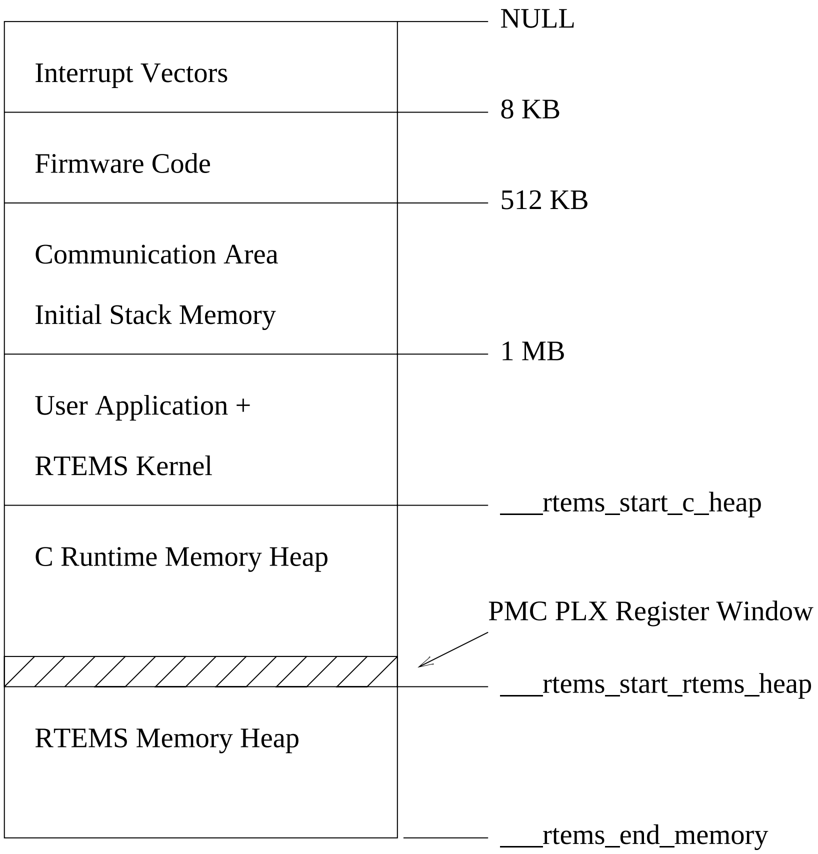


Figure 5.1: The RTEMS/ACE Memory Map

[1] The problems originating from the limits of that ’blessed’ window attracted some other names for that address range. Many of them should not be used on sacred grounds.

memory allocator are different. RTEMS uses its own memory management while user code is supposed to `malloc()` its storage needs. If both would use the same memory allocator code the division into two kinds of memory would be obsolete.

A positive aspect of the divided memory architecture is that the usual bugs introduced by dynamic memory allocation[2] are more likely to damage the user data structures, keeping the system alive. This is no solution to the problem of wild pointers but it can help letting the code run long enough to detect the problem and output some kind of assert(). Another important reason is that the FPGAs can only access their 'blessed' window and so all memory allocations which belong to the operating system should be handled by a memory allocator that does not use up the 'blessed' window memory. The user program can choose where to the dynamically allocated memory will come from. If the allocation is done by `malloc()`, the memory will be taken from the C-Heap. If the RTEMS memory functions are used the allocation is served by the RTEMS-Heap.

A user program can find out more about the currently active memory setup by reading the global variables which are pointing to interesting points in the memory map. These variables are :

```
char* ___rtems_start_c_heap;
char* ___rtems_start_rtems_heap;
char* ___rtems_end_memory ;
```

As figure 5.1 shows, between the actual RTEMS heap and the end of the C library heap is a gap of 64 KB for the PLC PCI registers.

5.4 ACE Firmware

As figure 5.1 shows, the lower megabyte of address space is occupied by the original ACE firmware. The firmware by default expects the user program at the load address of one megabyte and starts execution there. By keeping this, it was possible to use most of the firmware features for free. That greatly simplified the start process of the RTEMS operating system. There was no need to add the PCI bus setup to the startup code, no need to make flash updates and no need to bother with a lot of the interrupt handling problems.

[2] Statistically, the most frequent made mistakes in dealing with dynamically allocated memory are : writing out-of-bounds, writing to freed memory, writing via a stale pointer and freeing memory twice or not at all [6].

RTEMS copies the interrupt table of the firmware as one of the first things of its initialization and restores it again upon its exit. That way no exception should point to code no longer callable. By keeping the firmware alive during execution of RTEMS operating system, we also got the ability to source-level debug RTEMS for free. The firmware supports the GDB debugger in its remote-debugging mode. So, by being able to single-step through the process of the system initialization, most of the bugs were located and fixed quickly, without the need to use drastic debugging methods. Another benefit that came with keeping the firmware active was that the firmware has an interface for launching programs on the ACE. By keeping that, the tools which came with the ACE card were still usable to launch RTEMS applications. They only required minor changes for the memory based I/O and some RTEMS/ACE specific features.

5.5 Porting Problems

Porting an operating system is a complex task, and some problems which came up should be discussed here.

- Outdated Tool-chain
  The installed tool-chain that was provided on the system used for porting (make/grep/ ...) was too old to handle the configuration process. Because of unclear error messages which pointed to the source tree of RTEMS, finding the problem took some time. After adding a new ”bin” directory containing newer versions of all those tools to the search path it was possible to rebuild an already existing implementation of RTEMS in a matter of minutes. If RTEMS fails to build out-of-the-box in a matter of minutes for an already defined system, a check of the tool-chain versions is advised.

- Endianness issues
  After the first examples, such as ”hello world” and the `paranoia` test, worked on the ACE card no program that used multitasking or timing functions worked. It appeared that the timer setup was not as straightforward as the documentation suggested. The ACE BSP uses the internal timers of the microSparcIIep CPU for the multitasking timer and long-interval timer. The first timer is responsible for triggering an interrupt each millisecond to serve the scheduler and timing functions. Since these are part of the PCI logic of the microSparcIIep PCIC (PCI

Controller), they happen to be in little endian while the rest of the CPU works in big endian.

- Reboots
  Since a small hiccup of the communication layer in a badly written Unix driver can bring the whole host system into a state in which a crash might be imminent, several reboots of the host were required. Since this is not easily done with no access to the host, it took a lot of time. The host driver has been improved since.

- Hardware Bugs
  The ACE hardware has some bugs. One of the bugs which showed up rather early was that the VSP lines are buggy. They need a certain amount of data transfers until they sync between sender and receiver. Another bug in the hardware is that the RC local bus (see figure 2.1) does not support snooping correctly. The microSparcIIep CPU should be capable of snooping the bus for writes and dropping cache lines if their contents had been changed. However, this does not work reliably.

5.6 Hardwired Defaults

Due to the statically linked nature of RTEMS there are some limits which are compiled into the system. These limits can be modified to some degree by the user process at compile time. These limits are:

- Maximum open files

  ```
  #define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 40
  ```

- Maximum units per driver

  ```
  #define CONFIGURE_MAXIMUM_DEVICES                40
  ```

- Maximum number of tasks

  ```
  #define CONFIGURE_MAXIMUM_TASKS                 100
  ```

- Maximum number of timers

  ```
  #define CONFIGURE_MAXIMUM_TIMERS                 10
  ```

- Maximum number of semaphores

  ```
  #define CONFIGURE_MAXIMUM_SEMAPHORES             10
  ```

- Maximum number of message queues

  ```
  #define CONFIGURE_MAXIMUM_MESSAGE_QUEUES         10
  ```

- Maximum number of partitions

  ```
  #define CONFIGURE_MAXIMUM_PARTITIONS             10
  ```

- Maximum number of regions

  ```
  #define CONFIGURE_MAXIMUM_REGIONS                10
  ```

- Maximum number of ports

  ```
  #define CONFIGURE_MAXIMUM_PORTS                   5
  ```

- Maximum number of periods

  ```
  #define CONFIGURE_MAXIMUM_PERIODS                         5
  ```

- Maximum number of user extensions

  ```
  #define CONFIGURE_MAXIMUM_USER_EXTENSIONS                 5
  ```

- Microseconds per tick

  ```
  #define CONFIGURE_MICROSECONDS_PER_TICK                1000
  ```

- Ticks per timeslice

  ```
  #define CONFIGURE_TICKS_PER_TIMESLICE                    20
  ```

Changing these values does not make sense in all cases. Defining the 'Microseconds per tick' to something other than the supplied value will create confusion when the timing code inside the BSP is not changed to reflect that fact. To change these default values a recompile of the `rtems_client` is required. That module is supplied in source format for that reason. After the compile, many of these values are 'set in stone' and cannot be changed without recompiling.

5.7 Customizing

Some of the default values are also accessible and changeable by user code using the following interface:

```
typedef struct {
  void                          *work_space_start;
  unsigned32                     work_space_size;
  unsigned32                     maximum_extensions;
  unsigned32                     microseconds_per_tick;
  unsigned32                     ticks_per_timeslice;
  unsigned32                     maximum_devices;
  unsigned32                     number_of_device_drivers;
  rtems_driver_address_table    *Device_driver_table;
  unsigned32                    number_of_initial_extensions;
```

```
  rtems_extensions_table        *User_extension_table;
  rtems_multiprocessing_table   *User_multiprocessing_table;
  rtems_api_configuration_table *RTEMS_api_configuration;
  posix_api_configuration_table *POSIX_api_configuration;
} rtems_configuration_table;

/* extern symbols */
extern void ___rtems_set_limits(rtems_configuration_table*) ;
extern unsigned long ___config_rtems_memsize ;
```

The routine `___rtems_set_limits`, which gets a pointer to the configuration table as its only argument, is called very early in the startup process of RTEMS/ACE by the routine `bsp_start()`. It can not depend on any other data structure being initialized, its only purpose is to modify the configuration table and maybe the value `___config_rtems_memsize`.

The routine could look like this :

```
#include <config.h>
#define MB *(1024*1024)
void ___rtems_set_limits(rtems_configuration_table* me)
{
    /* For some reason we want some spare at the front
        of the memory list */

    me->work_space_start += 4 MB ;
}
```

The structure should be defined in user code by including `rtems-4.0.0/c/src/exec/sapi/headers/config.h`, not by cut&paste. Be sensible with modifications because RTEMS is not protected against abuse here. Changes to vital information here can make the whole system unusable.

Previous versions of RTEMS/ACE allowed for customizing the amount of memory which was going to be assigned to the RTEMS memory pool and the C library memory pool, but the gap in the usable memory map (ref. fig. 5.1) makes that impossible.

Chapter 6

The File Layer

The goal was to easily adapt existing software to the RTEMS/ACE without major changes. Since most software performs file I/O in one way or another[1], the I/O operations must follow the same semantics as on the host. For this reason, a file layer device driver was developed and implemented.

The major issues were, that the solution must fit into RTEMS without major changes to the rest of the OS code, and must be properly integrated into the code to avoid redundancies. The file layer is part of the RTEMS device driver table. In that way, it is clearly integrated into the system and accessible from RTEMS itself for its own purposes, as well as from the user application.

Usually an RTEMS application either declares a driver table of its own, or decides to use the default one. RTEMS/ACE declares this one:

```
/* Define file driver codes */
#define FILE_DRIVER_TABLE_ENTRY \
  { file_initialize, file_open, file_close, \
    file_read, file_write, file_control }

rtems_driver_address_table Device_drivers[] = {
  CONSOLE_DRIVER_TABLE_ENTRY,
  CLOCK_DRIVER_TABLE_ENTRY,
  FILE_DRIVER_TABLE_ENTRY,
  NULL_DRIVER_TABLE_ENTRY  /* end of table marker */
};
```

[1] A program doing no I/O at all is by definition rendered useless since it does not produce any result. It can therefore be ignored in this study

If a different driver table needs to be created, be sure to have the correct ordering: the console driver requires the file driver, but needs to be listed first! The reason for this is a bit obscure, but important.

6.1 Device Driver Lookup in RTEMS

If a file is being opened on RTEMS, the system traverses the list of device drivers and looks which one can actually handle the file. The decision is based on the full name of the file and the name the device driver has been mounted on.

So, if a device driver would be registered to answer to all names which start with ”/usr”, a driver that is specialized on names starting with ”/usr/local” would need to be mounted (= listed in Device-drivers) before that one. Otherwise, the driver ”/usr” will claim to be able to handle the request, and the specialized driver would never be considered in that case. If the specialized driver is mounted before the general one all accesses to names starting with ”/usr/local” will be handled by it, while the driver for ”/usr” would be used for the rest, such as ”/usr/share” for example.

The RTEMS console driver registers itself to the name `/dev/console`. The file driver registers itself to the name ””, an empty string. Thus, all requests which are not handled by the drivers which are installed before the file system driver are handed to it.

6.2 The Actual File Driver

The following sections will discuss the structure of the file layer, the actual data transfer protocol and the implementation.

Figure 6.1 shows which parts of the software and hardware are involved in the file layer. The box on the left in figure 6.1 is the ACE side while the right side contains the host part of the transfer mechanism. As can be seen, the user application on RTEMS/ACE has to use the RTEMS API or the POSIX API to talk to the I/O client on the RTEMS/ACE side of the system. This I/O client makes its service requests by using the VSP lines to the server system. These transfers are routed through the various bus systems which connect the ACE and the server. On the server side, the requests are handled by a modified TSI device driver which connects to the host’s file system.

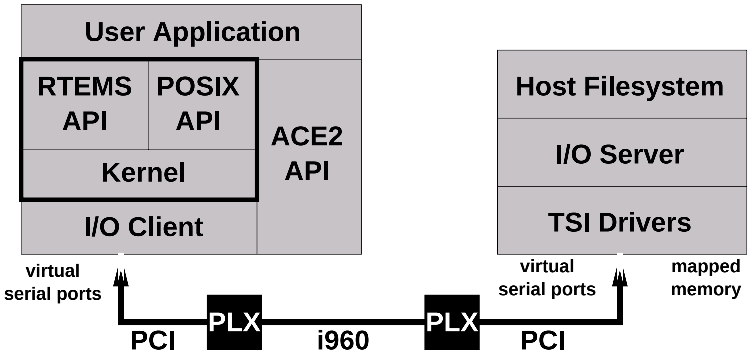


Figure 6.1: Route of data during I/O

6.3 The Structure of the File Layer

The file layer consists of two parts, which communicate to provide the same functionally as a native file system would provide to the RTEMS/ACE system. The client is located on the RTEMS/ACE side and translates RTEMS file system calls to requests to the server and accepts the handshakes from there.

On the server the requests made by the client are translated and handled by a Finite State Machine (FSM). The server performs the action requested by the client and answers the client with a handshake confirming that the operation requested is completed.

6.4 The Protocol

The protocol used to transfer data between the host and the server only knows one direction of command flow, from the RTEMS/ACE to the server software. Data transfer is then performed by reading and writing the memory on the ACE card .

The cornerstone of the protocol is a structure named `handle_client`, which contains all operational data for one file-handle.

```
struct handle_client {
  union ace_code ace_code ;     /* Parameters */
  long int result ;             /* result from last action */
  enum command comm ;           /* What is to be done ? */
  char allocated ;              /* 0, free. != 0, allocated */
} ;
```

The entry ”ace_code” is a union that has a different contents depending on the kind of service the client wants the server to perform. To distinguish the service types, the ”comm” entry exists, which holds one of the following command codes. These are :

```
enum command {
    com_none = 0,
    com_open,
    com_close,
    com_read,
    com_write,
    com_ioctl,
    com_done
};
```

Please note that there is no explicit entry for a seek operation. The seek operations are handled by RTEMS internally and are not visible to the file layer. Each read() or write() operation is accompanied by the correct offset from the beginning of the file. Thus, no separate seek operation needs to be performed because RTEMS changes the current file position in its own file description object instead. The only data which needs to be transfered to the server is now the index into the global array of ”struct handle_client”. This array is located in the ACE memory space at a fixed address at 512K (see Figure 5.1, 'Communication Area').

Actual I/O is initiated by the client setting up a communication structure for a file and telling the server to perform the desired action on that node. The file handle number is transfered by a VSP line and is only one byte in size. The server then accesses the structure in the mapped ACE memory (using his device file) and gets all information needed to handle the request. The transfer of data to be written to a file or read from a file involves further memory transfers with the payload data. After updating the structure with a new command entry (com_done) in the ACE memory, the client considers its requests as being served and continues execution. For stability reasons all operations are handled synchronously. Some operations could have been implemented using asynchronous actions on the server side, but that was not done because of the possibility of race conditions and because the speed increase would not have been worth the increased complexity.

Following is a description on how the protocol works in respect to the command flow and data flow. After that, an example for a typical action is explained in detail.

6.4.1 Command Flow

Commands are passed from the client to the server by using a VSP line. The client only transfers the number of the file's `handle_client` struct which is an offset into the global array (see Figure 5.1) of these structs.

On the server side the command is read from the `/dev/ace0ttyb` device file which the server opens on his start up. When a command is read, the value is treated as offset into the array. The `handle_client` struct which is addressed by that command is copied out from the clients memory to the host and inspected for what action has to be performed. The offsets `0,1,2` are treated specially because they correspond to the `stdin`, `stdout` and `stderr` file handles of the server. That way, the host console the server is started in is connected to the RTEMS/ACE console. However, some command values sent by the client do not map to file handles, they serve an administrative purpose. These values are:

- 120
  If the value 120 is sent as a command to the server, this means that RTEMS/ACE has reached the exit code and that it wishes to terminate. The server then resets the card , frees its resources, and exits.

- 122
  This is a special link-up value that is written by RTEMS/ACE on startup. A block of several bytes with this value is written so the VSP can synchronize. This is necessary because of a VSP driver problem that can lead to data loss of the first few bytes transfered (see section 5.5). To avoid that commands are dropped due to this initial unreliability, the value 122 is sent by the client (ACE) and ignored by the server.

- 0...99 These are valid file handle numbers. If a command is read, its correct `handle_client` I/O block is transfered from the ACE memory to the host side and the operation specified in that block is executed.

6.4.2 Data Flow

Transfer of all data is done in the memory of the ACE card. The memory space of the ACE card is accessible on the host by a device file. By reading and writing this file, the memory contents on the corresponding memory location on the ACE card is read from or written to. The server implements an abstraction layer above this mechanism. It has two functions called ”copy_in” and ”copy_out”, which handle all address translation required. They are used to read communication structs from the ACE memory to a local copy in the server and to re-transfer them back to the ACE memory. Between these two actions, additional transfers of payload data between the two bus systems occur. By providing this additional abstraction layer, the server is easily adapted to different ways of memory access.

6.4.3 The Host Side

The host side implementation of the protocol can be found in the server/server.c file. The server side is implemented as a Finite State Machine (FSM) with one central state where different input switches the state to the different service states. There is only one thread of execution. After the requests have been handled, the flow of control returns to the central state and waits for the next action to be performed. The routines for accessing the ACE memory are also found here along with the setup code for the RTEMS/ACE command line.

The server attaches a timeout to /dev/ace0ttyb, the line file descriptor that connects to the ACE card in use. This is important because the server would stay in the kernel read code waiting for incoming data even if the RTEMS/ACE application had crashed. To avoid this, the file descriptor has to be set to timeout after a short period of time so the read call returns with a read length of zero. When that happens, the server is in the user mode again and can be terminated interactively by ’control-c’. This is part of the server source code, which sets the file handle modes using ioctl. The code is stripped down a bit by removing the parts which are not vital for proper functionality.

```
unsigned char get_command()
{
  unsigned char x, c ;
  int tb, s, i;
  struct termios tios;
  static int sleep_val=0;

  do {
    /* Set a timeout of 1 sec */
    s = ioctl(fd, TCGETS, &tios);
    tios.c_cc[VMIN] = 0;
    tios.c_cc[VTIME] = 01;
    s = ioctl(fd, TCSETS, &tios);

    /* Do the read and check result */
    if (read(fd, &x, 1)==1) {
      /* Got something */
      return x ;
    }
    /* ... timed out */
  } while (1);
}
```

6.4.4 The RTEMS/ACE Side

The RTEMS/ACE side of the protocol is implemented in the file server/rtems_client/rtems_client.c. Since this is also the default startup code for RTEMS/ACE applications, additional operations are implemented here. Most noticeable is the file layer, which is implemented mostly in this file.

The protocol driver opens the VSP line to the server and sets up the argc and argv command line parameters for the main() function in user- supplied code. The initialization function of the protocol first establishes contact with the server by initializing the VSP line which is used for command transfer. After successfully initializing that line, the client starts writing a block of the commands with the value 122, which is to be ignored by the server if it is actually received by the host. This is done to sync the VSP lines.

After that the command line arguments, which were left at a fixed address by the server, are copied over to a dynamically allocated memory block because that address range is then used for the array of struct handle_client. The entries are all cleared and the first three are preallocated because they are hardwired to the stdin,stdout and stderr file handles of the server.

6.5 The Actual Implementation

The actual implementation of the protocol mentioned above is done in 'C', as is the rest of RTEMS/ACE and the download tool by TSI. This section will describe implementation details and what is where.

6.5.1 Parts of the File Driver

The complete RTEMS/ACE file driver system consists of the following parts:

- newlibif
  This stands for newlib interface. It is part of RTEMS. This file contains code that replaces Unix semantics with RTEMS semantics and calls the RTEMS driver code for the functions. It translates modes for opening files from the Unix semantics to the RTEMS key values and keeps track of file descriptors. The newlib [8] itself is a small C library that is designed for embedded systems and which provides basic features such as `malloc()` and `fopen()`. A normal libc from a Unix system would be much to big to fit in the limited resources of an embedded system.

- File Driver
  The file driver is a new part and is configured into the BSP that is the base platform for user programs. It registers itself as a RTEMS device driver and answers all calls which have not been acted upon yet by other drivers. The drivers are organized as a chain, and a driver answers to a name if it can. The file driver is the last one, and thus it is allowed to answer to all names coming in. RTEMS special names for device drivers, like "/dev/console" are resolved before the file layer is asked, and thus not passed to the file layer for service requests. Since RTEMS does not provide a directory structure here, reading e.g., the hosts "/dev/cdrom" is possible and will be resolved to the file I/O layer.

- Host Driver
  The host driver replaces the `ace2download` tool and starts the programs on the microSparcIIep . After transferring the binary image into the ACE memory, the command line arguments for the program are also copied over to the ACE memory space so the RTEMS/ACE application will find them. Not all options are copied over to the RTEMS/ACE side because some options can be for the server. Please refer to Appendix B for details. After that setup is complete the ACE firmware is told to start execution of the code.

- Communication Area
  At a fixed address (524288, or 512K) of the memory on the ACE card some memory serves as an array of communication channels. Each channel is a structure which contains the parameters of a call and a handshake tag. The channels are allocated by the `open()` call on the RTEMS side and stay connected with the file they are assigned to. They contain the information which is normally passed to the `open()`, `close()`, `read()`, `write()`, `seek()`,... calls plus special status information about the action to be performed by the host. The handshake word is used to signal RTEMS the host is done with I/O. For more details about the data structures please refer to section 6.4

6.5.2 Example call chain

As an example, here is a list of steps which is executed for an `open()` call by the user program until it returns:

- `rtems-4.0.0/c/src/lib/libc/newlibif.c | open()`
  The newlibif provides the entry point for `open()`. The call is simply forwarded to `__rtems_open()` which is part of `rtems-4.0.0/c/src/lib/libc/libio.c`.

- `rtems-4.0.0/c/src/lib/libc/libio.c | __rtems_open()`
  `__rtems_open()` translates the mode flags for the call to the RTEMS I/O manager from the Unix style to the one used internally by RTEMS. It also allocates an RTEMS file descriptor object out of a fixed storage array of descriptor objects. Object allocation is done under semaphore protection. According to the name of the object to be opened, the driver list is searched for the appropriate driver. That reference is remembered in the I/O object for faster dispatching of I/O. The new object is then initialized with the values for the RTEMS I/O subsystem (name, permission bits, access bits, ...) and passed to it via `rtems_io_open()` in `./c/src/exec/sapi/src/io.c`.

- `rtems-4.0.0/c/src/exec/sapi/src/io.c | rtems_io_open()`
  `rtems_io_open()` calls the drivers `open()` via a function pointer. The pointer can be obtained by referencing the driver information which is part of the I/O object. That pointer is set during initialization of the device driver and originates in the BSP package.

- `BSP | open()`
  This is the new driver. Its `open()` call allocates a new I/O channel structure from the global array and places the channel's index number inside a private extension slot which is part of the RTEMS I/O object. The channel structure is then set up with the command values (see 6.4 for them) for the Unix `open()` function. Finally the number of the communication channel is sent to the host via VSP.

  The RTEMS I/O control flags need to be translated to Unix flags here, the inverse of the translation already performed before. The translation that happens here, from Unix to RTEMS and back, must not be ommited because the `open()` does not neccesarily target the file I/O layer. It must translate the flags the user application passes in to the RTEMS flags. The file I/O layer now knows it is going to talk to a Unix machine, and so it needs to translate the RTEMS flags it receives back to legal UNIX flags.

  After the command is sent to the host, the client is waiting for the host handshake. This handshake is performed by updating the I/O channel structure in the ACE memory by the host. The host places a new value (`com_done`) in the command entry (struct handle_client) of the I/O channel struct, notifying the client that the I/O has been completed. Due to the polling mechanism used, that memory area has to be non-cacheable. The currently used method is to disable the data caches right before the loop and re-enable them again after successful completion of the handshake.

  Since manipulating the cache status requires dealing with hardware resources of un-shareable nature, that part is protected by a semaphore. Otherwise, a race-condition might occur with one task finishing I/O and re-enabling the caches. That would prevent an other task from receiving his reply because its `'struct handle_client'` would be cached.

- returning
  After the handshake and the re-enabling of the data caches, the call-chain is walked back. The link between the file and the communication structure is saved in the RTEMS file handler object. It remains valid for the whole lifetime of that file handle. For more details about the data structures please refer to Section 6.4

6.6 The Host Side

The Host side of the file layer consists of a slightly modified version of the download tool which was provided in source form by TSI-Telsys. Here, the host part is called *server* because, after startup of the application, it does nothing but answer requests from the ACE card . The binary representation of the RTEMS/ACE program and the server are combined by a Unix feature originally used for scripts.

Scripts are files of unknown architecture, that means the kernel does not know how to start them directly, but they are marked executable. Now the first line of the file is examined if it holds a valid path to an interpreter binary. Usually, that would be `/bin/sh`, but in our case, it is the path to the server binary `rtemsserver`. The server binary then gets started with the binary as an argument, plus any command line arguments given on the command line. It loads the binary, skips the leading interpreter path line, if present, and uploads the rest onto the ACE card.

After the application has been started and the contact has been established, the RTEMS/ACE side acts as a client. It uses the VSPs to transfer the commands to the server side. The server side waits for incoming commands and processes them one at a time.

Serialization is guaranteed by a semaphore in the RTEMS/ACE file layer which ensures that only one file I/O operation at a time can be issued and completed. This semaphore is allocated before an I/O command is sent to the server. The CPU caches are flushed to main memory to enforce data consistency. After the server has completed the requested transfer, and written to ACE memory, the CPU caches are flushed again to ensure that CPU cache is consistent with the main memory of the ACE card . Finally, the semaphore is released again.

The setup of the `'struct handle_client'`, however, is not protected by this mechanism. In order to minimize the latency of the file operations, the single-threaded part is kept as small as possible. Avoiding races in the file layer would only make the possibility of races smaller, but not eliminate them. The RTEMS I/O code was not designed to ensure that multiple threads can write to the same file handle. Code doing that is bound to fail on RTEMS/ACE sooner or later, and by making the window for the fail larger increases the possibility to find such bugs early in the testing period. Syncronisation of multiple threads doing I/O to the same file requires protection on the application level.

6.7 Console I/O

Console I/O for RTEMS/ACE was initially implemented solely using the VSP mechanism. After the file layer was completed, there was no reason why the console should not use the same way to transfer the data to the server console. After all, the file layer provides higher bandwidth than the VSP lines.

For this purpose the new console driver creates three files which are implicitly. They are connected to `stdin`, `stdout` and `stderr` on the server side. Because the new console uses the file layer, debugging output via `printf()` is not possible before the file handler is initialized. Should a new driver be implemented that needs to be initialized before the file layer, debugging output should use the `vsp_puts(const char*)` call which writes to the boot console. The boot console is connected to the `/dev/ace0ttya` VSP device file on the host side.

Chapter 7

Conclusion

The port of RTEMS to the ACE hardware was done to simplify software development for adaptive computing. RTEMS/ACE has been in use for some time in different places and is, so far, doing well. It has proven to be a reliable and efficient programming vehicle which frees the programmer from a lot of burdens and lets him focus on the main problem. This work took longer than planned due to the different time zones in which all parties are located. In addition, the requirements and specifications changed as research continued. Also, no program beyond a certain size is ever ready, there is always room for improvements.

The central points of the project were the porting of RTEMS to the ACE hardware and creating a file system emulation layer inside RTEMS. Both goals have been achieved. Porting RTEMS to the ACE hardware was easy. After all, the RTEMS kernel is designed to be portable. Actual porting guidelines were rare, so most of the porting process had to be figured out by code review. First versions of the RTEMS/ACE kernel booted soon after the porting began. Adding the file layer introduced also the need to guard against stale data in CPU caches due to multiple DMA busmasters.

To conclude, RTEMS/ACE has reached production quality level of reliability and can be used in daily research and production work.

A

Troubleshooting

This section will discuss how to determinate the cause(s) of abnormal behavior. One can read the boot messages of the RTEMS kernel by attaching a terminal program such as `tip` to the VSP[0] channel. An example command for Solaris 2.7 on the host accessing an ACE card could be

```
tip /dev/aceII0ttya
```

on the host workstation (under Solaris 7.7/sparc) used for the initial port and an ACE2 card . The ACE card (not ACE2) has a different naming of the device files. For the ACE card the command would be:

```
tip /dev/ace0ttya
```

Notice that the ACE cards lacks the `II` part of the name. The number digit is to distinguish different cards of the same type in a single host computer.

In some situations RTEMS/ACE will fail to perform as expected. What follows is a list of possible failures and their explanation along with hints on how to solve them.

- No memory size test messages appear, nothing happens at all
  The VSPs used for data transfers need some I/O to come in sync. For this purpose, RTEMS/ACE sends a set of ignore-me commands to the server by the command VSP first. If the VSPs do not sync, then no I/O is possible. This might be cured by lowering the load on the host, choosing different hardware for the host or by a fixed version of the VSP driver code in the host kernel.

- Program dies during memory size test
  That means one cannot read the reported memory size. Something is wrong with the installed memory. RTEMS/ACE does its own memory size calculation based on mirror detection[1]. If the memory address engine on the board is producing results differently from the ones planned for, then this might happen. It will also happen if there is not enough memory available to hold the initial image. Also, RTEMS/ACE expects at least 12 Megabytes of main memory to be present. The memory size test should tell the size of detected memory. If it differs from the actually installed size, the memory may be faulty.

- Insufficient memory installed
  If RTEMS/ACE finds the memory insufficient, it did not find enough free memory to hold the RTEMS memory pools and the libc memory area, RTEMS/ACE halts execution after displaying this error. This means the program declares a lot of static storage which might better be dynamically allocated. Another reason might be that RTEMS itself allocates a big address range for its own memory area. One may find the RTEMS configuration routine (see section 5.7) useful here.

- Server does not obey 'ctrl-c'
  The virtual serial port driver is buggy. The ACE server sets the file descriptors for the VSPs it allocates to a timeout read mode[2]. If that fails, the server waits inside the kernel read function for incoming data and is not interruptible. Because it still has a lock on several resources, it is not possible to reset that ACE card . Reboot the machine and update the VSP driver to one that supports timeouts.

- lock is set
  The server maintains a lock-file to make sure that only a single server accesses the same ACE card . If a RTEMS/ACE application or the server prints out "Error : lock is set . . . ", then the lock-file exists. This may be due to an already running application on that card , or it may be a leftover from the last server running that card . If a server gets killed by hand, it cannot remove the lock-file again. This is extremely annoying if that lock-file is owned by some other user and one cannot delete it even

[1] That means that the end of the physical memory is detected by observing the side effects a write outside the physical memory has. This write can be redirected to location NULL (mirroring the address range) or simply not work at all.

[2] This is done using `VMIN` and `VTIME` in an `ioctl()` The include file is `<tty.h>` or `<termios.h>`, depending on the Unix version in use.

if it is stale. The use of lock-files can be turned off by using the `-l` switch to the server.

A.1 Compatibility

The host driver silently assumes that the data types on both, the microSparcI-Iep and the host system, have the same representation and alignment restrictions. For the current version, which uses SPARC on host and client sides, this does not matter. A port to some little-endian hardware (e.g. Linux on Intel's) will need some attention in that area.

B

Usage

rtemsserver(1)

NAME

rtemsserver

SYNOPSIS

rtemsserver [flags] [file]

DESCRIPTION

The rtemsserver program is the host resident part of the RTEMS/ACE system. It is responsible for controlling the ACE card and executing the I/O transfers the RTEMS/ACE application requests. The server is derived from the original ACE tool `aceiidownload` which is supplied with the card .

The RTEMS/ACE server understands the following command line options:

-c <**card**>

Sets the card number the server should use to run the RTEMS/ACE application. Since more than one card may be present in a system they are enumerated and differ in the number which is also a part of the name of their device driver interface file.

-d

Enable debugging of the RTEMS/ACE application. The application is not launched but stopped at the first machine instruction. By using the GDB debugger, a capture of the application is possible.

-l

By issuing this switch the rtemsserver does not use lock files to make sure it does not try to use a card which is already in use by another rtemsserver.

-e <**address**>

Specifies the core address of the first instruction. The binary RTEMS/ACE application will be loaded to this address. No checks are done whether it is relocated for that address or not. Since the default relocation address is one megabyte, the default for the **-e** option is the same.

-v

Be verbose. The server is printing additional information to the console about what it is going to do. This option is off by default since it could be confusing to have the server console output mixed with the output of the RTEMS/ACE application.

-P <**path**>

Set the current directory to <**path**> after loading the RTEMS/ACE application binary. This improves usability in case of scripting the server separately from the binary.

-X <**prefix**>

Set the prefix of the /dev/ files to <**prefix**>, so an old ACE1 card can coexist with newer ones. The ACE1 had /dev/ace. . . entries while the ACE2 has /dev/aceII. . . This string defaults to ”aceII”.

-i

This means ”ignore” and tells the server to quit parsing the following options. Options given after this flag are passed to the RTEMS/ACE application. So, if the application needs to get an option which is also understood by the server, it can be placed after this ignore option and is then correctly transfered to the RTEMS/ACE.

If the binary starts with a line that specifies an interpreter binary to the shell, all the options set on the command line are transfered to the application. The option **-i** can be argued to be obsolete in most cases, but it can be useful nevertheless.

In addition to the command line flags the server understands a set of environment variables. These are evaluated before the command line is parsed, so their values can change the default settings but can still be overwritten by settings specified on the command line. The environment variables are :

ACE2CARDNUM
The card number the server tries to open defaults to 0. This default value can be changed by setting this variable to a different value. The previous default card is then not tried. The value can be overwritten on the command line.

ACE2PREFIX
The prefix under which the device driver files are searched defaults to ”`aceII`”. In case only ACE1 cards are installed the default can be set to ”`ace`” with this environment variable. Again, the command line can overwrite the value assigned to the default setting.

Example

```
sieve.run
```

This will run the application ’sieve’ without any arguments to either the server or the application.

```
sieve.run -c3 -Xace
```

The application will get the arguments -c3 -Xace, not the server !

```
rtemsserver -d cat.run logfile -v
```

This will start the application ”`cat`” with the parameter ”`logfile`”, the options -d will go to the server, -v will not. The server will load the application and place a breakpoint at the first instruction there, -v will be an argument to the application.

B.1 RTEMS/ACE

Now, how can RTEMS/ACE be put to use ? The normal way is to first create a cross compiler (gcc) for the microSparcIIep CPU. All the binutils and newlib are needed also. With these tools installed, the application can be compiled to object code. On the development system, a makefile might look like this:

```
# sample RTEMS/ACE2 makefile
# AHK991207

# the target program

PROG=hello

# path to the RTEMS installation directory
#   (containing bin, sparc-rtems etc.)

RTEMSBIN=$(RTEMSACE)/bin

# the RTEMS/ACE2 IO server

SERVER=$(RTEMSBIN)/rtemsserver

# the compile-flow tools

CC=$(RTEMSBIN)/sparc-rtems-gcc
LD=$(RTEMSBIN)/sparc-rtems-gcc
COPY=$(RTEMSBIN)/sparc-rtems-objcopy
CCOPTS= -g -fasm -specs bsp_specs -qrtems -O3
LDOPTS= -g -fasm -specs bsp_specs -qrtems -O3
COPYOPTS=-O binary

# make rules

default: all

$(PROG).exe:    $(PROG).o
        $(LD) -o $(PROG).exe $(PROG).o $(LDOPTS) -lm
.c.o:
        $(CC) -c $(CCOPTS) $<
```

```
$(PROG).bin: $(PROG).exe
        $(COPY) $(COPYOPTS) $(PROG).exe $(PROG).bin

all: $(PROG).bin
        echo "#!$(SERVER)"|cat - $(PROG).bin\
          > $(PROG).run
        chmod a+x $(PROG).run
        @echo "OK, RTEMS/ACE binary for $(PROG) built.\
              Run it using \n\t./$(PROG).run"
clean:
        rm $(PROG).exe $(PROG).bin $(PROG).run $(PROG).o
```

This Makefile is used to create a simple ”hello world” program to test the RTEMS part of RTEMS/ACE. During the build process some files are created with unusual suffixes. The initial compile creates a file that ends in ′.exe′. This is a normal executable binary file, it contains symbols and is needed for debugging purpose. The ′.exe′ file is then transformed into a raw binary which ends in ′.bin′. That raw binary contains only a raw dump of the code and data segments the ′.exe′ file held. All additional information is stripped. This file is the memory image for the ACE card .

After being loaded to the originate address of 1M, it can be started. To make the process of starting it simpler and intuitive, the ′.bin′ file is related with the rtemsserver program using the UNIX interpreter mechanism so it can be started by just typing its name in a shell window. The resulting file therefore ends in ′.run′.

B.2 Debugging RTEMS/ACE

As mentioned earlier, one positive aspect in keeping the ACE firmware alive is the bonus of the GDB debugging hook. To use it, 2 terminals are needed. They can be 2 console screens or 2 xterm windows. If one also wants to see the boot output, a third console / xterm is needed for that.

For example, lets assume you want to debug a program named ′′foo.run′′. Start your program. Because you need to give an extra option to the server, the binary cannot be started alone. You need the explicit invocation form.

```
rtemsserver -d foo.bin
```

In case `foo.bin` needs parameters and options, place them behind the name of the binary file, like in

```
rtemsserver -d foo.bin -optsToFoo ArgsToFoo
```

Change to the second console and start the debugger. The application is already loaded now and waits for the debugger to connect.

```
./projects/test> gdb
(gdb) sym foo.exe
(gdb) target remote /dev/aceII0debug
(gdb) break main
(gdb) c
```

The first command to GDB forces it to load the symbols from another file. Since the debugger has no knowledge what it is going to debug, it cannot know where to find the symbols. During the build process of `foo.bin` the standard executable binary file `foo.exe` was created. This file contains the symbols we are interested in. The line

```
(gdb) sym foo.exe
```

makes the debugger read the symbols from that file. Now we can work with names rather than numeric addresses. The next logical step is to connect the debugger with the remote debugger hook in the ACE firmware. The line

```
(gdb) target remote /dev/aceII0debug
```

does that, assuming the binary was started on an ACE2 card with number zero. Next, it should be a good idea to place a breakpoint so the program stops at an interesting point once we let it run. The GDB command

```
(gdb) break main
```

will place a breakpoint on the `main()` function. Since the program is already started, we do not need to start it with a `run` command. The `rtemsserver` placed a breakpoint on the very first instruction, so we simply need to *continue* the execution. The short form of the `continue` command in GDB is the simple `''c''`. So the use of this line

```
(gdb) c
```

will continue the program and stop it at the `main` function where a breakpoint had been placed. Please be advised to never let the debugger run free, because it will not recognize the termination of the program by itself. The resulting hang of the debugger can be removed by giving it a KILL signal and after that flooding its connection channel (VSP 1) with arbitarydata. The ACE binary `smack.bin` does that. Writing that one from scratch is easy.

While being able to source debug RTEMS/ACE is a good thing, there are limitations to the usability of the debugger. It is not always possible to debug an application in detail. The problem here is that GDB thinks of the RTEMS/ACE as one application, but this application does evil tricks which GDB cannot understand. One example is: Place a breakpoint on your `main()` function and after arriving there, call for a stack back trace. GDB will tell you that `main` is the outermost layer and no back trace exists. RTEMS implements a powerful multitasking, and with that comes the need for exchanging stack pointers. So, if a breakpoint is hit, that can be any task of RTEMS, and GDB might be a bit confused by the stack layout implemented by RTEMS and by the absence of a thread model it understands.

B.3 High Resolution Timers

To aid the process of benchmarking, RTEMS/ACE supports a fine granularity clock function which has a resolution of 1/4 of the CPU clock frequency. In case of the currently available ACE and ACE2 cards that means 25MHz timing frequency or 40 nanoseconds per clock tick. The file I/O driver knows about this clock and uses it to keep track of the delay time it spends waiting for the server to answer the requests. This idle time then can be subtracted from the runtime to get the time the CPU was actually doing work. For details, please refer to the example source code in section B.3.1 and B.3.2.

One issue is that the time spent inside the RTEMS scheduler is not taken into account. Other interrupt times are included, however. This is important because the SPARC architecture has window- underflow and -overflow traps. Following are the manual pages for the benchmarking support code.

B.3.1 RTEMSIO_getSystemTicks

NAME

RTEMSIO_getSystemTicks

SYNOPSIS

long long RTEMSIO_getSystemTicks(void);

DESCRIPTION

RTEMSIO_getSystemTicks() returns the number of 25 MHz clock ticks since the initialization of the scheduler clock. The time spent inside the scheduler is not accounted for.

EXAMPLE

```
#include <stdio.h>

extern long long RTEMSIO_getSystemTicks() ;

/* assume sieve being used to eat up some time */
extern int primes(int max);
int main(int ac, char** av)
{
  long long start, end, overhead ;
  int p ;
  overhead = RTEMSIO_getSystemTicks();
  start = RTEMSIO_getSystemTicks();
  p = primes(8192);
  end = RTEMSIO_getSystemTicks();
  printf(‘‘found %d primes\n’’, p);
  printf(‘‘time : %L\n’’, end-start-(start-overhead));
}
```

BUGS

The internal code uses an unsigned long integer to count the scheduler timeslices since scheduler initialization. This counter will overflow after 2^{32} milliseconds. That is more than 1193 hours. Please note that the return type of 'long long' is not a typo. The result is based on the number of timeslices (uint32) times ticks_per_timeslice. To prevent overflows, the calculation is done in the next higher ranged datatype.

B.3.2 RTEMSIO_getIdleTicks

NAME

RTEMSIO_getIdleTicks

SYNOPSIS

long long RTEMSIO_getIdleTicks(void);

DESCRIPTION

The function RTEMSIO_getIdleTicks returns the number of ticks which were spent in the file I/O code waiting for the server to handle a request. Normally, the time spent in the I/O code waiting for the server is counted by the benchmarking ticker in the total time statistics. To be able to estaminate how much time a program actually needed to fulfill its job, this idle time (similar to ”system time” on Unix) needs to be subtracted.

EXAMPLE

```
#include <stdio.h>

extern long long RTEMSIO_getIdleTicks() ;

int main(int ac, char** av)
{
  long long start, end, overhead ;
  FILE* in, *out ;
  int i,len ;
  char buff[1024] ;
  in = fopen(‘‘input’’,’’r’’);
  out = fopen(‘‘output’’,’’w’’);
  if (in && out) {
    overhead = RTEMSIO_getIdleTicks();
    start = RTEMSIO_getIdleTicks();
    do {
      len = fread(buff, sizeof(buff[0]),
        sizeof(buff)/sizeof(buff[0]), in);
      fwrite(buff, sizeof(buff[0]), len, out);
    } while (len == sizeof(buff)/sizeof(buff[0]));
    end = RTEMSIO_getIdleTicks();
    printf(‘‘Copy idle time : %L\n’’,
        end-start-(start-overhead));
    fclose(in);
    fclose(out);
  }
}
```

BUGS

The file I/O code which calculates this values is using `RTEMSIO_getSystemTicks()` to do the measurements, so if the wraparound of the timeslice counter falls in the time an I/O operation is currently waited on, the idle time will be corrupted.

C

Glossary

- Adaptive computation
  In the research field of adaptive computation the aim is to blur the boundary between software and hardware. A program no longer consists only of a set of processor instructions, but also a set of problem-specific hardware which is loaded into the hybrid computer as well. A special compiler can optimize the program by identifying parts of the code which can be executed in hardware. These parts then get translated into a description that can be loaded into the FPGA part. Speedups in an order of magnitudes have been observed up to now. The compiler technology for this field is currently a research field, too.

- BSP
  Board **S**upport **P**ackage.
  A set of support routines which enables RTEMS to talk to a specific platform. RTEMS only uses the well-defined interface to the BSP. The task of the BSP is to supply the hardware drivers required for the functions defined in the interface.

- Endian
  Different CPU types use different ways to store information in main memory. There are almost as many ways used to order the bytes of a multibyte value in memory as there are theoretical possibilities. The most widely used ways are little and big endian. Under big endian, the most significant byte is placed at the lowest memory address of the multibyte value while under little endian the most significant value is written to the highest address. So, for example the value 0x11223344 would look like 0x11,0x22,0x33,0x44 on a big endian system when read as a byte sequence starting at the address of the multibyte value. On a

little endian system the sequence would be 0x44,0x33,0x22,0x11 [1]. The little endian order is used in x86 and compatible CPUs while the big endian is used in the Motorola series, the PPC, Sparc and MIPS. Each side has its advantages and disadvantages.

- FPGA
  Field **P**rogrammable **G**ate **A**rray.
  This is a gate array which can be reprogrammed without bulky equipment like the fuse burners used to program normal PLAs (Programmable Logic Array). The connections between the gates are done by a software modifiable switch matrix which is loaded into the FPGA from a dedicated SRAM. This allows, for example, hardware updates by disk and also allows for custom hardware which is tailored to the currently running application.

- HAL
  Hardware **A**bstraction **L**ayer.
  Typically, a HAL provides equal access to different hardware. The HAL is one of the lowest layers of an operating system. Since all accesses to the hardware are done by the HAL, changed hardware onlys need a change to the HAL and avoids modifications spread over the entire system.

- RTEMS
  RTEMS stands for"Real Time Executive for Multiprocessor Systems". This means the operating system does enforce a fast interrupt dispatch and, if needed, a fast context switch to the task which gets a signal, provided it is currently waiting for it. RTEMS is portable and scalable, allowing for use on small system hardware up to multi-processor systems.

- NFS
  Network **F**ile **S**ystem.
  Invented by SUN to make the connection of multiple workstations easier by allowing for shared disk areas. Using NFS, it is possible to find the own home directory and common file base in place no matter which workstation is actually being used.

- OAR
  On-Line **A**pplications **R**esearch Corporation.

[1] On the PDP, a 32-bit value is stored as 0x33,0x44,0x11,0x22, but a working PDP is hard to find these days.

This corporation is the distributor of RTEMS. They can be contacted by Internet access to ”www.oarcorp.com”. The download section of their web-site has all the needed parts to build a complete cross development system. This is the status as of late 2000. Changes may be expected. The postal address is :
4910-L Corporate Drive
Huntsville, Alabama 35805 USA

- PCI
  Peripherial **C**omponent **I**nterconnect.
  A commonly used bus system in desktop PC systems and workstations. The PCI bus has a number of slots in which add on cards can be plugged to supply hardware features not included on the mainboard.

Bibliography

[1] John L. Hennessey / David A. Patterson *Computer Architecture : A Quantitative Approach*
Morgan Kaufmann Publishers,760 Pages, Aug. 1995, ISBN 1558603298

[2] Martin Fowler / Kendall Scott *UML Distilled*
Second Edition. Addison-Wesley, 185 Pages, 20. Aug. 1999, ISBN 020165783X

[3] Craig Larman *Applying UML and Patterns*
Prentice Hall, 507 Pages, 1. Mai 1998, ISBN 0137488807

[4] Andrew S. Tanenbaum *Moderne Betriebssysteme*
2. Auflage. Hanser Verlag, 1995, ISBN 3446184023

[5] Horst Langendörfer / Bettina Schnor *Verteilte Systeme*
Hanser Verlag, 1994, ISBN 3446174680

[6] Prof. Snelting *Software Engeneering*
Lecture at TU-BS in WS 1994/1995

[7] GNU binutis : *GNU binary utils and libraries*
ftp sourceware.cygnus.com
/pub/binutils/releases/binutils-2.9.tar.gz

[8] GNU newlib : *GNU linker library for small systems*
ftp sourceware.cygnus.com
/pub/newlib/newlib-1.8.2.tar.gz

[9] GNU CC : *GNU GCC portable C compiler*
ftp sourceware.cygnus.com
/pub/egcs/releases/gcc-2.95.2/gcc-2.95.2.tar.gz

[10] RTEMS Source : *RTEMS source codes & documentation*
www.oarcorp.com

Appendix D. Profiling Tools and Result Viewers for the Nimble Compiler Project

1 Loop procedure Hierarchy graph (LH)

Nimble tries to accelerate compute intensive loops into a configurable co-processor. As such, it is important to understand where time is being spent in a program. Typically, profiling programs like Pure Software's *Quantify* will provide a *procedure call hierarchy* with time spent in each procedure or possibly basic block / statement. Vector / Loop compilers for FORTRAN provide a *loop hierarchy* such as with SUNWPro's FORTRAN compiler. Nimble provides a unique *procedure call - loop hierarchy* graph.

Both procedure call and loop hierarchy are needed to understand effects such as loop distribution and inlining -- two example transformations that affect the program structure and the respective contributions to the overall execution time dramatically.

The Loop Procedure Hierarchy graph created by **`loophier`** shows procedures as square boxes and loops as circles. Each static call to a procedure or static invocation of a loop entry is shown with an arc. Note, that for this reason, there may be many arcs from one graph node to another.

The option **`-np`** is often used with **`loophier`** to remove system procedure calls (such as **`printf()`**) which tend to clutter the graph with needless information and turn a nice tree into a DCG/DAG (or rats nest). System calls, if present, are grey shaded boxes.

The option **`–target <targetname>`** is used to define the target platform, with **acev** being the default.

Note that the **`loophier`** report tool is called after an automatic inlining step. The inliner attempts to make more loops feasible by inlining procedure call hierarchies that do not contain any loops. One should turn off the automatically enabled smart inliner if one wishes to see the original program.

`loophier` can be used with or without profiling information. If profiling has occurred before **`loophier`** is called, the nodes in the graph are annotated with the amount of time the node contributed to the overall execution time of the application and scaled in size accordingly. This helps focus the eye directly to the areas of interest. Loops with execution time greater than 1% of program total exeuction time are shaded in yellow. Note, as with all profiling, results of relative application time can be very data value or command invocation parameter dependent. Option **`-prof`** is required in order to annotate the graph with profiling information. The default of **`loophier`** is to run it without profiling information.

Eventually, Nimble will focus optimizations by trying to increase the dwell time (or total time spent per entry) into the fewest number of loops. The goal of the compiler is to make the larger loops even larger to enable the maximum acceleration in the smallest amount of hardware kernels. Loop distribution, loop unrolling, loop fusion, and loop flattening are just some of the many optimizations planned as part of a heuristic approach to automatic transformation of the code.

loophier is based on static compile time analysis. If there are function pointers defined in the program, it can not infer what is the run time value of these function pointers. We introduce a "function pointer" node in the loop procedure hierarchy graph to represent all possible function pointers in the program. Each lexical invocation of a function pointer is shown with an arc going into the function pointer node.

2 Loop Entry trace Profile (LEP)

Very important information, if one is planning to dynamically reconfigure a shared resource, is to understand the path or trace through the code during execution; not just how much time was spent totally in a given loop. For example, lets say there are 4 loops, labeled A through D, that dominate the calculation and are being considered for a shared co-processor resource. Assume each consumes 20% of the execution time of the application. It is important to understand whether one starts in loop A and spends 20% of the time before moving onto B and so on. Or whether maybe .1% of the time is spent in A, then .1% in B, and so on to D before returning back to executing in A again. The former is a great candidate for a shared reconfigurable coprocessor as it only has to be reconfigured at 4 specific times. The latter will probably have too much overhead of switching contexts between kernels. The loop trace file gives the full trace of in-order loop entries for the given profiling run. Currently, the loop trace result is a text only file and not graphically presented.

The option **–ild** can be used to limit the profiling to only loops with execution time larger than 1% of application total time. The default for **lep** is to record the entry trace for all loops in an application.

There are two files that contain the resultant information: The ***.loop.info** file contains two part: the first part list the maximum lengths of forward paths in all procedures in the application. Forward paths do not account for feedback edges in loops. The maximum path lengths are useful in determining what K-value to use in HALT profiling. If the maximum path is very long, instead of using the default value of –1, we need to restrict K-value to be a small number, say 32, to limit the search space during the path profiling performed by HALT. The second part of the ***.loop.info** file contains a mapping of actual loop names to a numerically unique loop identifier. Then the ***.lep.trace** file contains the generated loop entry trace. Instead of just printing out the raw trace, which can take millions of bytes in storage, the loop trace profiling program compresses the trace at run time. The compression drastically reduces the trace size for most applications—this not only save storage space, but more importantly, the compressed trace allows fast traversing of the trace at later part of the Nimble Compiler flow. The compressed trace consists of loop numbers in the sequence they are invoked. The first file line lists the number of records (lines) that follow. Each record represents a single, compressed stream of behavior. For example, the template for each record format is:

```
n, m: x, z, d, … h, b, y
```

where

n is the repetition count for this record (that is, how many times did the following sequence occur)
m is the number of loops listed following the colon
x, z, d, … h, b, y is an ordered list of loop invocation occurrences. The whole list is repeated **n** times to get the full, uncompressed trace.

So, for example, the ***.lep.trace** file of:

```
1,10: 115 114 110 112 95 53 1 4 3 2
792,2: 55 54
1,1: 93
```

means that there was an initial series of 10 loops each executed once, followed by 792 occurrences of loops 55 and 54 being called in succession (note: this means 55 54 55 54 55 54 …. up to 792 times), followed by one loop (#93) being called at the end.

One could imagine finding sequences of repeating sequences though and further compressing the trace. One level of repeating hierarchy seems to be sufficient to capture most trace behavior in a reasonable, understandable file though.

Note that this is measuring loop entries, and not loop iterations or time spent in each iteration. Nested loops (Loop 1 being outer and iterating 32 times; loop 2 being inner iterating N times) might appear like:

```
1, 1: 1
16, 1: 2
```

Note that you cannot tell how many times loop 2 iterated in this case at all. Nor can you really tell how many times loop 1 may have iterated as loop 2 may be on a conditional (data dependent) path internal to loop 1.

3 Freq Dump

The Nimble compiler is actually doing a full path profile during the profile run utilizing a variation of Cliff Young's HALT tool (from Mike Smith at Harvards' HUBE group). The depth of the paths (number of basic blocks counted forward from any particular basic block) is parameterizable and defaults to infinite path lengths. A path length of 0 means simply count basic block executions without regard to where they were entered from or control goes to (path length of 0). **Freq_dump** in conjunction with **xvcg** is the method to view the basic block CFG and read the annotations of execution frequencies and paths down to this finer granularity. That is, **freq-dump** in conjunction with **Halt** provides a form of basic block tracing. Such a capability is mainly used for intra-loop optimizations to determine the frequency of potential exceptional exits, the cost of trimming basic block paths out of a conditional structure, and even for helping quickly estimate the software performance of the code without doing an accurate, detailed performance profile run.

freq_dump generates a separate graph for every procedure and contains all the basic blocks for that procedure. Once executed once, and the basic block numbers are detailed, it can be executed given a starting and stopping basic block number to report on. Not that the starting and stopping basic blocks must pre and post dominate each other; respectively. That is, when starting from the starting basic block you must always end up (no matter what path taken) at the stopping basic block. This is the only valid sub-set of a procedure CFG allowed.

4 Interesting Loop Detector (ILD)

Most of the statistics gathered or calculated are presented in the summary *Interesting Loop Detector* text table. General statistics about the whole application are given at the top. This is followed by a table with one row per interesting loop; sorted by estimated total time spent in that loop for the application. The format is further explained below.

4.1 ILD Options

- **`-h:`** print out usage information
- **`-v:`** print out detail operator counts for each loop
- **`-nosynth`**: do not include hardware synthesis information in the ILD report. Only print SW estimates for each loop.
- **`-target <targetname>:`** target platform. The default is acev.

4.2 Header

An example of the header is shown below. It goes the furthest to showing a number of items about the application overall.

```
####     Program totals: #loops    25,  total SW-time     463057, operators        625
#### Loops w/ t_per>10%: #loops     5,  total SW-time     66.79%, operators     14.40%
#### Loops w/ t_per>1% : #loops    12,  total SW-time     97.22%, operators     56.32%
#### Loops w/ HW kernel: #loops    21,  total SW-time     92.90%, operators     69.28%
```

The header has four rows of information, with each row consists of three metrics: number of loops, SW execution time and operator counts. In the first row, **`program totals`** include the total number of loops in the whole application, the program total SW time and the total operator counts. A loop is defined as a section of code with a back-edge. Any back-edge, even if formed by a goto (->) is considered a loop in the code.The **`Program Total SW Time`** is the quick *estimated* time that the program takes in cycles. This is based on using the path profiling, SUIF instruction counts per block, and other techniques. If a full, nanosecond clock is available on the platform of choice, an accurate performance profile of the application and selected important loops is available also. Operator counts are estimated based on SUIF operators as well.

The second row summarizes how many loops individually represent more than than **10% of the total application execution** time of the program. The number of such loops along with their **sum percentage** total of the applications execution time and sum percentage total of the application operator counts are listed in this row. Next, in the third row, the number of loops that represent **greater than 1%** each of the total application execution time are listed with their respective execution percentage and operator count percentage sum totals. Note that the latter statistics include all loops greater than 10% of the time also.

Embedded applications show the old adage described in Hennessey and Patterson's Computer Architecture book to the extreme -- that 90% of the execution time is spent in 10% or less of the code. The loop counts here tend to be less than 5 and 20; respectively, out of an application total loop count range of 30 to 200 loops. Rarely is the 1% loop count greater than 20 and usually exceeds the 90% performance mark with a count of 10 or less. Therefore, by just concentrating on a small number of loops, one can capture a majority of the application execution time. If an accurate performance profile is done on the major loops identified by this quick estimate step, a more accurate count can be obtained.

4.3 Loop Summary Table

After the header, the loop summary table is introduced. As the table generally needs to be printed out in landscape mode due to its wide width, the column headings are transposed to rows here for explanation purposes. The table below shows the column heading in each row followed by its explanation.

Note that most numbers in this table are quick estimates used to guide decision making and heuristics for automating transformations. Accurate performance profiling should be performed to obtain numbers accurate to within a percentage point or so. Also, the numbers are obtained before most transformations. So exceptional exit frequency and causes are just guidelines which may be dramatically changed by different optimizations. Finally, the quick software estimation does not take into account any cycle time associated with C stdlib calls. Although this can be significant, it is usually not the case for embedded, integral-number based applications.

`name`	Loop name. An identifier for the loop based on the static procedure it is contained in and basic block number that starts the loop. A compiler directive can be given as a C comment to provide a more usable name if desired.
`line`	Line number in source code file that starts the loop (to further aid the identification of the loop).
`frequency: iter`	Total number of iterations through loop body. When divided by the entries, gives a rough number of iterations per entry and therefore initial key to the dwell time (you need to understand if this loop was re-entered before entering another loop to really understand the dwell time).
`frequency: entries`	Number of times the loop was entered (should be determinable from loop entry trace, for example)

frequency: hw-en	Based on the projected exceptional exits out of blocks in the control flow of the loop body, the predicted number of re-entries into the HW co-processor version (likewise, the predicted number of exceptional exits; assumes one re-enters the HW on the next iteration after an exit)
time-percent: total	The total % of the application time taken in this loop (all blocks, whether feasible or not).
time-percent: feas	The total % of application time (a smaller number than the loops total percentage of time in the application) that the loop was in a HW feasible block. Based on estimates of exceptional exits, gives a rough lower bound one how much of the total application time taken by this loop could be accelerated in hardware.
software-time: total	The total number of cycles estimated for the software implementation of the loop. Includes accounting for "normalized" cache activity.
software-time: feasible	The total number of cycles estimated for the feasible (basic blocks that can be implemented in HW) of the loop.
Software-time: time-1	The estimated average number of cycles for the software implementation of the loop.
#op	Loop operator counts: static number of operators in the body of the loop.
Op%	Percentage of static operator count out of the total static operator count for the entire application.
memory - ld	Number of static memory loads in the loop. This is estimated based on SUIF instructions, and does not include any memory loads caused by register spills.
memory - st	Number of static memory stores in the loop. Like memory load, this is estimated based on SUIF instructions, and does not include any memory stores caused by register spills
hardware: total	The total estimated number of cycles spent in a hardware implementation of the loop (all iterations, etc.)
hardware: Time-1	The estimated time for one iteration of the loop implemented in hardware. Includes accounting for pipelining (that is, is the initiation interval if pipelined).
hardware: size	The number of datapath operator rows taken by the configuration to implement the loop. The maximum number available is platform dependent. The ACE 2 can have a maximum of 50. The GARP is 32 by design (the simulation can be reset to any number up to 1024).

hardware: speedup	Based on the columns to the right, the total speedup estimated obtained by hardware over the software version; including all (re)configuration times of the hardware. Note, many times this is blank in the initial compile stages because a trivial exceptional exit condition is not optimized out yet and thus makes the loop infeasible. This gives a lower bound on an expected gain unless there is unusal reconfiguration activity (very low dwell time). This is actually calculated as the **(Hardware: Time / Software Time: Feasible)**
hardware: qld	Number of queue loads if queue is implemented.
hardware: qst	Number of queue stores if queue is implemented.

Note that all the hardware related entries are reported only when **–nosynth** is not specified.

Finally, if an exceptional exit exists, a guess as to the root cause of the exit is made by looking into the basic block that is determined infeasible. This is reported in a line below the loop row. An example is:

```
B:    22 ->    137, f: 8e+05 call
```

and interpreted as:

> an exceptional exit exists in going from basic block 22 to 137. The exception (this path from one basic block to the other) was taken 8e+05 (80,000) times. The reason for the exception appears to be a function call that was not inlined (cast and float are other common occurrences).

Note that more than one exceptional exit path may be listed. Only when the number of times that path was taken is high (or closely matches the **frequency: hw-en**) is this of a concern. The exceptional path may be listed as never having been taken. This is often the case for error condition checks that then contain **printf()** or similar system calls as a result of the exception. This also implies the data set supplied in profiling never tripped the error condition causing the exception to be taken.

5 METER: Target Platform Performance Measurement

ILD uses estimation-base performance profiling to estimation loop and application performance. Because it uses a simple SUIF instruction based approach, it does not take into account instruction and data cache effect, pipeline stalls and the impact of compiler optimizations. Therefore, it is a fast but crude profiling mechanism, which can be used to aid initial analysis of applications. In order to more accurately measure application and loop performance on actual target platform, we implemented a package name METER to automatically instrument application program with various clock probing points to measure program performance at run-time. The measurement can be done at the application level, or loop level, or in between start and end points identified by the user. Note that instrumentation points inevitably introduce overhead cycles. METER estimates the possible overheads and deducts these cycles from the profiling results.

5.1 METER Flags

`Meter` is a SUIF pass that instruments application programs with performance measurement points, The instrumented program can then run on the target platform to obtain performance results. Meter can be used as a standalone pass, or used as part of the **`ncc`** flow. The latter is recommended because running meter usually requires the **`meter`** pass, the actual run and possibly other passes. **`ncc`** has packaged these passes together correctly. The user can pass various flags to meter to select different running modes and what loops to instrument:

Mode selection flags: only one of the following mode flags can be specified. The default is **`–sw`** is no mode flag is given.

`-sw`	Instrument loops for SW only measurement. This is the default mode. For SW mode, **`meter`** pass should run on **`.slb`** files.
`-hw`	Instrument loops for mixed HW and SW measurement. For any loop selected to be measured, whether to measure its SW or HW performance depends on the loop's HW (which kernel) or SW selection. HW mode requires **`meter`** to run after kernel selection (KS).

Loop selection flags: only one of the following loop selection flags can be specified. The default is **`–all`**.

`-feasilbe`	Instrument loops feasible for HW implementation. These include all loops that do not contain any operations not implementable in the HW, such as floating point operations. The feasibility analysis here does not consider HW size constraint.
`-ild`	Instrument ILD loops (with estimated SW time over 1% of estimated total program time). In order to detect ILD loops, profiling needs to run before the meter pass. **`branch.path`** and **`.branch.info`** files are required.
`-feasibleild`	Instrument loops that are both feasible and ILD (estimated SW time over 1% of estimated total program time). In order to detect ILD

	loops, profiling needs to run before the meter pass. **`.branch.path`** and **`.branch.info`** files are required.
`-hwselected`	Instrument only loops selected to run in HW. This needs to be combined with **`–hw`** option.
`-all`	Instrument all loops.
`-none`	Do not instrument any loops. Meter still instruments the application start and end, or the START_PROFILING and END_PROFILING specified by the user (see below).
`-loopfile filename`	Instrument loops specified in a file. The loop file should contain the names of the loop to be measured but nothing else.

Other flags:

`-sys`	Exclude the cost of system calls from the meter report.
`-target <targetname>`	Define the target platform. The default is **acev.**

5.2 START_PROFILING() and END_PROFILING()

`METER` instruments the start and the end of the program execution to measure the program total time. It also provides the **`START_PROFILING()`** and **`END_PROFILING()`** macro so that the user can specify where are the start and end points for program level measurement. **`START_PROFILING()`** and **`END_PROFILING()`** can be inserted at any lexical location of the program. However, if the user inserts them such that there are multiple **`START_PROFILING()`** and **`END_PROFILING()`** encountered at run time, METER will report the time between the first **`START_PROFILING()`** and the last **`END_PROFILING().`**

5.3 METER Result Files

`METER` results can be viewed in two files: **`.meter.info`** and **`.meter.out`**. METER pass itself outputs the **`.meter.info`** indicating what loops have been instrumented. After running the instrumented program on the target platform, user can look at the **`.meter.out`** file which contains the results of performance measurement at both the program level and the loop level.

`.meter.out` file starts with a header summarizing program level performance data. All timing data are in cycles.

`Application time w/o overhead` is the measured time for the entire application (including time spent in system calls), excluding estimated overhead spent in the measurement code. **`Application time w/o overhead & system time`** is the measured time for the entire application excluding time spent in system calls and excluding estimated measurement overhead. The percentage of time spent in system calls (**`system time`**) is also given at this line. **`Application time w/o overhead & system time`** is only available for ACE4k and ACEV platforms running RTEMS. If the user has specified **`START_PROFILING()`** and **`END_PROFILING()`** in the program, the measured time between the first **`START_PROFILING()`** and the last **`END_PROFILING()`** is listed in the next line.

The loop level performance results are summarized in a table. For each loop, the result contains the following fields:

`Name`	Loop name. Same as the name used in ILD report
`Total-%`	Percentage of time spent in this loop out of the total application time w/o measurement overhead.
`Total-time`	Total cycles spent in this loop, including SW time, HW time, as well as configuration time. This is printed for **`–hw`** mode only, as the SW time in **`–sw`** mode is the same as Total-time.
`SW-time`	Total cycles spent in the SW portion of the loop.
`HW-time`	Total cycles spent in the HW datapath portion of the loop. For **`–hw`** mode only.
`Config-time`	Total cycles spent in HW configurations of the loop, including both the initial configuration and any reconfigurations needed. For **`–hw`** mode only.

For **`–hw`** mode, the total percentage of time spent in HW and re/configuration are summed up for all loops and printed at the bottom of the **`HW-time`** and **`Config-time`** columns. The last line of the **`.meter.out`** file in **`–hw`** mode also provides the total number of configuration changes throughout the program execution.

6 SUIF to VCG (s2vcg)

`s2vcg` can be used along with **`xvcg`** to view the control-flow graphs of a user program at different stages of the compilation process as well as to see detailed information about the specific basic blocks such as feasibility, execution frequencies, basic block numbers, loop structure and others. **`s2vcg`** takes a SUIF file and generates a separate **`.vcg`** file for each procedure in it representing the CFG and the basic block properties. The **`.vcg`** file can be viewed later with **xvcg**. Here is a brief description of the command-line options that control the behavior of **s2vcg**:

- **-target** specifies the hardware target that **s2vcg** should use to determine the basic block feasibility.
- **-prof** lets **s2vcg** extract the profiling information from the **.branch.path** and **.branch.info** reports.
- **-instr**, **-event** and **-other** options can be used to have **s2vcg** include the instruction, the profiling and/or other basic block information (such as feasibility and basic block types) respectively in the main CFG. By default the basic blocks in the CFG contain only their basic block number, and all other detailed information can be accessed using **xvcg**'s *Node Information* menu choice.
- **-func** makes **s2vcg** output the CFG of a specified procedure. By default the CFG's of all procedures are generated.
- **-sfx** can be used to replace the default **.vcg** file extension.

To ease the readability of the CFG, **s2vcg** uses a color-coding scheme to distinguish between different kinds of blocks and graph edges. All forward edges use the default color (black), while the backedges are shown in magenta. The border color of the basic blocks depends on the basic block status and feasibility – non-kernel blocks are black, loop prologues and epilogues are purple, hardware blocks are green, software blocks are blue, version switches are cyan, and infeasible blocks can be easily recognized by their red border. In addition to that, loop entry blocks have light-gray background, and CFG start and end blocks have orange background. By exploring the differently colored regions of the graph, a user can easily identify any interesting or problematic structures in the IR.

Appendix E. Efficient Pipelining of Nested Loops: Unroll-and-Squash, an Innovative compiler Optimization Technique used in the Nimble Compiler

Note: Paper submitted for review.

Abstract

The size and complexity of current custom VLSI have forced the use of high-level programming languages to describe hardware, and compiler and synthesis technology to map abstract designs into silicon. Many applications operating on large streaming data usually require a custom VLSI because of high performance or low power restrictions. Since the data processing is typically described by loop constructs in a high-level language, loops are the most critical portions of the hardware description and special techniques are developed to optimally synthesize them. In this paper, we introduce a new method for mapping nested loops into hardware and pipelining them efficiently. The technique achieves fine-grain parallelism even on strong intra- and inter-iteration data-dependent inner loops and, by economically sharing resources, improves performance at the expense of a small amount of additional area. We implemented the transformation within the Nimble Compiler environment and evaluated its performance on several signal-processing benchmarks. The method achieves up to 2X increase in the area efficiency compared to the best known optimization techniques.

1 Introduction

Growing consumer market needs that require processing of large amount of data with a limited power or dollar budget have led to the development of increasingly complex embedded systems and application-specific IC’s. As a result, high-level compilation and sophisticated CAD tools are used to automate and accelerate the intricate design process. These techniques raise the level of abstraction and bring the hardware design closer and closer to the system engineer.

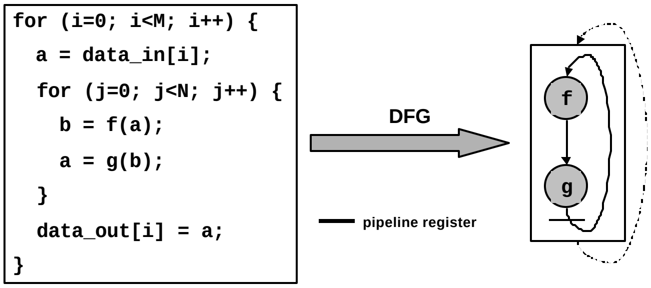
Since loops are the most critical parts of many applications (and, specifically, signal-processing algorithms 1), the new generation of CAD tools needs to borrow many transformation and optimization methods from traditional compilers in order to efficiently synthesize hardware from high-level languages. A large body of work exists on translating software applications for parallel execution on microprocessors. These techniques include software pipelining 1619 for exploiting parallelism within single processors and loop parallelization for multi-processors 14.

However, direct application of these techniques does not produce efficient hardware since the design tradeoffs in software compilation to a microprocessor are quite different from circuit synthesis from a program. For instance, while the number of extra operators (instructions) resulting from a particular software compiler transformation may not be critical as long as it increases the overall pararallelism in a microprocessor, the amount of additional area that the hardware synthesis produces may have much bigger impact on the performance and cost of a custom VLSI design. On the other hand, in contrast to traditional compilers for microprocessors, which are restrained by the paucity of registers in general-purpose processors and their limited capacity for data transfer between registers and memory, hardware synthesis algorithms usually have much more freedom in allocating registers and connecting them to memory.

When an inner loop has no loop-carried dependencies across iterations, many techniques such as pipelining will provide efficient and effective parallel performance for both microprocessors and custom VLSI. Unfortunately, a large number of loops in practical signal-processing applications have strong loop-carried dependencies. Many cryptographic algorithms, such as unchained Skipjack and DES for example, have a nested loop structure where an outer loop traverses the data stream while the inner loop transforms each data block. Furthermore, the outer loop has no strong inter-iteration data-dependencies while the inner loop has both inter- and intra-iteration dependencies that prevent synthesis tools employing traditional compilation techniques from mapping and pipelining them efficiently.

This paper introduces a new loop transformation that efficiently maps nested loops following this pattern into hardware. The technique, which we call *unroll-and-squash*, exploits the outer loop parallelism, concentrates more computation in the inner loop and improves the performance with little area increase by allocating the hardware resources without expensive multiplexing and complex routing. The technique was prototyped using the Nimble Compiler environment 1, and its performance was evaluated on several signal-processing benchmarks. Unroll-and-squash reaches comparable performance to traditional loop transformations with 2 to 10 times less area.

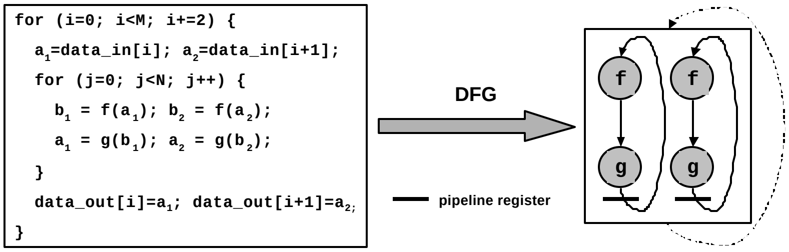
2 Motivation



```
for (i=0; i<M; i++) {
  a = data_in[i];
  for (j=0; j<N; j++) {
    b = f(a);
    a = g(b);
  }
  data_out[i] = a;
}
```

Figure 10: A simple example of a nested loop.

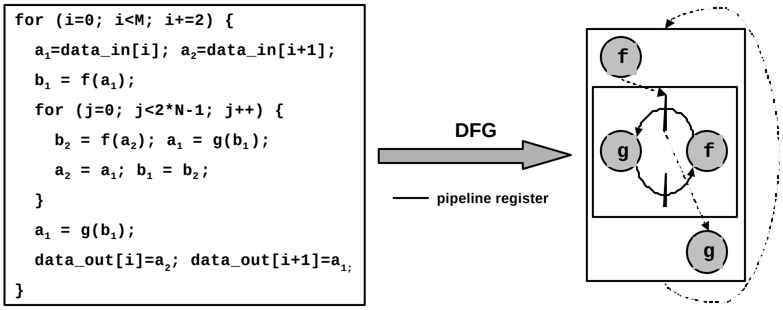
The importance of the new technique can be demonstrated using the simple set of loops shown in Figure 10. The outer loop walks through the input data and writes out the result, while the inner loop runs the data through several rounds of 2 operators – `f` and `g`, each completing in 1 clock cycle. Little can be done to optimize this program considering only the inner loop. Because of the cycle in the inner loop, it cannot be pipelined, i. e., it is not possible to execute several inner loop iterations in parallel. The interval at which consecutive iterations are started is called the *initiation interval (II).* As depicted in the DFG, the minimum II of the inner loop is 2 cycles, and the total time for the loop nest is *2´M´N*.



```
for (i=0; i<M; i+=2) {
  a1=data_in[i]; a2=data_in[i+1];
  for (j=0; j<N; j++) {
    b1 = f(a1); b2 = f(a2);
    a1 = g(b1); a2 = g(b2);
  }
  data_out[i]=a1; data_out[i+1]=a2;
}
```

Figure 11: A simple example: unroll-and-jam by 2.

Traditional loop optimizations such loop unrolling, flattening and permutation **Error! Reference source not found.** fail to exploit the parallelism and improve the performance for this set of loops. One successful approach in this case is the application of *unroll-and-jam* (Figure 11), which unrolls the outer loop but fuses the resulting sequential inner loops to maintain a single inner loop 28. After applying unroll-and-jam with a factor of 2 (assuming that *M* is even), the resulting inner loop has 4 operators (twice the original number). Although this transformation does not decrease the minimum II of the inner loop because the data-dependency cycle still exists, the ability to execute several operators in parallel has the potential to speed up the program. The II is 2 but the total time is half the original because the outer loop iteration count is halved – *2´(M/2)´N=M´N*. Thus, unroll-and-jam doubles the performance of the application at the expense of a doubled operator count.



```
for (i=0; i<M; i+=2) {
  a1=data_in[i]; a2=data_in[i+1];
  b1 = f(a1);
  for (j=0; j<2*N-1; j++) {
    b2 = f(a2); a1 = g(b1);
    a2 = a1; b1 = b2;
  }
  a1 = g(b1);
  data_out[i]=a2; data_out[i+1]=a1;
}
```

Figure 12: A simple example: unroll-and-squash by 2.

A more efficient way to improve the performance in this example is by applying the unroll-and-squash technique introduced in this paper (Figure 12). This transformation, similarly to unroll-and-jam, unrolls the outer loop but maintains a single inner loop. However, the data sets of the different outer loop iterations run through the inner loop operators in a round-robin manner, which allows the parallel execution of the operators. Moreover, the original operator count remains unchanged. Application of unroll-and-squash to the sample loop nest by a factor of 2 is similar to unroll-and-jam with respect to the transformation of the outer loop – the iteration count is halved, and 2 outer loop iterations are processed in parallel. However, the operator count in the inner loop remains the same as in the original program (2). By adding variable shifting/rotating statements and pulling appropriate prolog and epilog out of the inner loop, the transformation can be correctly expressed in software, although this may not be necessary if a pure hardware implementation is pursued. Since the final II is 1, the total execution time of the loop nest is *1´(M/2)´(2´N)=M´N*. Thus, unroll-and-squash doubles the performance without paying the additional cost of extra operators.

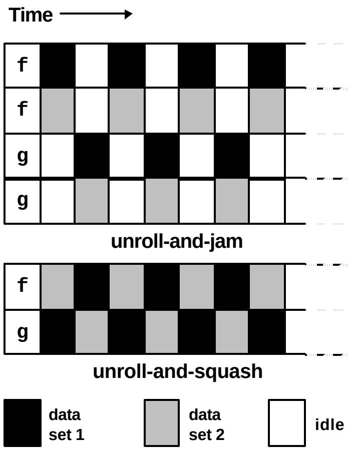


Figure 13: Operator usage.

Figure 13 shows the operator usage over time in the unroll-and-jammed and unroll-and-squashed versions of the program (it omits the prolog and the epilog necessary for unroll-and-squash). Besides the fact that unroll-and-squash makes better use of the existing operators than unroll-and-jam, another important observation is that it may be possible to combine both techniques simultaneously. Unroll-and-jam can be applied with an unroll factor that matches the desired or available amount of operators, and then unroll-and-squash can be used to further improve the performance and achieve better operator utilization.

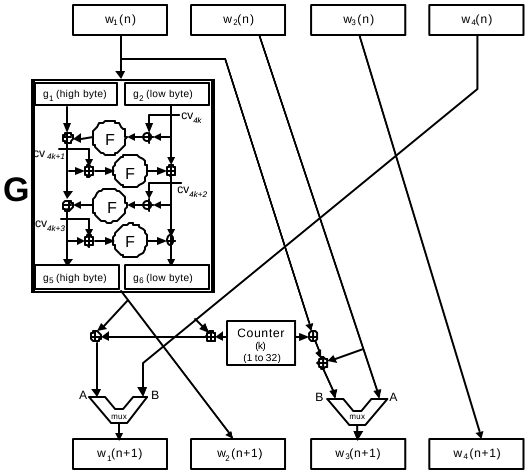


Figure 14: Skipjack cryptographic algorithm.

A good example of a real-world application of unroll-and-squash is the Skipjack cryptographic algorithm, declassified and released in 1998 (Figure 14). This crypto-algorithm encrypts 8-byte data blocks by running them through 32 rounds of 4 table-lookups (*F*) combined with key-lookups (*cv*), a number of logical operations and input selection. The *F*-lookups form a long cycle that prevents the encryption loop from being efficiently pipelined. Again, little can be done by optimizing the inner loop in isolation but, as with the simple example in the previous section, proper application of unroll-and-squash (separately or together with unroll-and-jam) on the outer, data-traversal loop can boost the performance significantly at a low extra area cost.

3 Method

The unroll-and-squash transformation optimizes the performance of 2-loop nests by executing multiple outer loop iterations in parallel. The inner loop operators cycle through the separate outer loop data sets, which allows them to work simultaneously. By doing efficient resource sharing, this technique reduces the total execution time without increasing the operator count. This section assumes that unroll-and-squash is applied to a nested loop pair where the outer loop iteration count is *M*, the inner loop iteration count is *N*, and the unroll factor is *DS* (*Data Sets*).

3.1 Requirements

This section outlines the general control-flow and data-dependency requirements that must hold for the proposed transformation to be applied to an inner-outer loop pair. In the next section, we show how some of these conditions can be relaxed by using various code analysis and transformation techniques such as induction variable identification, variable privatization, and others.

Unroll-and-squash can be applied to any set of 2 nested loops that can be successfully unroll-and-jammed 28. For a given unroll factor *DS*, it is necessary that the outer loop can be tiled in blocks of *DS* iterations, and that the iterations in each block be parallel. The inner loop should comprise a single basic block and have a constant iteration count across the different outer loop iterations. The latter condition also implies that the control-flow always passes through the inner loop.

3.2 Compiler Analysis and Optimization Techniques

A number of traditional compiler analysis, transformation and optimization techniques can be used to determine whether a particular loop nest follows the requirements, to convert the loop nest to one that conforms with them, or to increase the efficiency of unroll-and-squash. First of all, most standard compiler optimizations that speed up the code or eliminate unused portions of it can be applied before unroll-and-squash. These include constant propagation and folding, copy propagation, dead-code and unreachable-code elimination, algebraic simplification, strength-reduction to use smaller and faster operators in the inner loop, and loop invariant code motion. Scalarization may be used to reduce the number of memory references in the inner loop and replace them with register-to-register moves. Although very useful, these optimizations can rarely enlarge the set of loops that unroll-and-squash can be applied to.

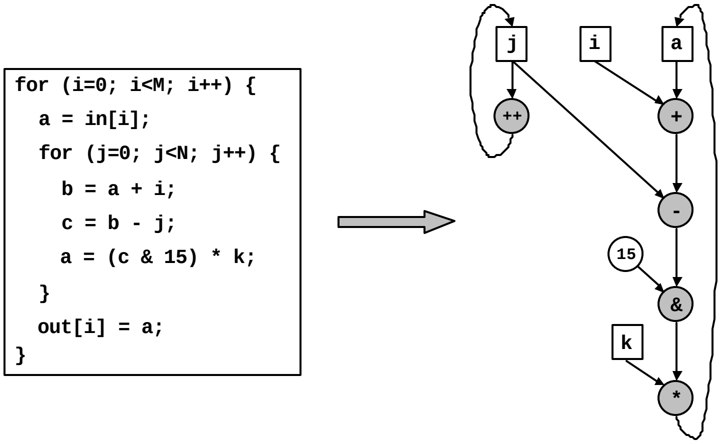
One way to eliminate conditional statements in the inner loop and make it a single basic block (one of the restrictions) is to transform them to equivalent logical and arithmetic expressions (if-conversion). Another alternative is to use code hoisting to move the conditional statements out of the inner-outer loop pair, if possible.

In order for the outer loop to be tiled in blocks of *DS* iterations, its iteration count *M* should be a multiple of *DS*. If this condition does not hold, loop peeling may be used, and *M mod DS* iterations of the outer loop may be executed independently from the remaining *M-(M mod DS)*.

The data-dependency requirement, i.e., the condition that the iterations of the outer loop should be parallel, is much more difficult to determine or overcome. Moreover, if the outer loop data dependency is an innate part of the algorithm that the loop nest implements, it is usually impossible to apply unroll-and-squash. One approach to eliminate some of the scalar variable data dependencies in the outer loops is by induction variable identification – it can be used to convert all induction variable definitions in the outer loop to expressions of a single index variable. Another method is modulo variable expansion, which replaces a variable with several separate variables corresponding to different iterations and combines them at the end. If the loops contain array references, dependence analysis 27 may be employed to determine the applicability of the technique and array privatization might be used to better exploit the parallelism. Finally, pointer analysis and other relevant techniques (such as converting pointer to array accesses) may be employed to determine whether code with pointer-based memory accesses can be parallelized.

3.3 Transformation

Once it is determined that a particular loop pair can be unroll-and-squashed by an unroll factor *DS*, it is necessary to efficiently assign the functional elements in the inner loop to separate pipeline stages, and apply the corresponding transformation to the software representation of the loop. Although it is possible to have a purely hardware implementation of the inner loop (without prolog and epilog in software), the outer loop still needs to be unrolled and have a proper variable assignment. The sequence of basic steps that are used to apply unroll-and-squash to a loop nest are presented below:



```
for (i=0; i<M; i++) {
  a = in[i];
  for (j=0; j<N; j++) {
    b = a + i;
    c = b - j;
    a = (c & 15) * k;
  }
  out[i] = a;
}
```

Figure 15: Unroll-and-squash – building the DFG.

- Build the DFG of the inner loop (Figure 15). Live variables are stored in registers at the top of the graph.
- Transform live variables that are used in the inner loop but defined in the outer loop (i.e., registers that have no incoming edges) into cycles, i.e., output edges from the register back to itself.
- “Stretch” the cycles in the graph so that the backedges start from the bottom and go all the way to the registers at the top.

- Pipeline the resulting DFG ignoring the backedges (Figure 16) producing exactly *DS* pipeline stages. Empty stages may be added or pipeline registers may be removed to adjust the stage count to *DS*.

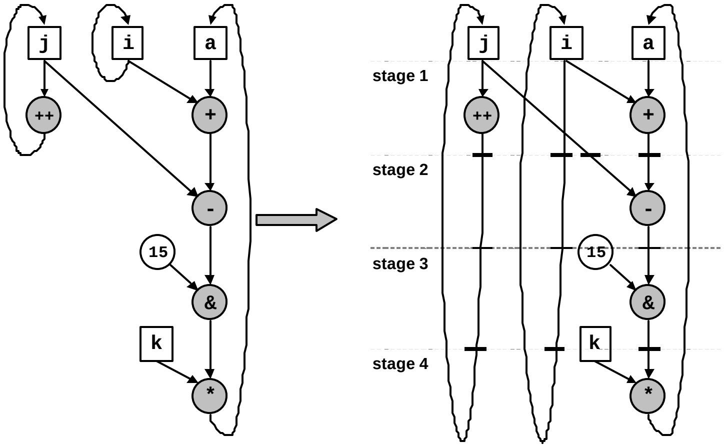


Figure 16: Stretching cycles and pipelining.

- Perform variable expansion – expand each variable in the inner/outer loop nest to *DS* versions. Some of the resulting variables may not actually be used later.
- Unroll the outer loop basic blocks (this includes the basic blocks that dominate and post-dominate the inner loop).
- Generate prolog and epilog code to fill and flush the pipeling (unless the inner loop is implemented purely in hardware).
- Assign proper variable versions in the inner loop. Note that some new (delay) variables may be needed to handle expressions split across pipeline registers.
- Add variable shifting/rotation to the inner loop. Note that reverse shifting/rotation may be required in the epilog or, alternatively, a proper assignment of variable versions.

The outer loop data sets pass through the pipeline stages in a round-robin manner. All live variables should be saved to and restored from the appropriate hardware registers before and after execution.

3.4 Algorithm Analysis

The described loop transformation decreases the number of outer loop iterations from *M* to *M/DS*. A software implementation will increase the inner loop iteration count from *N* to *DS ´N-(DS-1)* and execute some of the inner loop statements in the prolog and epilog in the outer loop. The total iteration count of the loop nest stays approximately the same as the original – *M ´N*.

There are several factors that need to be considered in order to determine the optimal unroll factor *DS*. One of the main barriers to performance increase is the maximum number of pipeline stages in the inner loop. In a software implementation of the technique, this number is limited by the operator count in the critical path in the DFG or may be smaller if different operator latencies are taken into account. A pure hardware implementation bounds the stage count to the delay of the critical path divided by the clock period. The pipeline stage count determines the number of outer loop iterations that can be executed in parallel and, in general, the more data sets that are processed in parallel the better the performance. Certainly, the calculation of the unroll factor *DS* should be made in accordance to the outer loop iteration count (loop peeling may be required) and the data dependency analysis discussed in the previous section (larger *DS* may eliminate the parallelism).

Another important factor for determining the unroll factor *DS* is the extra area and, consequently, extra power that comes with large values of *DS*. Unroll-and-squash adds only pipeline registers to the existing operators and data feeds between them and, because of the cycle stretching, most of them can be efficiently packed in groups to form a single shift register. This optimization may decrease the impact of the transformation on the area and the power of the design, as well as make routing easier – no multiplexors are added, in contrast to traditional hardware synthesis techniques. In comparison with unroll-and-jam by the same unroll factor, unroll-and-squash results in less area since the operators are not duplicated. The trade-off between speed, area and power is further illustrated in the benchmark report (Section 5).

4 Implementation

Recently, there has been an increased interest in hardware/software co-design and co-synthesis both in the academia and in the industry. Most hardware/software compilation systems focus on the functional partitioning of designs amongst ASIC (hardware) and CPU (software) components 567. In addition to using traditional behavioral synthesis languages such as Verilog and VHDL, synthesis from software application languages such as C/C++ or Java is also gaining popularity. Some of the systems that synthesize subsets of C/C++ or C-based languages include HardwareC 21, SystemC 22, and Esterel C 23. DeepC, a compiler for a variation of the RAW parallel architecture presented in 2, allows sequential C or Fortran programs to be compiled directly into custom silicon or reconfigurable architectures. Some other novel hardware synthesis systems compile Java 24, Matlab 26 and term-rewriting systems 25. In summary, the work in this field clearly suggests that future CAD tools will synthesize hardware designs from higher levels of abstraction. Some efforts in the last few years have been concentrated on automatic compilation and partitioning to reconfigurable architectures 8910. Callahan and Wawrzynek 3 developed a compiler for the Berkeley GARP architecture 4 which takes C programs and compiles them to a CPU and FPGA. The Nimble Compiler environment 1 extracts hardware kernels (inner loops that take most of the execution time) from C applications to accelerate on a reconfigurable co-processor. This system was used to develop and evaluate the loop optimization technique presented in this paper.

4.1 Target Architecture

Figure 17 demonstrates an abstract model of the new class of architectures that the Nimble Compiler targets. The Agile hardware architecture couples a general purpose CPU with a dynamically reconfigurable coprocessor. Communication channels connect the CPU, the datapath, and the memory hierarchy. The CPU can be used to implement and execute control-intensive routines and system I/O, while the datapath provides a large set of configurable operators, registers and interconnects, allowing acceleration of computation-intensive parts of an application by flexible exploitation of ILP.

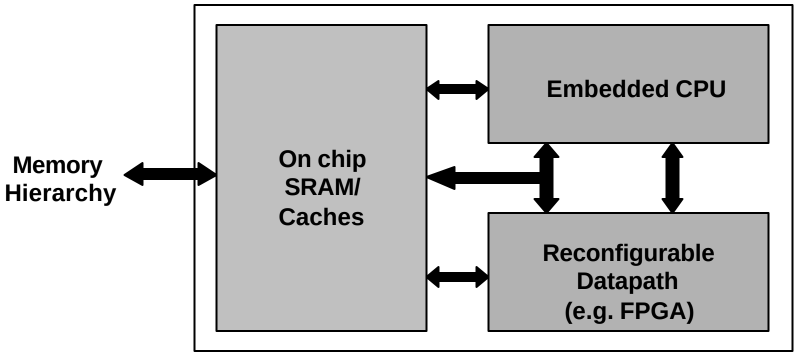


Figure 17: The target architecture – Agile hardware.

This abstract model describes a broad range of possible architectural implementations. The Nimble Compiler is retargettable, and can be parameterized to target a specific platform described by an Architecture Description Language. The target platforms that the Nimble Compiler currently supports include GARP, ACE2 card and ACEV. Berkeley's GARP is a single-chip architecture with a MIPS 4000 CPU, a reconfigurable array of 24 by 32 CLBs, on-chip data and instruction caches, and a 4-level configuration cache 4. The TSI Telsys ACE2 is a board-level platform and consists of a microSparc CPU and Xilinx 4085 FPGAs 13. The ACEV hardware prototype combines a TSI Telsys ACE card 12 with a microSparc CPU, and a PCI Mezzanine card 11, containing a Xilinx Virtex XCV 1000 FPGA. In the ACE card configurations, a fixed wrapper is defined in the FPGA to provide support resources to turn the FPGA into a configurable datapath coprocessor. The wrapper includes the CPU interface, memory interface, local memory optimization structures, and a controller.

4.2 The Nimble Compiler

The Nimble Compiler (Figure 18) extracts the compute-intensive inner loops (kernels) from C applications, and synthesizes them into hardware. The front-end, built using the SUIF compiler framework 1, profiles the program to obtain a full basic block execution trace along with the loops that take most of the execution time. It also applies various hardware-oriented loop transformations to concentrate as much of the execution time in as few kernels as possible, and generate multiple different versions of the same loop. Some relevant transformations include loop unrolling, fusion and packing, distribution, flattening, pipelining, function inlining, branch trimming, and others. A kernel selection pass chooses which kernel versions to implement in hardware based on the results from the profiling, feasibility, and a quick synthesis step. The back-end datapath synthesis tool takes the kernels (described as DFG's) and generates the corresponding FPGA bit streams that are subsequently combined with the rest of the C source code by an embedded compiler to produce the final executable binary.

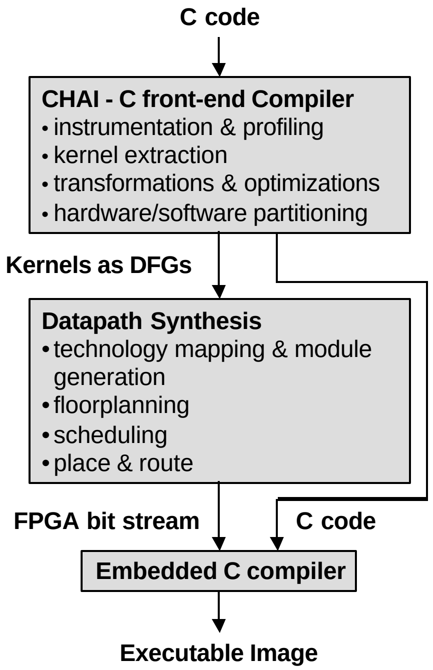

C code
CHAI - C front-end Compiler
• instrumentation & profiling
• kernel extraction
• transformations & optimizations
• hardware/software partitioning
Kernels as DFGs
Datapath Synthesis
• technology mapping & module generation
• floorplanning
• scheduling
• place & route
FPGA bit stream
C code
Embedded C compiler
Executable Image

Figure 18: Nimble Compiler flow.

Unroll-and-squash is one of the loop transformations that the Nimble Compiler considers before kernel selection is performed. This newly discovered optimization benefits the Nimble environment in a variety of ways. First of all, outer loop unrolling concentrates more of the execution time in the inner loop and decreases the amount of transitions between the CPU and the reconfigurable datapath. In addition, this transformation does not increase the operator count and, assuming efficient implementation of the register shifts and rotation, the FPGA area is used optimally. Finally, unroll-and-squash pipelines loops with strong intra- and inter-iteration data dependencies and can be easily combined with other compiler transformations and synthesis optimizations.

5 Experimental Results

We compared the performance of unroll-and-squash on the main computational kernels of several signal-processing benchmarks to the original loops, pipelined original loops, and pipelined unroll-and-jammed loops. The collected data shows that unroll-and-squash is an effective way to speed up such applications at a relatively low area cost and suggests that this is a valuable compiler and behavioral synthesis technique in general.

5.1 Target Architecture Assumptions

The benchmarks were compiled using the Nimble Compiler with the ACEV target platform. Two memory references per clock cycle were allowed, and no cache misses were assumed. The latter assumption is not too restrictive for comparison purposes because the different transformed versions have similar memory access patterns. Furthermore, a couple of the benchmarks have been specially optimized for hardware and have no memory references at all.

5.2 Benchmarks

Benchmark	Description
Skipjack-mem	Skipjack cryptographic algorithm: encryption, software implementation with memory references
Skipjack-hw	Skipjack cryptographic algorithm: encryption, software implementation optimized for hardware without memory references
DES-mem	DES cryptographic algorithm: encryption, SBOX implemented in software with memory references
DES-hw	DES cryptographic algorithm: encryption, SBOX implemented in hardware without memory references
IIR	4-cascaded IIR biquad filter processing 64 points

Table 5: Benchmark description.

Our benchmark suit consists of two cryptographic algorithms (unchained Skipjack and DES) and a filter (IIR) described in Table 5. Two different versions of Skipjack and DES are used. Skipjack-mem and DES-mem are regular software implementations of the corresponding crypto-algorithms with memory references. Skipjack-hw and DES-hw are versions specifically optimized for hardware implementation – they use local ROM for memory lookups and domain generators for particular bit-level operations. Finally, IIR is a floating-point filter implemented on the target platform by modeling pipelinable floating-point arithmetic operations.

5.3 Results and Analysis

Table 6 presents the raw data collected through our experiments. It compares ten different versions of each benchmark – an original, non-pipelined version, a pipelined version, unroll-and-squashed versions by factors of 2, 4, 8 and 16, and, finally, pipelined unroll-and-jammed versions by factors of 2, 4, 8 and 16. The table shows the initiation interval in clock cycles, the area of the designs in rows and the register count. One should note that if the initial loop pair iteration count is *M ´N*, after unroll-and-jam by a factor *DS* it becomes *M ´N/DS*.

Benchmark		original	pipelined	squash(2)	squash(4)	squash(8)	squash(16)	jam(2)	jam(4)	jam(8)	jam(16)
Skipjack-mem	II (cycles)	22	21	12	9	8	7	23	28	38	70
	Area (rows)	49	57	62	91	143	256	111	219	435	867
	Registers (count)	6	13	18	44	92	197	25	49	97	193
Skipjack-hw	II (cycles)	19	19	11	7	4	3	19	19	19	19
	Area (rows)	41	41	56	86	143	262	80	158	314	626
	Registers (count)	8	8	21	50	105	218	16	32	64	128
DES-mem	II (cycles)	16	13	9	7	5	5	17	25	41	73
	Area (rows)	69	72	84	143	174	263	141	279	555	1107
	Registers (count)	5	8	19	60	99	174	15	29	57	113
DES-hw	II (cycles)	8	5	5	3	3	2	5	5	5	5
	Area (rows)	27	30	36	56	99	141	57	111	219	435
	Registers (count)	5	8	13	33	73	115	15	29	57	113
IIR	II (cycles)	56	13	29	15	9	5	13	18	33	65
	Area (rows)	106	131	118	138	177	258	253	497	985	1961
	Registers (count)	2	26	14	34	73	154	48	92	180	356

Table 6: Raw data – initiation interval (II), area and register count.

The normalized data corresponding to the figures in Table 6 is presented in Table 7. The base case is the original, non-pipelined version of the benchmarks. A detailed analysis of these values follows.

Benchmark		original	pipelined	squash(2)	squash(4)	squash(8)	squash(16)	jam(2)	jam(4)	jam(8)	jam(16)
Skipjack-mem	Speedup	1.00	1.05	1.83	2.44	2.75	3.14	1.91	3.14	4.63	5.03
	Area	1.00	1.16	1.27	1.86	2.92	5.22	2.27	4.47	8.88	17.69
	Registers	1.00	2.17	3.00	7.33	15.33	32.83	4.17	8.17	16.17	32.17
	Speedup / Area	1.00	0.90	1.45	1.32	0.94	0.60	0.84	0.70	0.52	0.28
Skipjack-hw	Speedup	1.00	1.00	1.73	2.71	4.75	6.33	2.00	4.00	8.00	16.00
	Area	1.00	1.00	1.37	2.10	3.49	6.39	1.95	3.85	7.66	15.27
	Registers	1.00	1.00	2.63	6.25	13.13	27.25	2.00	4.00	8.00	16.00
	Speedup / Area	1.00	1.00	1.26	1.29	1.36	0.99	1.03	1.04	1.04	1.05
DES-mem	Speedup	1.00	1.23	1.78	2.29	3.20	3.20	1.88	2.56	3.12	3.51
	Area	1.00	1.04	1.22	2.07	2.52	3.81	2.04	4.04	8.04	16.04
	Registers	1.00	1.60	3.80	12.00	19.80	34.80	3.00	5.80	11.40	22.60
	Speedup / Area	1.00	1.18	1.46	1.10	1.27	0.84	0.92	0.63	0.39	0.22
DES-hw	Speedup	1.00	1.60	1.60	2.67	2.67	4.00	3.20	6.40	12.80	25.60
	Area	1.00	1.11	1.33	2.07	3.67	5.22	2.11	4.11	8.11	16.11
	Registers	1.00	1.60	2.60	6.60	14.60	23.00	3.00	5.80	11.40	22.60
	Speedup / Area	1.00	1.44	1.20	1.29	0.73	0.77	1.52	1.56	1.58	1.59
IIR	Speedup	1.00	4.31	1.93	3.73	6.22	11.20	8.62	12.44	13.58	13.78
	Area	1.00	1.24	1.11	1.30	1.67	2.43	2.39	4.69	9.29	18.50
	Registers	1.00	13.00	7.00	17.00	36.50	77.00	24.00	46.00	90.00	178.00
	Speedup / Area	1.00	3.49	1.73	2.87	3.73	4.60	3.61	2.65	1.46	0.75

Table 7: Normalized data – estimated speedup, area, registers and cost (speedup/area).

Unroll-and-squash achieves better speedup than regular pipelining, and usually wins over the worse case unroll-and-jam (Figure 19). However, with large unroll factor unroll-and-jam outperforms unroll-and-squash by a big margin in most cases. Still, an interesting observation to make is the fact that, for several benchmarks, unroll-and-jam fails to obtain a speedup proportional to the unroll factor for larger factors (Skipjack-mem, DES-mem and IIR). The reason for this is that the increase of the unroll factor proportionally increases the operator count and, subsequently, the number of memory references. Since the amount of memory references is limited to two per clock cycle, more memory references increase the II and decrease the relative speedup. Unlike unroll-and-jam, unroll-and-squash does not change the number of memory references – the initial amount of memory references form the lower bound for the minimum II. Therefore, designs with many memory references may benefit from unroll-and-squash more than unroll-and-jam at greater unroll factors. Additionally, unroll-and-squash, in general, performs worse on designs with small original II (Skipjack-hw and DES-hw) because there is not much room for improvement.

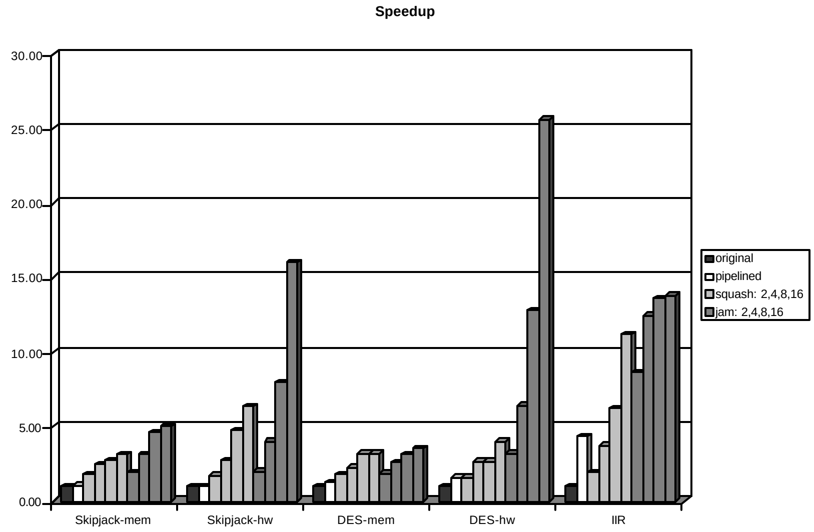


Figure 19: Speedup.

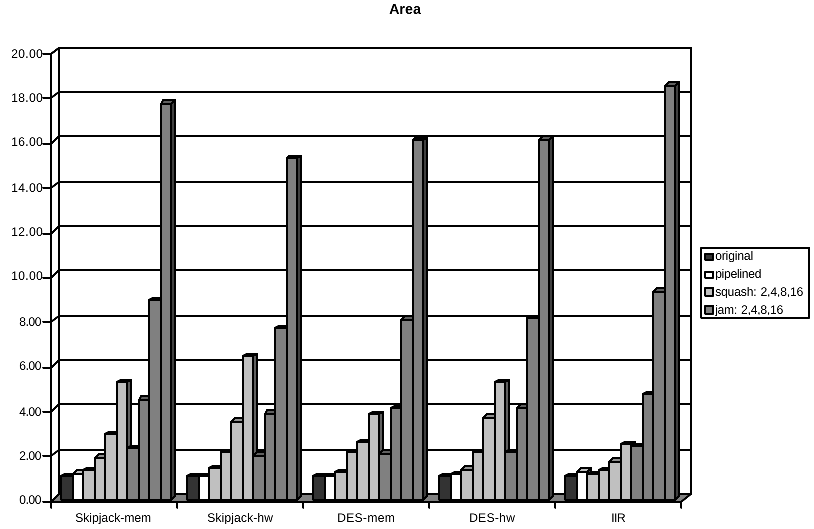


Figure 20: Area.

The speedup from the different transformations comes at the expense of additional area (Figure 20). Undoubtedly, since unroll-and-squash adds only registers while unroll-and-jam also increases the number of operators in proportion to the unroll factor, unroll-and-squash results in much less extra area. This can be very clearly seen from the results of the floating point benchmark (IIR) depicted in Figure 20.

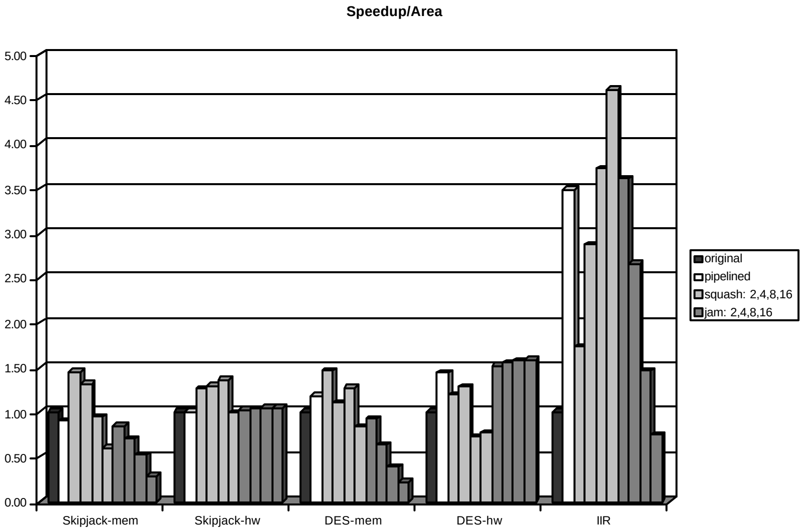


Figure 21: Execution efficiency (speedup/area) – higher is better.

In order to evaluate which technique is better, we can look at the speedup to area ratio (Figure 21). This value captures the performance of the design per unit area – higher speed and smaller design leads to larger ratio, while lower speed and larger area results in smaller ratio. By this measure, unroll-and-squash wins over unroll-and-jam in most cases, although some interesting trends can be noted in this regard. The ratio decreases with increasing unroll factor when unroll-and-jam is applied to benchmarks with memory references – this is caused by the higher II due to a congested memory bus. However, for designs without memory references unroll-and-jam increases the operator count with the unroll factor and does not change the II, so the ratio stays about constant. The ratio for unroll-and-squash stays about the same or decreases slightly with higher unroll factors in most cases. An obvious exception is the floating point benchmark where higher unroll factors lead to larger ratios. This can be attributed to the large original II and small minimum II that unroll-and-squash can achieve – a much higher unroll factor is necessary to reach to the point where the memory references limit the II.

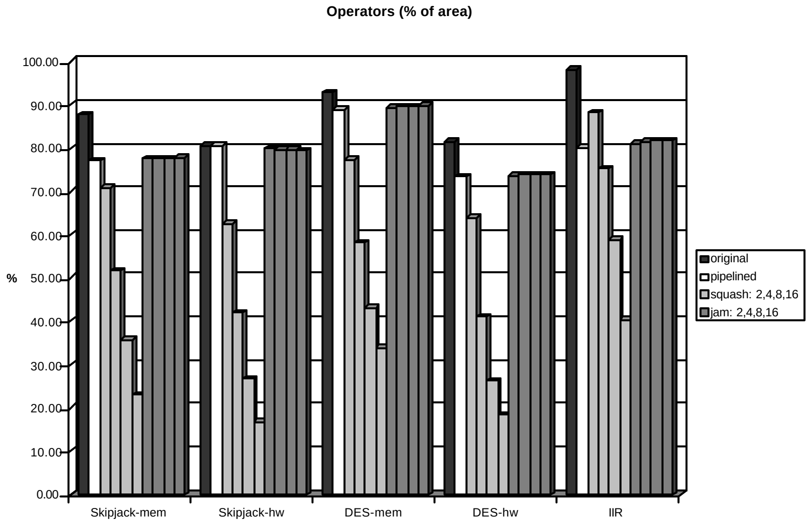


Figure 22: Operators as percent of the area.

Finally, it is interesting to observe how the operator count as a proportion of the whole area varies across the different transformations (Figure 22). While this value remains about the same for unroll-and-jam applied with different unroll factors, it sharply decreases for unroll-and-squash with higher unroll factors. This is important to note because our prototype implements the registers as regular operators, i.e., each taking a whole row. Considering the fact that they can be much smaller, the presented values for area are fairly conservative and the actual speedup per area ratio will increase significantly for unroll-and-squash in the final hardware implementation. Furthermore, many of the registers in the unroll-and-squashed designs are shift/rotate registers that can be implemented even more efficiently with minimal interconnect.

6 Related Work

An extensive survey of the available software pipelining techniques such as modulo scheduling algorithms, perfect pipelining, Petri net model and Vegdahl's technique, and a comparison between the different methods is given in 17. Since basic-block scheduling is an NP-hard problem 18, most effort on the topic has been concentrated on a variety of heuristics to reach near-optimal schedules. The main disadvantage of all these methods when applied to loop nests is that they consider and transform only inner-most loops resulting in poor exploitation of parallelism as well as lower efficiency due to setup costs. Lam's hierarchical reduction scheme aims to overlap execution of the prolog and the epilog of the transformed loop with operations outside the loop 19. The original Nimble Compiler approach to hardware/software partitioning of loops may pipeline outer loops but considers inner loop entries as exceptional exits from hardware 1. In general, all techniques that perform scheduling across basic block boundaries do not handle nested loop structures efficiently 1520.

7 Conclusions

In this paper we showed that high-level language hardware synthesis needs to employ traditional compilation techniques but most of the standard loop optimizations cannot be directly used. We presented an efficient loop pipelining technique that targets nested loop pairs with iteration-parallel outer loop and strong inter- and intra-iteration data-dependent inner loop. The technique was evaluated using the Nimble compiler framework on several signal-processing benchmarks. Unroll-and-squash improves the performance at a low additional area cost through efficient resource sharing and proved to be an effective way to exploit parallelism in nested loops mapped to hardware.

References

1. Y. Li, T. Callahan, E. Darnell, R. Harr, U. Kurkure, and J. Stockwood. Hardware-software co-design of embedded reconfigurable architectures, *Proc. 37th Design Automation Conference*, pp. 507-512, Los Angeles, CA, 2000.
2. J. Babb, M. Rinard, A. Moritz, W. Lee, M. Frank, R. Barua, and S. Amarasignhe. Parallelizing Applications into Silicon. *Proc. IEEE on FCCM*, Napa Valley, April 1999.
3. T. Callahan, and J. Wawrzynek. Instruction level parallelism for reconfigurable computing, *Proc. 8th International Workshop on Field-Programmable Logic and Applications*, September 1998.
4. J. R. Hauser, and J. Wawrzynek, Garp: A MIPS processor with a reconfigurable coprocessor, *Proc. FCCM '97*, 1997.
5. W. Wolf. Hardware/software co-design of embedded systems, *Proc. IEEE*, July 1994.
6. B. Dave, G. Lakshminarayana, and N. Jha. COSYN: hardware-software co-synthesis of embedded systems, *Proc. 34th Design Automation Conference*, 1997.
7. S. Bakshi, and D. Gajski. Partitioning and pipelining for performance-constrained hardware/software systems, *IEEE Transactions on VLSI Systems*, 7(4), December 1999.
8. R. Dick, and N. Jha. Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems, *Proc. Intl. Conference on Computer-Aided Design*, 1998.
9. M. Kaul, *et al*. An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications, *Proc. 36th Design Automation Conference*, 1999.
10. M. Gokhale, and A. Marks. Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays, *Proc. FPL*, 1995.
11. Alpha Data Parallel Systems, *ADM-XRC PCI Mezzanine Card User Guide. Version 1.2*, 1999.

12. TSI Telsys, *ACE Card Manual*, 1998.

13. TSI Telsys, *ACE2 Card Manual*, 1998.

14. M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion and M. S. Lam. Maximizing Multiprocessor Performance with the SUIF Compiler, *IEEE Computer*, December 1996.

15. Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, CA, 1997.

16. B. R. Rau, and J. A. Fisher. Instruction-level parallel processing: history, overview, and perspective. *The Journal of Supercomputing*, 7, pp. 9-50, 1993.

17. Vicki H. Allan, Reese B. Jones, Randall M. Lee, and Stephen J. Allan. Software Pipelining. In *ACM Computing Surveys*, 27(3):367-432, September 1995.

18. Michael R. Garey, and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., San Francisco, CA, 1979.

19. Monica Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *Proceedings in SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pp. 318-328, 1988.

20. Andrew Appel, and Maia Ginsburg. *Modern Compiler Implementation in C.* Cambridge University Press, Cambridge, United Kingdom, 1998.

21. David Ku, and Giovanni De Micheli. *High Level Synthesis of ASICs under Timing and Synchronization Constraints*, Kluwer Academic Publishers, Boston, MA 1992.

22. SystemC, http://www.systemc.org.

23. Luciano Lavagno, Ellen Sentovich. ECL: A Specification Environment for System-Level Design, *Proc. DAC '99*, New Orleans, pp. 511-516, June 1999.

24. Xilinx, http://www.lavalogic.com.

25. Arvind and X. Shen. Using Term Rewriting Systems to Design and Verify Processors, *IEEE Micro Special Issue on Modeling and Validation of Microprocessors*, May/June 1999.

26. M. Haldar, A. Nayak, A. Kanhere, P. Joisha, N. Shenoy, A. Choudhary and P. Banerjee. A Library-Based Compiler to Execute MATLAB Programs on a Heterogeneous Platform, *ISCA 13th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS-2000)*, August 2000.

27. Dror E. Maydan, *Accurate Analysis of Array References*, Ph.D. thesis, Stanford University, Computer Systems Laboratory, September 1992.

28. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June 1990.

29. F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, Prentice-Hall, 1972

Appendix F. The Hardware-Software Partitioning Approach of the Nimble Compiler

Paper Published in Proceedings, 37th Design Automation Conference (DAC 2000), June, 2000. Copyright ACM/IEEE.

Hardware-Software Co-Design of Embedded Reconfigurable Architectures

Yanbing Li, Tim Callahan[*], Ervan Darnell[**], Randolph Harr, Uday Kurkure, Jon Stockwood

Synopsys Inc., 700 East Middlefield Rd. Mountain View, CA 94043
* Department of EECS, Univ. of California, Berkeley, CA 94720
** Silicon Spice, 415 East Middlefield Rd., Mountain View, CA 94043

Abstract

In this paper we describe a new hardware/software partitioning approach for embedded reconfigurable architectures consisting of a general-purpose processor (CPU), a dynamically reconfigurable datapath (e.g. an FPGA), and a memory hierarchy. We have developed a framework called Nimble that automatically compiles system-level applications specified in C to executables on the target platform. A key component of this framework is a hardware/software partitioning algorithm that performs fine-grained partitioning (at loop and basic-block levels) of an application to execute on the combined CPU and datapath. The partitioning algorithm optimizes the global application execution time, including the software and hardware execution times, communication time and datapath reconfiguration time. Experimental results on real applications show that our algorithm is effective in rapidly finding close to optimal solutions.

1. Introduction

Reconfigurable computing using FPGAs is emerging as an alternative to conventional ASICs and general-purpose processors[1]. Reconfigurable architectures can be post-fabrication customized for a wide class of applications, including multi-media, communications, networking, graphics and cryptography, to achieve significantly higher performance over general or even special-purpose processor alternatives (such as DSPs). For convenience, we will use the term FPGA to refer to any type of reconfigurable datapath, whether implemented using FPGAs or other forms of reconfigurable logic.

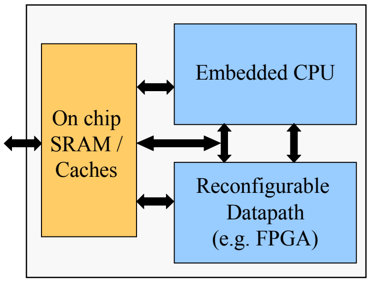


Figure 1. The target architecture.

Recent developments in reconfigurable architectures have demonstrated that a tightly coupled reconfigurable co-processor with a general purpose CPU can achieve significant speedup on a general class of applications[6]. An abstract model of this new class of architecture is shown in Figure 1. The architecture also contains memory hierarchy and communication channels that connect the CPU, datapath, and memory. The CPU can be used to implement control-intensive functions and system I/O, leaving the datapath to accelerate computation-intensive parts of an application. This class of architecture defines a common, reusable platform for a wide range of applications, and potentially provides a better transistor utilization than a single CPU or combined CPU and ASIC of comparable silicon area.

To exploit the potential performance gain provided by this class of architectures, we have developed a retargetable framework named **Nimble** that automatically compiles system-level applications specified in C to executables running on these platforms. At the core of the Nimble Compiler is a **hardware/software partitioning** algorithm that partitions applications onto the CPU and the datapath. As opposed to many co-synthesis algorithms that work at moderate to coarse granularities (such as task-level and function level) and extract task-level parallelism [2][7][10], our algorithm performs fine grain partitioning at the loop and basic block levels to exploit potential **instruction-level parallelism (ILP)** to significantly accelerate important loops in the FPGA.

There have been considerable research efforts in co-design of conventional embedded hardware/software architectures containing ASICs, which we will briefly review in Section 2. However, the partitioning problem for architectures containing reconfigurable FPGAs has a different requirement: it demands a two-dimensional partitioning strategy, in both **spatial** and **temporal** domains, while the conventional partitioning involves only spatial partitioning. Here, spatial partitioning refers to physical implementation of different functionality within different areas of the hardware resource. For dynamically reconfigurable architectures, besides spatial partitioning, the partitioning algorithm needs to perform temporal partitioning, meaning that the FPGA can be reconfigured at various phases of the program execution to implement different functionality.

In this paper, we focus on the temporal partitioning aspect. The input to the algorithm is a set of candidate loops for hardware, termed **kernels**, that have been extracted from the source application. Each loop has a software version and one or more hardware versions that represent different delay and area tradeoffs. The partitioning algorithm selects which loops to implement in the FPGA, and which hardware version of each loop should be used to achieve the highest application-level performance. Key issues with this approach are:

- The partitioning algorithm must effectively capture the dynamic reconfiguration costs. This is difficult as the number of reconfigurations for one kernel depends on which other kernels may go into the hardware.
- The algorithm must integrate compiler optimizations and hardware design space exploration into the hardware/software partitioning process.
- Partitioning must be guided by various forms of profiling information to accurately assess the tradeoffs between hardware and software implementations.

The rest of the paper is organized as follows. Section 2 reviews related work. In Section 3, we first present an overall picture of the Nimble Compiler framework, of which our partitioning algorithm is a component, and then describe the formulation of the hardware/software partitioning problem itself. Section 4 explains the details of the partitioning algorithm. Section 5 presents the experimental results on several benchmarks.

2. Previous Work

Related work includes studies from general areas of hardware/software co-design and reconfigurable computing.

Earlier work in hardware-software co-design mainly focused on hardware-software partitioning. Most of the partitioning algorithms model the system based on an architectural template of a CPU (software) and an ASIC (hardware)[4] [5][7][11]. Recent work in co-synthesis has used a more generalized model consisting of heterogeneous multiprocessors with various communication topologies [2][9][10]. Although some of the above techniques use highly abstract architecture models that might be retargetable to reconfigurable architectures, none of them can represent the special characteristics of the platform such as the reconfiguration overhead or possibility of both spatial and temporal partitioning.

Some recent efforts in reconfigurable computing address automatic compilation and partitioning to reconfigurable architectures. Callahan *et al.* [1] developed a compiler for the Berkeley GARP architecture[6] which compiles source applications in C to a CPU and FPGA. They use a feasibility-driven approach that does not take performance into account during the hardware/software partitioning process. Gokhale *et al.* worked on compiling C onto reconfigurable processors but did not address the hardware-software partitioning problem directly[12].

Dick and Jha proposed the CORDS algorithm to synthesize real-time tasks onto distributed systems containing dynamically reconfigurable FPGAs [3]. CORDS uses a coarse, task-level input represented by acyclic graphs and exploits task-level parallelism. Kaul *et al.* [8] recently proposed an ILP based algorithm for temporal partitioning of reconfigurable designs that also starts with acyclic task graph specifications. The algorithm finds optimal solutions but has a very high computation cost. Its acyclic task inputs only allow a single configuration of a task and therefore use a simple configuration cost model. Both works assume a single implementation of a task in the hardware and do not explore compiler optimizations and the hardware design space to evaluate tradeoffs between different implementations of the same task.

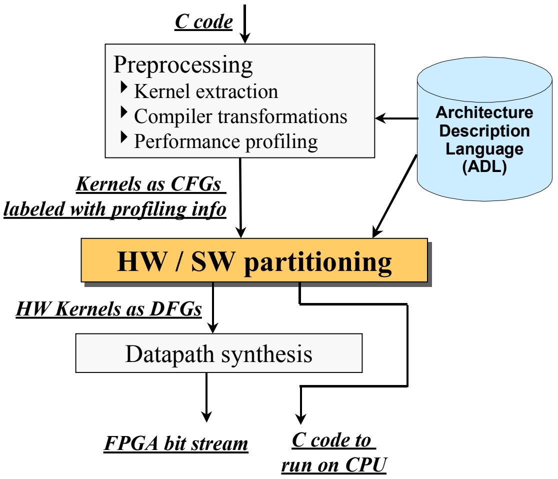


Figure 2. The Nimble Compiler flow: an overview.

3. Problem Formulation

In this section, we first give an overview of the Nimble Compiler environment. This provides a context for the problem formulation and cost function formulation of our hardware/software partitioning algorithm, which is the focus of this paper.

3.1 Nimble Compilation Overview

The Nimble Compiler environment depicted in Figure 2 focuses on extracting hardware kernels from the source application to accelerate on the FPGA. It is originally based on an early version of the GARP compiler[1].

The Nimble Compiler is **retargetable** and can be parameterized to the target platform described by the Architecture Description Language (**ADL**). ADL defines the components and parameters of the system such as the type of processor being used, characteristics of the reconfigurable array, memory hierarchy, etc.

Our studies show that loops represent a significant portion of application execution time, and yet are usually compact enough to implement in modest hardware resources. Therefore, the compiler focuses on finding the most profitable loops to extract out as hardware kernels. At the front end of the Nimble flow, the C program is preprocessed to extract the loop-level representations and parameters needed by the partitioning algorithm. Optimizations are applied to concentrate much of the execution time in as few loops as possible. A preprocessing step provides the partitioning algorithm a set of hardware candidates (kernels) to select from. Preprocessing not only extracts loops as kernels, but it also applies various hardware-oriented **compiler transformations** to generate multiple optimized versions of the same loop. These transformations are important because transformed code has different performance and area tradeoffs when implemented in hardware. For example, an unrolled loop requires more area in the FPGA, but it may accelerate the execution by exposing increased instruction-level parallelism. Potentially useful transformations include loop unrolling, fusion, pipelining, procedure inlining, and branch trimming, just to name a few. Profiling performed on the application and each extracted kernel to estimate the software performance, memory bandwidth need, trace behavior etc. A quick synthesis is done to estimate the delay and area of the hardware implementations.

The extracted kernels, internally represented as basic block control flow graphs (CFGs), are fed to the hardware/software partitioner which decides which kernels will go into the hardware. The selected hardware kernels are then input into our backend datapath synthesis tool to generate the corresponding FPGA bit streams, which are then used to configure the FPGA for a kernel's execution at runtime.

3.2 Hardware/software Partitioning Problem Formulation

We now define the hardware/software partitioning problem. The input to the partitioning step comprises two parts: the target architecture, and the set of loops/kernels extracted from the source application. The algorithm uses a fixed FPGA total size constraint as described in ADL, along with other parameters, such as configuration times and memory bandwidth.

Representing possible hardware candidates is a set of loops L, with each loop L_i having multiple kernels K_j, which include an original software version and several hardware versions generated from compiler transformed code. Figure 3 shows an example with two versions for one loop. Figure 3(a) is the original CFG implemented totally in hardware. Figure 3(b) shows a transformed version after unrolling the loop once and trimming off an infrequently executed branch (marked A) by keeping it in software. *Con, En,* and *Ex* are overheads incurred by putting a kernel in hardware and they refer to configuration cost, hardware entry and exit costs, respectively.

The kernels and the basic blocks are labeled with profiling information obtained in the preprocessing step of the Nimble flow (Figure 2). Profiling data includes the total software execution time for each kernel, average time for each basic block and execution frequencies of basic blocks. Hardware implementation data includes the hardware area and delay for each kernel. Details of our profiling approach is beyond the scope of this paper and are not discussed further here.

As pointed out earlier, we aim to exploit ILP in loops instead of task-level parallelism. Therefore, the compiler currently only supports mutually exclusive execution of the CPU and FPGA. This simplifies the partitioner since it does not have to consider multiple loops fired off simultaneously. Loops are executed purely sequentially according to their original C specification even if pulled off onto the FPGA for acceleration.

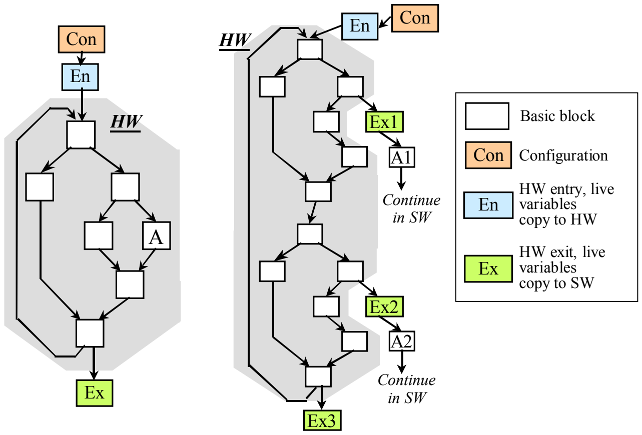
The goal of the partitioning algorithm is to select whether to put each loop into software or hardware, and if a loop is selected as hardware, which version to use, such that the execution time for the whole application is minimized. Note that while the partitioning is generally done at loop-level, the partitioner can make basic-block level decisions by putting only a subset of the basic blocks of a kernel CFG into the hardware.

3.3 Global Cost Function

As the algorithm tries to maximize the overall application performance, it uses a global cost function that incorporates the hardware and software execution times, hardware kernel entry and exit delay, and hardware reconfiguration time. Equation 1 shows the global cost of all loops T_{all_loops}, which is the sum of time spent in each individual loop $T(L_i)$. $T(L_i)$ denotes the total time spent in loop L_i, including all its iterations and entries.

$$T_{all\_loops} = \sum_{i \in L} T(L_i) \qquad (1)$$

$$T(L_i) = T_{sw}(L_i) \bullet Iter(L_i)\text{, if Li is in software.} \qquad (2)$$



(a) (b)

Figure 3. Multiple hardware kernels for one loop.

$$\begin{aligned} T(L_i, K_j) = {} & T_{hw}(L_i, K_j) \bullet Iter(L_i) \\ & + T_{sw}(L_i, K_j) \bullet Iter(L_i) \\ & + T_{sw2hw}(L_i, K_j) \bullet En(L_i, K_j) \\ & + T_{hw2sw}(L_i, K_j) \bullet Ex(L_i, K_j) \\ & + T_{config}(L_i, K_j) \\ & \quad \text{if kernel Kj of Li is in hardware} \qquad (3) \end{aligned}$$

$$\begin{aligned} T_{config}(L_i, K_j) = {} & N_{miss}(L_i) \bullet T_{miss}(L_i, K_j) \\ & + N_{hit}(L_i) \bullet T_{hit}(L_i, K_j) \end{aligned} \qquad (4)$$

As shown in Equation 2, if L_i is selected to be implemented in software only, its execution time can be characterized as the average time per iteration $T_{sw}(L_i)$ times its number of iterations $Iter(L_i)$. The computation of hardware time is more complex. Suppose we put kernel version K_j of loop L_i in hardware. The hardware loop time shown in Equation 3 is composed of several terms:

1. **Execution time spent in the hardware** itself. Similar to software time, it is the average hardware time per iteration $T_{hw}(Li, K_j)$ times the number of iterations $Iter(L_i)$.
2. **Execution time spent in the software** if kernel K_j only implements a portion of the loop in the FPGA. (See Figure3(b) for an example of a partial loop in hardware.)
3. **Communication time** between hardware and software, which involves the copying of live variables to and from the FPGA. Since variable transfer only happens when the program enters or exits from hardware, it is obtained by multiplying the cost per transfer (T_{hw2sw} or T_{sw2hw}) and the number of hardware entries *En(Li, Kj)* and exits *Ex(Li, Ki),* respectively.
4. **Configuration time** of the loop on the FPGA. Unlike the previous terms which only depend on decisions made about the current loop L_i, configuration time depends on decisions made for other loops that interleave with L_i during application execution. Some architectures (such as the GARP[6]) utilize a configuration cache to store recent configurations, so that they can be quickly reconfigured. The configuration cost for the cache miss *(T_{miss} (Li, Kj))* and hit *(T_{hit} (Li, Kj))* can be dramatically different, therefore, they must be computed separately as shown in Equation 4. The numbers of configuration cache hits and misses *(N_{hit} (Li) and*

N_{miss} *(Li))* for a loop depend what hardware/software partitioning decisions are made for all loops.

If configuration time is not included, optimizing execution time can be reduced to locally selecting the fastest implementation of each loop that satisfies the FPGA size constraint. However, because of the complexity of computing configuration cost, the partitioning problem is NP-complete, and involves evaluating loops in a global cost function to find the optimal solution.

4. Algorithm Flow

Since the total number of kernels can be large for many applications, we need to deploy a heuristic algorithm to efficiently solve the hardware/software partitioning problem. The two key heuristics that we have applied are:

1. Reducing the number of loops and kernels that the algorithm needs to analyze, by focusing solely on "interesting" loops that contribute significantly to the application time.
2. For the remaining loops, partitioning them into small clusters and performing optimal selection in each loop cluster.

Based on the above heuristics, the partitioning algorithm consists of the following main steps.

1. Loop entry trace profiling (LEP). LEP generates a complete trace that records all loops entries, such that the configuration cost for all loops can be inferred.
2. Interesting loop detection (ILD). ILD screens all hardware candidate loops and only selects a subset of "interesting" loops.
3. Intra-loop kernel selection. This selects the best hardware kernel among the multiple versions of a loop implementation.
4. Inter-loop selection. Selects among loops and decides which go into hardware and software, respectively.

Steps 2 and 3 apply the first heuristic, in an attempt to cut down the number of loops and kernels to be considered. Step 4 applies the second heuristic and is the most critical step of the algorithm. The rest of this section describes these steps in detail.

4.1 Loop Entry Trace Profiling and Compression

When a hardware loop is entered for the first time, it needs to be configured onto the FPGA. If it is entered again before being overwritten by another loop, it does not require reconfiguration. To compute configuration cost, we need to know the exact runtime sequence of all hardware candidate loops (i.g. the entries to these loops). Loop entry trace profiling (LEP) identifies and instruments loop entries to generate a trace. The trace can potentially be huge, e.g. encoding four frames using standard MPEG-2 generates ~200M bytes of loop entry trace. LEP incorporates an online compression scheme to encode the trace. Loop trace compression not only saves storage space, but more importantly, the compact representation allows fast traversing of the trace in later steps of the algorithm. For the MPEG-2 encoding example, the trace size is reduced to several Kbytes after compression.

4.2 Interesting Loop Detection

While the goal of the partitioning algorithm is to select loops to implement in the FPGA to achieve maximum overall acceleration, Amdahl's law implies that we should focus on loops that represent a large portion of the application total time. We have implemented an interesting loop detector (ILD), which reports the percentage contribute a loop makes to total application time. Table 1 shows the ILD results for several benchmarks. The third column of the table shows the number of loops that contribute to more than 1% of the total program execution time. The fourth column shows the total contribution of these >1% loops. Table 1 suggests that, even though the total number of loops in an application may be large, only a small number of these loops (2—15 in the examples) contribute to most of the applications' execution time (90+%). Therefore, if we focus on these few loops, we can expect the computation cost for the algorithm to reduce significantly, yet still achieve comparable quality of results because we are accelerating most of the significant loops of the program. Furthermore, even if all loops can be accelerated by the FPGA, any speedup for insignificant loops is usually negated by the configuration overhead.

Benchmarks	*# loops*	*# loops >1%*	*Total % (>1%)*
Wavelet image compression	25	13	99%
EPIC encoding	132	13	92%
UNEPIC decoding	62	15	99%
Media Bench ADPCM	3	3	98%
MPEG-2 encoder	165	14	85%
Skipjack encryption	6	2	99%

Table 1. Interesing loop detection for benchmarks.

4.3 Intra-Loop Selection

Since each hardware candidate loop can have multiple kernels generated by compiler transformations, we apply intra-loop selection, to evaluate these multiple hardware versions, and select the best one that fits within the FPGA size constraint. This further cuts down the number of kernels to be considered in the next step—inter-loop selection. The decision of whether to put a loop in hardware or software can not be made until the inter-loop selection step. Therefore, along with the best hardware version, we also keep the original software version for further evaluation in Step 4.

The criterion for intra-loop selection is the total loop execution time, not including configuration time. This is because the number of configurations for a loop is not available until we know the hardware/software partitioning result for all loops.

Figure 4 illustrates intra-loop selection. A—D, P and Q represents several points in the hardware design space for a loop. Kernels P and Q do not satisfy the hardware size constraint. We can trim off infrequently executed branches in P and Q by keeping these branches in software to obtain the more compact implementations P* and Q*. For all kernels within the area limit, the fastest one (in

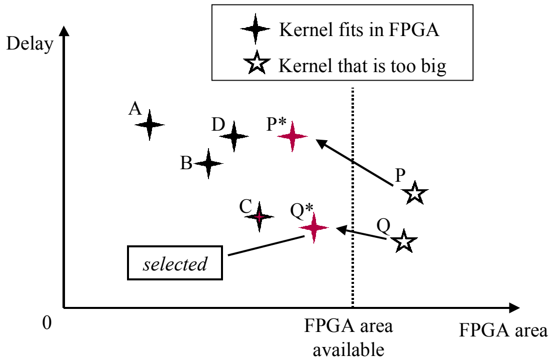


Figure 4. Multiple hardware versions of a loop, in the area and delay design space.

this case, kernel Q*) is selected.

4.4 Inter-Loop Selection

Inter-loop selection is the most critical step of the algorithm, as the final partitioning decision has to be based on the global cost function described in Section 3.3. Selections in previous steps eliminate loops/kernels based on execution time metrics for each individual loop, while in this step, we analyze the interaction among all loops, and optimize execution time and configuration time.

Even though the number of loops (say *n*) left after steps 1 and 2 may not be very large, the number of configuration possibilities is exponential (2^n). We introduce a clustering technique to partition loops into small clusters to allow us to solve the partitioning problem optimally for each cluster.

4.4.1 Hierarchical Loop Clustering Based on the Loop-Procedure Hierarchy Graph

Clustering of loops is based on the loop-procedure hierarchy graph (LPHG) which represents the procedure call and loop nest relations in the application. Figure 5 shows the LPHG for the wavelet image compression benchmark. A square node indicates a procedure definition, and a circular node indicates a loop. Edges into a procedure node represent calling instances to that procedure. An edge from a procedure to a loop indicates the loop is defined within the procedure. An edge from a loop *a* to another loop *b* indicates that *b* is nested inside loop *a*. There may be multiple incoming edges for a procedure, indicating multiple calling instances of the same procedure. Recursive procedures create cycles in the graph.

An LPHG captures loops and their relative positions in the application and therefore provides a navigation tool for the partitioning algorithm to traverse the loops. We define the shortest distance from a node to the root node (*main*) as the *level* of that node. We can make the following observations:

- If two loops have different first-level predecessors, they appear in a disjoint part of the LEP trace and do not compete for the FPGA configuration. For example, in Figure 5, all entries of loop FW3 appear strictly before those of RLE2. These loops can be partitioned into different clusters.
- Conversely, loops sharing common loop or procedure predecessors tend to compete with each other, and therefore should be placed in the same cluster. In the example, entries of FW3 and FW4 interleave and hence compete for the FPGA resource.

Based on the above observations, we have developed a hierarchical loop clustering algorithm based on the LPHG.We predefine a size limit for the loop clusters to ensure that the clusters are small enough for feasible optimal selection. The loop clustering algorithm traverses the loop-procedure hierarchy graph in a top-down fashion and recursively clusters loops until the sizes of all clusters are within the pre-defined limit. The algorithm works as follows.

1. Starting from the first level of the loop-procedure hierarchy, loops with a common predecessor at this level are clustered together. In Figure 5, the unshaded loop nodes are discarded after ILD. Clusters {R4}, {FW2, FW3, FW4, FW5, FW6, FW7}, {Q3, Q6}, {RLE2, RLE3}, and {E4, E3} are generated based on their level 1 predecessors.
2. If the size of any cluster exceeds the cluster size limit, we need to traverse down a level in the hierarchy and refine the clusters by grouping loops again with common predecessors at the new level. For example, the FW loop cluster has six loops. If we set the size limit at 5, we need to go down a level, to level 2, and recompute the clusters. All the other clusters are within the size limit and need no further refinement.

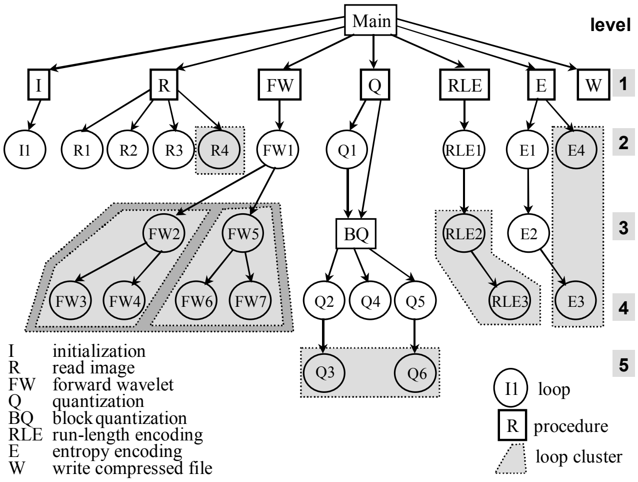


Figure 5. Loop-procedure hierarchy graph for wavelet image compression benchmark.

3. Repeat step 2 until all loop clusters satisfy the cluster size limit. In the example, at level 2, the FW loops still can not be resolved into smaller clusters and it is necessary to go down to level 3. The clustering result is shown in the figure, {FW2, FW3, FW4} and {FW5, FW6, FW7}.

4.4.2 Optimal Selection in Loop Clusters

After the loops have been partitioned into smaller clusters, our algorithm performs optimal hardware/software partitioning for each individual loop cluster. The approach adopted is to exhaustively search the solution space of all partitioning possibilities, evaluate each of these possibilities, and select the one with the best overall performance for all loops in the cluster.

To evaluate the overall performance, we must compute the number of reconfigurations needed in each partitioning possibility. This is achieved by walking through the compressed loop entry trace. The state of the configuration cache (if there is one) is taken into account to estimate the number of hits and misses.

5. Experimental Results

The hardware/software partitioning algorithm has been applied on real benchmarks, as part of the Nimble compilation flow. The Nimble flow takes off-the-shelf C code and compiles it onto a target architecture of a combined CPU and FPGA. The flow is fully implemented and completely automated.

In order to show the result quality and computation efficiency of our partitioning algorithm, we compare it here with a local optimization algorithm that selects loops by evaluating individual loop cost, instead of the global cost function used by our algorithm. The local optimization uses a greedy approach: if a loop shows acceleration in the FPGA, assuming it is configured once, then it is put in hardware.

We also compare the result quality of our algorithm with an absolute performance upper bound. The upper bound is obtained via the following method: For each loop (not limited to ILD loops), we use the performance of its best hardware kernel, regardless of what size it is, to estimate its hardware execution time, and we make the idealizing assumption that only one

Benchmarks	#loops	Our algorithm		Local-optimal algorithm		Performance upper-bound (cycles)
		CPU time (sec)	Result performance (cycles)	CPU time (sec)	Result performance (cycles)	
Wavelet compression	25	0.17	1.74e+5	0.05	5.10e+5	1.74e+5
MPEG-2 encoder	165	1.92	7.47e+8	0.49	1.58e+9	7.17e+8
MediaBench ADPCM	16	0.08	7.09e+4	0.04	8.00e+4	7.00e+4
Unepic decoding	62	1.53	8.57e+6	0.28	1.47e+7	8.42e+6
Skipjack encryption	6	0.04	8.00e+4	0.01	1.10e+5	8.00e+4

Table 2. Results of our algorithm compared to a local-optimal algorithm and the absolute performance upper bound.

configuration is needed. The lesser of the software version time and the hardware version time (combined hardware execution time and configuration time) is used as the estimated time for that loop. This estimate provides an absolute lower bound on the execution time for that loop. This bound is optimistic: even an optimal algorithm may not always achieve this performance upper bound because of the single configuration assumption used in obtaining the bound.

The benchmarks we have used include the wavelet image compression algorithm, an MPEG2 encoder and decoder from the MPEG Simulation Group, the MediaBench ADPCM, the Unepic benchmark from MIT, and the Skipjack encryption algorithm, among other smaller test programs. All are off-the-shelf C code and compiler directly using the Nimble framework.

We experimented with the partitioning algorithm targeting two different platforms: GARP[6] and the ACEII card[13]. GARP is a single-chip architecture with a MIPS 4000 CPU, a reconfigurable array of 21 by 32 CLBs, on-chip data and instruction caches, and a 4-level configuration cache. ACEII is a board-level platform developed by TSI Telsys. It consists of a uSparc CPU and Xilinx 4085 FPGAs. There is no configuration cache on the ACEII.

Table 2 shows the experimental results of our partitioning algorithm on the GARP architecture. The table includes the performance of the partitioned design and the CPU time spent in the partitioning algorithm. These results are compared to that of the local-optimal partitioning algorithm, and the absolute performance upper bound. The results indicate that while our algorithm consumes comparable CPU time to that of a greedy local-optimal algorithm, it generates close-to-optimal hardware/software partitions in all the benchmarks shown.

6. Conclusions

We have presented a hardware-software partitioning algorithm that targets dynamically reconfigurable architectures consisting of a single CPU and an FPGA co-processor. The algorithm applies heuristics to achieve high computation efficiency yet finds optimal or near optimal solution in most cases. Using the algorithm in a fully automated framework on real off-the-shelf benchmarks demonstrate its effectiveness.

We plan to extend our work in the following directions: 1) instead of allowing only one loop in hardware at any time, we consider introducing multiple kernels into the same hardware configuration to improve hardware utilization; 2) improve the algorithm by more closely coupling compiler optimizations with the hardware/software partitioning, e.g. the partitioning algorithm should provide directives on what are the best optimizations to perform.

7. Acknowledgement

This work is partly sponsored by DARPA/AFRL under grant F33615-98-2-1317. The authors would like to thank our collaborators at Lockheed Martin ATL and UC Berkeley.

8. References

[1] T. J. Callahan and J. Wawrzynek, “Instruction level parallelism for reconfigurable computing,” *Proc. 8th Intl. Workshop on Field-Programmable Logic and Applications*, Sept. 1998.

[2] B. Dave, G. Lakshminarayana, and N. Jha, “COSYN: hardware-software co-synthesis of embedded systems,” *Proc. 34th Design Automation Conference,* 1997.

[3] R. P. Dick and N. K. Jha, “Cords: hardware-software co-synthesis of reconfigurable real-time distributed embedded systems,” *Proc. Intl. Conference on Computer-Aided Design*, 1998.

[4] R. Ernst, J. Henkel, and T. Benner, “ Hardware-software cosynthesis for microcontrollers,” *IEEE Design and Test of Computers*, vol.10, no.4, pp.64-75, Dec. 1993.

[5] R. Gupta and G. De Micheli, “Hardware-software cosynthesis for digital systems,” *IEEE Design and Test of Computers*, vol.10, no.3, pp.29-41, Sept. 1993.

[6] J. R. Hauser and J. Wawrzynek, “Garp: A MIPS processor with a reconfigurable coprocessor,” *Proc. FCCM '97,* 1997.

[7] A. Kalavade and E. A. Lee, “A global criticality/local phase driven algorithm for the constrained hardware/software partitioning problem,” *Proc. International Workshop on Hardware-software Co-design*, pp. 42-48, 1994.

[8] M. Kaul *et al.*, “An automated temporal partitioning and loop fission approach for FPGA based reconfigurable synthesis of DSP applications,” *Proc. 36th Design Automation Conference*, 1999.

[9] Y. Li and W. Wolf, “Hardware/software co-synthesis with memory hierarchies*,” IEEE Transactions on CAD*, vol. 18, no.10, pp.1405-1417, Oct. 1999.

[10] S. Prakash and A. Parker, “SOS: synthesis of application-specific heterogeneous multiprocessor systems,” *Journal of Parallel and Distributed Computing*, vol.16, pp.338-351, 1992.

[11] W. Wolf. “Hardware/software co-design of embedded systems,” *Proceedings of the IEEE*, July 1994.

[12] M. B. Gokhale and A. Marks. “Automatic synthesis of parallel programs targeted to dynamically reconfigurable logic arrays,” *Proc. FPL,* 1995.

[13] TSI Telsys, “ACE2 Card Manual”, 1998.

Appendix G. Xima - The Nimble Datapath Compiler

Abstract

In the context of the Nimble compiler, datapath portions of a C program are mapped on to an FPGA. The goal of the datapath compiler is to efficiently map the unscheduled dataflow graph description of the datapath to an FPGA. The involved steps include performing scheduling, technology mapping, module generation, and datapath floorplanning. For the TSI-Telsys ACE board with an ADM-XRC daughter card, that has been selected as prototype platform, the target device is a Xilinx Virtex 1000. To leverage existing datapath compiler technology and to quickly provide a prototype solution, the U.C. Berkeley work on the Garp architecture, in particular, their datapath compiler tool Gama has been selected as starting framework. As modified for the Xilinx architecture, the tool is called Xima.

1 Introduction

For a prototype environment for the Nimble Compiler, this project initially focused on providing a compiler for the reconfigurable TSI-Telsys ACE II board. The board includes a MicroSparc, two Xilinx XC4085XL devices, and dedicated SRAM and DRAM. The U.C. Berkely BRASS group introduced a unique single chip architecture (Garp) containing a reconfigurable array and a MIPS processor. The Garp compiler (garcc) was developed to provide an automatic path from a C program to the Garp architecture. A part of the Garp compiler is the datapath compiler gama which performs the module mapping, scheduling, and placement of the datapath portions of programs to the reconfigurable array of the Garp architecture. A very restricted version, gamax, providing limited technology mapping support for the Xilinx 4000 series FPGAs, was developed in the initial phase of the project. This work leveraged much of the technology from the Garp datapath compiler.

The current Xilinx datapath compiler known as *Xima*, further improved upon *gamax* by providing full technology mapping for integer ANSI C and for some domain-specific functions, with an extensible and isolated generator library. It also included target support for the Virtex 1000 series parts, on a ADM-XRC daughter card mounted on TSI-Telsys ACE I boards. Within the context of the Nimble Compiler, the XC4085XL / ACE II combination is referred to as the "ace" target, and the Virtex 1000 / ACE I combination is referred to as the "acev" target.

The input to *xima* is an unscheduled dataflow graph (in a format known as "AFL"), consisting of processing nodes and data (or control) communication edges. The output consists of:

a. A ".ro" file, containing symbolic and other information on the datapath rows

b. A ".xnf" file, which is the datapath netlist

c. ".edn" files for each operator module required by the datapath

The following sections describe the xima datapath architecture framework, followed by a description of the datapath compiler for it.

2 Datapath Architecture for Xilinx

The datapath implementation model selected for the Xilinx implementation has been derived from the Garp architecture and the computation model used to implement datapath(s) on its reconfigurable array.

2.1 Garp Features

The reconfigurable array of Garp is a two-dimensional array of small computing elements interconnected by a network of wires. The following figure gives a high level view of the basic architecture. For a more detailed description see ‘The Garp Architecture’ John Hauser.

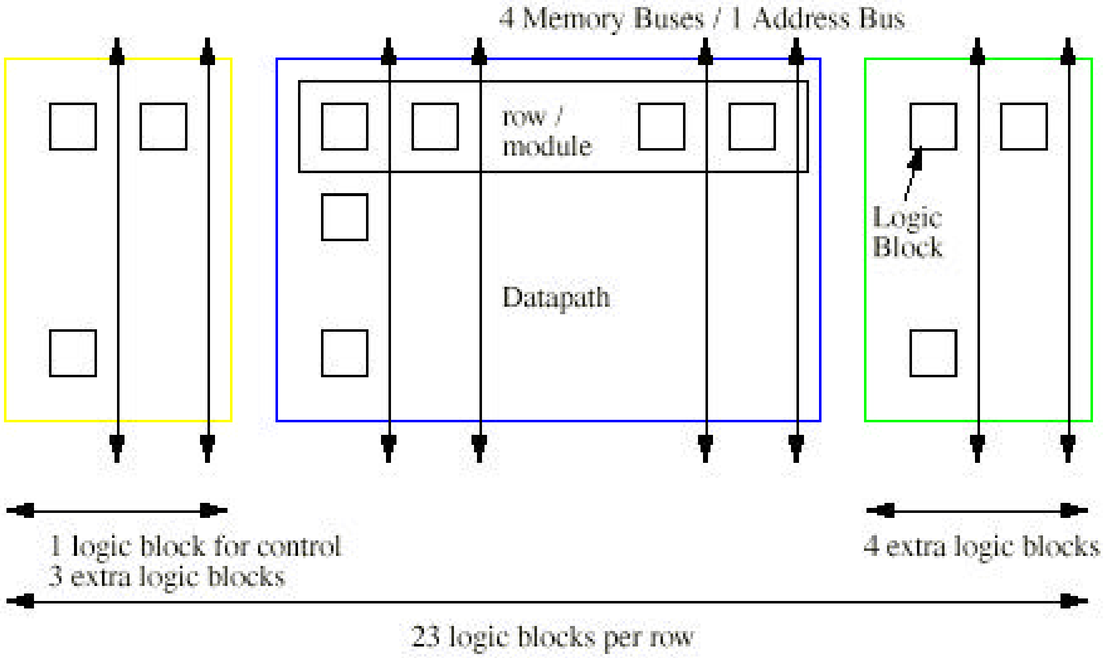


Figure 1: The Garp Architecture

Each row / module can implement a 32 bit arithmetic function such as "add", "minus", "or", "and". The output of each module is registered. One set of the extra logic blocks is used to implement a sequencer for enabling the registers of the modules according to an execution schedule. The control block is used to generate HALT, and predicate memory access control signals. The memory buses allow for concurrent reading / writing to multiple modules. At this time the software supports single read/write accesses with concurrent provision of the address.

2.2 Xilinx Features

The Xilinx XC4085XL device is a general purpose FPGA in the Xilinx XC4000 series. It shares the same basic logic cell - 4 input lookup table - with the Garp architecture. The main difference is in the cell arrangement and the interconnect. The Xilinx Virtex 1000 parts share enough similarities to the XC4000 series parts to allow for use of a unified module generator library for the two Xilinx target architectures. There are differences in the support logic between the 4085 and Virtex targets. This document focuses primarily on the later, since it is the more capable and refined target.

2.3 Overall Architecture

Several of the Garp features have to be implemented on the FPGA using the general available resources to achieve a similar timing predictability and to allow memory access functionality. The following diagram is an illustration of the primary datapath features, with some possible row operations and sequencer paths annotated:

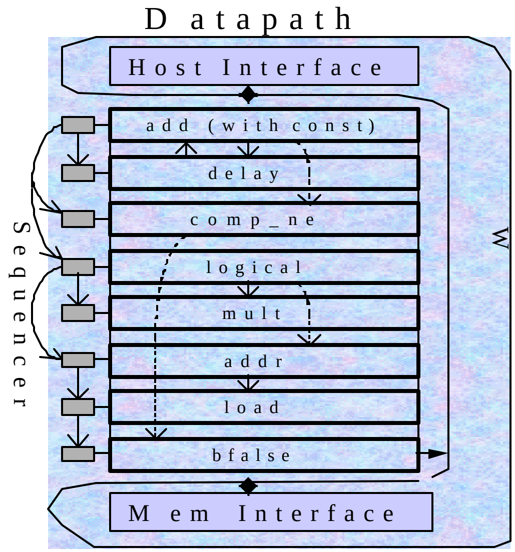


Figure 2: Xilinx Datapath Overall Architecture

As shown in figure 2, the datapath consists of an interface and control wrapper, operator modules (add, delay, etc...), and a sequencer. Each operator module also requires some control and glue logic.

2.3.1 The Wrapper

The wrapper provides a control and data interface between the host and the datapath as well as support for straight SRAM access and cached DRAM access (although only the latter is used).

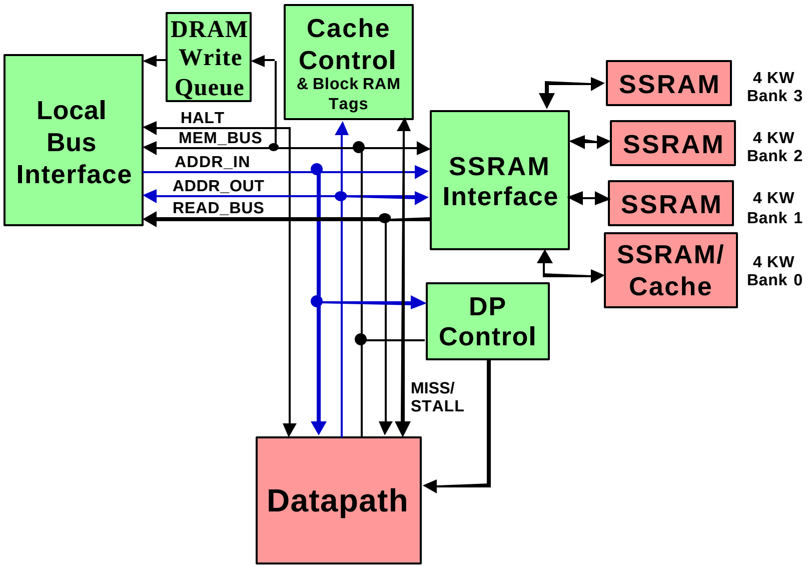


Figure 3 illustrates the main features of the wrapper:

Figure 3: The Datapath Wrapper

The host accesses the wrapper functionality through a memory-mapped interface. On the ADM-XRC card for the Virtex target, the memory map is based at the "S0" memory space, which is currently defined to be 31000000 (hex). Consult the ADM-XRC and PLX-9080 documentation for details on memory map and capabilities.

2.3.1.1 The Wrapper's Host / Datapath Interface

The following table details the host access of datapath control and row registers:

Offset (hex)	Read/Write	Operation
404000	R/W	Clear necessary datapath and wrapper registers
404400	R/W	Start the datapath sequencer at its initial state
404800	R/W	Stop the datapath sequencer
405000	W	Write data to the row specified in bits 9-0
406000	R	Read data from the row specified in bits 9-0

Row data can actually be read from offset 405000h as well as 406000h, but this was not the original definition. Rows correspond to address multiples of 4. So, row-0 would be read from offset 406000h, and row 1 would be read from offset 406004h. A read of row 511 (offset 4067FCh) retrieves the exit status. The least significant bit which is high corresponds to the row which triggered the exit. The datapath alerts the wrapper of an exit by driving the "halt" line active low.

2.3.1.2 The Wrapper's Host / SRAM Interface Memory Map

The following table details the host access to the SRAM through the wrapper (offset from S0):

Offset (hex)	Read/Write	Operation
000000	R/W	SRAM Bank 0 (cache in lower 4k words)
100000	R/W	SRAM Bank 1
200000	R/W	SRAM Bank 2

300000	R/W	SRAM Bank 3

2.3.1.3 The Wrapper's Datapath / Memory Interface

The datapath can access the full SSRAM memory space, as well as the DRAM (through the cache). The following table defines the memory map for the memory access as seen from the datapath:

Address (hex)	Read/Write	Operation
0000000-0FFFFFF	R/W	DRAM Space (16 MB)
1000000-10FFFFF	R/W	SRAM Bank 0 (1 MB)
1100000-11FFFFF	R/W	SRAM Bank 1 (1 MB)
1200000-12FFFFF	R/W	SRAM Bank 2 (1 MB)
1300000-13FFFFF	R/W	SRAM Bank 3 (1 MB)

2.3.1.4 The Wrapper's Cache

The datapath's access to DRAM is provided using a write-though cache. The cache's tag bits are located in Virtex Block SRAM and the data is maintained in external SRAM. The total cache size is 4096 words. It is direct mapped, with 32 words loaded from DRAM on a load miss. The wrapper stalls the datapath and starts the DRAM access as soon as the miss is detected, and datapath operation resumes as soon as the final word of the 32 is received. Loads are performed in pipeline fashion with one cycle per load when there is a cache hit. There is a pipeline latency associated with loads, which is currently 5 cycles but could be reduced to 4 cycles. The five cycles are:

1. Register on address output from datapath (this could be eliminated)
2. I/O register on the address output to SSRAM
3. Register of the ZBT flow-through SSRAM
4. I/O register on the data input to the FPGA from SSRAM
5. Register on the data input to the datapath (this is in the datapath, not in the wrapper)

Stores to the cache take one cycle for word (32-bit) stores and two cycles for short (16-bit) and byte stores. The cache keeps track of valid data on a byte basis, which made it necessary to perform a read-modify-write when storing bytes and shorts. It should be possible to modify the logic so that short and byte stores are also one cycle. The cache is write-through with stores placed in a queue for write-back to DRAM. The wrapper writes back the queued data to DRAM whenever the queue is not empty. This queue for write-back to DRAM is implemented in Block SRAM and is 256 words deep--a size choice that was driven by the Block SRAM dimensions. The datapath is stalled on store only if this queue fills, otherwise, the datapath is not slowed by stores. At the end of datapath activity the "halt" interrupt is delayed until any pending write-backs are completed.

2.3.2 The Datapath

The datapath consists of "rows" (modules implementing operator functionality), a sequencer, and control logic. The word "row" here, is a left-over from garp, since the actual module bit-wide orientation is vertical on the Xilinx parts. The following figure illustrates the datapath architecture:

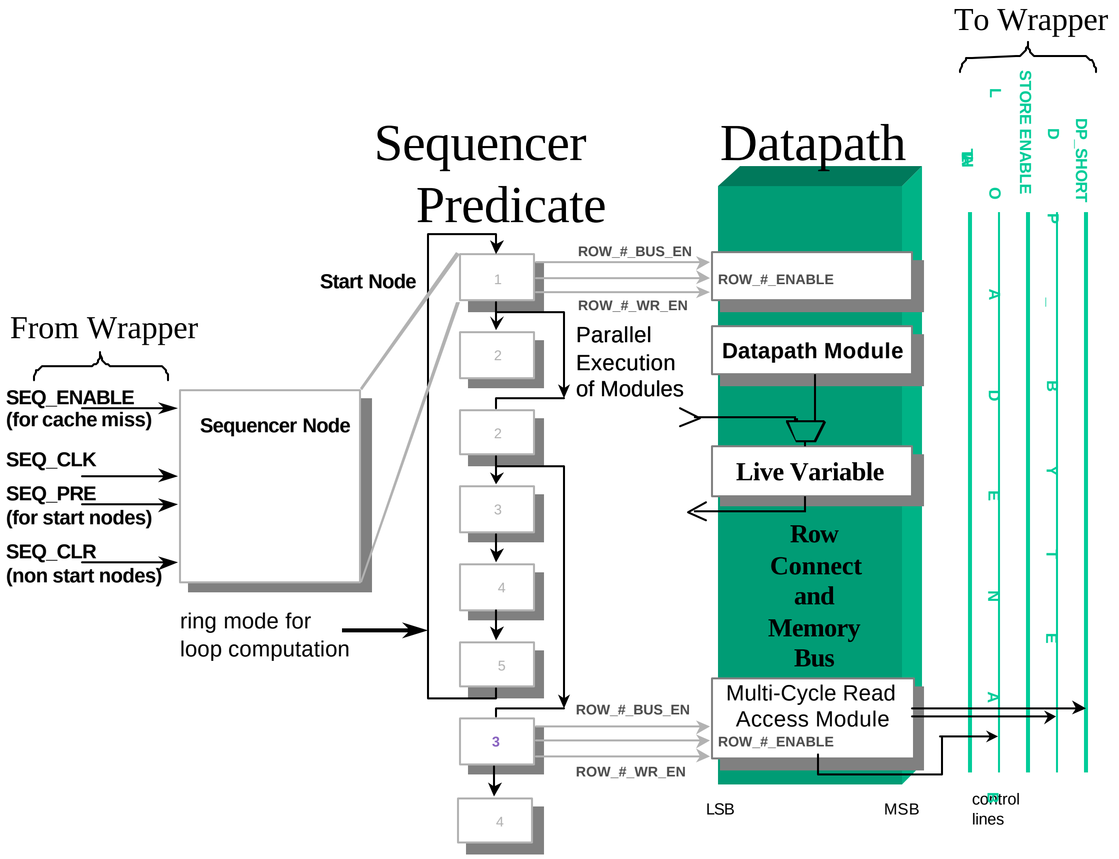


Figure 4: Detailed Datapath Architecture

2.3.2.1 The Datapath Rows (Module Generators)

The module generators, written in a Java-based language called JHDL, build macros for the Xilinx targets which perform sets of operations as mapped from the input AFL. After Xima has determined the mapping and placement for the row modules, a placed macro is called out in the datapath's top-level "xnf" netlist. The macro generation is invoked via command line, with the proper parameters (sign, bit-width, etc...), resulting in an EDIF netlist placed in a directory named "Cells" under the current design directory. For detailed information on the module generators, consult the Specification for FPGA Macro Generators for the Nimble Compiler. The modules have defined input, output, and control interfaces to make interconnection simpler for Xima as shown in figure 4.

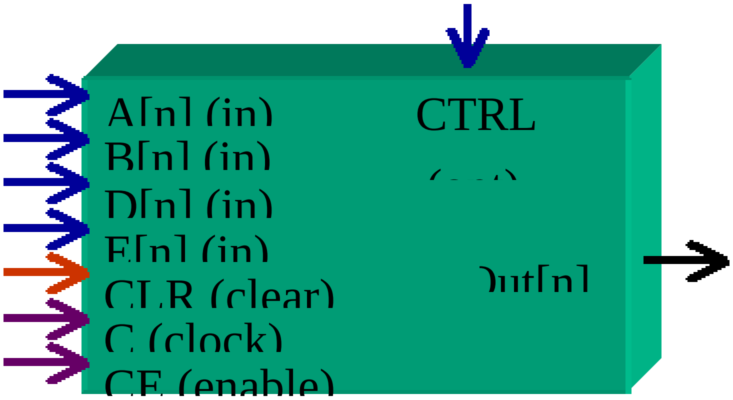


Figure 5: Common Module Interface

Xima defines some additional common circuitry and signaling around the module generators, which includes the memory interface and host access signals.

Data "flow" for a given operator module is from left to right (top to bottom in the Garp "row" way of thinking), although modules may take input from any location. It is the job of the floorplanning and placement to minimize the distances between module connections.

2.3.2.2 The Datapath Sequencer

The datapath sequencer triggers operator modules according to a precomputed schedule. The sequencer can be started, paused (for cache miss), and stopped by the wrapper (see section 2.3.1). Modules, once fired, may issue signals to the wrapper, which control sequence halting and interrupt generation, memory access, and data size/alignment. The sequencer is built of subtree sequencers which run in parallel.

2.3.2.3 Datapath Memory Subsystem Architecture

One major difference between the Garp architecture and the Xilinx targets is the available bus resources. While the Garp architecture provides multiple dedicated memory and address buses per logic block, the Virtex and XC4085 have limited tristate resources, restricting the memory bus architecture. The datapath for the Virtex target has separate data write (MEM_BUS), data read (READ_BUS), and address (ADDR_BUS) signaling, to avoid contention for these resources. Memory reads and writes are broken up into separate address and data phases. This increases required area and schedule allotment, but allows the address and data to be placed closer to the rows which feed and/or use them. A write data, read data, and read or write address can all be scheduled for on the same clock cycle. Since the READ_BUS is only has one driver, it is not a tristate bus. The ADDR_BUS and MEM_BUS are tristate, since multiple rows may issue address or write data. The active-low memory control signals consist of:

DP_BYTE	Indicates byte data access
DP_SHORT	Indicates short data access
LOAD_ENABLE	Indicates start of load address
STORE_ENABLE	Indicates start of store address

If no memory accesses are used, the four memory control lines are always high.

2.3.2.3.1 Memory Write

A memory write begins with the datapath driving the tristate STORE_ENABLE signal low for one clock cycle. This is done by tying the enable of the tristate driver to the inversion of the clock enable (ENABLE_ROW_#) signal of the corresponding address row, and grounding the input. For a predicated write, the output from the corresponding condition row is included, so the STORE_ENABLE is never generated if the predicate condition is not satisfied. On the second clock cycle of the write, the address is driven onto the MEM_ADDR_BUS. This is done by including the enable of the nearest successor in the product terms for the address row's TENABLE_ROW_# signal. Also during this cycle, the DP_BYTE or DP_SHORT signal may be asserted, depending on the data size. On the third clock, the write data is driven, with the TENABLE_ROW_# signal derived from the closest successor to the write data row.

2.3.2.3.2 Memory Read

A memory read begins with the (possibly predicated) LOAD_ENABLE signal going low for one cycle. On the second cycle, the address is driven onto the MEM_ADDR_BUS, and DP_BYTE or DP_SHORT may be asserted. Cycles three through five are fill (and can be additional read/write addresses and data, and on cycle six, the enable signal (ENABLE_ROW_#) is driven active high for the load data row, completing the access.

3 Xilinx Datapath Compiler (Xima)

The datapath compiler for the Xilinx targets has to perform the scheduling of the computational tasks, the technology mapping and module generation, the floorplanning, and the control logic and sequencer generation. It reads in a data flow graph file in a human-readable format known as "AFL", and outputs a ".ro" row description file, an ".xnf" netlist of the datapath, and ".edn" EDIF netlists of the generator modules. To utilize existing technology, the Berkeley *gama* tool has been selected as development platform. The following sections elaborate the additions and changes to the base algorithms that have been made to perform the mapping to Xilinx XC4085XL and Virtex 1000 devices rather than the Garp architecture.

The following is the command-line usage of xima:

```
usage: xima <options> input{.afl}
```

The following are the command-line arguments available for xima:

```
debug options:
   -v: verbose output
   -c: show cover (shows the process of grammatically processing AFL nodes)
   -d: turn on yydebug (internal debug use)
   -x: keep going even if no cover
output options:
   -g: .ga output (garp only)
   -G: .ga output with control! (garp only)
   -8: 8-bit wide datapath (default 32) (deprecated)
postpass optimization options:
   -M: module reordering
   -E: Eigenvector-based module reordering
   -C: do compression optimization
prelabel and labeling options:
   -L: tree layout reordering
   -r: modifies -L for better routability
   -s <size>: size cutoff for replication
   -m: no reuse (minimize delay)
   -a: cost function: minimize area
   -S: super-auto-dynamic cost function! (for experimental use)
   -f <dist>: assume rows <dist> or more rows apart need long wire
   -F: inhibit adjacent tree matches
   -W [weight]: cost function: weighted
       "I'd rather have <weight> more rows than
        an extra cycle of delay (default 8)"
near-obsolete options:
   -w: write modified .afl file
   -r: dump routing postscript
   -p: optimize for pseudo-constants
   -e: do pre-estimate and reorder trees
   -D: like -e, plus dynamic cost fn
```

In most cases, calling xima on an AFL file with no options suffices. Using the "-v" and "-c" options are useful when doing an in-depth analysis of xima operation.

3.1 Flow Graph Input Parse and Translate

The initial task which xima performs is to parse the AFL file into an internal node-based representation. This is done in two steps.

3.1.1 Initial AFL Read

In the first parsing step, the function "GraphInputWithInit()" from the "flowLib" library is used to read the AFL into a flow-graph structure with nodes and edges, a data structure which similarly represents the data from the input file.

3.1.2 Flow Graph Translation

The second parsing step, the function "translate_flowgraph()" in the file "input.c" translates the data from "flowLib" into the internal node-based representation. The nodes facilitate the building of data-flow subtrees, each having an array of children (kids[]) indicating data operations whose results feed into the current node. The function which performs the bulk of the computation in this step is "translate_node()" in "input.c". The translation performs child determination, attribute initialization, and operation code assignment. Nodes of type "load" and "store" are split into separate nodes for address and data phases, and special handling is provided for certain nodes, including queue operations, delays, and inputs.

3.2 Tree Collection

After the flow graph data has been parsed into the internal node representation, a series of preprocessing steps is performed on the data, the goal of which is to make it possible for grammatical tree coverage. The first step (prepass_all_nodes() in "input.c")removes unnecessary nodes, initializes successors, annotates control information, places nodes in topological order, and marks live variable information. The second step (find_trees(),unshare_and_fix_trees(), and reorder_trees() in "input.c") determines the separable trees in the data-flow, organizes them, and estimates initial schedule parameters.

3.3 Mapping, Scheduling and Placement

Mapping, Scheduling, and Placement all take place concurrently (in maptrees() of "node.c"), since the modules chosen during mapping affect the time allocations in the schedule. For the technology mapping of the dataflow graph to module primitives, the grammar package developed with tburg in *gamax* has been reused. The grammar (contained in the file "xgrammar.m4") defines how operations of a subtree map to generator modules. The goal of the grammatical approach is to cover as many operations in as few clock cycles and as little area as possible, and to minimize inter-row connection delays. The amount of time and area (rows) each module takes is computed in the file "cost.c"

The basic ASAP scheduling applied in *gama* still applies to the datapath architecture implementation for the Xilinx targets. However, the schedule allotments imposed by the different memory and bus architecture and the different module and wire timing have to be taken into account:

- address of memory write access one clock before data
- address of memory read access four clocks before expected data return

Due to the separate address, read data, and write data busses, the only constraint on memory access is that a single address may be scheduled to drive the address bus at any given time slot. Cycle-by-cycle accesses of same or different types are allowed and handled properly by the wrapper. Only one exit row is allowed to fire per time slot. The ASAP value for a multi-cycle module indicates the cycle on which the module has completed. This usually corresponds to the cycle on which the enable of the module's output register should be active. Memory address rows are handled specially, in that the address row is always enabled on the first clock cycle, and the ASAP (derived from the row's op_delay field) is used merely to push the dependent data phase into the correct cycle.

3.4 Control Information Annotation

Before netlist generation can proceed, control information must be gathered for the proper sequencing of the datapath. This is done in the function xiAddControlInfo() of the file "xi_control.c". Control annotation is slightly complicated by pipelining, which allows rows to have both a lag and an ASAP. The lag is a delay measured in terms of loop iterations, and the ASAP is a cycle offset from this lag boundary. First, the bottom exit and an input row are found. Then, the bottom exit is moved to the last clock cycle, and all the delays are moved to the last cycle of their pipeline segment. Lastly, each module is annotated with its closest successor and predecessor modules for later use by sequencer generation layout.

3.5 Module Generation

Once all of the modules, positions, and schedule have been determined, the steps to build the target netlist begin. After an initial prolog is generated, for each module, control logic and the module itself are emitted.

3.5.1 Proglog Generation

The first generation step is prolog information, which is common in every netlist generated by Xima. The prolog first includes some directive information, and some informational comments. Next, the following global signals are set up:

Control Inputs From Wrapper

• DP_FF_CLR	Clears the host-access row decode registers
• SEQ_CLK	The sequencer clock
• DP_CLK	The clock to the datapath's row registers
• CLK_EN	Clock enable for stalling datapath during cache misses
• SEQ_PRE	Sets the bits for rows scheduled on cycle 1
• SEQ_CLR	Clears other sequencer bits
• SEQ_ENABLE	Clock enable for the sequencer
• ROW_WRITE_ENABLE	Enables data to be written into a row/module
• ROW_OUTPUT_ENABLE	Enables rows to activate tristate output driver
• IO_OPERATION	Indicates a host I/O operation
• IO_ADDRESS_BIT(7:0)	Row address for host access
• EXIT_IDENTIFY	Row causing exit drives bit onto MEM_BUS

Memory Interface

MEM_BUS_BIT(31:0)	Tristate write-data bus
READ_BUS_BIT(31:0)	Read data bus, fanned out to all load rows
ADDR_BUS_BIT(31:0)	Tristate address bus for both reads and writes
DP_BYTE	Datapath informs wrapper that this is byte access
DP_SHORT	Datapath informs wrapper that this is short access
LOAD_ENABLE	Issued before load address is driven
STORE_ENABLE	Issued before store address is driven
DATA_VALID	Not used

Miscellaneous

Halt	Datapath informs wrapper that loop has terminated

In addition, the prolog instantiates some pullups and glue logic.

3.5.2 Control Logic for Each Module

The control logic for modules consists of a sequencer segment, host I/O access logic, and possible memory or halt control.

3.5.2.1 Sequencer

The sequencer segment for a row drives the ENABLE_ROW_# signal high (where "#" indicate row number) during the scheduled clock cycle. Provisions are made to prevent rows scheduled on the first clock cycle from firing after a halt. The information annotated in xiAddControlInfo() (see section 3.4) is used to place the proper number of sequencer bits to delay the row's execution from it's nearest temporal and spatial companion. A shift register chain is used to implement the sequencer. Typically, a row's "onehot_source" is used to feed the input to the sequencer chain, and the row's "onehot_delay" defines how many sequencer bits to place in the chain for a particular module. The exception is for rows which fire on cycle offset 1, marked by a "onehot_delay" value of -1. The sequencer bits for these rows are preset (high) during datapath initialization, and they receive a "loop-back" from a sequencer bit scheduled at the end of the iteration interval.

3.5.2.2 Host I/O Access

The host I/O access is provided for initially setting the values of the live variables, and for reading out the live variables after the loop has completed. In addition, the ability to read or write to any row except memory access rows is useful when debugging. Memory access rows are not readable by the host since their tristate output is connected to ADDR_BUS, as opposed to MEM_BUS. Host access is decoded on the IO_ADDRESS bus, taken from bits 9-2 of the local bus, providing logical room for up to 256 separate rows. Due to physical arrangement, decode access is currently limited to 92 sites. This does not confine the datapath to 92 operators, however, since grammar is used to pack multiple operators into single modules. Multi-row modules will pack nicely, using all available "slices". A design with many single-row modules will leave slices unoccupied. This is due to the availability of tristate access resources in a given row. In order to allow write access to live variables, delay rows are actually multiplexors, accepting input either from the MEM_BUS or from another datapath row.

3.5.2.3 Memory and Halt Control

Logic is provided so that the tristate control lines are driven at the proper times. The following row types require this additional control logic:

- Exit Drives the HALT signal when condition is satisfied
- Pmem Condition from predicate is used to mask load/store enables
- Mem Drives LOAD_ENABLE or STORE_ENABLE
- Load Drives DP_BYTE or DP_SHORT when address is driven
- Store Drives DP_BYTE or DP_SHORT when address is driven

3.5.3 Module Generation

Currently, emit_xi_row() in the file "xi.c" emits tristate buffers in ".xnf" format for each module requiring them and determines inputs and control for each module type. Following this, the routine emit_xi_gen() in "xi.c" makes calls the appropriate generator with the necessary parameters (signed/unsigned, bitwidth, etc...) via command-line interface. Special handling is provided to determine control and data inputs to the generator modules. Data inputs may be paths (tree segments) which contain shifting and other masking type operations which may be implemented by routing in the FPGA. The grammar and the function emit_xi_input_pin() in "xi.c" collaborate to produce the module interconnect.

The following module generators are currently implemented for datapath width up to 32 bits:

add	Binary Adder *(+)*
comp	Compare function *(<, <=, >, >=, ==, !=)*
div	Binary Divider *(/)*
logic	Up to 4-input logic specified by table *(&, |, ~, ^, &&, ||, !, ^^, ?)*
mul	Multiplier _(*)_
mux	Two-to-one registered multiplexor *(live variables)*
neg	Negate unary operator *(-)*
reg	Register *(for memory address, load, store, inputs, patches)*
rem	Remainder *(%)*
shift	Shifter with variable shift count *(<<, >>)*
sub	Subtractor *(-)*
abs	Fixed point absolute value: *(a>0) ? a : -a*

bytesel	Friendly endian byte select: *a >> ((3-byte&3)*8)&255*
fir	FIR filter
parcnt	A 32-bit counter which skips over parity (LSb) bits of each byte
permute	General bit permute, set, and clear
ram	CLB-based RAM (not in external memory)
rom	CLB-based ROM
sbox	The DES sbox computation with “P” permute
sjg	The skipjack “G” function

3.5.4 Domain Generators

One of the easiest and rewarding extensions to Xima is to implement additional domain specific operators or blocks, which can be called out from "C". You simply create a circuit using Java/JHDL, integrate it with Xima, and then reference is as a function call with the "nimble_" prefix. See the Nimble Compiler Domain Generator Tutorial for information on incorporating and taking advantage of this powerful feature.

4 Summary

The Xima datapath compiler for Nimble has been described, including its derivation from Gama (Garp mapper), through its interim implementation supporting the Xilinx 4000 series as instantiated on the ACE 2 platform (gamax), to its final, delivered form (Xima) supporting the ACE-V (ACE board with ADM-XRC daughter card) platform. Important concepts relating the general method that Nimble employs to map code to reconfigurable hardware and provide necessary interfacing have been described. These include the construction of the datapath, the functionality implemented in the wrapper, the control approach used for the sequencer, and the general operation of the memory subsystem. Finally, the methods used to convert the data flow graph (AFL file) provided by the Nimble front end through a series of processes, resulting in a placed and annotated .xnf file used as input to the Xilinx tool for routing, optimization and bitfile generation, are described.

5 References

[1] Timothy J. Callahan and John Wawrzynek, Instruction Level Parallelism for Reconfigurable Computing, FPL'98, Tallinn, Estonia, September 1998. http://brass.cs.berkeley.edu/documents/fpl98.html

[2] John R. Hauser, The Garp Architecture http://brass.cs.berkeley.edu/documents/GarpArchitecture.html

[3] Randy Harr: A Nimble Compiler Environment for Agile Hardware,l DARPA ACS PI Meeting, Oct 2000, http://www.dyncorp-is.com/darpa/registration/agenda.asp?regCode=acs00oct

[4] Virtex 2.5V FPGAs (XCV00), http://www.xilinx.com/partinfo/ds003.pdf

[5] Alphadata Parallel Systems Ltd, ADM-XRC, Xilinx Reconfigurable Computing PMC Card, http://www.alphadata.co.uk/dsheet/adm-xrc.html

[6] TSI Telsys ACE Documentation – Installed on delivered nimblex machine in directory: /disks/local/ace/doc/manuals

[7] PLX PCI-9080 Data Book - http://www.plxtech.com/products/toolbox/9080.htm

Appendix H. Domain Generator Tutorial for the Nimble Compiler Project

1 Introduction

This tutorial documents the process of creating a new domain generator, integrating it, and using it within the Nimble Compiler framework. The ability to create and add custom domain generators to the Nimble Compiler allows the user provides a quick way to develop a digital circuit and then test its functionality by calling it out in a C program. Especially critical portions of a program can be custom implemented in hardware, and then simply invoked by a function call. The convention for function name is to append "nimble_" to the beginning of the generator name.

1.1 Background

For creating and integrating new domain generators, it is helpful to be generally familiar with the C and Java programming languages. Also, knowledge of Xilinx FPGAs (Virtex) and tools is required. The generators are written in JHDL, a Java-based description language for structural composition of the operators, so it might be necessary to review the JHDL tutorials and reference material available from "http://www.jhdl.org". It is, however, possible to write a domain generator in any language, as long as it conforms to the physical and calling interface standards which are detailed in this document.

1.2 Development Environment

JDK version 1.1.8 must be installed and referenced by the environment (system path, CLASSPATH, etc…). JDK 1.1.7 has also been tested. JDK 1.2 should work with little effort, but has not been tested. The JHDL toolkit must also be installed and referenced by the CLASSPATH environment variable. The generator library has been compiled with JHDL version 0.2.16, but has been tested with earlier and later versions. The required JHDL ships with the Nimble distribution. For simple unary domain operators, this is all that is required. For more complicated domain ops (unary with constant parameters, or binary), it will be necessary to modify Xima source code, so a C compiler is required. Finally, a recent (2.1i or later) version of the Xilinx tools is required.

1.3 Integration

The integration of a general domain op with the Nimble compiler is currently not an exact science. There are a couple of options available which make the integration process definable within the context of this tutorial. The focus is placed on the simpler unary and binary operators, and hopefully provides a good starting point for further investigation.

2 Developing the Generator

Currently, the generator must have a cell interface which is predefined, because the back end compiler simply calls out generator macros heuristically. The following table summarizes the cell interface which must be defined in the generator source code:

Pin Name	*Dir*	*Description*
a	in	The first operand input. The only operand input for unary ops.
b	in	The second operand input.
out	out	Output from the module.
ce	in	Clock enable for synchronous control.

In addition, a pin "clk" will be on the physical interface in the netlist. It is automatically inserted by JHDL, and does not need to be defined in the cell interface. The above list shows the signals required for the special domain generators. For the more general operator, there are a few more signals which could potentially be used (start, done, ci, co, etc…), but these are not considered in this tutorial.

2.1 The Java Source Files

There typically two java source files which go with each generator. The first serves as a JHDL "testbench", and the second is the actual circuit definition. The Java source files which comprise the module generators for the Nimble Compiler can be found in the *$NIMBLE_ENV/acesyn/gen* directory.

2.1.1 The Testbench File

The top level, or "testbench" file inherits from the GenLogic class which simplifies command-line processing, simulation, and netlisting. This file also instantiates the actual instance of the underlying circuit, as specified by the command-line parameters. The following is an example of the "testbench" file for the "abs" absolute value domain generator in the file named "abs.java":

```
/**
 * Absolute value generator top level
 * @see abs_inst
 * @author JCR
 * @see copyright.java
 */
package gen;
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import gen.tech;

/**
 * Absolute value module generator
 */
public class abs extends GenLogic implements TestBench {
  public static void main(String argv[]) {
    inputs=1;
    initialize("abs",argv);
    abs top = new abs(hw);
    finish();
  }

  public abs(Node parent) {
    super(parent);
    cell = new abs_inst(this, a, ce, out, cout ? co : null, reg, name);
  }
}
```

2.1.1.1 The Testbench "main" Routine

The following steps are taken in "main":

A. The assignment "inputs=1" is used to alter the number of inputs from its default value of two.

B. The call to "initialize" processes command line parameters and sets up netlisting and simulation.

C. The "abs" class is instantiated.

2.1.1.2 The Testbench Constructor

The testbench cunstructor simply calls super(parent) which constructs the superclass testbench, and then instantiates the cell, with a call to the abs_inst class constructor. Note that there are provisions for an optional carry-out port in this call. In this case, abs_inst will include a carry-out port if cout is true.

2.1.2 The Instance File

The instance file defines the cell interface and creates the logic contained in the module. An example of the instance for the absolute value generator, named "abs_inst" follows:

```
/**
 * Absolute value instance
 * @author JCR
 * @see copyright.java
 */
package gen;
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;

public class abs_inst extends Logic {
  public static CellInterface[] cell_interface = {
    in("a", "width"),
    out("out", "width"),
    in("ce",1),
    in("en",1),
    out("co",1),
    param("width", INTEGER)
  };
  public String name = "abs_cell";
  public String getCellName() { return(name); }
  public static int num_inst = 1;

/**
 * Absolute value
 * @param parent The parent node
 * @param a The "A" input
 * @param ce The clock enable
 * @param out The output
 * @param co Optional carry out
 * @param reg Flag indicating registering on output
 * @param name_in The name for the cell
 */
  public abs_inst(Node parent, Wire a, Wire ce,
             Wire out, Wire co, boolean reg, String name_in) {
    super(parent);
    boolean virtex = ((GenLogic)parent).part.equalsIgnoreCase("Virtex");

    // Get the width of Wire a so that the input ports a, b & out can be sized
    int width = a.getWidth();
    bind("width", width);

    // Update the cell name
    name = (name_in != null && name_in != "") ? name_in : name +"_"+num_inst++;

    // Connect the ports to output wires
    connect("a", a);
    connect("ce", ce);
    connect("out", out);
    if(co != null) { connect("co", co); }
    Wire out_nr = wire(width,"out_nr");
    Wire co_nr = wire(1,"co_nr");
    Wire zeros = wire(width,"zeros");
    Wire sign = a.getWire(width-1);
    for (int i=0; i<width; i++) gnd_o(zeros.getWire(i));
    addsub_o(zeros,a,sign,not(sign),out_nr,co_nr,"abs_jhdl");
    place(out_nr.getWire(0),0,(virtex ? 0:-1));

    if(reg) {
      regce_o(out_nr,ce,out, "abs_reg");
      if(co != null) regce_o(co_nr,ce,co,"abs_co_reg");
      for (int i=0; i<width; i++) place(out.gw(i) ,0,width/2-1-i/2);
    }
    else  {
      buf_o(out_nr, out);
      if (co != null) buf_o(co_nr, co);
    }
  }
}
```

2.1.2.1 The Instance Cell Interface

The cell interface, specified in the array of type CellInterface, defines the module ports which will appear in the resulting netlist. Even though it is not included in the cell interface a "CLK" port will also be included by JHDL. Typically, a cell interface will have an "a" input, possibly a "b" input, a "ce" clock enable input, and an "out" output. Also included in the cell interface is the "width" parameter, because it affects the width of the "a" and "out" ports. The "co" port is a carry output, and the "en" port is an unused port which is reserved for tristate enabling (even though this is not currently implemented in the generators).

2.1.2.2 Miscellaneous Declarations

The following three lines set up some general information required by the generator:

```
public String name = "abs_cell";
public String getCellName() { return(name); }
public static int num_inst = 1;
```

The name is used as the instance name, and is modidified if there is more than one instance used.

2.1.2.3 The Constructor

The constructor for abs_inst first invokes the superclass constructor, and then performs cell interface initialization. This includes binding the "width" parameter, and connecting the ports. Next, needed wires are created and the JHDL primitives are hooked up. In this case, we use JHDL's addsub module. The first input to the addsub module is connected to all grounds, and the second input is connected to the "A" input of the absolute value generator. The add/sub control is connected to the inversion of the input's sign bit. Optionally, a register is connected to the output, and some placement directives are provided.

2.1.3 Final Steps for the Generator

Once coding is complete, the generator files must be compiled, using "javac" on the command line. Simply typing: "javac abs.java" would compile both the testbench and the instance file. Once any errors are corrected, you can simulate the file using the following command line:

java gen.abs netlist=0 sim=1

The GenLogic.java file contains some default handling for simulation, so you should get some usefule results. For more custom simulation, you will have to overload the clock(), reset(), and other such methods. Examples of overloading the simulation methods can be found in more complicated generators such as "div.java".

Once simulation has verified functional operation, you will need to test the command line operation of the generator and verify its relative placement. To do this, in the case of the abs generator, type:

java gen.abs part=virtex pads=1

This will produce a netlist of the generator targetted toward the virtex part, and pads will be inserted so that when it is sent through place and route, the entire circuit won't be optimized away by the tools. Once this is completed successfully, type the following commands to put the netlist through place and route:

ngdbuild -p xcv1000-4-bg560 abs_32_s

map -p xcv1000-4-bg560 abs_32_s

par abs_32_s -w abs_32_s_routed

You can then analyze the design in fpga_editor (type '**fpga_editor abs_32_s_routed.ncd**") to ensure proper construction and placement.

3 Integrating the Generator to Xima

Adding more complicated generators is not an exact science, and requires extensive modifications to obscure portions of the Xima source code. This section describes the process of adding simple generators for which most of the infrastructure is already in place. These include binary operators, unary operators, and unary operators with a second constant parameter (such as permute with a permute table, or rom with rom contents).

For the special case of a unary operator with no parameters (like our "abs" example), there is no need to integrate the generator with Xima. It can be used immediately in Nimble programs, by defining a constant integer array whose elements contain the ASCII values for the generator name, like:

```
const int ABS[4] = {'a', 'b', 's',0};
```

The generator may then be referenced in a loop as follows:

```
a=0;
for (i=0, i<16; i++) { /* ks: yes */
  a += nimble_unary(i, ABS);
}
```

When implementing a generator for a binary operator, or a unary generator which is configured by constant parameters (like a ROM), it is currently required to modify Xima source code as described in the following text. Note that this requires that the Xima source code be available. Also note that the interface for adding such generators to Xima is currently unrefined.

3.1 Adding to "opcodes.h"

In the file "opcodes.h", you must add the opcode to the if_ops enumeration. For our example "abs" generator, you would add "io_abs" to the enumeration (but don't do this, since it's already there).

3.2 Modifications to "opcodes.c"

In the file "opcodes.c", you must add the opcode to the if_op_names[] array and to the afl_op_names[] array. In our example, we would add "nimble_abs" to the two (again, this is already in place). In the op_family[] array, a "fam_other" entry must be added in the position corresponding to the added opcode.

Next, if adding a unary operator, you will need to add another term to the is_nimble1() function. If adding a binary operator, add the corresponding term in the is_nimble2() function. For the "abs" example, the term o==io_abs is already in the equation in is_nimble1().

3.3 Modifications to "cost.c"

If the new operator only takes one clock cycle and one row, no modifications are needed to "cost.c". Otherwise, code must be added to the xil_nimble1() or xil_nimble2() function for unary or binary operators, respectively. Following the example of the multi-cycle or multi-row opcodes that are currently present is the easiest way to add new cost functionality.

4 Using the New Domain Generator

After creating the generator, testing it, and integrating it with Xima, the generator may be used in a "C" program and the Nimble Compiler. The generator is referenced in the "C" source by prefixing the generator name with "nimble_". So in the case of the abs generator, we would simply make a call to nimble_abs(). You must also provide an equivalent function of the same name, so that the software-only target will function. The following example illustrates a simple program which uses the "abs" generator:

```
/*
  abs test
*/
#include <stdio.h>

int nimble_abs(int i) { return(i<0 ? -i : i); }

int main(int argc, char *argv[])
{
  int i,j;

  j = 0;
  for (i=-8; i<2; i++) { /* ks:yes */
    j += nimble_abs(i);
  }
  printf("j=% \n",j);
}
```

Appendix I. Specification for FPGA Macro Generators for the Nimble Compiler Project

1 Introduction

These macro generator libraries were developed to support the full set of ANSI C integer operators. In addition, an initial set of domain specific generators are provided to facilitate acceleration of cryptographic and general computational functions. Additional API documentation is provided with the generator distribution.

1.1 Generator List and Operator Coverage

1.1.1 Generators Supporting Intrinsic Operators

add	Binary Adder *(+)*
comp	Compare function *(<, <=, >, >=, ==, !=)*
div	Binary Divider *(/)*
logic	Up to 4-input logic specified by table *(&, |, ~, ^, &&, ||, !, ^^, ?)*
mul	Multiplier (*)
mux	Two-to-one registered multiplexor *(live variables)*
neg	Negate unary operator *(-)*
reg	Register *(for memory address, load, store, inputs, patches)*
rem	Remainder *(%)*
shift	Shifter with variable shift count *(<<, >>)*
sub	Subtractor *(-)*

1.1.2 Generators Supporting Domain Specific Functions

abs	Fixed point absolute value: *(a>0) ? a : -a*
bytesel	Friendly endian byte select: a >> ((3-byte&3)*8)&255
fir	FIR filter
parcnt	A 32-bit counter which skips over parity (LSb) bits of each byte
permute	General bit permute, set, and clear
ram	CLB-based RAM (not in external memory)
rom	CLB-based ROM
sbox	The DES sbox computation with “P” permute
sjg	The skipjack “G” function

1.2 *Intrinsic Operator to Generator Mapping*

1.2.1 Arithmetic

Op	Description	Generator
+	Unary plus	none
-	Unary minus	neg
++	Unary increment	add
—	Unary decrement	sub
+	Binary add	add
-	Binary subtract	sub
*	Binary multiply	mul
/	Binary division	div
%	Binary modulus	rem

1.2.2 Relational

Op	Description	Generator
==	Binary Equal	comp type=e
!=	Binary Not equal	comp type=ne
>	Binary Greater than	comp type=g
<	Binary Less than	comp type=l
>=	Binary Greater than or equal	comp type=ge
<=	Binary Less than or equal	comp type=le

1.2.3 Bitwise

Op	Description	Generator
~	Unary complement (invert)	logic
&	Binary And	logic
|	Binary Or	logic
^	Binary Xor	logic
<<	Binary Left shift	logic
>>	Binary Right shift	logic
>>>	Binary Right shift with zero fill	logic

1.2.4 Logical

Op	Description	Generator
!	Unary not	logic
&&	Binary and	logic
||	Binary or	logic

^^	Binary xor	logic

1.2.5 Control Flow

Op	Description	Generator
?	ternary operator	logic

1.3 Invoking the Generators from the Command Line

The generators have a simple calling interface, which is introduced with the following example:

```
  java gen.comp name=testcomp type=e sign=0 reg=1 width=16 sim=1
part=virtex
```

The preceding line will produce an "equals" compare function which is unsigned (irrelevant), registered, 16-bits wide, and targeted toward a Virtex part. Note that there are no spaces within a given parameter assignment. In this case, an EDIF netlist of the cell "testcomp" would be generated in a file named "testcomp.edn".

In general, parameters all have sensible default values, so one could just enter the following and still get valid results (a netlisting of a registered 32-bit equals comparitor named "comp_32_e_s"):

```
  java gen.comp
```

The following parameter assignments are common to all generators:

Parameter	default	effect
Width=n	32	bitwidth of the generator
reg=0|1	1	turns output registering on/off
Sign=0|1	1	invokes unsigned/signed version of generator
Netlist=0|1	1	turns netlisting on/off
Pads=0|1	0	turns on insertion of pads for lone P&R
sim=0|1	0	turns simulation on/off
Part=virtex|XC4000	XC4000	chooses the target part
Name=string	auto	sets the name of the cell and netlist file
Disp=dec|hex|bin	hex	sets the display format for simulation output
Verbose=0|1	0	Prints out useful info to stdout

The word "true" can usually be substituted for "1", and "false" for "0" above.

Some parameters have special meanings for specific generators:

Generator	parameter	default	meaning
Comp	type=e|ne|l|le|g|ge	e	type of comparitor
Logic	logic=n	6 (xor)	Decimal int for logic table (LUT)
Logic	inputs=n	2	Number of inputs
Mul	stages=n	1	Number of multiplier stages per clock
Permute	table=n,n,n,...	-2,-2,...	In bit wired to out. -1=VCC -2=GND
Rom	init=n,n,n,...	0,0,...	Contents of ROM
Shift	type=l|r	l	Direction of shift

1.4 Generator Cell Interface

Invoking the generators from the command line results in an EDIF netlist for the module, if netlisting is not specifically disabled. The module's cell interface follows a convention which is common across all generators, unless otherwise noted. This makes it easy for a higher level application to instantiate and connect the modules. The standard cell interface is as follows:

Name	*Dir*	*Description*
a,b,d,e	in	Up to four main inputs to the module
Ctrl	in	A control input (for variable shift and mux only)
Out	out	Output from the module
Clk	in	Clock input providing for pipelined operation
Ce	in	Clock enable for synchronous control
Start	in	A pulse which begins operation of synchronous modules
Done	out	A pulse indicating module completion (output is valid)
Ci	in	Optional carry input for adder-type modules
Co	out	Optional carry output for adder-type modules

Unary operator modules, such as "abs" or "neg", would only have one input, namely "a". Binary operator modules, such as "add" or "sub", would have both "a" and "b" inputs. Currently, only the logic modules might use more than two main inputs. Only modules which are sequential (ie, take more than one cycle to complete) have the "start" and "done" signals.

1.5 Generator Area/Timing Information

Limited generator area and timing information can be obtained. The "gen.tech" class can be used to print out or programmatically obtain timing/area info for a particular generator. The parameter syntax is the same as the generator command line invocation, except that the first parameter specifies the generator.

```
  java gen.tech add width=16
```

The above example would produce a timing report for a 16-bit add generator.

In addition, if you are interfacing directly to the generator classes, each generator class supports three cost reporting routines:

```
public static int getRows(String s)     // datapath rows used
public static int getColumns(String s)  // datapath columns used (out
of 16)
public static int getCycles(String s)   // cycles for sequential ops
public static double getDelay(String s) // combinational delay or min
period
```

2 Intrinsic Operator Specification

2.1 Generator: add

Binary Adder

2.1.1 Command-line Generation:

java gen.add name=*string* width=*int* reg=*bool*

2.1.2 Instantiation:

add_inst(Node parent, boolean plus, Wire a, Wire b, Wire ci, Wire ce, Wire out, Wire co, boolean req, String name)

Pass in null for "ci" or "co" to bypass.

2.1.3 Operation:

out = a + b

The two input operands are added together.

2.1.4 Implementation:

Binary adder cells are used, with dedicated carry logic.

2.1.5 Special Options:

None.

2.1.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[a_width]	R	in	a_width specifies bit width	Input to adder
b[b_width]	R	in	b_width specifies bit width	Input to adder
out[width]	R	out	width specifies bit width	Output
clk	O	in	Present if registered	Clock input
ce	O	in	Present if registered	Clock enable
ci	O	in	Direct instantiation only	Carry input
co	O	out	Direct instantiation only	Carry output

2.2 Generator: comp

Integer compare functions

2.2.1 Command-line Generation:

java gen.compare name=*string* width=*int* type=*string* reg=*bool*

2.2.2 Instantiation:

comp_inst(Node parent, Wire a, Wire b, Wire ce, Wire out, String type, boolean signed,

boolean req, String name)

2.2.3 Operation:

```
For cell type "e":    out = (a==b)
For cell type "ne":   out = (a!=b)
For cell type "g":    out = (a>b)
For cell type "ge":   out = (a>=b)
For cell type "l":    out = (a<b)
For cell type "le":   out = (a<=b)
```

Outputs a logic high if the selected comparison of A and B is true, or logic low otherwise.

2.2.4 Implementation:

The "e" and "ne" cell types compare four bit-pairs per CLB, skipping alternate CLBs, radix-4 ANDing up the results in central locations.

The "g", "ge", "l", and "le" cell types use a subtractor. Inputs are swapped for "l" and "le" compares. The active-low borrow is used for "g" and "l" compares. Signed compares have special logic for handling the most significant bit.

2.2.5 Special Options:

1. type=*string*

   Specifies the type of compare function:

   "e" - Equal "ne" - Not equal

   "g" - Greater than "ge" - Greater than or equal

   "l" - Less than "le" - Less than or equal

2.2.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[width_a]	R	in	width_A controls bit width	Input A into comparator
b[width_b]	R	in	width_B controls bit width	Input B into comparator
out[width]	O	out	width controls bit width	Compare output
clk	O	in	Present if registered	Clock input
ce	O	in	Present if registered	Clock enable

2.3 Generator: div

Binary Divider

2.3.1 Command-line Generation:

java gen.div name=*string* width=*int* signed=*bool*

2.3.2 Instantiation:

div_inst(Node parent, Wire a, Wire b, Wire start, Wire ce, Wire out, Wire done,

Wire zero, boolean signed, String name)

2.3.3 Operation:

out = a/b

A is divided by B to produce result. Divide-by-zero is produced if required.

"DONE" flag is cleared on issue of "START" and is set upon divide completion.

2.3.4 Implementation:

An iterative add/subtract is performed on the remainder and divisor, based on the state of the remainder. See "div_inst.java" source code for implementation details.

2.3.5 Special Options:

None.

2.3.6 Interface:

Name	*Req/ Opt*	*Dir*	*Variability*	*Description*
a[width_a]	R	in	width_a sets bit-width	Dividend
b[width_b]	R	in	width_b sets bit-width	Divisor
out[width_out]	R	out	width_out sets bit-width	Result from divide
zero	O	out	options sets presence	Divide-by-zero indicator
start	R	in	NA	Triggers start of divide
done	R	out	NA	Indicates divide complete
clk	R	in	NA	Clock input
ce	R	in	NA	Clock enable

2.4 Generator: logic

Up to four-input logic evaluation specified by table.

2.4.1 Command-line Generation:

java gen.logic width=*int* inputs=*int* logic=*int*

2.4.2 Instantiation:

logic_inst(Node parent, Wire a, Wire b, Wire d, Wire e, Wire ce, Wire out,

long table[], int num_inputs, boolean reg, String name)

2.4.3 Operation:

```
for (out=0, i=0; i<width; i++)
  out |= ((logic>>((a>>i)&1 + (b>>i<<1)&2 + (d>>i<<2)&4 +
(e>>i<<3)&8))&1)<<i;
```

2.4.4 Implementation:

A Lookup table is composed of the input logic number, and addressed by concat(e,d,b,a) for each bit.

2.4.5 Special Options:

1. logic=*int*

The bits of the specified decimal integer specify the contents of the lookup table implementing the logic function.

2. inputs=int

Specifies the number of inputs into the logic function (minimum 1, maximum 4)

2.4.6 Interface:

Name	*R/O*	*Dir*	*Variability*	*Description*
a[width]	R	in	width specifies replication	Input to the logic block
b[width]	O	in	width specifies replication	Input to the logic block
d[width]	O	in	width specifies replication	Input to the logic block
e[width]	O	in	width specifies replication	Input to the logic block
out[width]	R	out	width is bit width of output	Output from the logic block
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable

2.5 Generator: mul

Binary Multiplier

2.5.1 Command-line Generation:

java gen.mul width=*int* stages=*int*

2.5.2 Operation:

out = a * b

Operands A and B are multiplied together to produce result.

2.5.3 Implementation:

A Booth encoding algorithm is used, processing two bits per stage per cycle. See "mult_inst.java" source code for details.

2.5.4 Special Options:

1. stages=*int*

   The number of shift/add stages included, each processing two bits.

2.5.5 Interface:

Name	***R/O***	***Type***	***Variability***	***Description***
a[width_a]	R	in	*width_a* sets bit-width	Multiplicand
b[width_b]	R	in	*width_b* sets bit-width	Multiplier
out[width]	R	out	*width* sets bit-width	Result from multiplier.
Start	R	in	NA	Starts the multiplication sequence
Done	O	out	NA	Indicates completion of multiplication
Clk	O	in	NA	Clock
Ce	O	in	NA	Clock

2.6 Generator: mux

Two-to-One Multiplexor

2.6.1 Command-line Generation:

java gen.mux width=*int* reg=*bool*

2.6.2 Instantiation:

```
mux_inst(Node parent, Wire a, Wire b, Wire ctrl, Wire ce, Wire out,
        boolean reg, String name)
```

2.6.3 Operation:

out = ctrl ? b : a

If the single-wire "ctrl" signal is high, the wide output gets "b", otherwise, "a".

2.6.4 Implementation:

The mux is implemented with the following logic: a & ~ctrl | b & ctrl

2.6.5 Special Options:

None.

2.6.6 Interface:

Name	***R/O***	***Dir***	***Variability***	***Description***
a[width]	R	in	width specifies bitwidth	Input selected when ctrl is low
b[width]	R	in	width specifies bitwidth	Input selected when ctrl is high
Ctrl	R	in	none	Control input
out[width]	R	out	width specifies bitwidth of output	Output from the multiplexor
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock input

2.7 Generator: neg

Twos-Complement negator

2.7.1 Generation:

java gen.neg name=***string*** width=***int*** reg=***bool***

2.7.2 Instantiation:

neg_inst(Node parent, Wire a, Wire ce, Wire out, Wire co, boolean req, String name)

2.7.3 Operation:

out = -a

2.7.4 Implementation:

The input, "a", is fed to the b-input of a subtractor module with its a-input grounded.

2.7.5 Special Options:

None.

2.7.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[width]	R	in	width specifies bit width	Input to neg
out[width]	R	out	width specifies bit width	Output from neg
CLK	O	in	Present if reg=true	Clock input
CE	O	in	Present if reg=true	Clock enable
CO	O	out	Direct instantiation only	Carry output

2.8 Generator: reg

Simple register

2.8.1 Generation:

java gen.reg name=***string*** width=***int***

2.8.2 Instantiation:

reg_inst(Node parent, Wire a, Wire ce, Wire out, boolean req, String name)

2.8.3 Operation:

out = a

2.8.4 Implementation:

The input is fed to registers with clock enable inputs.

2.8.5 Special Options:

None.

2.8.6 Interface:

Name	***Req/Opt***	***Dir***	***Variability***	***Description***
a[width]	R	in	width specifies bit width	Input to reg
out[width]	R	out	width specifies bit width	Output from reg
CLK	O	in	Present if reg=true	Clock input
CE	O	in	Present if reg=true	Clock enable

2.9 Generator: rem

Integer remainder function (actually, "%" in C)

2.9.1 Command-line Generation:

java gen.rem name=*string* width=*int* signed=*bool*

2.9.2 Instantiation:

mod_inst(Node parent, Wire a, Wire b, Wire start, Wire ce, Wire out, Wire done,
    Wire zero, boolean signed, String name)

2.9.3 Operation:

out = a%b

A is divided by B and the remainder becomes the result. Divide-by-zero is produced if required. "DONE" flag is cleared on issue of "START" and is set upon divide completion.

2.9.4 Implementation:

A divide operation is performed (see "div") and the remainder is used as the result.

2.9.5 Special Options:

None.

2.9.6 Interface:

Name	***Req/ Opt***	***Dir***	***Variability***	***Description***
a[width_a]	R	in	width_a sets bit-width	Dividend
b[width_b]	R	in	width_b sets bit-width	Divisor
out[width_out]	R	out	width_out sets bit-width	Result from rem
Zero	O	out	options sets presence	Divide-by-zero indicator
Start	R	in	NA	Triggers start
Done	R	out	NA	Indicates rem complete
Clk	R	in	NA	Clock input
Ce	R	in	NA	Clock enable

2.10 Generator: shift

Shifter with variable shift count

2.10.1 Command-line Generation:

java gen.shift type=*string* width=*int* sign=*bool*

2.10.2 Instantiation:

shift_inst(Note parent, Wire a, Wire ctrl, Wire ce, Wire out, Sting type,
boolean signed, boolean reg, String name)

2.10.3 Operation:

For cell type "l": out = a << ctrl

For cell type "r": out = a >> ctrl

For right shifts, sign extension is performed if "sign" is true.

2.10.4 Implementation:

The barrel shifter is produced with stacked selectable shift-by-1, shift-by-2, shift-by-4, shift-by-8, and shift-by-16. The order is mixed to optimize timing (1,16,2,8,4 for a 32-bit shifter).

2.10.5 Special Options:

```
1.   type=string
     Selects the type of shifter ("l" for left, "r" for right)
```

2.10.6 Interface:

Name	*R/O*	*Dir*	*Variability*	*Description*
a[width]	R	in	width specifies bitwidth	Input to shifter
Ctrl	R	in	log2(width) is bitwidth	Shift amount
out[width]	R	out	width_out specifies bit width	output from shifter
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable

2.11 Generator: sub

Binary subtractor.

2.11.1 Command-line Generation:

java gen.sub name=*string* width=*int* reg=*bool*

2.11.2 Instantiation:

sub_inst(Node parent, boolean plus, Wire a, Wire b, Wire ci, Wire ce, Wire out,
Wire co, boolean req, String name)

Pass in boolean false for "plus" to indicate subtraction. Pass in null for "ci" or "co" to bypass.

2.11.3 Operation:

out = a - b

"b" is subtracted from "a".

2.11.4 Implementation:

Binary subtract cells are used, with dedicated carry logic.

2.11.5 Special Options:

None.

2.11.6 Interface:

Name	***Req/Opt***	***Dir***	***Variability***	***Description***
a[a_width]	R	in	a_width specifies bit width	Input to sub
b[b_width]	R	in	b_width specifies bit width	Input to sub
out[width]	R	out	width specifies bit width	Output
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable
Ci	O	in	Direct instantiation only	Carry input
Co	O	out	Direct instantiation only	Carry output

3 Group 2 Domain Specific Functions

For the domain generators, an equivalent C function is provided where applicable for behavioral/source modeling.

3.1 Generator: abs

Absolute value

3.1.1 Generation:

java gen.abs name=***string*** width=***int*** reg=***bool***

3.1.2 Instantiation:

abs_inst(Node parent, Wire a, Wire ce, Wire out, Wire co, boolean req, String name)

3.1.3 Operation:

int nimble_abs(int a) { return(a<0 ? -a : a); }

If the sign bit of "a" is set, then it is negated (twos complemented).

3.1.4 Implementation:

The input, "a", is fed to the a-input of an adder/subtractor module with its b-input grounded. The add/sub control (add if high) is tied to the inversion of the sign bit.

3.1.5 Special Options:

None.

3.1.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[width]	R	in	width specifies bit width	Input to ABS
out[width]	R	out	width specifies bit width	Output from ABS
CLK	O	in	Present if reg=true	Clock input
CE	O	in	Present if reg=true	Clock enable
CO	O	out	Direct instantiation only	Carry output

3.2 Generator: bytesel

Byte Select

3.2.1 Command-line Generation:

java gen.bytesel name=***string*** width=***int*** reg=***bool***

3.2.2 Instantiation:

bytesel_inst(Node parent, Wire a, Wire ce, Wire out, boolean req, String name)

3.2.3 Operation:

unsigned int nimble_bytesel(unsigned int a, int b) { return(a<<b*8>>24); }

Return the indicated byte of the input (MSB is byte 0, LSB is byte 3).

3.2.4 Implementation:

The input is fed to selectable shift-by-16 and shift-by-8 stages. The shift-by-16 is enabled if bit-1 of "b" is high. The shift-by-8 is active if bit-0 of "b" is high.

3.2.5 Special Options:

None.

3.2.6 Interface:

Name	***Req/Opt***	***Dir***	***Variability***	***Description***
a[width]	R	in	width specifies bit width	Input to bytesel
out[width]	R	out	width specifies bit width	Output
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable

3.3 Generator: fir

Finite Impulse Response filter

3.3.1 Command-line Generation:

java gen.fir name=*string* a_width=*int* coef_width=*int* taps=*int* init=*int*,*int*,...

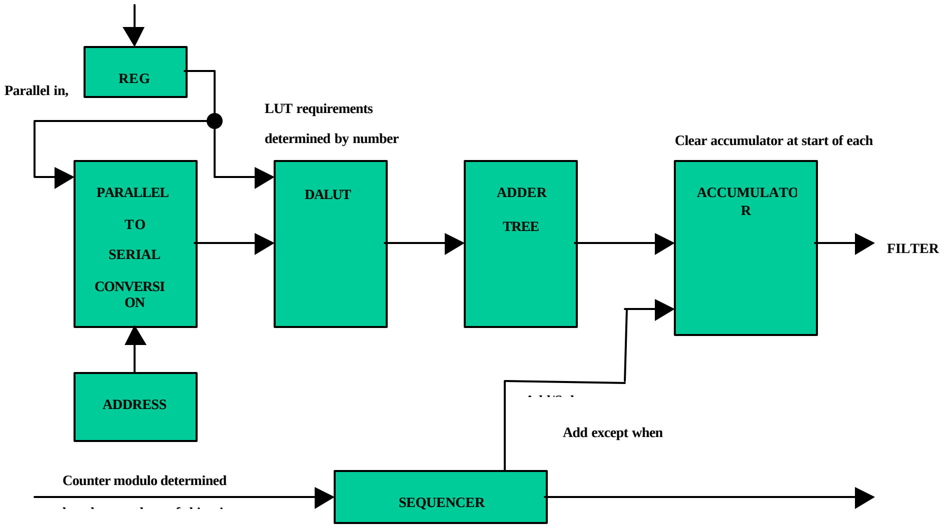
3.3.2 Instantiation:

```
fir_inst(Node parent, Wire a, Wire ce, Wire seq_clr, Wire out, Wire done,
         int num_taps, int coef_width, long init[], String name)
```

3.3.3 Operation:

Performs and FIR filter function, performing a weighted sum on a queue containing current a previous input samples.

3.3.4 Implementation:



This uses a distributed arithmetic approach for the FIR:

3.3.5 Special Options:

1. init=*int*,*int*,... Specifies filter taps values
2. coef_width Specifies the coefficient width
3. taps Specifies the number of taps

3.3.6 Interface:

Name	***Req/Opt***	***Dir***	***Variability***	***Description***
a[a_width]	R	in	a_width specifies bitwidth	Data input
out[width]	R	out	width specifies bit width	Output of FIR
Clk	O	in	NA	Clock input
Ce	O	in	NA	Clock enable
seq_clr	O	in	NA	Sequencer clear
Firdone	O	in	NA	Indicates complete

3.4 Generator: parcnt

A 32-bit counter which skips the parity bits (the LSb) of each byte.

3.4.1 Generation:

java gen.parcnt name=***string*** width=***int*** reg=***bool***

3.4.2 Instantiation:

parcnt_inst(Node parent, Wire a, Wire ce, Wire out, boolean req, String name)

3.4.3 Operation:

```
unsigned int nimble_parcnt(unsigned int a)
{
  a = (a>>1)&0x7f | (a>>2)&0x3fc0 | (a>>3)&0x1fe000 |
(a>>4)&0x0ff00000;
  a++;
  a = (a<<1)&0xfe | (a<<2)&0xfe00 | (a<<3)&0xfe0000 |
(a<<4)&0xfe000000;
  return(a);
}
```

3.4.4 Implementation:

Bits 31-25, 23-17, 15-9, 7-1 are fed to an incrementor. The result is padded back up to 32 bits.

3.4.5 Special Options:

None.

3.4.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[width]	R	in	width specifies bit width	Input to counter
out[width]	R	out	width specifies bit width	Output from counter
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable

3.5 Generator: permute

General bit position changer with set and clear masks

3.5.1 Command-line Generation:

java gen.permute name=*string* width=*int* reg=*bool* table=int,int,int,...

3.5.2 Instantiation:

permute_inst(Node parent, Wire a, Wire ce, Wire out, int[] table, boolean req,
String name)

3.5.3 Operation:

```
unsigned int nimble_permute(unsigned int a, const int *tab)
{
  unsigned int i, r=0;

  for (i=0; i<32; i++)
    r |= ((tab[i]>=0 ? ((a>>tab[i])&1) : tab[i]==-1 ? 1 : 0)<<i);
  return(r);
}
```

The indexed output bit gets the input bit corresponding to the indexed table entry. An table entry of –1 routes VCC to the indexed output bit and –2 routes GND. Index "0" corresponds to the least significant bit.

3.5.4 Implementation:

FPGA routing resources are used. Power and grounds are made in CLBs.

3.5.5 Special Options:

1. table=*int,int,int*... or type=*int,int,int*...

   Specifies the permute table using a comma separated list of decimal integers.

3.5.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[width_a]	R	in	a_width specifies input width	Input to permute
out[width]	R	out	width specifies output width	Output from permute
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock enable

3.6 Generator: ram

Random Access Memory

3.6.1 Command-line Generation:

java gen.ram name=*string* width=*int* depth=*int* init=*int*,*int*,...

3.6.2 Instantiation:

ram_inst(Node parent, Wire a, int addr_width, Wire we, Wire addr, Wire ce,

Wire out, long init[], String name)

3.6.3 Operation:

Performs as a Random Access Memory.

3.6.4 Implementation:

This RAM is implemented in CLBs. Muxing is provided for rams deeper than 32.

3.6.5 Special Options:

1. init=*int*,*int*,...

   Specifies initial contents of the RAM.

3.6.6 Interface:

Name	***Req/ Opt***	***Dir***	***Variability***	***Description***
a[width]	R	in	width specifies bit width	Data input
We	R	in	NA	Write Enable
addr[addr_width]	R	in	Bitwidth is log2(depth)	Address
out[width]	R	out	width specifies bit width	Output from RAM
Clk	O	in	Present if registered	Clock input
Ce	O	in	NA	Clock enable

3.7 Generator: rom

Read Only Memory

3.7.1 Command-line Generation:

java gen.rom name=*string* width=*int* depth=*int* init=*int*,*int*,...

3.7.2 Instantiation:

rom_inst(Node parent, Wire a, int addr_width, Wire ce, Wire out, long init[],
String name)

3.7.3 Operation:

```
int nimble_rom(int addr, const int *init) { return(init[addr]); }
```

3.7.4 Implementation:

This ROM is implemented in CLBs with multiplexing provided for depths greater than 32.

3.7.5 Special Options:

1. init=*int*,*int*,...

   Specifies contents of the ROM.

3.7.6 Interface:

Name	*Req/Opt*	*Dir*	*Variability*	*Description*
a[a_width]	R	in	a_width specifies bitwidth	Address input
out[width]	R	out	width specifies bit width	Output from RAM
Clk	O	in	NA	Clock input
Ce	O	in	NA	Clock enable

3.8 Generator: sbox

The S-Box computation for the DES cryptographic function

3.8.1 Command-line Generation:

java gen.sbox name=*string* reg=*bool*

3.8.2 Instantiation:

sbox_inst(Node parent, Wire a, Wire b, Wire ce, Wire out, boolean reg, String name)

3.8.3 Operation:

```
unsigned int nimble_sbox(unsigned int a, unsigned int b)
{
  int ret=0, box, lr, i;
  const int P[32] = {
    7,28,21,10,26,2,19,13,23,29,5,0,18,8,24,30,22,1,14,27,6,9,17,31,15,4,20,3,11,12,25,16
  };

  int S[8][64] = {
    {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
     4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13},
    {15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10,3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5,
     0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15,13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9},
    {10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8,13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1,
     13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7,1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12},
    {7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15,13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9,
     10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4,3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14},
    {2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9,14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6,
     4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14,11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3},
    {12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11,10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8,
     9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6,4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13},
    {4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1,13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6,
     1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2,6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12},
    {13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7,1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2,
      7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8,2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11}
  };

  for (box=1; box<9; box++) {
    lr = ((box<5 ? a:b) >> (3-((box-1)%4))*6)&0x3f;
    i = lr&32 | (lr>>1)&15 | (lr<<4)&16;
    ret |= (S[box-1][i] << ((8-box)*4));
  }
  return(nimble_permute(ret,P));     /* do the "P" permute */
}
```

3.8.4 Implementation:

The sbox function is composed of ROMs and routing. 32 bits of output are formed by feeding 6 bits at a time (from l to r) into s1 through s8.

3.8.5 Special Options:

None.

3.8.6 Interface:

Name	*R/O*	*Dir*	*Variability*	*Description*
A	R	in	NA	The left (MSB) 24 bits of the 48-bit SBOX input word
B	R	in	NA	The right (LSB) 24 bits of the 48-bit SBOX input word
out[32]	R	out	NA	Output
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock input

3.9 Generator: sjg

The "G" function for the Skipjack cryptographic algorithm

3.9.1 Command-line Generation:

java gen.sjg name=*string* reg=*bool*

3.9.2 Instantiation:

sjg_inst(Node parent, Wire a, Wire b, Wire ce, Wire out, boolean reg, String name)

3.9.3 Operation:

```
unsigned int nimble_sjg(unsigned int a, unsigned int b)
{
  const unsigned char f[256] = {
    0xa3,0xd7,0x09,0x83,0xf8,0x48,0xf6,0xf4,0xb3,0x21,0x15,0x78,0x99,0xb1,0xaf,0xf9,
    0xe7,0x2d,0x4d,0x8a,0xce,0x4c,0xca,0x2e,0x52,0x95,0xd9,0x1e,0x4e,0x38,0x44,0x28,
    0x0a,0xdf,0x02,0xa0,0x17,0xf1,0x60,0x68,0x12,0xb7,0x7a,0xc3,0xe9,0xfa,0x3d,0x53,
    0x96,0x84,0x6b,0xba,0xf2,0x63,0x9a,0x19,0x7c,0xae,0xe5,0xf5,0xf7,0x16,0x6a,0xa2,
    0x39,0xb6,0x7b,0x0f,0xc1,0x93,0x81,0x1b,0xee,0xb4,0x1a,0xea,0xd0,0x91,0x2f,0xb8,
    0x55,0xb9,0xda,0x85,0x3f,0x41,0xbf,0xe0,0x5a,0x58,0x80,0x5f,0x66,0x0b,0xd8,0x90,
    0x35,0xd5,0xc0,0xa7,0x33,0x06,0x65,0x69,0x45,0x00,0x94,0x56,0x6d,0x98,0x9b,0x76,
    0x97,0xfc,0xb2,0xc2,0xb0,0xfe,0xdb,0x20,0xe1,0xeb,0xd6,0xe4,0xdd,0x47,0x4a,0x1d,
    0x42,0xed,0x9e,0x6e,0x49,0x3c,0xcd,0x43,0x27,0xd2,0x07,0xd4,0xde,0xc7,0x67,0x18,
    0x89,0xcb,0x30,0x1f,0x8d,0xc6,0x8f,0xaa,0xc8,0x74,0xdc,0xc9,0x5d,0x5c,0x31,0xa4,
    0x70,0x88,0x61,0x2c,0x9f,0x0d,0x2b,0x87,0x50,0x82,0x54,0x64,0x26,0x7d,0x03,0x40,
    0x34,0x4b,0x1c,0x73,0xd1,0xc4,0xfd,0x3b,0xcc,0xfb,0x7f,0xab,0xe6,0x3e,0x5b,0xa5,
    0xad,0x04,0x23,0x9c,0x14,0x51,0x22,0xf0,0x29,0x79,0x71,0x7e,0xff,0x8c,0x0e,0xe2,
    0x0c,0xef,0xbc,0x72,0x75,0x6f,0x37,0xa1,0xec,0xd3,0x8e,0x62,0x8b,0x86,0x10,0xe8,
    0x08,0x77,0x11,0xbe,0x92,0x4f,0x24,0xc5,0x32,0x36,0x9d,0xcf,0xf3,0xa6,0xbb,0xac,
    0x5e,0x6c,0xa9,0x13,0x57,0x25,0xb5,0xe3,0xbd,0xa8,0x3a,0x01,0x05,0x59,0x2a,0x46
  };
  unsigned int g1,g2;

  g1 = (a>>8) ^ f[(a  ^ (b>>24))&255];
  g2 = a&255 ^  f[(g1 ^ (b>>16))&255];
  g1 ^=         f[(g2 ^ (b>>8)) &255];
  g2 ^=         f[(g1 ^  b)     &255];
  return( (g1<<8)| g2 );
}
```

3.9.4 Implementation:

The "G" function is composed of ROMs, XORs, and routing.

3.9.5 Special Options:

None.

3.9.6 Interface:

Name	*R/O*	*Dir*	*Variability*	*Description*
A	R	in	NA	The "w1" input to the G function
B	R	in	NA	The 4 bytes of key to be used in the G function
out[32]	R	out	NA	Output
Clk	O	in	Present if registered	Clock input
Ce	O	in	Present if registered	Clock input

Appendix J. Final Technical Deliverables Status

In the section, we review the status of deliverables listed in the Nimble project agreement. We list each deliverable (in *italic*) followed by a summary of its status.

1. *Study of FPGA co-processor implementation techniques (4QFY98 Draft, 4QFY99 Final) (Task 1.2) (NSC)*

Finished. Studied and compared multiple Agileware platforms. The final implementation of Nimble supports Garp, ACEII, and ACEV platforms. Detailed documentation regarding the ACEV platform is included in Appendices A and B.

2a. Library of Java based generator functions released and made available on the WWW (Intrinsic operator elements: 3QFY99, Domain specific elements: 3QFY00) (Task 5.3) (UCBerkeley)

Finished. Java-based (JHDL, a Java-base hardware description language) generator functions are implemented and integrated into the Nimble environment. It is included in the Nimble Phase 1+ release as well as the final release. The related documents (Appendix H and Appendix I) can be made available on the WWW once submitted to the government. Also refer to Section 2.5 of this report and the two supporting documents Appendix H and Appendix I for technical discussions on this item.

2b. Specification of a vendor neutral API to the function generator library (3QFY98 Draft, 2QFY99 Preliminary, 1QFY00 Final) (Task 1.3) (UCBerkeley)

Finished. Refer to Section 2.5 of this report, and the supporting document Appendix H.

2c. List of base library generator functions. (1QFY99 Draft, 3QFY99 Preliminary, 2QFY00 Final) (Task 1.4) (NSC)

Finished. Refer to Section 2.5 of this report. List of the base library generator functions is also reported in Appendix I.

3a. [D4] Operator library implementation of base library functions on Xilinx 4000 family device. (2QFY99) (Task 5.2) (NSC)

Finished. Java-based (JHDL) generator functions are implemented and integrated into the Nimble environment. The final target reconfigurable datapath is the more powerful Xilinx Vertex 1000 family of devices. Refer to Section 2.5 of this report, and two supporting documents Appendix H and Appendix I.

3b. D4 language specification and user guide published and available openly. (1QFY99) (Task 5.1) (NSC)

Finished. Final document is included as a supporting document to this report. See Appendix L in Vol II.

4. Specification of Agileware Architecture Definition Language (1QFY99) (Task 3.2) (Synopsys)

Finished. Final document is included as a supporting document to this report. See Appendix K in Vol II.

4b. Update to specification of Agileware Architecture Definition Language and initial code processing of the language (3QFY99) (Task 3.2) (Synopsys)

Finished. See item 5.

4c. Update to specification of Agileware Architecture Definition Language and cost function inclusion into the processing (2QFY00) (Task 3.2) (Synopsys)

Finished. See item 5.

6. Language style guide for the Nimble Compiler (1QFY99 baseline, 3QFY99 w/ parallel, 1QFY00 w/ domain) (Task 1.5) (Synopsys)

Finished. The final document is included as Appendix L in Vol II.

6. Demonstration of the initial (phase 1) new tool environment on an ACS architecture with at least one application from the cryptography algorithm set. Featuring tentatively the user pragma specification of enhancements, target optimizations of code, basic FPGA datapath component assembly with manual optimal placement. Up to five (5) unsupported, binary licenses of tool set to be made available to ACS research program participants (universities and potential customers) (4QFY99) (Task 2 - 6) (Synopsys)

Finished. Demonstrated Nimble on cryptography algorithms including several skipjack implementations and the DES algorithm. Results are reported in the final benchmark report (Appendix M in Vol II). Binary licenses along with the actual hardware have been shipped to several research institutes. Tutorials were given at the October 2000 PI meeting.

7. Demonstration of a number of higher level signal and image processing function generator library elements that require complete datapath construction (for example: FIR filters, I/DCT, and Hough transforms). Results available on the WWW. (4QFY00) (Task 6) (UCBerkeley)

Finished. Demonstrated Nimble on a number of signal and image processing applications from different sources (MediaBench, ACS benchmarks, and SPEC benchmarks). Results are reported in the final benchmark report (Appendix M in Vol II). Results can be published on the WWW once submitted to the government.

8. *Demonstration of the final (phase 2) Nimble C Compiler environment and libraries on at least one ACS architecture and one other architecture; including one cryptography and one multimedia algorithm set. Featuring tentatively the capacity (size) and rank-order cost function hardware selection, bit width interpolator, domain libraries for quick synthesis and composition, and automatic placement of the datapath. Up to five (5) unsupported, binary licenses of tool set to be made available to ACS research program participants (universities and potential customers) (4QFY00) (Task 2 - 6) (Synopsys)*

Finished. Demonstrated Nimble on the Garp, ACEII and ACEV architectures. The final Nimble environment includes compiler optimizations, comprehensive profiling and analysis framework, performance driven hardware-software partitioner, domain-specific libraries, quick synthesis capability and automatic placement of the datapath. Binary licenses along with the actual hardware have been shipped to several research institutes. Tutorials were given at the October 2000 PI meeting.

9. *Detailed Program Plan (3QFY98, 1QFY99, 1QFY00) and Final Report (4QFY00) (Task 1.1) (Synopsys)*

Finished. Plans and detailed schedules were submitted for three phases: Phase 0, Phase 1, and Phase1+/Phase2. This document along with about 400 pages of supporting documentation serves as the final report.

10. *Quarterly Reports and Yearly PI meeting (as needed) (Task 1.1) (Synopsys)*

Finished. Submitted quarterly reports and attended all the PI meetings during the project period.

11. *Demonstrate prototype C FE to mixed Processor and Xilinx 4K (4QFY98) (Task 2.1, Task 4.1) (Synopsys)*

Finished. Demonstrated the prototype C front-end to mixed process and Xilinx 4K (the ACEV II platform) at the February 1999 DARPA/AF review.

12. *Benchmark report that compares the Phase 1 and 2 demonstration platforms with Phase 0 and with other programmable processor only solutions and utilizes the demonstration platform results (2QFY99 preliminary, 1QFY00 Phase 1 draft, 4QFY00 Final) (Task 6.1) (Synopsys)*

Finished. Final benchmark report is included in Appendix M in Vol II.