AFRL-IF-WP-TR-2003-1546

ASC3: ALGORITHMIC STRATEGIES FOR COMPILER CONTROLLED CACHES

Benjamin Goldberg

New York University
Courant Institute
251 Mercer Street
New York, NY 10012-2105

Krishna V. Palem

Georgia Institute of Technology

Michael O. Rabin

Harvard University

OCTOBER 2003

Final Report for 01 July 1999 – 30 November 2002

STINFO FINAL REPORT

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

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.

/s/

ANDREW W. HYATT, 2Lt, USAF
Program Monitor

/s/

ROBERT A. EHRET, Chief
Collaborative Simulation Technology & Applications Branch
Information Systems Division
Information Directorate

Do not return copies of this report unless contractual obligations or notice on a specific document require 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)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
October 2003	Final	07/01/1999 – 11/30/2002

4. TITLE AND SUBTITLE

ASC3: ALGORITHMIC STRATEGIES FOR COMPILER CONTROLLED CACHES

5a. CONTRACT NUMBER
F33615-99-1-1499

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)

Benjamin Goldberg (New York University)
Krishna V. Palem (Georgia Institute of Technology)
Michael O. Rabin (Harvard University)

5d. PROJECT NUMBER
ARPI

5e. TASK NUMBER
FT

5f. WORK UNIT NUMBER
09

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

New York University
Courant Institute
251 Mercer Street
New York, NY 10012-2105

Georgia Institute of Technology

Harvard University

8. PERFORMING ORGANIZATION REPORT NUMBER

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

Information Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson AFB, OH 45433-7334

DARPA/IPTO
3701 Fairfax Drive
Arlington, VA 22203-1714

ONRRO Boston
495 Summer Street, Room 627
Boston, MA 02210-2109

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

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-IF-WP-TR-2003-1546

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

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The ASC3 effort was a collaboration among research groups. The effort focused on strategies for improving the performance of data-intensive and memory-bound programs through the innovation of algorithmic approaches in the following areas: 1) The management of programmable memory hierarchies, 2) Data remapping and speculative execution for improved cache performance, and 3) Compiler optimizations that are tolerant to memory aliasing. The techniques developed by the ASC3 effort were applied to a number of data-intensive applications, including automatic target recognition, database management, image matching, neural network simulation and scientific computation. The strategies were validated using industry-strength simulation and emulation tools, based on the Trimaran EPIC Research Infrastructure. The optimizations developed in the program exploited trends in microprocessor design and are applicable to current platforms, emerging EPIC architectures, and future generations of COTS software. The ASC3 effort also extended the functionality of the Trimaran Compiler EPIC infrastructure.

15. SUBJECT TERMS
Data Intensive Systems, Data mapping in Caches, management of programmable memory hierarchies

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**	SAR	30	2Lt Andrew W. Hyatt
Unclassified	Unclassified	Unclassified			**19b. TELEPHONE NUMBER** *(Include Area Code)* (937) 904-9162

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

ASC^3 : Algorithmic Strategies for Compiler Controlled Caches

Abstract

The ASC^3 effort within the Darpa Data Intensive Systems Program (DIS) was a collaboration among research groups at the Georgia Institute of Technology, New York University, and Harvard University. The effort focused on strategies for improving the performance of data-intensive and memory-bound programs through the innovation of algorithmic approaches in the following areas:

1. The management of programmable memory hierarchies,

2. Data remapping and speculative execution for improved cache performance,

3. Compiler optimizations that are tolerant to memory aliasing,

The techniques developed by the ASC^3 effort were applied to a number of of data-intensive applications, including automatic target recognition, database management, image matching, neural network simulation and scientific computation. The strategies were validated using industry-strength simulation and emulation tools, based on the Trimaran EPIC Research Infrastructure. The optimizations developed in the program exploited trends in microprocessor design and are applicable to current platforms, emerging EPIC architectures, and future generations of COTS systems.

In addition, the ASC^3 effort extended the functonality of the Trimaran Compiler EPIC infrastructure to provide a more realistic experimental framework for validating the algorithms developed in this program. In March of 2002, Tritanium – an Itanium-specific version of Trimaran – was released to the research community.

ASC^3 : Algorithmic Strategies for Compiler Controlled Caches

Final Technical Report

Benjamin Goldberg

New York University

Krishna V. Palem

Georgia Institute of Technology

Michael O. Rabin

Harvard University

1 Introduction

The ASC^3 effort within the Darpa Data Intensive Systems Program (DIS) was a collaboration among research groups at the Georgia Institute of Technology, New York University, and Harvard University. The effort was focused on strategies for improve the performance of data-intensive and memory-bound programs through the innovation of algorithmic approaches in the following areas:

1. The management of programmable memory hierarchies,

2. Data remapping and speculative execution for improved cache performance,

3. Compiler optimizations that are tolerant to memory aliasing,

The techniques developed by the ASC[3] effort were applied to a number of of data-intensive applications, including automatic target recognition, database management, image matching, neural network simulation and scientific computation. The strategies were validated using industry-strength simulation and emulation tools, based on the Trimaran EPIC Research Infrastructure. The optimizations developed in the program exploited trends in microprocessor design and are applicable to current platforms, emerging EPIC architectures and future generations of COTS systems. This report provides a summary of the results obtained in the areas enumerated above, along with citations to full papers and technical reports providing the details of the work.

In addition, the ASC[3] effort extended the functonality of the Trimaran Compiler EPIC infrastructure to provide a more realistic experimental framework for validating the techniques described in this report in the context of EPIC architectures. In March of 2002, Tritanium – an Itanium-specific version of Trimaran – was released to the research community. The Tritanium system is the result of an ongoing collaboration between the ASC[3] team members, the National University of Singapore and George Washington University.

2 The management of programmable memory hierarchies

In this section, we describe several approaches that were used to improve performance via the explicit management of the memory hierarchy through prefetching.

Data Prefetching is an important technique for addressing this growing problem, and is the focus of this research. In this research, a new paradigm that utilizes extensive profiling and powerful off-line learning algorithms to produce stochastic prediction models of hidden memory access patterns is proposed. By adapting those prediction models to newly proposed hardware and software-based prefetching schemes, we were able to overcome limitations of

current hardware or software-based data prefetching schemes which depending solely on run-time information.

The main contributions of this research are:

- A novel framework to perform off-line trace analysis that permits a wide range of learning algorithms to produce Markovian models to capture hidden memory access patterns.

- Proposal of a prefetching micro-architecture that is low in hardware requirement and overhead - Hardware-assisted prefetching mechanism.

- A software-based adaptation of this technique coupled with lightweight architectural extensions ubiquitous to generic EPIC processors to eliminate the need for additional hardware support.

Overall we were able to improve performances of conventional as well as irregular workloads from 11

3 Methodology

In the proposed framework, an off-line learning algorithm gathers a data access profile, processes it at compile time, and builds a model of the memory access patterns (MAP) for an application - this latter step shall be referred as cartography . The model is characterized by a connected graph, where nodes in the graph are addresses and edges between nodes indicate a transition from one address to the next. Using the memory profile, the probability of each transition is computed and associated with the appropriate edge in the graph. Various subsequent optimizations prune and compress the graph and glean necessary information to select candidates for prefetching.

Unlike the related work in this area, the cartography (learning phase) of the proposed optimization, as well as all subsequent decision making, are carried out at compile time. By contrast, prior related work performs similar tasks at run-time, and is thus restricted by the

available resources of the machine and hence limited in scope. The methodology proposed here provides a larger opportunity for program analysis, affords better decisions, and thus helps deliver the promise of Moore's Law to the end user.

We begin with the collection of load miss traces for a program. This is trivially achieved by augmenting a cache simulator to record the addresses that result in cache misses. Each load miss trace is a sequence of ¡lx_i , x_i¿ pairs, where lx_i is the uniqueoperation identifier in the program, and x_i is the memory address, fetched by l, which resulted in a cache miss at a particular level of the memory hierarchy. Recent studies have shown that only a small number of load instructions - the delinquent loads - are responsible for the bulk of the cache misses. Hence, the analysis only gathers information for a few operations as explained further when newly proposed hardware-assisted prefetching scheme is described in Chapter 3. Each basic block that contains a delinquent load is known as a hot spot and is a candidate for further analysis. A sequence of delinquent load operations will be the focus of newly proposed hardware-assisted and compiler directed software-based prefetching algorithm in this research as further explained in the following paragraphs.

After certain hot spots are identified by above procedure (hot spot filtering), a sequence of delinquent load operations for each hot spot in the target applications is fed into learning module to produce a Markov model for each hot spot. The Markov model is a probability-based automaton where each state corresponds to a memory address, and the state transitions realize a sequence of addresses. Furthermore, the automaton is annotated with the likelihood of occurrence of any given transition. Once built, this Markov model as shown in Chapter 3 generates a prediction model for each hotspot. Past studies have shown that in any address trace, there is a high probability that the current address is related or determined by past addresses. Furthermore, this phenomena is fairly stable in most applications, and is generally not sensitive to variations of the input workload. These properties make off-line Markovian-based cartography a suitable tool for extracting memory access patterns - which subsequently serve as a guide to data prefetching.

Once each prediction model per hotspot is created, it will be used for inserting dynamic

prefetch requests by prefetch engine for proposed hardware-assisted prefetching scheme or by the compiler to insert predicated prefetch operations for proposed software-based prefetching scheme (figure 1)

4   Hardware-assisted data prefetching

Hardware-based predictors operate in two phases - a learning phase and a prediction phase. In the learning phase, the prediction facility is trained. Typically, this involves the updating of a prediction table or automaton. In the prediction phase, the learned table or automaton is used to make prefetch requests. In some schemes, during the prediction phase, the prediction table or automaton may also be updated, i.e. the learning and prediction phases are interleaved.

A major drawback of existing hardware-based schemes is the need to perform learning and prediction both at run time. This severely limits the type and scope of learning schemes that one can use. Also, this may necessitate a significant investment in hardware due to complex logic, or there may be an impact on the critical path of instruction processing. In the worst case, it can be both. Thus, overcoming this limitation by taking the learning phase off-line is proposed in this research. In this chapter, it is shown that off-line learning not only improves overall performance of our prediction model but also makes possible through simple hardware logic for prediction engine since all the complex analysis and learning phase are preprocessed by off-line software module. This in turn make possible to realize a proposed prefetching scheme with much less extra hardware cost. We call our proposed scheme in this section as hardware-assisted prefetching scheme to differentiate with traditional hardware-based prefetching scheme since our proposed scheme has less hardware involvement in terms of logic and cost. The hardware is just assisting to issue and effectively store prefetched data what off-line software-learning module analyze and compute. We adopted Markovian algorithm for the learning phase. For a certain load miss address x in hot spot, consider all miss addresses within the specific window size w that immediately follows x and we call it a

Window Markov Predictor. More specifically, let y1, y2, , is the sequence of miss addresses that follows x, we compute N'(x) = P(yi — x) — i w, x, yi T . The set N' consists of the miss addresses yi which followed the miss address x. The variable w corresponds to the window size controlling the scope of the analysis.

Table 1 describes the average performance improvements of L1 & L2 prefetch over all benchmarks (SPEC + Olden) tested by many optimizations. Baseline setup was 32K L1 cache and 256K L2 cache without any prefetching scheme. Second column(64K L1 cache) shows the performance improvement when the size of L1 cache was increased 2 times from 32K to 64K bytes. Third column shows similar results when L1 cache size was tripled. RPT stands for "Reference Prediction Table" and it is one of traditional data prefetching scheme originally designed by Chen and Baer. We tested 3 variations of Markov Predictor and SMP(Simple Markov Predictor) is one using only next address after certain miss address. WMP(Window Markov Predictor) is one I described above and window size was set to 5 in this experimentation. We also adopted Hidden Markovian algorithm to our purpose and implemented and named as HMP(Hidden Markov Predictor) and those results are shown in Table 1. Using proposed optimization techniques, we could reduce the stall cycles and it directly leads to reduction in total execution cycles and achieving overall performance improvements. As Table 1 shows, WMP outperforms the entire optimization scheme tested in this experiments. The gap in memory and processor speed is increasing which eventually results in larger miss penalties, which in turn promises even larger performance improvements in the future systems.

4.1   Adaptive Compiler Directed Prefetching for EPIC Processors

After hardware-assisted prefetching scheme explained in section 3 was implemented and tested, we noticed the effectiveness of its framework and applied computational learning methods. Even though, the cost of building proposed framework is far less than the one of having bigger caches, we still need extra hardware involvement and naturally the next step we focused on was how to minimize the extra hardware cost and build more flexible scheme

without sacrificing too much on performance improvement shown in the existing framework.

There has been increasing popularity of EPIC architectures (ex. Itanium from Intel) which presents couple of novel architectural features which are exposed to the compiler, speculation and predication. Based on those features, we could propose software-based prefetching scheme, which lightly extends existing EPIC architecture (ISA extensions). We implemented couple of new special purpose registers (MPR:Miss Predicate Register, MAR:Miss Address Register) and new prefetch instruction. In our implementation, when a load operation misses primary cache, we set the MPR and enter the address causing the miss to MAR. New predicated prefetch instructions, which are inserted by compiler, are

-prefetch0 addr if MPR -If MPR is set, non-binding load (prefetch) the address addr(MAR + -const)

Each prefetch instruction in the proposed scheme is also a (non-binding) speculative data load. It is issued earlier than the control flow would normally allow and any resulting exceptions are generally masked. The advantages over traditional software-based prefetching scheme are the followings

- Low instruction overhead

- Adapts to run-time environments

- Avoids unnecessary bandwidth consumption and cache pollution

Table 2 shows Average performance improvement in several optimization schemes and SMP(Software Markov Predictor) in the last column shows the result for the proposed software-based prefetching scheme. The results reveals performance improvements due to larger cache sizes are not significant enough when compared to the architectural investments necessary to realize high capacity memory structures and proposed software prefetching scheme(SMP) improved performance 14.8% - 64% improvement over traditional stride based prefetching(RPT) alone.

5 Lessons Learned

By simply increasing L1 cache size alone did not help much in improving overall data cache performance and this trend is more significant as application shows more irregular memory access behavior. Thus we need new optimization technique for data intensive applications and a new paradigm that utilizes extensive profiling and powerful off-line learning algorithms which is proposed in this research shows noticeable performance benefits and it is expected to show more promise in the future systems as the gap between processor and memory widens.

5.1 Cache Sensitive Instruction Scheduling

As part of the ASC[3] effort, we designed and implemented a feedback-driven compiler-based instruction scheduling algorithm to mask the long latencies of memory accesses. The algorithm is named *Cache Sensitive Scheduling* (CSS). CSS is targetted towards VLIW instruction set architectures found in embedded systems as well as high-performance workstations. These instruction set architectures (ISAs) support high degrees of instruction level parallelism (ILP). A single VLIW instruction contains a set of operations that are executed in parallel. These ISAs rely on the compiler to derive efficient instruction schedules; and as result provide the compiler with information regarding the number and type of functional units as well as the latencies of individual operations.

CSS is a localized instruction scheduler that benefits from the creation of large regions of code containing no branches. This provides CSS with more opportunities to shuffle operations to mask the latencies of load operations. Examples of these regions are hyperblocks and superblocks depending on how they are formed. Forming these regions requires the ability to convert branches resulting from if-statements to sequences of operations with no branches. Therefore, CSS requires ISAs to provide instruction predication, which is found in many embedded processors (ARM and SHARC) as well as high-performance explicitly parallel instruction computer (EPIC) workstations such as Itanium.

CSS is extends traditional compiler-based rank-function instruction scheduling tech-

niques, such as CPFS[3], that focus on maintaining the critical path and improving ILP to mask the latency of delinquent load operations. CSS indirectly uses average latencies of load operations which are gathered from light-profiling. This makes CSS sensitive to the impact that a given load operation has on instruction schedule. CSS lessens or hides the impact these delinquent loads while still maintaining a high degree of ILP.

When an instruction scheduler becomes more aware of the latencies of load instructions and uses such information during scheduling, then we say it is a *Load Sensitive Scheduler*. The algorithm we are presenting here, CSS, is a load-sensitive scheduler. The state-of-art instruction schedulers today will treat all load operations as equals. Either the load operations are assumed to always experience a cache hit latency (*Optimistic Scheduler*), or the load operations are expected to experience a cache miss latency (*Pessimistic Scheduler*). The example in figure 1 illustrates the problems with these approaches. As shown in part A, the pessimistic scheduler chooses to schedule the load instructions with longer latencies that resemble miss latencies. The result is that when data arrives early, the uses of that data are unnecessarily delayed in beginning execution. As shown in part B, an optimistic scheduler will choose to schedule the instruction earlier; however, if the data arrives in a later cycle the processor will be stalled until the data arrives. This approach will shorten the schedule, but increase the execution time. In addition, the optimistic scheduler is not concerned with hiding latencies; and therefore, there are no gaps to insert other useful non-dependent instructions.

Consider the example in figure 2. This example is based on code from an implementation of matrix-multiplication. The original inner loop requires two floating point load operations, one multiply operation, and one add operation. After loop-unrolling, it requires 5 times as many operations. This scenario provides an opportunity for uncovering ILP in the schedule. Due to the number of load operations, there are likely to be varying degrees of latencies and some delinquent loads. Ideally, the loads producing cache misses should be scheduled earlier in the iteration, while scheduling the operations that use the data later in the iteration. The cycles in between can be used to schedule the other operations that do not depend on the
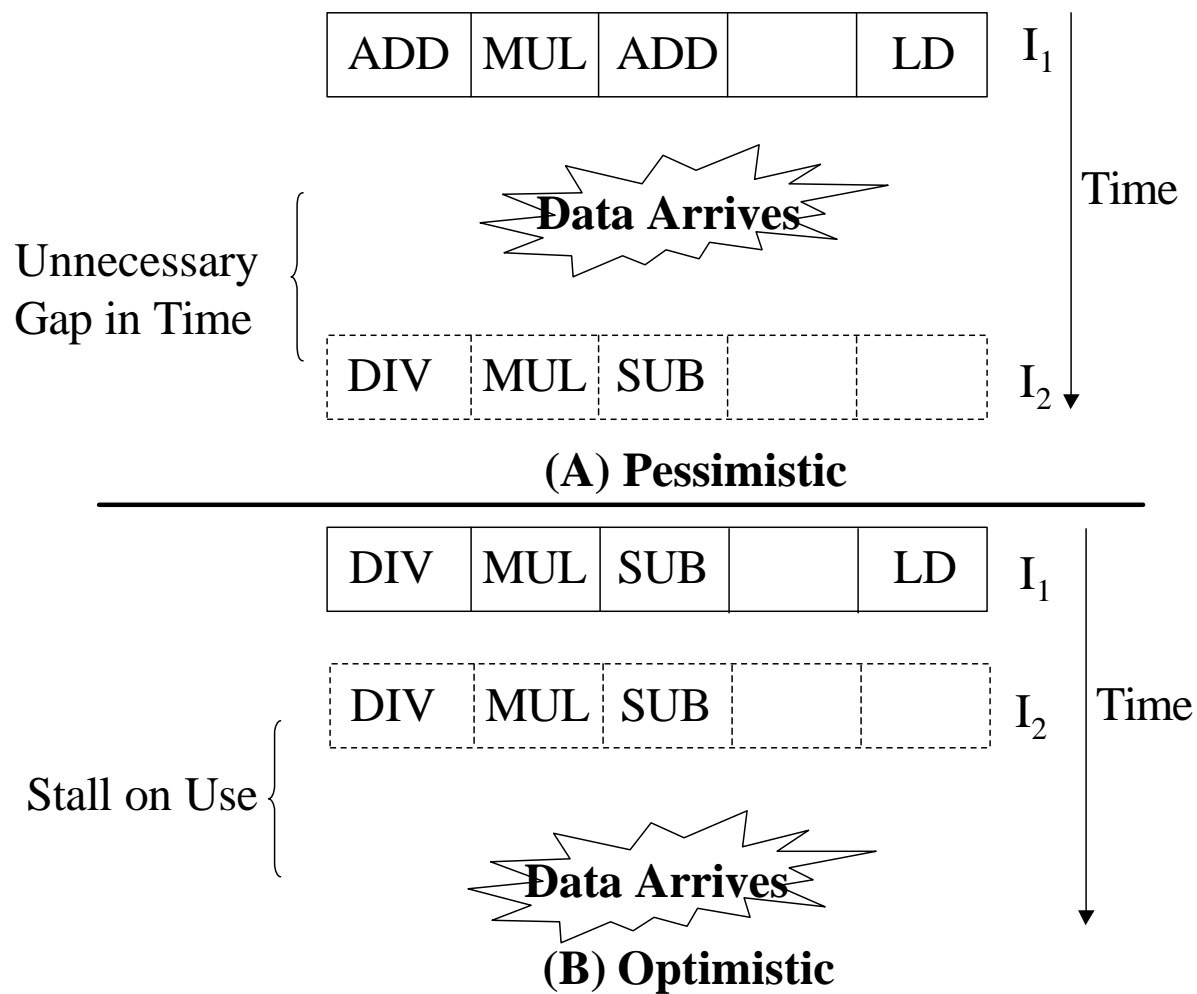
Figure 1: How conventional instruction schedulers handle load instructions

```
for k = 1 to N step 5
    S1 += A[i][k]*B[k][j];
    S1 += A[i][k+1]*B[k+1][j];
    S1 += A[i][k+2]*B[k+2][j];
    S1 += A[i][k+3]*B[k+3][j];
    S1 += A[i][k+4]*B[k+4][j];
endfor
```

Figure 2: The above code is from the inner loop of a matrix-multiply implementation. The inner loop has been unrolled by 5.

offending load operations. This reduces the effect of processor stalls related to dependent operations attempting to access data that has not arrived. At the same time, this increases the amount of ILP in the program. This is the essence of what CSS is designed to accomplish using a rank-based instruction scheduler framework. In other words, CSS uses realistic latencies of operations to determine how much of a gap should be placed between the load operation and the use of the data. The gap is tailored to the behavior of each load operation to provide the proper time for that particular load operation to load its data, while minimizing the effects found in optimistic and pessimistic scheduling.

$$
\begin{aligned}
CSSrank(i) = \alpha \quad &* \quad height(i) \quad + \quad \beta \quad * \\
&fanOut(i) \quad + \quad \gamma \quad * \\
&avgLatency(i) \quad - \quad \delta \quad * \\
&predLatency(i)
\end{aligned}
\tag{1}
$$

The CSS rank function as shown in equation 1 refers to $height(i)$, defined to be the distance from i (in a flow graph) to the last instruction, and $fanOut(i)$, defined to be the number of instructions waiting for the result of i. It is these two properties of each instruction that are used in current instruction schedulers such as CPFS. However, the CSS function has other components, mainly $avgLatency(i)$ and $predLatency(i)$. Our unique contribution

11

is these two components and the way in which they are incorporated. Our unique handling of these components gives us the ability to balance ILP gains with the ill-effects of poor memory utilization.

1. $avgLatency(i)$ refers to the average latency experienced by operation i during profiling. Our profile framework is able to capture the latencies, hit/miss quantities, and hit/miss ratios for individual load operations. The average latency gives a more realistic latency experienced by the load than the optimistic or pessimistic latency.

2. $predLatency(i)$ for operation i refers to the maximum $avgLatency(j)$, where $j \in preds(i)$. If i is dependent on longer latency operations then its priority should be decreased to allow the scheduling of operations that depend on operations with lower latencies.

Profiling is used to determine the appropriate latencies, and was one of central points of our method.

5.1.1 Results

The results are based on comparison with traditional critical path schedulers that used and did not use profiling as well as some other variants. This section summarizes the results.

The data in the table above summarizes our results for the most commonly used metrics for this area of research. We also developed other metrics:

1. *Closeness to etime(i)*: Captures how early operations are being scheduled relative to the earliest possible schedule time. The earliest possible schedule time is constrained by data dependences, as well as block and branch boundaries. This metric approaches 0 for schedulers that are producing compact schedules (i.e. optimistic schedulers).

2. *ILP Efficiency*: Measures how the amount of ILP being extracted by the scheduler during the execution of load operations. It is a ratio of actual ILP to the available ILP given an infinite resource machine.

Benchmark	% Compute Cycle Reduction	% Stall Cycle Reduction
DARPA DIS Benchmarks	18.18%	44.48%
Olden Benchmarks	6.35%	5.17%
Spec2000 Benchmarks	5.88%	6.46%

Table 1: Average percentage reductions in execution and stall cycles experienced by CSS in relation to the baseline scheduler over all selected benchmarks from a given suite.

The results of these metrics are shown in the following tables and graphs:

Table 2 contains the data for the E-Time Closeness metric. This metric was developed to determine whether or not load operations were being scheduled as early as possible. If load operations are being scheduled as early as possible then they cannot be scheduled earlier due to data dependences, block and branch boundaries, or resource constraints. The exact nature is beyond the scope of this paper. As values approach 0, then it can be concluded that operations are being scheduled as early as possible without violating any of the above constraints. In comparing CPFS and CSS, it is apparent that both schedulers schedule instructions very close to the e-time. If it were the case that CPFS was scheduling operations further from their e-time, then it could be argued that if CPFS simply scheduled operations earlier within the slack time, then it would compete with CSS. However, the data suggest that the operations need to be reordered in a way to balance the masking of load latencies and ILP.

The graphs in figures 3 and 4 compare the ILP Efficiency of CPFS and CSS for each of the benchmark suites. The y-axis is the quantity of load operations, and the y-axis is
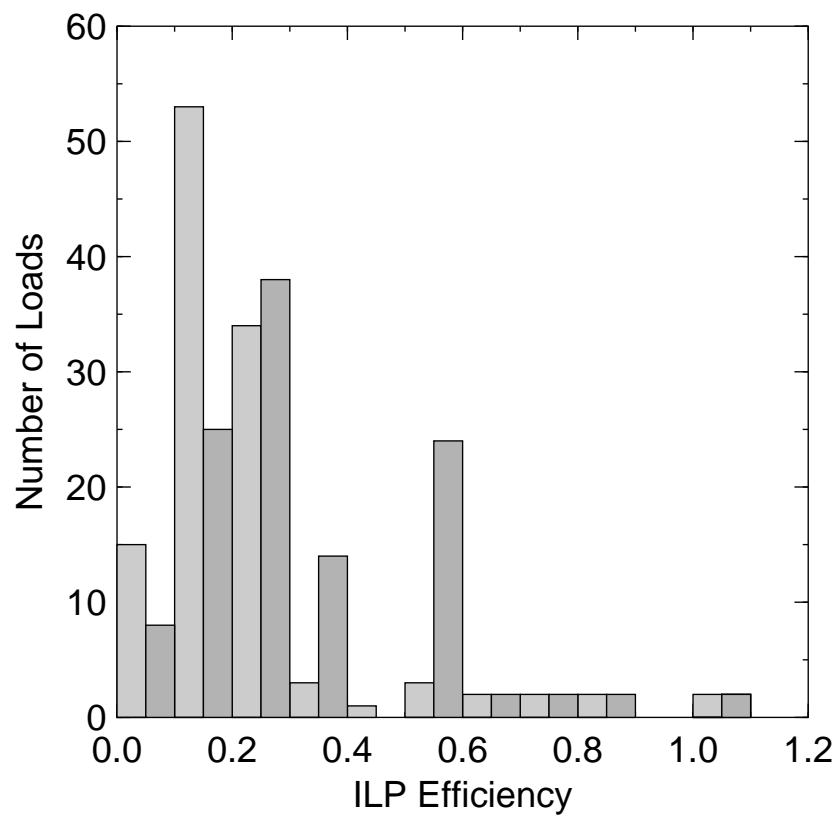
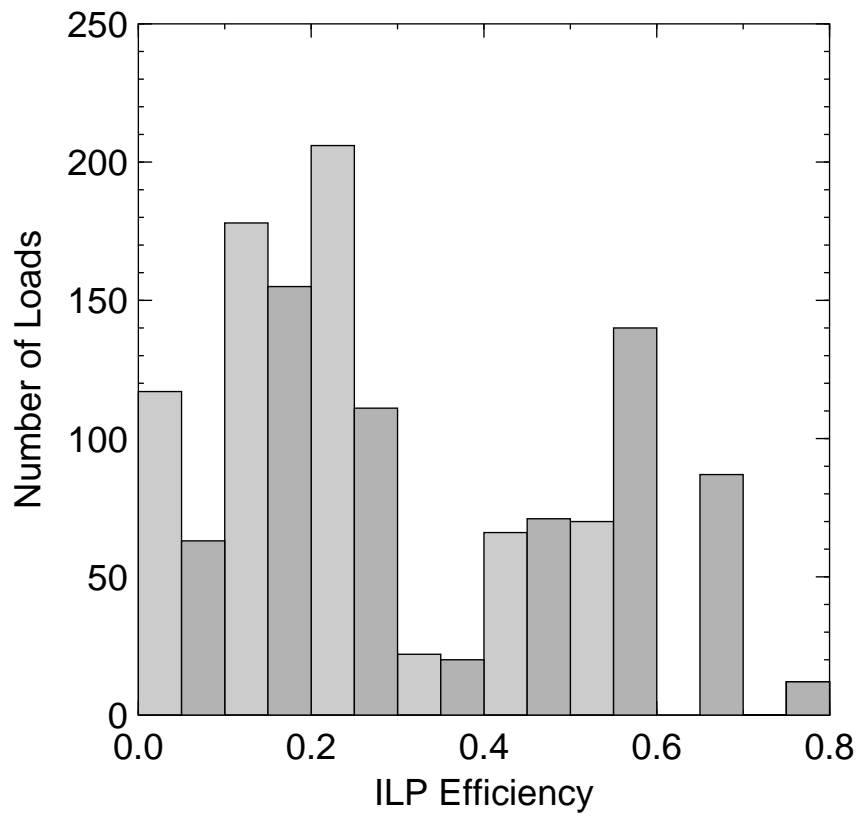Figure 3: Compares the CPFS algorithm(light) with the CSS algorithm(dark)

Figure 4: ILP Efficiency for DIS

Benchmark	E-time-Closeness (CPFS)	E-time-Closeness (CSS)	Difference (# of cycles)
Data Management	1.11	1.56	0.45
Matrix	0.62	0.88	0.26
Update	0.61	0.7	0.09
Neighborhood	1.52	1.75	0.23
Bisort	0.18	0.18	0
Health	0.09	0.14	0.05
Mst	0.09	0.14	0.05
Perimeter	0.09	0.14	0.05
Treeadd	0.09	0.14	0.05

Table 2: This table summarizes etime(i) closeness metric to determine if load operations are being scheduled as early as possible within data dependence, block/branch boundaries, and resource constraints

the percentage of functional units used by the schedule while load operations are executing. The trend in all of the benchmark suites is that CSS is able to extract more ILP on average during the execution of load operations. Overall there is an average of 11.5% improvement.

5.1.2   Lessons Learned

The work presented here showed that there were opportunities for scheduling instructions in a cache sensitive manner. However, the results indicate that these optimizations should be coupled with other locality optimizations to see the true benefit. Future work in this area should attempt to take advantage of programmable cache features that enable the compiler to have more control over the cache behavior, as well as eliminating the reliance on profile-data.

Another area would be to have a compilation/simulation environment that can not only handle other optimizations, but have an array of scheduling options as well. This will allow a more comprehensive comparision of the optimizations.

We feel that the future of this area of research will be in the ability to analytically determine program behavior vs profiling, the ability to reduce energy dissipation via scheduling, and analyses that allow you to study the effects of various optimizations on the performance goals.

A more complete description of this work can be found in [3].

6 Data remapping and speculative execution for improved cache performance

This section gives an overview of the results of the ASC^3 effort to improve the performance of data intensive programs via data remapping and speculative execution.

6.1 Ameliorating the Memory Bottleneck by Speculative Execution: The Load Dependence Graph

The effective use of the memory hierarchy is crucial for achieving good performance in all modern processors. In many applications, however, a very small number of delinquent memory operations are responsible for the bulk of the cache misses incurred by an application. As part of the ASC^3 effort, we developed an innovative, lightweight, and effective compiler framework that deals with delinquent loads by leveraging architectural features that exist in the class of Explicitly Parallel Instruction Computing (EPIC) processors – of which the Intel Itanium is an important representative.

We developed at compile-time program representation called the Load Dependence Graph, and showed how we can effectively make use of it to insert prefetching code in an application such that (i) the prefetch is highly precise, (ii) the technique is applicable to both numerical

and pointer-intensive applications, (iii) we require no new hardware support, and (iv) the overhead is negligible. Our results showed that when implemented in the Trimaran EPIC research infrastructure, we achieve a speedup of 26% on average. Experiments conducted on an Itanium processor demonstrate a 11.67% reduction in total execution time. Similar experiments performed on an Itanium II processor showed an average performance reduction of 7.14%. Our application test bed was drawn from the well-known Olden and SPEC2000 suites of integer and floating point benchmarks.

The full details of this work can found in [4].

6.2   Data Remapping for Design Space Optimization of Embedded Memory Systems

As part of the ASC^3 effort, we developed a novel linear time algorithm for data remapping that is (i) lightweight,(ii) fully automated and (iii) applicable in the context of pointer-centric programming languages with dynamic memory allocation support. All prior work in this area lacked one or more of these features. We demonstrated a novel application of this algorithm as a key step in optimizing the design on an embedded memory system. Specifically, we show that by virtue of locality enhancements via data remapping, we can reduce the memory subsystem needs of an application by up to 50%, and hence concomitantly reduce the associated costs in terms of size, power, and dollar-investment (by up to 61%). Such a reduction overcomes key hurdles in designing high performance embedded computing solutions.

Memory subsystems are very desirable from a performance standpoint, but their costs have often limited their use in embedded systems. Thus, our innovative approach offers the intriguing possibility of compilers playing a significant role in exploring and optimizing the design space of a memory subsystem for an embedded design. To this end and in order to properly leverage the improvements afforded by a compiler optimization, we identified a range of measures for quantifying the cost-impact of popular notions of locality, prefetching, regularity of memory access and others. The proposed methodology will become increasingly

important, especially as the needs for application specific embedded architectures become prevalent.

In addition, we demonstrated the wide applicability of data remapping using several existing microprocessors, such as the Pentium and UltraSparc. We showed that remapping can achieve a performance improvement of 20% on the average. Similarly, for a parametric research HPL-PD microprocessor, which characterizes the new Itanium machines, we achieve a performance improvement of 28% on average. All of our results were achieved using applications from the DIS, Olden and SPEC2000 suites of integer and floating-point benchmarks. A full description of this work can be found in [5].

7 Aliasing-Tolerant Compiler optimizations

As part of the ASC[3] project, we initiated a research effort aimed at developing, refining, and validating compiler techniques for utilizing the novel hardware of emerging architectures to maintain the effectiveness of compiler optimizations despite the presence of unanalyzable memory aliasing. Of interest to us is the kind of aliasing for which static analyses have been ineffective and generally cause the compiler to abandon optimizations that would otherwise be effective. We refer to such aliases as *dynamic* aliases, to reflect both the fact that static analysis techniques are ineffective for such aliasing and the fact that the aliasing properties may change over the lifetime of the program.

The hardware features that we believe to be particularly useful are those — such as dynamic memory disambiguation, predication, and support for speculative execution — that are being included in new processors such as those of the EPIC class of architectures exemplified by the Intel IA-64. The optimizations that we hope to be able to preserve range from the classical optimizations — CSE, loop invariant code motion, dead code elimination, etc. — to more recent techniques such as software pipelining, vectorization and parallelization, and advanced instruction scheduling and register allocation.

The ASC[3] work in this area proceeded as follows:

1. We identified compiler optimizations that are particularly sensitive to aliasing. Such optimizations were drawn from the classical optimization literature as well as from more recent work.

2. We developed techniques for adapting some of these optimizations to exploit novel hardware features so that their effectiveness is retained despite dynamic aliasing. As a legacy of the ASC[3] effort, we are continuing to develop techniques to important optimizations tolerant to aliasing.

3. We implemented several of these new techniques, followed by experimentation to quantify their effectiveness on a wide range of programs.

Our main result in this area, completed during the ASC[3] effort, was the development of a technique called called *Software Bubbling*. It is described below.

7.1 Software Bubbles: Using Predication to Compensate for Aliasing in Software Pipelines

Software bubbling is a technique for supporting software pipelining in the presence of dynamic aliasing. It is particularly useful for EPIC architectures, such as Intel's IA-64 and HP Laboratories' HPL-PD, that support both instruction-level parallelism and predication.

Software pipelining is a technique, commonly used in modern optimizing, for exploiting parallelism in loops by overlapping (in time) the execution of the iterations of the loop. Software pipelining and other methods for parallelizing loops rely on the compiler's ability to find loop-carried dependences between iterations. Often the presence or absence of these dependences cannot be determined statically, due to aliasing.

The idea behind software bubbling is that the compiler, when encountering a loop that could not be pipelined due to aliasing, generates a single version of the pipeline that is constrained only by the static dependences and resource constraints known to the compiler. The operations within the pipelined loop, however, are predicated in such a way that if dynamic dependences arise, the predicate registers are set in a regular but rotating pattern

that causes some of the kernel operations to be disabled each time the kernel is executed. With this disabling of operations, the effective overlap of the iterations in the pipeline is reduced sufficiently to satisfy the dynamic dependences. We refer to the disabled operations as *software bubbles*, drawing an anology with bubbles in processor pipelines. Experiments using the SPEC95 Alvinn benchmark and the Livermore Loops Kernel2 benchmark demonstrated performance improvements of 40% on each benchmark and 80% on the specific loops to which software bubbling was applied.

The details of the software bubbling work were published in [2].

8 Randomized Techniques for Memory Bandwidth Optimization

Work pursued under subcontract at Harvard focussed on the key idea of using probabilisitc or randomixed sampling to help ameliorate the memory bottleneck challenge. Based on this idea, the Harvard team abstracted and demonstrated that for the *k nearest vector problem* which involves, given a fixed vector with integer entries, finding its k nearest "neighbors" in a well-defined metric space, hihgly bandwidth efficient randmozied algorithms are possible. By interpreting the entries in these vectors as denoting features, the notion of distance has been shown to be useful in finding good matches in such domains as face recognition, fingerprrting matching and other such very practical methods. Preliminary software implementations have been completed and the investigators are currently developing a hardware prototype to validate the efficacy of these ideas in a real-time patter matching setting. The main concept that this work demonstranted is to show that the randomized approach can yield and exponential improvement to the bandwidth requirements while yielding a match with very high (exponential) probability. Details can be found in [1].

References

[1] Carl Bosley and Michael O. Rabin. Selecting closest vectors through randomization. Technical Report TR-01-00, Computer Science Group Harvard University, 2000.

[2] B. Golderg, E. Crutcher, C. Huneycutt, and K. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *Proceedings of the 2002 Symposium on Parallel Architectures and Compilation Techniques (PACT'02)*, September 2002.

[3] C. Hardnett, R. Rabbah, K. Palem, , and W-F. Wong. Cache sensitive instruction scheduling. Technical Report GIT-CC-02-18, Georgia Institute of Technology, 2002.

[4] R. Rabbah, M. Ekpanyapong, and W-F. Wong. Ameliorating the memory bottleneck by speculative execution: The load dependence graph. Technical Report GIT-CC-02-57, Georgia Institute of Technology, November 2002.

[5] R. Rabbah and K. Palem. Data remapping for design space optimization of embedded memory systems. *ACM Transactions in Embedded Computing Systems*, 2(2), May 2003.