RICE UNIVERSITY

Exploiting Instruction-Level Parallelism for Memory System Performance

by

Vijay S. Pai

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Sarita Adve, Chair
Associate Professor in Electrical and Computer Engineering

Keith D. Cooper
Professor of Computer Science

Kenneth W. Kennedy, Jr.
Ann and John Doerr Professor in Computational Engineering

Willy E. Zwaenepoel
Noah Harding Professor of Computer Science and Electrical and Computer Engineering

Houston, Texas

August, 2000

Exploiting Instruction-Level Parallelism for Memory System Performance

Vijay S. Pai

Abstract

Current microprocessors improve performance by exploiting instruction-level parallelism (ILP). ILP hardware techniques such as multiple instruction issue, out-of-order (dynamic) issue, and non-blocking reads can accelerate both computation and data memory references. Since computation speeds have been improving faster than data memory access times, memory system performance is quickly becoming the primary obstacle to achieving high performance. This dissertation focuses on exploiting ILP techniques to improve memory system performance. This dissertation includes both an analysis of ILP memory system performance and optimizations developed using the insights of this analysis.

First, this dissertation shows that ILP hardware techniques, used in isolation, are often unsuccessful at improving memory system performance because they fail to extract parallelism among data reads that miss in the processor's caches. The previously-studied latency-tolerance technique of software prefetching provides some improvement by initiating data read misses earlier, but also suffers from limitations caused by exposed startup latencies, excessive fetch-ahead distances, and references that are hard to prefetch.

This dissertation then uses the above insights to develop compile-time software transformations that improve memory system parallelism and performance. These transformations improve the effectiveness of ILP hardware, reducing exposed latency by over 80% for a latency-detection microbenchmark and reducing execution time an average of 25% across 14 multiprocessor and uniprocessor cases studied in simulation and an average of 21% across 12 cases on a real system. These transformations also combine with software prefetching to address key limitations in either latency-tolerance technique alone, providing the best performance when both techniques are combined for most of the uniprocessor and multiprocessor codes that we study.

Finally, this dissertation also explores appropriate evaluation methodologies for ILP shared-memory multiprocessors. Memory system parallelism is a key feature determining ILP performance, but is neglected in previous-generation fast simulators. This dissertation highlights the errors possible in such simulators and presents new evaluation methodologies to improve the tradeoff between accuracy and evaluation speed.

Acknowledgments

I would like to thank my advisor, Sarita Adve, for research direction, challenges, and a real commitment to my professional growth. I also thank my officemate, co-author, and friend, Parthasarathy Ranganathan, for many interesting discussions and adventures (technical and otherwise) since we started working together. It has been a great pleasure to work with these two people for the past six years.

My Ph.D. thesis committee of Keith Cooper, Ken Kennedy, and Willy Zwaenepoel have given me excellent feedback on this work since the time of my proposal, and this feedback has been instrumental in shaping both the technical content of this work and my presentation of it. Some of this work draws from my M.S. thesis, for which I also profited from the suggestions of my committee of Alan Cox and Bob Jump.

I would also like to thank all the others with whom I have had a chance to interact over the years, with whom I have been a co-author, and who have given me feedback on the research leading to this dissertation and on my presentations of this research. In particular, this dissertation draws heavily from joint work that I pursued with Murthy Durbhakula, Dan Sorin, Mary Vernon, and David Wood. I thank all those who read my paper drafts and attended my practice talks over the years, and I feel that their comments have greatly helped refine both the content and presentation of this work. Vikram Adve, Chen Ding, John Mellor-Crummey, and Shubu Mukherjee particularly went far beyond the call of duty here. Credit also goes to all the RSIM team members over the years, including Hazim Abdel-Shafi, Dennis Geels, Jon Hall, Tracy Harton, Chris Hughes, and Praful Kaul. Doug Moore's Hilbert-curve library helped make Mp3d an interesting and tractable application for the course of this study. Donald Yeung provided the source code for the Em3d application used in this study.

I cannot thank my family enough for everything they have given me over the years. I owe a tremendous debt to my parents, Sadananda and Sharda Pai, for very practical contributions of guidance, support, and encouragement over the past twenty-six years. I also thank my grandmother, Janabai Kamath, for support and many nutritious meals. My brother Vinay and my sister-in-law Aarti have also given their support, and his kids have provided a great source of amusement and amazement on their visits. Finally, my brother Vivek and I shared most of the years of our graduate school experience. I have profited tremendously from his suggestions, discussions, and feedback on technical and non-technical matters, and I am very glad that we now have an opportunity to be colleagues as well.

My graduate career was supported by a Fannie and John Hertz Foundation Fellowship from September 1994 through May 1999. The research presented in this dissertation was also supported by IBM Corporation, Intel Corporation, the National Science Foundation under Grant No.CCR-9410457, CCR-9502500, CDA-9502791, CDA-9617383, CCR-0096126, the Texas Advanced Technology Program under Grants No. 003604-016 and 003604-025, and the Alfred Sloan Research Foundation. This dissertation was initiated to *Early Underground* by Moby and concluded to *Fight Club* by the Dust Brothers.

Contents

Illustrations

Tables

Chapter 1

Introduction

1.1 Motivation

Current commodity microprocessors improve performance by using aggressive techniques to exploit high levels of instruction-level parallelism (ILP). These techniques include multiple instruction issue, out-of-order (dynamic) scheduling, non-blocking reads, and speculative execution. We refer to these techniques collectively as *ILP techniques* and to processors that exploit these techniques as *ILP processors.* ILP processors are being used not only for uniprocessor systems, but also for high-performance shared-memory multiprocessors that focus on data-intensive scientific and commercial applications.

Most previous studies of shared-memory multiprocessors, however, have assumed a *simple processor* with single-issue, in-order scheduling, blocking reads, and no speculation. This assumption allows the use of direct-execution simulation, which is significantly faster than the detailed simulation currently required to model an ILP processor pipeline. A few multiprocessor architecture studies model state-of-the-art ILP processors [Gor95, NHO96, ONH^{+}96, PRAH96], but do not analyze the impact of ILP techniques. Further, many uniprocessor studies focus on instruction-level parallelism without specifically targeting optimizations toward memory system performance. Memory system performance is quickly becoming the primary factor determining overall system performance, so optimization approaches that target this subsystem are becoming more important.

To fully exploit recent advances in ILP technology for memory system performance, a detailed analysis is required to indicate how ILP techniques affect memory system performance in high-performance uniprocessor and multiprocessor systems and how they interact with memory system optimizations that are not specific to ILP. Such an analysis is also required to assess the validity of the continued use of direct-execution simulation with simple processor models to study next-generation shared-memory architectures. This dissertation performs such an analysis and uses the insights of this analysis to address the limitations of aggressive ILP-based systems, both in memory system performance and in evaluation methodology.

1.2 Contributions

This dissertation makes three key contributions to the state-of-the-art. First, we analyze the impact of ILP hardware techniques on data memory system performance. Second, we use the insights of that analysis to propose and evaluate new ILP-specific optimizations to improve data memory system performance. Third, we also analyze and improve the tradeoff between accuracy and speed in evaluating ILP-based systems. This work was enabled by the development of RSIM (the Rice Simulator for ILP Multiprocessors) as a part of this thesis. RSIM was the first publicly-available shared-memory multiprocessor simulator to model processors that aggressively exploit instruction-level parallelism.

The following discusses each of these topics more thoroughly.

1.2.1 Analyzing the Impact of ILP Techniques on Memory System Performance

This work characterizes the effectiveness of state-of-the-art ILP processors in shared-memory multiprocessor and uniprocessor systems, using a detailed ILP multiprocessor and uniprocessor simulator (RSIM) driven by full scientific applications [PRA97b, PRA96, PRAA99]. We show that all of our applications see performance improvements from the use of current ILP techniques in multiprocessors. However, the improvements achieved vary widely. In particular, ILP techniques successfully and consistently reduce the CPU component of execution time, but their impact on the memory (read) stall component is lower than their impact on CPU time and is also more application-dependent. As a result, data read stall time becomes a larger bottleneck than in previous-generation systems.

These deficiencies in the impact of ILP techniques on memory stall time arise primarily because of insufficient potential in our applications to overlap multiple read misses, as well as system contention from more frequent memory accesses. This portion of the dissertation (Chapter 2) draws from joint work with Parthasarathy Ranganathan.

This work also analyzes the interaction between software prefetching and ILP techniques in shared-memory multiprocessors [RPAA97, PRAA99]. Software-controlled non-binding prefetching has been shown to be an effective technique for hiding memory latency in simple processor-based shared memory systems [Mow94]. We show several important factors that can limit the effectiveness of prefetching and relate these to the ILP techniques in general and our applications in particular. On the whole, we find prefetching to be less effective in reducing memory stall time in ILP-based systems than in systems with simple processors. This portion of the dissertation (Chapter 3) draws from joint work with Parthasarathy Ranganathan and Hazim Abdel-Shafi.

Overall, our results suggest that, despite the inherent latency-tolerating mechanisms in ILP processors, multiprocessors built from ILP processors actually exhibit a greater potential need for additional latency reducing or hiding techniques than previous-generation multiprocessors. We identify clustering of read misses together within the ILP processor's instruction window as a potentially effective optimization for better exploiting the ILP features of current processors.

1.2.2 ILP-Specific Code Transformations to Improve Multiprocessor Performance

The second contribution of this dissertation is to propose software code transformations to improve read miss clustering for systems with out-of-order processors, while preserving cache locality [PA99, PA00]. We exploit code transformations already known and implemented in compilers for other purposes, providing the analysis needed to relate them to read miss clustering. The key transformation we use is unroll-and-jam, which was originally proposed for improving floating-point pipelining and for scalar replacement [AC72, CCK88, CK94, Nic87]. We develop an analysis and transformation framework that maps the read miss clustering problem to floating-point pipelining.

We evaluate the clustering transformations applied by hand to a latency-detection microbenchmark and seven scientific applications running on simulated and real uniprocessor and multiprocessor systems. These transformations reduce data memory latency stall time by over 80% for the latency-detection microbenchmark. For the scientific applications, the transformations reduce execution time by 5–39% (averaging 20%) in the simulated multiprocessor and 11–49% (averaging 30%) in the simulated uniprocessor. A substantial part of these execution-time reductions arise from improving memory parallelism, particularly as memory stall time becomes more significant. We confirm the benefits of our transformations on a real system (Convex Exemplar), where they reduce application execution time by 9–38% for 6 out of 7 applications.

We then show that the techniques of read miss clustering and software prefetching are actually mutually beneficial, each helping to overcome the performance limitations of the other. Thus, the combination can expose opportunities for latency tolerance missed by either technique alone. We evaluate read miss clustering, software prefetching, and their combination both with a detailed simulator (RSIM) and on a real system (Convex Exemplar). We apply miss clustering by hand in both cases and apply prefetching by hand for the simulation. We consider prefetching for both regular and irregular applications [KDS00, LM96, Mow94, RS99]. Experimental results show that clustering alone outperforms prefetching alone for most of the applications and systems that we study, and that the combination of read miss clustering and prefetching yields better execution time benefits than either technique alone in most cases. The combination of clustering and prefetching yields a significant improvement in latency tolerance over prefetching alone (the state-of-the-art implemented in systems today), with an average of 21% reduction in execution time across all cases studied in simulation and an average of 16% reduction in execution time for 5 out of 10 cases on the Exemplar. The experimental results also show reductions in execution time relative to clustering alone averaging 15% for 6 out of 11 cases in simulation and 20% for 6 out of 10 cases on the Exemplar.

1.2.3 Analyzing and Addressing the Impact of ILP on Multiprocessor Evaluation Methodology

The third contribution of this thesis focuses on evaluation methodologies for shared-memory multiprocessors built from ILP processors. We first study the validity of current simple-processor direct-execution simulation based models to model ILP multiprocessors [PRA97b]. For applications where our ILP multiprocessor fails to significantly overlap read miss latency, a simulation using a simple previous-generation processor model with a higher clock speed for the processor and the L1 cache provides a reasonable approximation to the results achieved with a more detailed simulation system. However, when ILP techniques effectively overlap read miss latency, all of our simple-processor-based simulation models can show significant errors for important metrics. Overall, for total simulated execution time, the most commonly used simulation technique gave an average error of 137% (ranging 9%–438%), while even the best approximate model saw an average execution time discrepancy of 46% (ranging 0%–128%). These errors depend on both the application and the ILP characteristics of the system; thus, models that do not properly capture these effects may not be able to effectively characterize an ILP-based multiprocessor system. Nevertheless, direct-execution simulations appear attractive since they are as much as an order of magnitude faster than the detailed simulator RSIM [DPA99]. This portion of the study draws from joint works with Murthy Durbhakula and Parthasarathy Ranganathan.

This dissertation then presents two evaluation methodologies to speed up the characterization of ILP multiprocessors. The first is a novel adaptation of direct-execution simulation that substantially speeds up simulation of shared-memory multiprocessors with ILP processors, without much loss of accuracy [DPA99, DPA98]. We have developed a new simulator, DirectRSIM, based on this new technique. This thesis evaluates the accuracy and speed of DirectRSIM by comparing it with RSIM and with two representative simple-processor based direct-execution simulators. For a variety of system configurations and applications, we find that DirectRSIM, on average, is 3.6X faster than RSIM with an error in reported execution time relative to RSIM of only 1.3% (range of -3.9% to 2.2%). Simple-processor based simulators remain an average of 2.7X faster than DirectRSIM. However, this additional speed comes at the cost of their much larger and more unpredictable errors. This result suggests a reconsideration of the appropriate simulation methodology for shared-memory systems. Earlier, the order-of-magnitude performance advantage of the simple-processor based simulators over detailed simulators like RSIM made a compelling argument for their use in spite of their potential for large errors. It is not clear that those errors are still justifiable given only a 2.7X performance advantage relative to DirectRSIM. This portion of the dissertation draws from joint work with Murthy Durbhakula.

The second new evaluation methodology presented in this dissertation is a high-level simulation methodology called FastILP [SPA$^+$98]. FastILP interacts with a new analytical model of ILP-based multiprocessors. Analytical modeling is an attractive alternative to simulation because of its high speed. However, most analytical models are driven by synthetic parameters that may not represent any real workload. FastILP generates the parameters needed to allow this new analytical model to capture relevant information about the workload. Specifically, FastILP characterizes an application in terms of fundamental ILP parameters, rather than attempting to measure the time required for the application. The parameters measured are the distribution of read misses outstanding at the time that processor action stalls (an indicator of overlap) and the rate and variance by which external memory requests are sent from the active processor (indicators of contention). Since FastILP does not need to measure the exact cycle count for an execution, it can achieve very high performance by abstracting both the ILP processor and the memory system, and modeling only enough state to generate the required ILP parameters. The combination of FastILP and the new analytical model can be used to compute performance estimates very quickly while still accounting for important workload characteristics. FastILP and the analytical model for ILP multiprocessors were developed jointly with Daniel Sorin, Mary Vernon, and David Wood.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 analyzes the impact of current techniques that exploit instruction-level parallelism and their effectiveness in targeting memory system performance. Chapter 3 shows the interaction between software prefetching and ILP. Chapter 4 proposes software code transformations that aim to improve the interaction of ILP with memory system performance by increasing memory parallelism. We also evaluate these transformations on both a simulator and a real machine. Chapter 5 compares and combines the software latency tolerance techniques discussed in the previous chapters (software prefetching and read miss clustering), and explains how each of these techniques can address limitations in the other. Chapter 6 analyzes the accuracy and performance of simulators for ILP multiprocessor systems and presents a new simulation methodology to improve the tradeoff between accuracy and simulation speed. Chapter 7 discusses our fast high-level evaluation methodology that characterizes applications in terms of fundamental ILP parameters instead of specific timings. Chapter 8 discusses related work, and Chapter 9 describes the conclusions of this study and planned future work.

Chapter 2

The Impact of ILP on Memory System Performance

Most multiprocessor architectural studies have used simple-processor-based models for the evaluation of their benefits. Some studies have modeled the effects of ILP, but have not detailed the benefits of techniques to exploit ILP in multiprocessors [NHO96, ONH^{+}96, GGH92, PRAH96, ZB92]. Many other works study the impact of ILP on uniprocessor systems without specifically analyzing its interaction with memory system performance. Our work seeks to provide a detailed analysis of the impact of ILP on data memory system performance, both to assess the efficacy of techniques to exploit ILP and to identify limitations to high performance.

This chapter evaluates the impact of instruction-level parallelism on memory system performance. Section 2.1 describes the architectural models and applications studied, as well as the quantitative metrics used. Section 2.2 focuses on multiprocessor systems, comparing the performance of systems based on simple and ILP processors and characterizing the effectiveness of current techniques to exploit ILP. Section 2.3 shows the impact of ILP on uniprocessor systems. Section 2.4 describes the sensitivity of our experimental results to system latencies as the processor-memory speed gap continues to grow. Finally, Section 2.5 summarizes the key findings of these studies and describes additional issues in the impact of ILP on shared-memory multiprocessor performance.

2.1 Methodology

This section describes the ILP and simple processor based systems that we investigate in our study, our simulation infrastructure, and the evaluation workload we use to characterize our systems.

2.1.1 Simulated Architectures

To determine the impact of ILP techniques on multiprocessor performance, we compare two CC-NUMA multiprocessor systems – *ILP* and *Simple* – equivalent in every respect except the processor used. The ILP system uses state-of-the-art ILP processors while the Simple system uses simple processors, as described below. We compare the ILP and Simple systems not to suggest any architectural tradeoffs, but rather, to understand how aggressive ILP techniques impact multiprocessor performance. Therefore, the two systems have identical clock rates, and include identical aggressive memory and network configurations suitable for the ILP system. Table 2.1 summarizes all the system parameters.

Processor models. The ILP system uses state-of-the-art processors that include multiple issue, out-of-order (dynamic) scheduling, non-blocking reads, and speculative execution. The Simple system uses previous-generation simple processors with single issue, in-order (static) scheduling, and blocking reads, and represents commonly studied shared-memory systems. Since we did not have access to a compiler that schedules instructions for our in-order Simple processor, we assume single-cycle functional unit latencies (as also assumed by most previous simple-processor based shared-memory studies). Both processor models include support for software-controlled non-binding prefetching to the L1 cache (discussed in Chapter 3).

Memory Hierarchy and Multiprocessor Configuration. We simulate a hardware cache-coherent, non-uniform memory access (CC-NUMA) shared-memory multiprocessor using an invalidation-based, three-state MSI directory coherence protocol. We extend our MSI protocol to issue reads in exclusive-mode when a write instruction to the same cache line appears later in the processor's instruction window. We model the release consistency memory model because previous studies have shown that it achieves the best performance [PRAH96].

Figure 2.1 shows the topology and features of our CC-NUMA system. The processing nodes are connected using a two-dimensional mesh network. Each node includes a processor, a primary instruction cache, a primary data cache, a unified secondary cache, a portion

Processor parameters	
Clock rate	500 MHz
Fetch/decode/retire rate	4 per cycle
Instruction window	64 instructions in-flight
Memory queue size	32
Outstanding branches	16
Number of functional units	2 ALUs, 2 FPUs, 2 address generation units
Functional unit latencies	1 cycle
Memory hierarchy and network parameters	
L1 D-cache	16 KB, direct-mapped, 2 ports, 10 MSHRs, 64-byte line
L1 I-cache	16 KB, direct-mapped, 64-byte line
L2 cache	64 KB (for Erlebacher, FFT, LU, and Mp3d) or 1 MB (for Em3d, MST, and Ocean), 4-way associative, 1 port, 10 MSHRs, 64-byte line, pipelined
Memory banks	4-way, permutation-based interleaving
Bus	167 MHz, 256 bits, split transaction
Network	2D mesh, 250MHz, 64 bits, flit delay of 2 network cycles per hop
System latencies in absence of contention	
L1 hit	1 processor cycle
L2 hit	10 processor cycles
Local memory	85 processor cycles
Remote memory	180-260 processor cycles
Cache-to-cache transfer	210-310 processor cycles

Table 2.1 : Base system configuration used in evaluating impact of ILP techniques on multiprocessor performance.

of the global shared-memory and directory, and a network interface. A split-transaction bus connects the network interface, directory controller, and the rest of the system node. Both data caches are non-blocking, cache with a write-allocate write-back policy, and use miss status holding registers (MSHRs) [Kro81] to store information on outstanding misses and to coalesce multiple requests to the same cache line. The cache sizes are scaled based on application input sizes according to the methodology of Woo et al. [WOT$^+$95]. Realistic application inputs would typically require an impractical amount of simulation time. How-

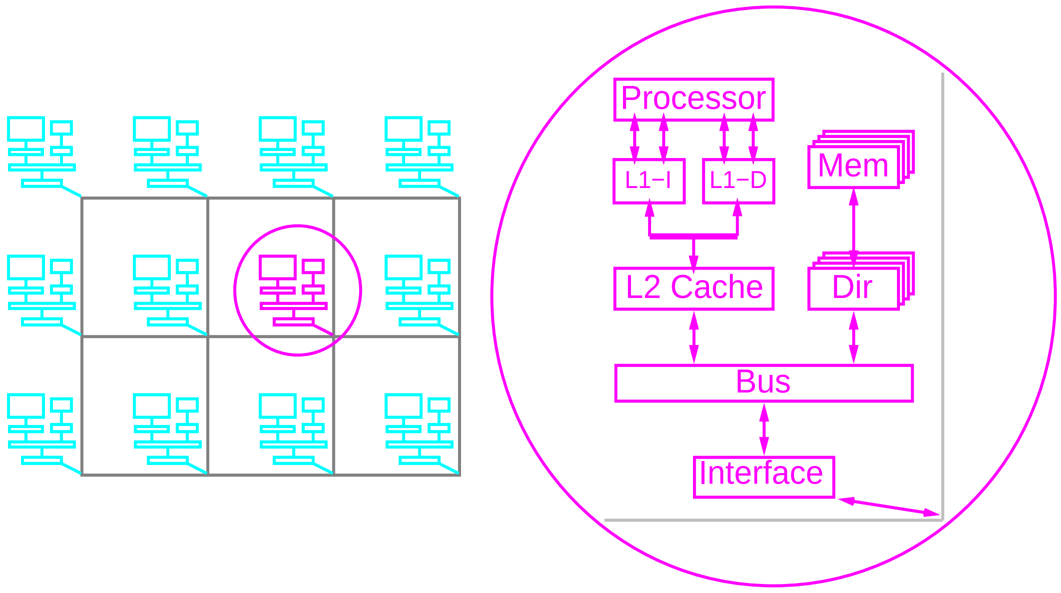


Figure 2.1 : Layout of CC-NUMA multiprocessor used in simulatioon studies, along with detailed diagram of individual processing node.

ever, smaller input sizes may lead to overly optimistic views of cache performance. Thus, this methodology suggests scaling the cache sizes down according to the simulated working sets, and is widely used in multiprocessor architectural studies. The primary working sets for our applications typically fit in the L1 cache, while the secondary working sets do not fit in the L2 cache. The memory banks use permutation-based interleaving on a cache-line granularity to support a variety of strides [Soh93].

Simulation environment. We use RSIM, the Rice Simulator for ILP Multiprocessors, to model the systems studied [PRA97a]. RSIM is an execution-driven simulator that models the out-of-order processor, memory system, and interconnection network in detail, including contention at all resources. It takes SPARC application executables as input.

2.1.2 Performance Metrics

Our primary metric for determining the impact of ILP is the total execution time. To characterize the performance bottlenecks in our system, we also categorize simulated execution time as follows: data read miss stall (for stalls caused by data references to main memory), data read hit or write stall (usually exposed only in the event of high resource contention), CPU (busy time and functional-unit stalls), synchronization, and instruction memory stall times. We account for stall cycles in the ILP processor as follows, similar to previous work (e.g., [LM96]). For each cycle, we calculate the ratio of the instructions retired from the instruction window to the maximum retire rate and attribute this fraction of that cycle to busy time. The rest of the cycle is attributed as stall time to the first instruction that could not retire that cycle, or as instruction memory stall if no instruction is available in the window because of an I-cache stall.

We also define a derivative metric to characterize the impact of ILP, which we call the *ILP speedup*. This is the ratio of the execution time with the Simple system relative to that achieved by the ILP system. For detailed analysis, we analogously define an ILP speedup for each component of execution time.

2.1.3 Evaluation Workload

Table 2.2 summarizes the evaluation workload for the simulated system. The number of processors used for the multiprocessor experiments is based on application scalability, with a limit of 16. The input sizes were chosen to ensure reasonable simulation times. Since a SPARC compiler for our ILP system does not exist, we compiled our applications with the commercial Sun SC 4.2 or the gcc 2.7.2 compiler (whichever gave better simulated ILP system performance) with full optimization turned on. The compilers' deficiencies in addressing the specific instruction grouping rules of our ILP system are partly hidden by the out-of-order scheduling in the ILP processor.

FFT, LU, Ocean, and Radix are array-based codes from the SPLASH-2 suite consisting primarily of loops and loop nests [WOT+95]. FFT, LU, and Ocean are *regular* codes:

Application	Input Size	Processors
Em3d	32K nodes, deg. 20, 20% rem.	1,16
Erlebacher	64x64x64 cube, block 8	1,16
FFT	65536 points	1,16
LU	256x256 matrix, block 16	1,8
Mp3d	100000 particles	1,8
MST	1024 nodes	1
Ocean	258x258 grid	1,8
Radix	1024 radix, 512K keys, max 512K	1,8
Water	512 molecules	1,16

Table 2.2 : Evaluation workload for simulated system.

the array indices used in these codes are affine functions of the controlling loop variables. Radix has one important phase with irregular references formed by indirection. For better load balance, LU is modified slightly to use flags instead of barriers.

Mp3d and Water are from the SPLASH suite [SWG92]. To eliminate false-sharing in the irregular, asynchronous, and communication-intensive Mp3d application, key data structures were padded to a multiple of the cache line size. To reduce true-sharing and improve locality in Mp3d, the data elements were sorted by position in the modeled physical world [MWK99].

Erlebacher is a shared-memory port of a code by Thomas Eidson at the Institute for Computer Applications in Science and Engineering (ICASE). Like FFT, LU, and Ocean, Erlebacher is also a regular array-based code dominated by loop nests.

Em3d is a shared-memory adaptation of a Split-C application [CDG$^+$93]. This code spends most of its time in loop nests and includes both regular and indirect references.

MST is the minimal-spanning tree algorithm from the Olden benchmarks [RCRH95]. MST is dominated by linked-list traversals for lookups in a hash-table. This is an example of a *linked data structure* application, as opposed to the other codes which are based on arrays. MST has an irregular access pattern. We do not run a multiprocessor version of MST because of excessive synchronization overhead.

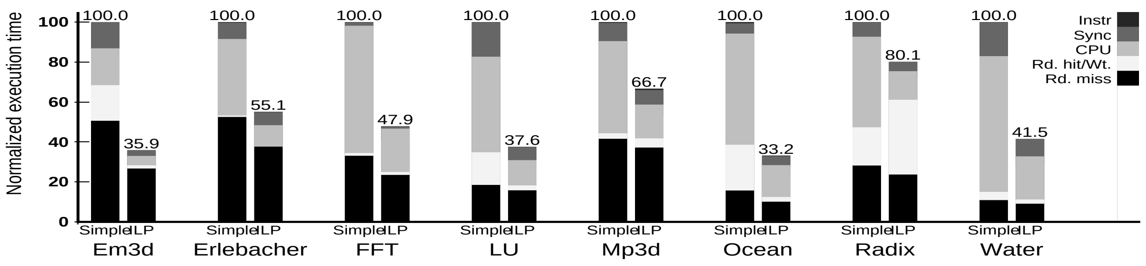


Figure 2.2 : Impact of ILP on multiprocessor execution time and its components.

2.2 Experimental results on multiprocessor system

This section analyzes the impact of ILP techniques on multiprocessor performance by comparing the Simple and ILP multiprocessor systems.

2.2.1 Overall Results

Figure 2.2 illustrates our key overall results for multiprocessor systems. For each application, Figure 2.2 shows the total execution time and its three components for the Simple and ILP systems (normalized to the total time on the Simple system).

This figure show two key trends:

- ILP techniques improve the execution time of all our applications. However, the impact of ILP is variable, ranging from 20% to 67% reduction in execution time. The average reduction in execution time is only 50%.
- The data memory stall component is generally a larger part of the overall execution time in the ILP system than in the Simple system.

We next investigate the reasons for the above trends.

2.2.2 Factors Contributing to ILP Speedup

Figure 2.2 shows that the most important components of execution time are CPU time and data memory stalls. Thus, ILP speedup will be shaped primarily by CPU ILP speedup and

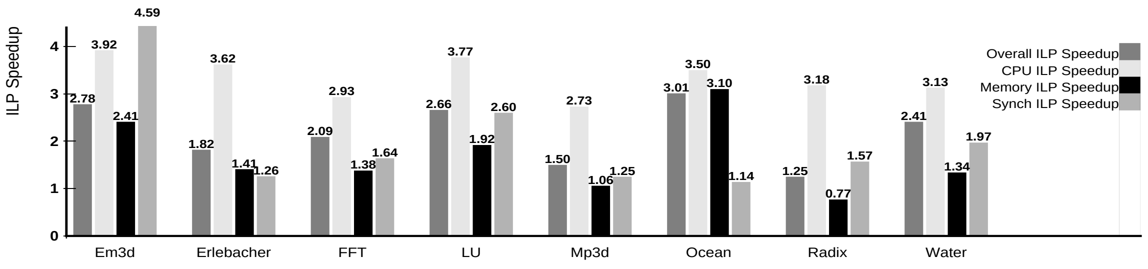


Figure 2.3 : ILP speedup of total execution time and components as seen in multiprocessor system.

data memory ILP speedup. Figure 2.3 summarizes these speedups (along with the total ILP speedup). The figure shows that the low and variable ILP speedup for our applications can be attributed largely to insufficient and variable data memory ILP speedup; the CPU ILP speedup is similar and significant across all applications (ranging from 2.73 to 3.92). Figure 2.2 particularly shows that for most of our applications, data memory stall time is dominated by stalls due to reads that miss in the L2 cache. We therefore focus on the impact of ILP on L2 read misses below.

The data read miss ILP speedup is the ratio of the stall time due to data read misses in the Simple and ILP systems, and is determined by three factors, described below. The first factor tends to increase the speedup, the second tends to decrease it, and the third may either increase or decrease it.

Memory parallelism. Since the Simple system has blocking reads, the entire read miss latency is exposed as stall time. In the ILP system, read misses can be overlapped with other useful work, reducing stall time and increasing the ILP read miss speedup. Figures 2.4(a) and 2.4(b) show instructions being processed in the instruction window (reorder buffer) of an out-of-order issue processor. Instructions in the window can issue and complete out-of-order. However, to maintain precise interrupts, instructions must commit their results and retire from the window in program order after completion [SP88]. (Stores may retire before completing at the memory hierarchy in systems that support write-buffering.)

As described in Section 2.1, an external cache miss may require hundreds of proces-

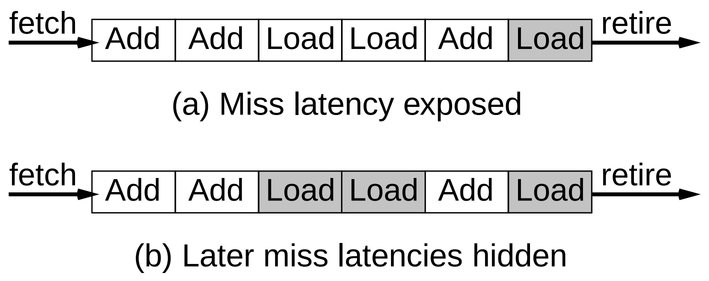


Figure 2.4 : The instruction window (reorder buffer) of an out-of-order issue processor. Older instructions are shown at the right, and cache misses are shaded.

sor cycles. However, current out-of-order processors typically have only 32–80 element instruction windows [Hun95, Kel96, MIP96]. Figure 2.4(a) shows an instruction window in which an outstanding read miss has reached the head of the window and the other instructions in the window are all low latency (such as typical computation and read hits). Then, the other instructions in the window will not be sufficient to completely overlap the oldest read's miss latency. Since the instructions retire in program order, the instruction window fills up and blocks the processor. Thus, ILP features such as out-of-order issue and non-blocking reads are insufficient to hide the read's miss latency.

Figure 2.4(b) shows an otherwise identical instruction window where other independent read misses are scheduled into the window behind the oldest outstanding miss. Here, the later misses issue in parallel with the first miss, and their latencies are overlapped with the stall time of the first miss. Thus, read miss latencies are typically only effectively overlapped in parallel with other read misses. Achieving such *memory parallelism* requires read misses to multiple cache lines to cluster together within the same instruction window. We call this phenomenon *read miss clustering*, or simply *clustering*.

Contention. Compared to the Simple system, the ILP system typically sees longer latencies from increased contention due to the higher frequency of misses, thereby negatively affecting data read miss ILP speedup.

Change in the number of misses. The ILP system may see fewer or more misses than the Simple system because of reordering of memory accesses or speculation, thereby positively or negatively affecting read miss ILP speedup.

We define the following equations to mathematically express the relations of these factors to read miss ILP speedup.

$$Read\ Miss\ Speedup = \frac{M_{Simple} \times l_{Simple}}{M_{ILP} \times l_{ILP,exposed}} \tag{2.1}$$

Equation 2.1 expresses the read miss speedup of these applications in terms of the miss counts and average exposed miss latencies seen in the system. M_{Simple} and M_{ILP} represent the count of L2 data read misses in the Simple and ILP systems, respectively. The term l_{Simple} represents the average data read miss latency in Simple system, which is entirely exposed because of the single-issue, static scheduling and blocking reads of the system. The term $l_{ILP,exposed}$ represents the average exposed data read miss latency in ILP system.

$$Read\ Miss\ Speedup = \frac{M_{Simple}}{M_{ILP}} \times \frac{l_{Simple}}{l_{ILP,exposed}} \tag{2.2}$$

$$Miss\ Factor = \frac{M_{Simple}}{M_{ILP}} \tag{2.3}$$

Equations 2.2 and 2.3 split off the portion of read miss speedup caused by a change in miss count from the portion caused by a change in the exposed latencies. Equation 2.3 defines the *miss factor*, which represents the ratio of the miss count in the Simple and ILP systems. A miss factor greater than 1 contributes positively to read miss speedup, while a lower miss factor hinders read miss speedup.

All of our applications except Radix see a similar number of cache misses in both the Simple and ILP systems. Radix sees 15% fewer misses in ILP because of a reordering of accesses that otherwise conflict in the L2 cache. When the number of misses does not change, the ILP system sees (> 1) read miss ILP speedup if the exposed ILP latency is less than the Simple latency. This occurs when the benefits of memory parallelism in the ILP system outweigh any additional latency from contention.

$$Exposed\ Factor = \frac{l_{ILP,exposed}}{l_{Simple}} \tag{2.4}$$

Equation 2.4 defines the *exposed factor,* which represents the ratio of the average exposed latency in the ILP system to the average Simple system latency. Since the read miss speedup varies inversely with this factor, an exposed factor less than 1 contributes to higher read miss speedup, while a greater exposed factor leads to lower read miss speedup.

$$Exposed\;Factor = \frac{l_{ILP} - l_{ILP,overlapped}}{l_{Simple}} \tag{2.5}$$

$$Exposed\;Factor = \frac{l_{Simple} + l_{ILP,extra} - l_{ILP,overlapped}}{l_{Simple}} \tag{2.6}$$

$$Exposed\;Factor = 1 - \left(\frac{l_{ILP,overlapped}}{l_{Simple}} - \frac{l_{ILP,extra}}{l_{Simple}}\right) \tag{2.7}$$

Equations 2.5–2.7 then expand the exposed factor to express it in terms of l_{Simple}, l_{ILP} (the average data read miss latency in the ILP system), $l_{ILP,overlapped}$ (the average data read miss latency overlapped by the ILP system; equivalent to $l_{ILP} - l_{ILP,exposed}$), and $l_{ILP,extra}$ (the average data read miss latency increase caused by the ILP system; equivalent to $l_{ILP} - l_{Simple}$). Since a low exposed factor leads to higher read miss ILP speedup, Equation 2.7 indicates that the exposed factor will be beneficial whenever the latency overlapped by the ILP system exceeds any increase in latency.

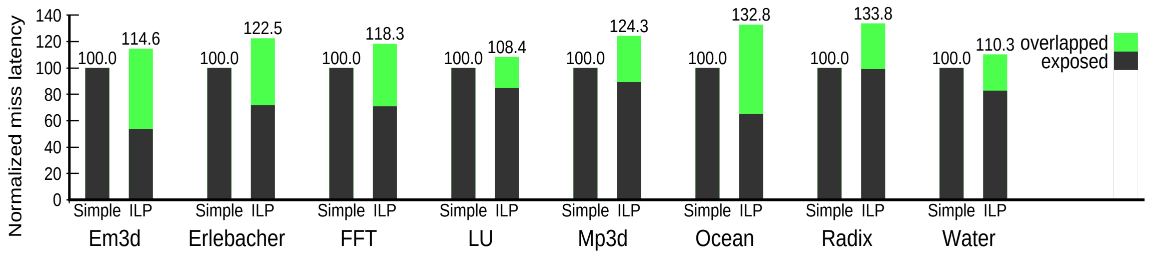
$$Overlapped\;Factor = \frac{l_{ILP,overlapped}}{l_{Simple}} \tag{2.8}$$

$$Extra\;Factor = \frac{l_{ILP,extra}}{l_{Simple}} \tag{2.9}$$

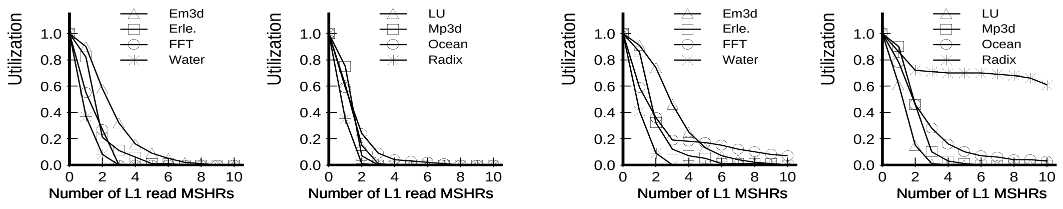
Equations 2.8 and 2.9 extract two new terms from the Equation 2.7: the *overlapped factor* and the *extra factor*. These represent the ratio of the overlapped latency and extra latency in the ILP configuration, respectively, relative to the total Simple latency.

$$Read\;Miss\;Speedup = \frac{Miss\;Factor}{1 - (Overlapped\;Factor - Extra\;Factor)} \tag{2.10}$$

Equation 2.10 summarizes the above equations by expressing the L2 data read miss ILP speedup in terms of three key parameters that we can directly gauge from our simulation results: the miss factor (Equation 2.3), the overlapped factor (Equation 2.8), and the extra factor (Equation 2.9). The miss factor expresses the impact of ILP on L2 read miss count



(a) Impact of ILP on average data read miss latency.



(b) MSHR occupancy due to reads

(c) Total MSHR occupancy

Figure 2.5 : Factors shaping data read miss ILP speedup. The latency figures indicate overall overlap and contention, while the MSHR graphs show specific sources of overlap (read MSHRs) and contention (total MSHRs).

through speculation and reordering, the overlapped factor quantifies the benefits of memory parallelism, and the extra factor indicates contention.

Figure 2.5(a) provides the average L2 read miss latencies for the applications in the Simple and ILP systems, normalized to the Simple system latency. The latency shown is the total miss latency, measured from address generation to data arrival, including the overlapped part (in ILP) and the exposed part that contributes to stall time. The difference in the bar lengths of Simple and ILP indicates the additional latency due to contention in ILP. When normalized to the length of the Simple bar, the exposed part of the ILP bar is the exposed factor of Equation 2.4, the overlapped part of the bar is the overlapped factor of Equation 2.8, and the difference in bar lengths is the extra factor of Equation 2.9.

All of the applications see some latency increase from resource contention in ILP. However, some of the applications (such as Em3d and Ocean) can overlap all of their additional latencies, as well as a large portion of their base (Simple) latencies. Such applications see a high memory ILP speedup. On the other hand, Radix cannot overlap its additional latency.

Radix sees a slight ILP read miss speedup only because of its miss factor; overall, it sees a data memory slowdown because of its increase in contention (explained in more detail below).

We use the data in Figures 2.5(b) and 2.5(c) to further investigate the causes for the memory parallelism and contention-related latencies in these applications. (As described above, all applications studied except for Radix have a miss factor near 1; thus, memory parallelism and contention are the primary factors shaping data read miss ILP speedup.)

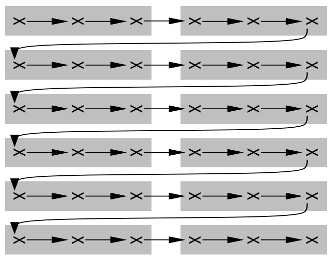
Sources of memory parallelism. Figure 2.5(b) shows the ILP system's L1 MSHR occupancy due to read misses for the applications. Each curve shows the fraction of total time for which at least N MSHRs are occupied by read misses, for each possible N (on the X axis). This figure shows that Em3d achieves significant overlap of read misses, with up to 7 read miss requests outstanding simultaneously at various times. In contrast, Radix almost never has more than 1 outstanding read miss at any time. This difference arises because read misses are clustered together in the instruction window in Em3d, but typically separated by too many instructions in Radix. Among the other applications, Erlebacher, FFT, Mp3d, and Ocean have moderate overlap, while LU and Water have poor overlap.

To understand the sources of poor read miss clustering in typical code, we consider a loop nest traversing a 2-D matrix. Figures 2.6(a) and 2.6(b) show a matrix traversal optimized for spatial locality, following much compiler research. The pseudocode is shown in row-major notation, while the graphic shows the matrix in row-major order, with crosses for data elements and shaded blocks for cache lines. In this row-wise traversal, L successive loop iterations access each cache line, where L is the number of data elements per cache line. While this traversal maximizes spatial locality, it minimizes clustering. For example, an instruction window that holds L or fewer iterations never holds read misses to multiple cache lines, preventing clustering. This problem is exacerbated by larger cache lines or larger loop bodies.

Sources of contention. Figure 2.5(c) extends the data of Figure 2.5(b) by displaying the total MSHR occupancy for both read and write misses. Write miss overlap in the system

```
for(···j++)
 for(···i++)
  ···A[j,i]
```

(a) Pseudocode



(b) Cache behavior

Figure 2.6 : Pseudocode and representation of cache-behavior for a matrix traversal optimized for cache locality. The pseudocode is shown with row-major notation, and the graphic shows the matrix in row-major order. Crosses in the graphic represent matrix elements, and shaded blocks represent cache lines.

is illustrated by the difference between the read MSHR occupancy and the total MSHR occupancy in Figures 2.5(b) and (c). The figure indicates that Radix has high write miss overlap. This overlap does not contribute to an increase in memory ILP speedup since write latencies are already hidden in both the Simple and ILP systems due to release consistency. The write miss overlap, however, increases contention in the memory hierarchy, causing cache hits to see exposed stalls and resulting in a 29% ILP memory slowdown in Radix. Contention-related latencies in most of the other applications come primarily from read misses, but these effects are offset by the resulting benefits in memory stall time.

ILP features can also reduce or hide synchronization time by reducing computation time, overlapping data read misses, and overlapping synchronization with other operations. ILP features can increase synchronization stall time because of additional contention in the memory system. These factors lead to synchronization ILP speedups ranging from 1.07 to 4.59 for our applications; however, synchronization has a small impact on total execution time and thus also contributes little to overall ILP speedup.

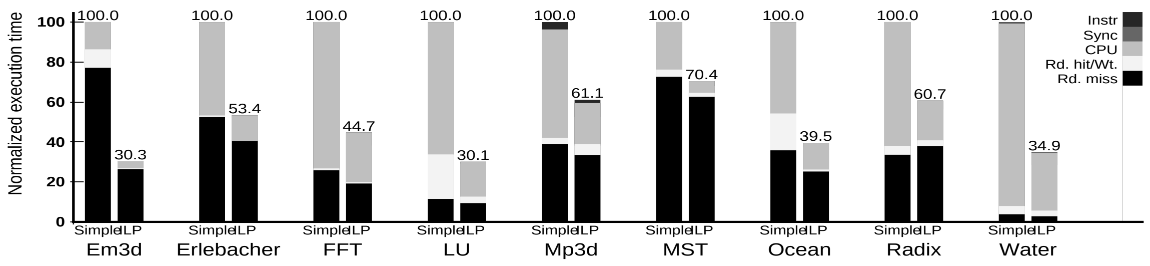


Figure 2.7 : Impact of ILP on uniprocessor execution time and its components.

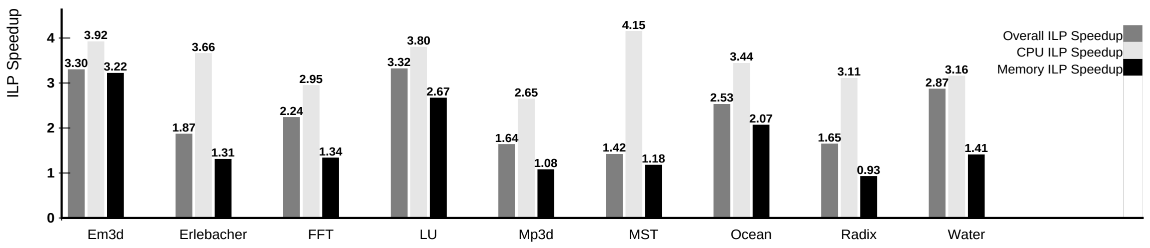
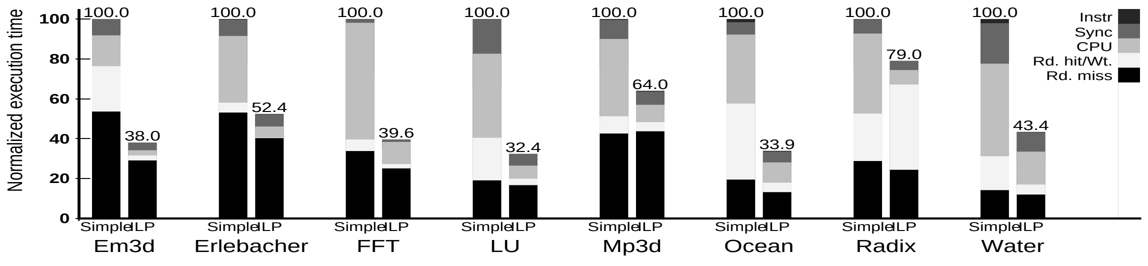


Figure 2.8 : ILP speedup of total execution time and components as seen in uniprocessor system.

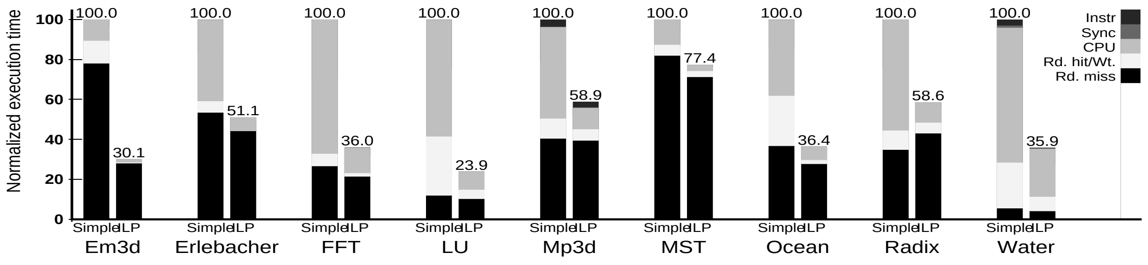
2.3 Experimental results on uniprocessor system

Figure 2.7 shows the impact of ILP on uniprocessor execution time. As in the multiprocessor, this impact is highly variable, providing 30–70% reduction in execution time. However, the average execution time reduction is somewhat higher (56% for the codes which have both uniprocessor and multiprocessor versions, 53% across all codes), as are the individual execution time reductions in all applications with comparable multiprocessor versions except for Ocean.

Figure 2.8 provides the ILP speedups for the uniprocessor configuration for reference. The uniprocessor also generally sees lower memory ILP speedups than CPU ILP speedups. However, the impact of the lower memory ILP speedup is greater in the multiprocessor because the longer latencies of remote misses and increased contention result in a larger relative memory component in the execution time (relative to the uniprocessor). Second, the dichotomy between local and remote miss latencies in a multiprocessor causes some



(a) Multiprocessor



(b) Uniprocessor

Figure 2.9 : Impact of ILP on execution time for systems with faster processors.

applications (most notably, Em3d and LU) to see substantially lower memory ILP speedup in the multiprocessor than in the uniprocessor. For effective latency tolerance, read misses must be overlapped with other read misses *that have similar latencies*; this is trivial in the uniprocessor, but not always true in a CC-NUMA multiprocessor system*.

2.4 Sensitivity to system latencies

Processor speeds and external memory latencies diverge further for processors in the gigahertz frequency range. To model this trend, we also performed experiments that modified our base configuration to include 1 GHz processors without changing any absolute memory hierarchy times (in ns or MHz). The results in Figure 2.9 show behavior qualitatively similar to Figures 2.2 and 2.7, with ILP having a greater impact on CPU time than data

*This would also not necessarily be true in an SMP system, since cache-to-cache transfers have different characteristics from ordinary data transfers.

memory time and data memory time making up a much larger fraction of total execution time in the ILP system than the Simple system.

2.5 Summary and Implications

This study finds that for the applications and systems that we investigate, ILP techniques effectively address the CPU component of execution time, but are less effective in reducing the data memory component of execution time, which is dominated by read misses. This disparity arises in our applications primarily because of insufficient opportunities to overlap multiple read misses and to a lesser extent because of system contention from more frequent accesses in the ILP system. As a result, data read miss latency actually appears as a greater relative performance bottleneck in ILP systems than in previous-generation systems, despite the latency-tolerating techniques incorporated in ILP processors.

These observations suggest that ILP systems have a greater need for both conventional and novel additional techniques to tolerate or reduce memory latency. A commonly used technique for better latency tolerance is software-controlled non-binding prefetching [CKP91, MG91, MLG92, Mow94, TE95, LM96]. Chapter 3 evaluates the interaction of this technique with instruction-level parallelism. Our results also motivate a specific technique novel to ILP processors: application-level clustering of read misses. Chapter 4 proposes and evaluates code transformations that improve miss clustering and also shows that these techniques can facilitate improved software prefetching.

Hardware enhancements can also increase read miss overlap; for example, through a larger instruction window. However, such hardware may be increasingly insufficient or infeasible as the processor-memory speed-gap continues to grow. Targeting contention requires increased system hardware resources or other latency reduction techniques (e.g. [AHAA97]).

Chapter 3

Interaction of Software Data Prefetching with Instruction-Level Parallelism

Chapter 2 shows that the ILP system sees a greater bottleneck from memory latency than the Simple system. To reduce memory stall time, many current processors support software-controlled non-binding prefetching [ERB$^+$95, Hun95, MIP96, Sun97]. With this technique, the compiler or programmer schedules an explicit prefetch instruction for a location that will be accessed by the processor at a later time, with the goal of bringing the location into the processor's cache before it issues a demand memory access for that location [CKP91]. Previous studies have shown that software-controlled non-binding prefetching can eliminate a large fraction of memory stall time in shared-memory multiprocessors and uniprocessors [MG91, MLG92, Mow94, TE95, LM96]. However, the multiprocessor studies used previous-generation processors with single-issue, static scheduling, and blocking reads. Some of the uniprocessor studies modeled ILP processors, but did not specifically relate their benefits or limitations to ILP features. Consequently, such studies do not account for the interactions between software prefetching and the other latency-tolerating techniques already incorporated in ILP-based systems. An analysis of these interactions is required to assess the effectiveness of current software prefetching strategies for state-of-the-art systems.

This chapter seeks to understand how software prefetching interacts with ILP, and to identify its limitations. Section 3.1 gives detailed information on the prefetching algorithms used in this study. Section 3.2 describes the systems, applications, and metrics used in this study. Section 3.3 gives the experimental results of the study. Section 3.4 describes the sensitivity of the results in this study to our simulated system latencies. Section 3.5

summarizes the results of the study and discusses possible solutions to the limitations of prefetching.

3.1 Software prefetching algorithm

The following discusses the best known and implemented software prefetching algorithm for regular memory references, as well as prefetching algorithms that support irregular references.

3.1.1 Algorithm for Adding and Scheduling Software Prefetches

The best known software prefetching algorithm implemented in a compiler is the loop-based algorithm of Mowry et al. [Mow94, MG91, MLG92]. The *analysis* phase of the algorithm identifies the static references which can miss (leading references). Then, the *scheduling* phase uses loop peeling, unrolling, and strip-mining to insert prefetches only for the dynamic instances of leading references that are expected to miss. The innermost loop for a miss reference is software pipelined to schedule a prefetch ahead of the demand access by a certain number of iterations, called the *prefetch distance*. The prefetch distance (d) is computed as:

$$d = \min(L \times \lceil \frac{D}{CL} \rceil, I_i) \tag{3.1}$$

The terms in Equation 3.1 are defined as follows:

D	expected miss latency in cycles
C	estimates the shortest possible path through an iteration (in cycles)
L	number of successive inner-loop iterations that share a cache line
I_i	total number of inner-loop iterations (the upper limit on inner-loop software pipelining)

The term $L \times \lceil \frac{D}{CL} \rceil$ represents the distance needed to completely overlap the prefetch latency.

The software pipelining applied produces a prologue, steady-state, and epilogue from the original inner loop. The prologue consists only of prefetches to cover the data of the first d iterations of the original loop. The steady-state includes both prefetches (scheduled

```
for(j=0;j<$I_o$;j++)
  for(i=0;i<$I_i$;i++)
    $\cdots$A[j,i]
```

(a) Original code

```
for(j=0;j<$I_o$;j++)
  for(i=0;i<$d$;i+=$L$)
    PF(&A[j,i])
  for(i=0;i<$I_i - d$;i++)
    if(i$\bmod L \equiv 0$) PF(&A[j,i+$d$])
    $\cdots$A[j,i]
  for(;i<$I_i$;i++)
    $\cdots$A[j,i]
```

(b) After prefetching

Figure 3.1 : Pseudocode of a 2-D matrix traversal, before and after applying software prefetching. All pseudocode uses row-major notation.

according to the prefetch distance) and computation for $I_i - d$ iterations of the original loop. Either strip-mining, unrolling, or a conditional test is used to insure that a prefetch is issued for a reference only once for every L iterations of the original loop. The epilogue includes only computation for the last d iterations of the original inner loop; no prefetches are issued since all the inner-loop iterations have already been prefetched in the prologue and steady-state. Figures 3.1(a) and 3.1(b) illustrate a matrix traversal before and after applying software prefetching, respectively.

3.1.2 Prefetching Algorithms for Irregular Memory References

The above algorithm handles only affine references, but newer prefetching algorithms have been developed to support two classes of irregular references. The first class supports references formed by indirection through an affine reference [Mow94]. Such references require two prefetches: one for the affine reference (which gives the address of the indirect reference), and one for the indirect reference itself. Since the second prefetch uses the data of the first, the first prefetch must be scheduled before the second according to the prefetch distance. This effectively doubles the prefetch distance and further decreases steady-state length.

More recent research has focused on linked data structures based on pointer-chasing (for example, linked lists). These structures hinder the above prefetching algorithm because their future addresses are more difficult to calculate. Instead, recent software prefetching techniques for linked data structures use *jump pointers*, naturally-occurring or artificially-created pointers to later elements expected in the traversal [LM96, RS99]. A recent study also adds a *prefetch array* containing pointers to the first elements of a linked data structure, thus allowing a prefetching prologue [KDS00]. However, the prefetch arrays themselves can increase the needed working set and cause new cache misses. Prefetch arrays were actually seen in that study to degrade performance on some bandwidth-limited systems.

Each of the various prefetching schemes for linked data structures can be limited by the nature of the structures and the available types of jump pointers. For example, each node of a singly-linked list has a single naturally-occurring jump pointer to the next element in the traversal. Consequently, prefetching techniques will be limited to a prefetch distance of 1 iteration if they only use naturally-occurring jump pointers within the singly-linked list (e.g., *greedy prefetching* [LM96]). Further, even artificial jump pointers may not help in such cases as hash-tables, as these are typically dominated by traversals of very short singly-linked lists and see little work per iteration. The length of the list itself limits the prefetching distances in these cases. We consider two prefetching schemes for linked data structures: greedy prefetching (which uses only naturally-occurring jump pointers) [LM96] and prefetch arrays [KDS00]. Since the application we study for linked data structures is dominated by accesses to hash tables with short lists, schemes based on longer artificial jump pointers are not applicable.

3.2 Measuring prefetching effectiveness

This section describes the systems, metrics, and prefetching algorithms we use to evaluate the effectiveness of software prefetching for the codes that we investigate.

3.2.1 Simulated systems

The systems simulated in this chapter are identical to the ILP and Simple systems simulated in Chapter 2. Again, the goal of the comparison between Simple and ILP systems is to understand how ILP techniques interact with software prefetching. Both systems are simulated using RSIM, a detailed execution-driven simulator for ILP multiprocessors.

Both systems support software-controlled non-binding prefetching, with both exclusive-mode and shared-mode prefetches. Prefetch instructions retire as soon as they reach the top of the instruction window, but occupy a slot in the processor's memory queue until they are issued. Prefetches are not dropped even if resource constraints block their issue, and prefetched lines are brought into the highest level of the memory hierarchy.

3.2.2 Performance Metrics

In addition to execution time and its components, measured as in Chapter 2, we also consider statistics specifically related to prefetching here. For detailed analysis, we divide prefetches into various categories, as summarized in Table 3.1. These categories are *useful*, in which a prefetched line arrives on time and is used by a demand access; *late*, in which a prefetched line arrives after the demand access; *early*, in which the prefetched line is replaced or invalidated before use, or is never used; and *unnecessary*, in which the line being prefetched is already present in either the cache or the MSHRs. We also discuss the component of execution time caused by *late prefetch stalls* – references which were prefetched late and have come to the head of the instruction window before their prefetches have responded with data.

3.2.3 Applications and Prefetching

This chapter studies the nine applications studied in Chapter 2, with identical input sizes and system configurations. Since we do not have a SPARC compiler that implements software-prefetching for C programs, we insert prefetches by hand, conservatively following the algorithms in Section 3.1, with one exception: we assume that locality is preserved

Category	Description
Useful	Arrives on time; used by demand access
Late	Arrives after demand access (i.e. latency only partially hidden)
Early	Replaced or invalidated before use (or unused)
Unnecessary	Hits in cache or MSHR

Table 3.1 : Classification of prefetches

across synchronization when this is beneficial.

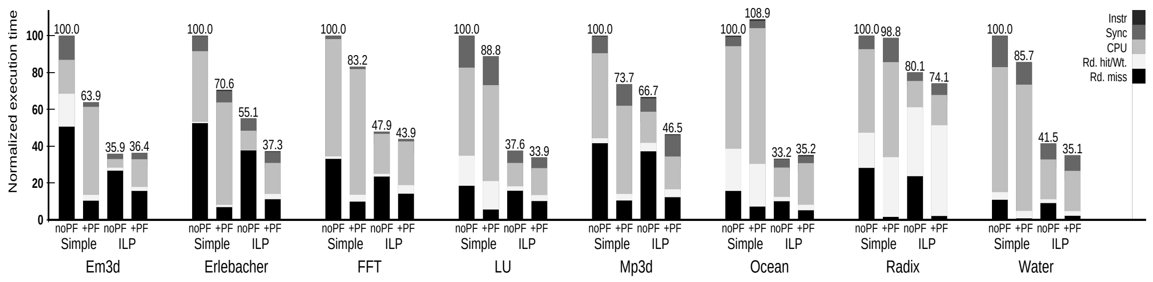
We assume a default prefetch distance of 400 instructions to model the representative latency seen in the ILP system. Since write latency is already hidden in our release-consistent systems, we do not issue prefetches for cache lines that are only written in a given loop nest; however, we use exclusive prefetches for reads of lines that will also be written.

Erlebacher, FFT, LU, Ocean, Radix, and Water use only prefetching for array references formed through affine functions on the controlling loop variables. (The only significant set of read miss references in these codes of a non-affine form is in the bit-reverse phase of FFT, and is not analyzable for prefetching.) Em3d and Mp3d include references formed by indirection through an affine reference. MST is based on linked data structures (short linked lists for hash table lookups), and is prefetched with each of greedy prefetching and prefetch arrays.

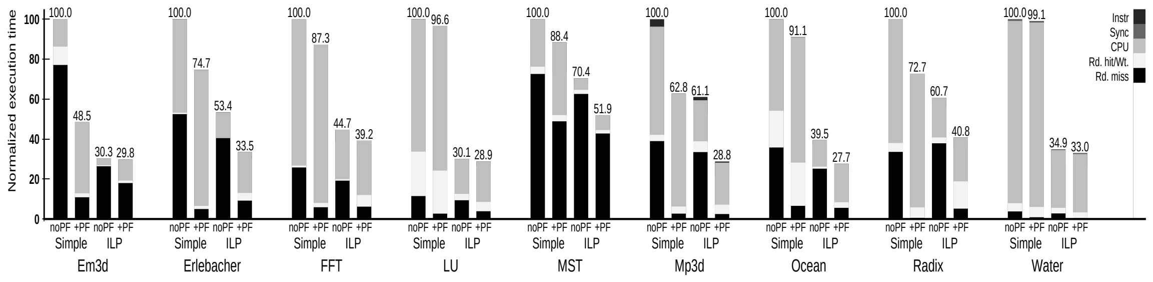
3.3 Experimental results

3.3.1 Overall Results

Figure 3.2 graphically presents the key results from our experiments. The figure shows the execution time (and its components) for each multiprocessor and uniprocessor application on Simple and ILP, both without software prefetching (noPF) and with software prefetching (+PF). Execution times are normalized to the time for the application on Simple without prefetching. (For MST, this chart only shows prefetch arrays. Greedy prefetching is described in Section 3.3.3.)



(a) Multiprocessor



(b) Uniprocessor

Figure 3.2 : Interaction of ILP with software prefetching in Simple and ILP systems with base configuration.

Software prefetching achieves significant reductions in execution time on the ILP system for six multiprocessor applications (ranging 7% to 32%) and seven uniprocessor applications (ranging 5% to 53%). In most of the cases, these execution time reductions are similar to or greater than those in the Simple system for these applications. However, software prefetching is typically less effective at reducing memory stalls on ILP than on Simple. In the ILP multiprocessor, the data memory stall time is reduced an average of 40% (ranging 16%–63%), compared to an average 56% for the Simple multiprocessor (ranging 21%-85%). In the uniprocessor, prefetching reduces the ILP data memory stall time an average of 49% (ranging 28–81%) compared to an average of 61% in the Simple system (ranging 24–88%). The net effect is that even after prefetching is applied, the percentage of time spent on data memory stalls is still greater on the ILP system than on the Simple system for most of the applications that we study. The ILP multiprocessor still sees an average of 39% data memory stall time, compared to only 16% in the Simple multiprocessor.

Application	% exec. time reduction		% data stall reduction		% remaining in data stall		% late pref. stall time	
	Simp.	ILP	Simp.	ILP	Simp.	ILP	Simp.	ILP
Em3d	36	-1	80	37	21	49	4	33
Erlebacher	29	32	85	63	12	38	9	29
FFT	17	8	60	24	16	43	8	23
LU	11	10	40	26	24	40	6	21
Mp3d	26	30	68	60	19	36	0	0
Ocean	-9	-6	21	34	28	23	0	3
Radix	1	7	28	16	34	69	0	0
Water	14	15	68	58	6	13	0	4
Average	19 (for 7 of 8)	17 (for 6 of 8)	56	40	16	39	4	13

Table 3.2 : Detailed data on effectiveness of software prefetching in multiprocessor system.

Similarly, the ILP uniprocessor sees 40% data memory stall time on the average, compared to a Simple uniprocessor average of 20%. (The uniprocessor numbers are slightly higher than the multiprocessor numbers primarily because of MST, which spends 59% of its Simple time and 86% of its ILP time in data memory stalls after prefetching.) For reference, Table 3.2 provides application-by-application details on the effectiveness of software prefetching in the multiprocessor.

3.3.2 Factors Contributing to the Effectiveness of Software Prefetching

We next identify several factors that affect software prefetching for ILP systems. Effective software prefetching depends on issuing each prefetch sufficiently in advance of its demand access and on the benefits of latency tolerance exceeding any overheads from this technique. The prefetching algorithms of Section 3.1 often suffer from the following limitations:

Prologue late prefetches. The prefetching algorithm of Mowry et al. generates a prologue to cover the references of the first few steady-state iterations [Mow94, MG91, MLG92]. However, the prologue is unlikely to completely overlap their prefetch latencies since it contains no computation. Thus, the data requested by some of these prefetches arrives after their demand references, leading to *late* prefetches. The first steady-state iteration will typically see such prologue late prefetches, with at least one full reference latency exposed. Since the prologue is invoked each time the inner loop starts, prologue late prefetches arise on each outer-loop iteration of a loop nest*. (Prefetching schemes that do not include a prologue would also see such exposed latencies at the beginning of each steady-state.) This problem can affect both Simple and ILP systems, but deficiencies in addressing data memory stalls have a greater relative impact on the ILP system than on the Simple system because of the greater percentage of execution time spent on data memory stalls in the ILP system.

Short steady-states. Because of the deficiencies in the prologue, only the prefetches in the steady-state are expected to be scheduled the correct distance ahead of their demand references. Thus, the effectiveness of the prefetching algorithm depends on most inner-loop iterations fitting in the steady-state. Using the calculation of the prefetch distance d given in Equation 3.1, we see that the steady state has $\max(0, I_i - L \times \lceil \frac{D}{CL} \rceil)$ iterations. The steady state is even shorter for prefetching of indirect references, since they effectively double the needed prefetch distance. Additionally, any technique that aims to address late prefetches in the prologue or steady-state by statically or dynamically increasing the prefetch distance further shrinks the steady state.

The above shows that a large steady-state requires an inner loop with a large number of iterations or a large amount of computation per iteration. Many loops do not meet these requirements. First, loops blocked for cache locality or fine-grained communication tend to have few iterations. Second, each loop iteration often includes little actual computation. Inner-loop unrolling also cannot help, since increases in C are offset by decreases in I_i. Modern ILP processors further exacerbate these problems by aggressively executing mul-

*Previous work by Saavedra et al. attempted to reduce the number of separate prologue invocations by merging prologues into earlier epilogues, but found no performance benefits because of an increase in cache conflicts [SMP+96].

tiple instructions per cycle, reducing C. Further, processor clock speeds tend to improve faster than memory latencies, increasing D.

Hard-to-prefetch references. Some references have addresses that are difficult to calculate sufficiently in advance, making these references hard to prefetch. Examples include greedy prefetching of linked lists (which limits the available prefetch distance to one iteration) and references left unprefetched because their address generation sequences are unanalyzable. ILP systems may be able to tolerate these latencies through the use of non-blocking read misses, if the applications can support the needed memory parallelism.

Instruction overhead. Prefetch instructions and their required address generation overhead increase dynamic instruction count, increasing the amount of time spent on CPU instructions. As a result, the latency-tolerance benefits of prefetching do not always translate directly to overall execution time benefits. Multiple issue and multiple functional units tend to reduce the negative impact of instruction overhead for ILP systems.

Other factors. Early prefetches can hinder demand accesses by invalidating or replacing needed data from the same or other caches without providing any benefits in latency reduction. An ILP system allows less time between a prefetch and its subsequent demand access, potentially decreasing the likelihood of an intervening invalidation or replacement. Some of our applications see such a reduction in early prefetches.

In ILP, prefetch instructions can be speculatively issued past a mispredicted branch. Speculative prefetches can potentially hurt performance by bringing unnecessary lines into the cache, or by bringing needed lines into the cache too early. Speculative prefetches can also help performance by initiating a prefetch for a needed line early enough to hide its latency. Some prefetching schemes (such as prefetch arrays) can also introduce additional data fetches, increasing resource contention.

3.3.3 Application-Specific Details

Em3d sees short steady states because of its short traversals of its `from_nodes` structures and its indirect prefetching. Erlebacher, FFT, and LU all have important phases blocked for cache locality and/or load balance: the fine-grained wavefront pipeline in Erlebacher, the transpose in FFT, and the entire code of LU. None of these blocked portions achieves a steady-state. Thus, all of these codes see a disproportionately large fraction of their prefetches in the prologue. Consequently, they spend a large fraction of their time stalling on late prefetches.

Mp3d sees an increase in total prefetches in ILP because of misspeculations. However, most of the misspeculated prefetches are to lines also accessed on the correct path; thus, the later reference on the correct path will merely coalesce with the earlier prefetch or use its data. These mispredicted prefetches thus increase the number of unnecessary prefetches and cause some contention for cache ports, but do not have a substantially negative impact on performance here.

In MST, prefetch arrays provide substantial benefits but also increase the needed working set and cause new misses. These new misses cannot be prefetched well because the index into the array is calculated through a hash function just before the traversal, and the arrays are too short to allow prefetching of the remaining elements. Additionally, the prefetch arrays always fetch several items of the list being traversed, even though a hash match might arise within the first 1 or 2 list entries. Thus, the remaining prefetches are useless, increasing contention, early prefetches, damaging prefetches, and CPU overhead without tolerating any latency. Other linked data structure prefetching schemes such as greedy prefetching do not increase the needed working set. Figure 3.3 includes results with greedy prefetching as well, with GPF and PFA denoting greedy prefetching and prefetch arrays, respectively. Greedy prefetching suffers from hard-to-prefetch references, since the prefetch distance for its linked-list traversals is limited to 1 iteration, and each iteration has very little computation. (Schemes that use longer artificial jump pointers would be inapplicable, since the linked-lists in MST are very short.)

Ocean sees a large number of unnecessary prefetches because of its inter-nest locality and stencil computations with variable-length inner loops. Our conservative locality analysis does not detect either of these types of locality, and thus schedules unnecessary

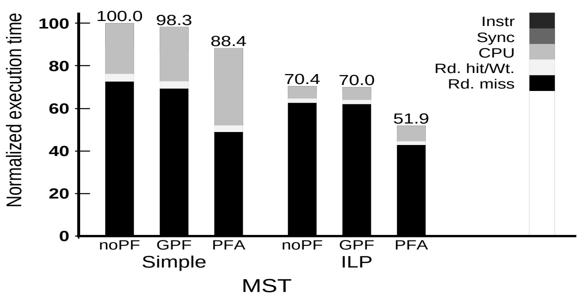


Figure 3.3 : Comparing greedy prefetching and prefetch arrays for MST on simulated Simple and ILP uniprocessors. Execution times are normalized to the Simple system without prefetching.

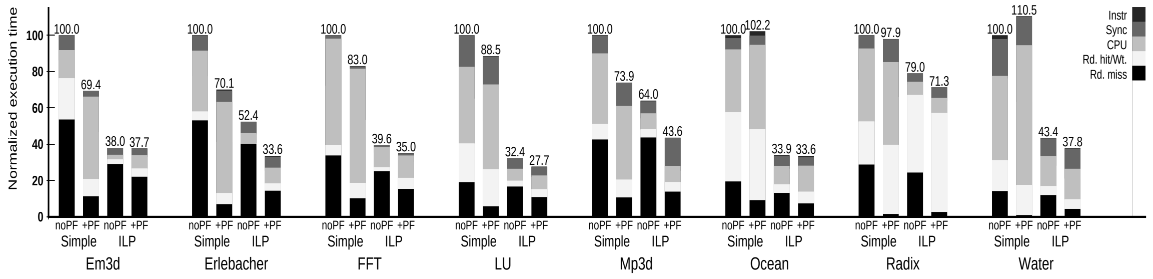
prefetches. These add a significant instruction overhead for both the prefetch instructions and their address-generation overhead.

In Radix, the deficiencies in prefetching are exposed as cache hit stalls. These ultimately result from write MSHR saturation (as described in Chapter 2) and subsequent write miss blockage. Blocked write misses can eventually cause read hits to stall as well for resources such as memory queue entries and cache ports.

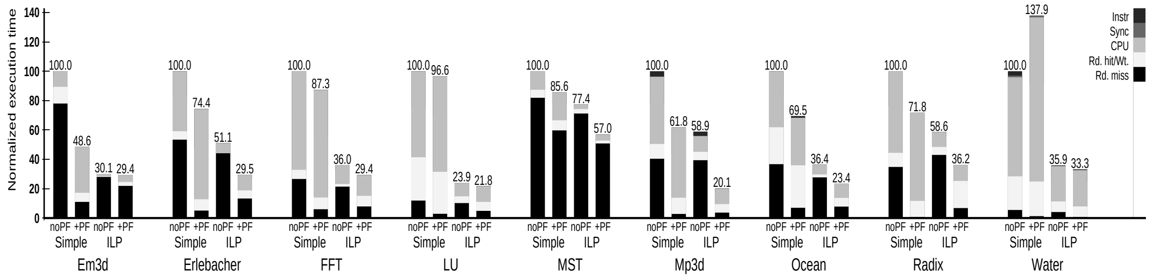
In Water, the data memory overhead of the base code is not sufficient to justify the instruction overhead of prefetching.

3.3.4 Impact of Software Prefetching on Execution Time

Despite its reduced effectiveness in addressing memory stall time, software prefetching achieves significant execution time reductions with ILP in most of our applications for two main reasons. First, memory stall time contributes a larger portion of total execution time in ILP. Thus, even a reduction of a small fraction of memory stall time can imply a reduction in overall execution time similar to or greater than that seen in Simple. Second, ILP systems see less instruction overhead from prefetching compared to Simple systems, because ILP techniques allow the overlap of these instructions with other computation.



(a) Multiprocessor



(b) Uniprocessor

Figure 3.4 : Interaction of ILP with software prefetching in Simple and ILP systems with faster processors.

3.4 Sensitivity to System Latencies

Figure 3.4 show the effectiveness of prefetching for Simple and ILP processors in a system configuration with 1 GHz processors and all other absolute memory hierarchy times (in ns or MHz) unchanged from the base. The problems of late prefetches and short steady states are now further exacerbated because of the increase in the number of cycles required to overlap the prefetch latencies.

3.5 Summary and Implications

While software prefetching improves memory system performance with ILP processors, it does not change the memory-bound nature of these systems for most of the applications studied primarily because of late prefetches in the prologue, short prefetching steady-states, and references that are hard to prefetch. Additionally, the instruction overhead of prefetching offsets some of its benefits in latency tolerance (although this overhead is less pronounced on the ILP system than on the Simple system). As described in Section 3.3.2, ILP systems have the potential to tolerate latencies left behind by prefetching if the applications provide memory parallelism. Thus, we expect that ILP-specific techniques to increase read miss parallelism will allow software prefetching to take advantage of overlapped accesses. Chapter 5 shows how transformations to improve memory parallelism can help to address some of the limitations of prefetching.

Chapter 4

Code Transformations to Improve Memory Parallelism

Chapter 2 established that applications often have insufficient potential to overlap multiple read misses, thus limiting the system's ability to hide read miss latency. Chapter 3 examined software prefetching as a potentially beneficial latency-tolerance technique, but found some important limitations to its effectiveness. This chapter discusses an alternative latency-tolerance technique that specifically seeks to take advantage of ILP through code transformations that increase an application's potential for read miss overlap. In particular, read misses that are to be overlapped must appear together within the same instruction window. We continue to refer to this phenomenon as read miss clustering or simply as clustering.

Section 4.1 explains in detail why applications currently see a low degree of read miss clustering, and then illustrates how read miss clustering can be improved through code transformation. Section 4.2 presents a formal analysis and transformation framework for miss clustering suitable for implementation in a compiler. Section 4.3 describes the methodology and metrics used in the experimental analysis of these techniques. Section 4.4 evaluates the benefits of these transformations on a latency-detection microbenchmark. Section 4.5 presents and analyzes experimental performance results for these transformations on applications running in a simulated environment, and Section 4.6 confirms the benefits of these transformations on a real system. Section 4.7 summarizes these transformations and discusses the implications and insights of this study.

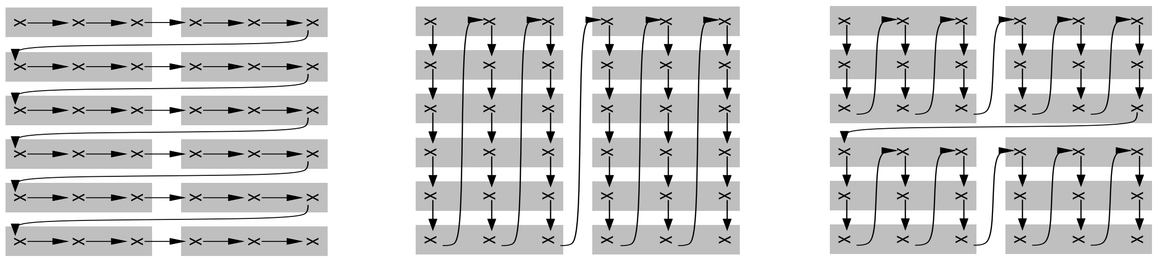

(a) Exploits locality (b) Exploits clustering (c) Exploits both

Figure 4.1 : Impact of matrix traversal order on miss clustering. Crosses represent matrix elements, and shaded blocks represent cache lines. The matrix is shown in row-major order.

4.1 Improving Read Miss Clustering in ILP Processors

Section 2.2.2 showed that long-latency data read misses should appear clustered together within the instruction window of an out-of-order processor in order to effectively exploit latency tolerance through memory parallelism. This section uses an illustrative example to demonstrate how code transformations can improve read miss clustering by reordering the cache misses in an application and scheduling them closer together.

To understand the sources of poor read miss clustering in typical code, we consider a loop nest traversing a 2-D matrix. Figure 4.1 graphically represents three different matrix traversals. The matrix is shown in row-major order, with crosses for data elements and shaded blocks for cache lines. Figure 4.2 relates these matrix traversals to code generation, with pseudocode shown in row-major notation.

Figures 4.1(a) and 4.2(a) show a matrix traversal optimized for spatial locality, following much compiler research. In this row-wise traversal, L successive loop iterations access each cache line, where L is the number of data elements per cache line. While this traversal maximizes spatial locality, it minimizes clustering. For example, an instruction window that holds L or fewer iterations never holds read misses to multiple cache lines, preventing clustering. This problem is exacerbated by larger cache lines or larger loop bodies.

Read miss clustering can be maximized by a column-wise traversal, since

```
for(···j++)
 for(···i++)
  ···A[j,i]
```
(a) Base code

```
for(···i++)
 for(···j++)
  ···A[j,i]
```
(b) Interchange

```
for(···jj+=N)
 for(···i++)
  for(j=jj;j<jj+N;j++)
   ···A[j,i]
```
(c) Strip-mine and interchange

```
for(···j+=N)
 for(···i++){

  ···A[j,i]
  ···A[j+1,i]
  ······
  ···A[j+N-1,i]
```
(d) Unroll-and-jam

Figure 4.2 : Pseudocode for Figure 4.1 matrix traversals (row-major notation).

successive iterations held in the instruction window access different cache lines. Figures 4.1(b) and 4.2(b) show such a column-wise traversal, obtained by applying loop interchange to the code in Figure 4.2(a). Each cache line is now accessed on multiple successive outer-loop iterations. However, the traversal passes through every row before revisiting an older cache line. If there are more rows than cache lines, this traversal could lose all cache locality, potentially overwhelming any performance benefits from clustering.

The above example suggests a tradeoff between spatial locality (favored by current code-generation schemes) and miss clustering. We seek a solution that achieves the benefits of clustering while preserving spatial locality. A column-wise traversal can maximize clustering; however, it must stop before losing locality. In particular, the column-wise traversal can stop as soon as the miss clustering resources are fully utilized. For example, a processor that allows ten simultaneous cache misses sees the maximum memory parallelism when ten independent miss references are clustered. The traversal could then continue in a row-wise fashion to preserve locality. Figure 4.1(c) shows a matrix traversal that exploits clustering and locality in this way. Figure 4.2(c) expresses this traversal by applying strip-mine and interchange to Figure 4.2(a).

Since the column-wise traversal length (N) of Figure 4.2(c) is based on the hardware resources for overlap (typically 3–12 today), the strip size is small, and the innermost loop can be fully unrolled. Figure 4.2(d) shows the result of that unrolling. Now, the code reflects the transformation of unroll-and-jam applied to Figure 4.2(a). This transformation unrolls an outer loop and fuses (jams) the resulting inner loop copies into a single inner loop. Previous work has used unroll-and-jam for scalar replacement (replacing array memory operations with register accesses), better floating-point pipelining, or cache locality [AC72, CCK88, CK94, Nic87, Car96]. Using unroll-and-jam for read miss clustering requires different heuristics, and may help even when the previously studied benefits are unavailable.

We prefer to use unroll-and-jam instead of strip-mine and interchange for two reasons. First, unroll-and-jam allows us to exploit additional benefits from scalar replacement. Second, unroll-and-jam does not change the inner-loop iteration count. The shorter inner loops of strip-mining can negatively impact techniques that target inner loops, such as dynamic branch prediction. By increasing inner-loop computation without changing the iteration count, unroll-and-jam can also help software prefetching (to be discussed in Chapter 5).

Unroll-and-jam creates an N-way unrolled steady-state, followed by an untransformed postlude of leftover iterations. To enable clustering in the postlude, we simply interchange the postlude when possible. This should not degrade locality, since the postlude originally has fewer outer-loop iterations than the unroll-and-jam degree.

4.2 Analysis and Transformation Framework

This section provides a formal framework to apply memory parallelism transformations in a compiler.

4.2.1 Dependences that Limit Memory Parallelism

We first describe a dependence framework to represent limitations to memory parallelism. As in other domains, dependences here indicate reasons why one operation will not issue in parallel with another. However, these dependences are not ordinary data dependences, since memory operations can be serialized for different reasons. We build this framework to gauge performance potential, not to specify legality. Thus, we optimistically estimate memory parallelism and specify dependences only when their presence is known. The transformation stages must then use more conventional (and conservative) dependence analysis for legality. For simplicity, we only consider memory parallelism dependences that are either loop-independent or carried on the innermost loop. We can then exploit previous work with the same simplification [CCK88].

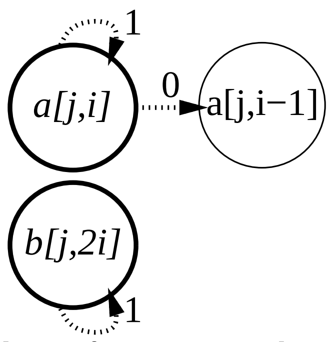
Since we focus on parallelism among read misses, we first require locality analysis to determine which static references can miss in the external cache (*leading references*), and which leading references are known to exhibit spatial locality across successive iterations of the innermost loop (*inner-loop self-spatial locality*). Known locality analysis techniques can provide the needed information [WL91]. Currently, we do not consider cache conflicts in our analysis and transformations.

We use the above information to identify limitations to read miss parallelism. We focus on three kinds of limitations, which we call *cache-line dependences*, *address dependences*, and *window constraints*.

Cache-line dependences. If a read miss is outstanding, then another reference to the same cache line simply coalesces with the outstanding miss, adding no read miss parallelism. Thus, we say that there is a *cache-line dependence* from memory operation A to B if A is a leading reference and a miss on A brings in the data of B. The cache-line dependence is a new resource dependence class, extending input dependences to support multi-word cache lines.

The following code illustrates cache-line dependences. In all examples, leading references known to have inner-loop self-spatial locality will be italicized, while other leading references will be boldfaced. The accompanying graph shows static memory references as nodes and dependences as edges. Each edge is marked with the inner-loop dependence distance, the minimum number of inner-loop iterations separating the dependent operations specified.

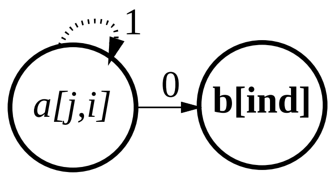
```
for(···j++)
 for(···i++)
  b[j,2*i] = b[j,2*i] + a[j,i] + a[j,i-1]
```



Note that there are no cache-line dependences from one leading reference to another; such a dependence would make the second node a non-leading reference. Additionally, any leading reference with inner-loop self-spatial locality has a cache-line dependence onto itself. That dependence has distance 1 for any stride, since the address of the miss reference will be closer to the instance 1 iteration later than to an instance farther away.

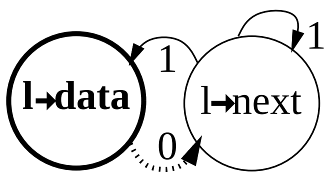
Address dependences. There is an address dependence from memory operation A to B if the result of A is used in computing the address of B, serializing B behind A. Address dependences typically arise for irregular accesses, such as indirect addressing or pointer-chasing. The following code segments show address dependences. The graphs show address dependences as solid lines and cache-line dependences as dotted lines. The first example shows the indirect addressing typical of sparse-matrix applications.

```
for(···j++)
 for(···i++){
  ind = a[j,i]
  sum[j] = sum[j] + b[ind]
```



The above shows one leading reference that exhibits cache-line dependences, connected through an address dependence to another leading reference. The following code shows address dependences from pointer dereferencing.

```
for(···i++){
 l = list[i]
 for(···l=l→next)
  sum[i] += l→data
```



The above assumes that the `data` and `next` fields always lie on the same cache line and that separate instances of `l` are not known to share cache lines. Even though `l`$\rightarrow$`next` is a non-leading reference, it is important since a dependence flows from this node to the leading reference.

Window constraints. Even without other dependences, read miss parallelism is limited to the number of independent read misses in the loop iterations simultaneously held in the instruction window. We do not include these resource limitations in our dependence graphs, since they can change at each stage of transformation. We will, however, consider these constraints in our transformations.

Control-flow and memory consistency requirements may also restrict read miss parallelism. We do not consider these constraints, since their performance impact can be mitigated through well-known static or dynamic techniques such as speculation. However, these dependences may still affect the legality of any code transformations.

Of the three dependence classes that we consider (cache-line, address, and window), only address dependences are true data-flow dependences. Window constraints can be eliminated through careful loop-body scheduling, possibly enhanced by inner-loop unrolling. Such scheduling would aim to cluster together misses spread over a long loop body. Loop-carried cache line dependences can be made loop-invariant through inner-loop unrolling by a multiple of L, where L iterations share each cache line. Then, no cache line is shared across unrolled loop iterations. However, the inner-loop unrolling degree may need to go as high as $N \times L$ to provide clustered misses to N separate cache lines. This can be excessive, particularly with long cache lines. We therefore leave these loop-carried cache-line dependences in place and seek to extract read miss parallelism with less code expansion through outer-loop unroll-and-jam.

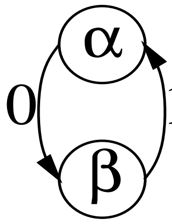
We will address memory parallelism limitations in loop nests by first resolving recurrences (cycles in the dependence graph), and then handling window constraints. A loop nest may suffer from one or both problems, and recurrence resolution may create new window constraints.

4.2.2 Background on Floating-point Pipelining

Section 4.1 showed in an informal way that unroll-and-jam could be used to increase miss clustering without degrading locality. Unroll-and-jam has previously been used to improve floating-point pipelining in the presence of inner-loop floating-point recurrences [CCK88, Nic87]. This section explains how unroll-and-jam has been used in this context.

Consider an inner loop that carries a floating-point recurrence (a cycle of true dependences). The operations of later iterations can stall for the results of earlier iterations, preventing maximum pipeline throughput. Further, inner-loop unrolling and scheduling cannot help, as later inner-loop iterations are also in the cycle. The following pseudocode has an inner-loop recurrence between statements α and β. The graph shows floating-point true dependences and dependence distances.

```
for(···j++)
 for(···i++){
  α: b[j,i] = a[j,i-1] + c[i]
  β: a[j,i] = b[j,i] + d[i]
```



The above recurrence has two floating-point operations, and needs 1 iteration for a complete cycle (the sum of the dependence distances). Thus, the system must serialize 2 floating-point operations (the number in the recurrence) to complete 1 iteration (the length of the cycle), regardless of the pipelining supported. Callahan et al. described floating-point recurrences as follows [CCK88]*:

- l_p : number of stages in the floating-point pipeline
- ρ : ratio of the number of nodes (static floating-point operations) in the inner-loop recurrence (R) to the number of iterations to traverse the cycle (τ)
- f : static count of floating-point operations in an innermost loop iteration

*Their notation was slightly different, with $\#R$, $\tau(R)$, $\rho(R)$, and f_L instead of R, τ, ρ, and f, respectively.

Since R floating point operations must be serialized in τ iterations, the recurrence requires at least ρ pipeline latencies ($= l_p \times \rho$ pipeline stages) per iteration. Without dependences, each iteration would require only the time of f pipeline stages. Thus, the recurrence limits pipeline utilization to $\frac{f}{l_p \times \rho}$. Unroll-and-jam introduces independent copies of the recurrence from separate outer loop iterations, increasing f without affecting ρ [CCK88]. To fill the pipeline, unroll-and-jam must be applied until $f \geq l_p \times \rho$. (The maximum ρ should be used for a loop with multiple recurrences, since each recurrence limits pipeline utilization.)

Certain dependences can prevent unroll-and-jam, but they are not directly related to the recurrences targeted. Previous work more thoroughly discusses legality and the choice of outer loops to unroll for deeper nests [CCK88, CK94, Nic87].

4.2.3 Resolving Memory Parallelism Recurrences

We seek to use unroll-and-jam to target loop nests with memory-parallelism recurrences, which arise for such common access patterns as self-spatial or pointer-chasing leading references. We map memory parallelism to floating-point pipelining, exposing several key similarities and differences between these problems. This section thus shows how to automate the process described in Section 4.1, which used unroll-and-jam to increase miss clustering without degrading locality.

In Section 4.2.2, unroll-and-jam used only the number of pipeline stages, not the latency. The number of pipeline stages simply represents the number of floating-point operations that can be processed in parallel. Thus, we can map this algorithm to memory parallelism: the goal is to fully utilize the miss clustering resources, not to schedule for some specific miss latencies. Here, l_p corresponds to the maximum number of simultaneous outstanding misses supported by the processor. The rest of the mapping is more difficult, as not all memory operations utilize the resources for miss parallelism — only those instances of leading references that miss at run-time do. This difference affects ρ and f.

Characterizing recurrences (ρ). We refer to recurrences with only cache-line dependences as *cache-line recurrences* and recurrences with at least one address dependence as *address recurrences*. Recurrences with no leading miss references are irrelevant here and can be ignored, since they do not impact read miss parallelism.

As discussed in Section 4.2.2, ρ is computed from two values: R and τ. We count only leading references in R, as only these nodes can lead to serialization for a miss. We count τ as in Section 4.2.2, since this paramter specifies the number of iterations over which all the miss instances in a recurrence are serialized. Our discussion has focused on tolerating only read miss latencies, since write latencies are typically hidden through write buffering. However, our algorithm must count both read and write miss references in R and f. This is because cache resources that determine the maximum number of outstanding misses (such as the miss status holding registers) are typically shared between reads and writes[†]. Nevertheless, we will not apply unroll-and-jam on an outer loop if it only adds write misses, since write-buffering is typically sufficient to hide their latencies.

Counting memory parallelism candidates (f). For floating-point pipelines, the f parameter counts the static instructions in an innermost loop iteration. We cannot use this same definition here for two key reasons, described below.

Dynamic inner-loop unrolling. An out-of-order instruction window of W instructions dynamically unrolls a loop body of i instructions by $\lceil \frac{W}{i} \rceil$. (For simplicity, we assume no outer-loop unrolling, although this could arise if the inner loop had fewer than $\lceil \frac{W}{i} \rceil$ iterations.) Such unrolling exposes no additional steady-state parallelism for loops with address recurrences, since these are analogous to the recurrences of floating-point pipelining. However, this unrolling can actually break cache-line recurrences. In particular, if L_m successive iterations share a cache line for leading reference m, dynamic inner-loop unrolling creates $\lceil \frac{W}{iL_m} \rceil$ independent misses from the original recurrence. Leading references outside recurrences can also contribute multiple outstanding misses ($L_m = 1$, since

[†]This is not true of no-write-allocate caches. However, we are most concerned with lower levels of the cache hierarchy that incur long latency misses. Such caches are usually write-allocate.

no cache-line sharing is known). Thus, we define C_m, the number of copies of m that can contribute overlapped misses:

$$C_m = \begin{cases} \lceil \frac{W}{iL_m} \rceil & \text{loop with no address recurrences} \\ 1 & \text{otherwise} \end{cases} \tag{4.1}$$

Miss patterns. A simple count of leading references can overestimate memory parallelism, since not all leading reference instances miss in the cache. To determine which leading reference instances miss together, we must know the miss patterns (sequences of hits and misses) for the different leading references and their correlation with each other. Such measures can be difficult to determine in general. In this work, we make some simple assumptions, described below.

We split the leading references into two types: regular (arrays indexed with affine functions of the loop indices) and irregular (all others). For regular references, we assume that at least some passes through the inner loop experience misses on each cache line accessed, and that different regular leading references experience misses together. These assumptions lead to maximum estimated parallelism for regular leading references.

For irregular accesses, the miss pattern is not typically analyzable. We assume no correlation, either among instances of the same reference or across multiple references. Thus, we only need to know the overall miss rate, P_m, for each reference m. P_m can be measured through cache simulation or profiling. These assumptions allow more aggressive transformation than the more common assumption of no locality for irregulars.

We can now estimate the f parameter, accounting for both dynamic inner-loop unrolling and miss patterns:

$$f = f_{reg} + f_{irreg} \tag{4.2}$$

$$f_{reg} = \sum_{m \in RLR} C_m \tag{4.3}$$

$$f_{irreg} = \lceil \sum_{m \in ILR} P_m \times C_m \rceil \tag{4.4}$$

We split f into regular and irregular components, with RLR and ILR the sets of regular and irregular leading references respectively. The terms C_m in Equation 4.3 and $P_m \times C_m$

in Equation 4.4 give the maximum expected number of overlapped misses to separate cache lines contributed by leading reference m. We round up f_{irreg} to insure that some resources are held for irregular references when they are present.

The floating-point pipelining algorithm applied unroll-and-jam until $f \geq l_p \times \rho$. We should be more conservative for memory parallelism, as the cache can see extra contention when the resources for outstanding misses (MSHRs) fill up. Thus, we aim to apply unroll-and-jam as much as possible while maintaining $f \leq l_p \times \rho$ (using the maximum ρ for the loop).

After applying unroll-and-jam, we must recompute f for two reasons. First, unroll-and-jam can introduce new leading references and increase the iteration size. On the other hand, some leading reference copies may become non-leading references because of scalar replacement or group locality. For similar reasons, we must repeat the locality and dependence analysis passes.

Since f varies as described above, we may need to attempt unroll-and-jam multiple times with different unrolling degrees to reach our desired f. We can limit the number of invocations by choosing a maximum unrolling degree U based on the resources for memory parallelism, code expansion, register pressure, and potential for cache conflicts. If we unroll only one outer loop, we can choose the unrolling degree by binary search, using at most $\lceil \log_2 U \rceil$ passes [CK94]. Generalized searching for unrolling multiple outer loops can follow the strategies described in previous work [CK94]. We also refer to previous work for legality issues [CCK88, CK94, Nic87]. We add only that we prefer not to unroll-and-jam loops that only expose additional write miss references, since buffering can hide write latencies.

To revisit the motivating example of Section 4.1, note that the matrix traversal of Figure 4.2(a) has a cache-line recurrence with $\rho = 1$. Since modern caches typically have lines of length 32 to 128 bytes, L_m typically ranges from 4 to 16 for stride-1 double-word accesses. With current instruction window sizes ranging from 32 to 80, $\lceil \frac{W}{iL_m} \rceil$ is thus most likely to be 1 for a loop body with a moderate amount of computation. Thus, $f = f_{reg} = 1$

for the initial version of this loop. This example has no scalar replacement opportunities, so each recurrence copy created by unroll-and-jam contributes a leading reference to the calculation of f. Assuming U is chosen to be at least l_p, the search algorithm will find that unroll-and-jam by l_p leads to $f = l_p \times \rho$. Since $\rho = 1$ in this example, the final version of this loop will see l_p static leading references in an iteration.

4.2.4 Resolving Window Constraints

We now address memory parallelism limitations from window constraints. These can arise for loops with or without recurrences. Further, recurrence resolution can actually create new window constraints, since unroll-and-jam can spread its candidates for read miss parallelism over a span of instructions larger than a single instruction window. We proceed in two stages: first using loop unrolling to resolve any inter-iteration window constraints, then using local instruction scheduling to resolve intra-iteration constraints.

As discussed in Section 4.2.3, an instruction window of W instructions dynamically unrolls an inner-loop body of i instructions by $\lceil \frac{W}{i} \rceil$. Inter-iteration window constraints arise when the independent read misses in $\lceil \frac{W}{i} \rceil$ iterations do not fill the resources for memory parallelism (typically because of large loop bodies). Since any recurrences have already been resolved, we can now use inner-loop unrolling to better expose independent misses to the instruction scheduler. We can directly count the maximum expected number of independent misses in $\lceil \frac{W}{i} \rceil$ iterations, using the miss rate P_m to weight the irregular leading references. We then unroll until the resources for memory parallelism are filled, recomputing the exposed independent miss count after each invocation of unrolling.

Now we resolve any intra-iteration window constraints stemming from loop bodies larger than a single instruction window (possibly because of unroll-and-jam or inner-loop unrolling). In such cases, the instruction scheduler should pack independent miss references in the loop body close to each other. The technique of balanced scheduling can provide some of these benefits [KE93, LE95], but may also miss some opportunities since it does not explicitly consider window size. Nevertheless, this heuristic worked well for the code sequences we examined. More appropriate local scheduling algorithms remain the subject of future research.

4.3 Measuring the Impact of Optimizations

This chapter focuses only on ILP-based systems, since memory parallelism transformations rely on the non-blocking reads and out-of-order scheduling features of ILP processors. This section discusses the architectures, applications, and metrics used for evaluation of these transformations.

4.3.1 Evaluation environments

Since we now focus only on ILP multiprocessors, we have the option of performing our experiments both in simulation and on current state-of-the-art hardware. We focus primarily on simulation results both for the purposes of detailed analysis and to show the benefits that these transformations can provide for future-generation systems. The only difference between the base simulated configuration used in this chapter and the ILP configuration of Chapters 2 and 3 is the use of realistic functional unit latencies. Since the comparisons in this chapter only deal with out-of-order scheduled ILP processors, the configuration can model realistic functional unit latencies without excessive dependence on the compilers. The functional unit latencies are chosen based on reasonable current system parameters and are summarized in Table 4.1. This modification did not significantly impact the results.

We also confirm the benefits of our transformations on a Convex Exemplar SPP-2000 system with 180 MHz HP PA-8000 processors [Hew97, Hun95]. Each processor has 4-way issue, a 56-entry out-of-order instruction window, and a 1 MB single-level data cache with 32-byte lines and up to 10 simultaneous misses outstanding. The system's memory banks support skewed interleaving with a cache-line granularity [HJ87]. The system supports the sequential consistency model with latency tolerance via speculative read execution, write prefetching, and write buffering [Lam79, GGH91, RPAA97]. Although the Exemplar supports a CC-NUMA configuration with SMP hypernodes, we perform our tests within

Functional Unit Latencies	
Operation type	**Latency**
Address generation	1 cycle
Most integer	1 cycle
Integer multiply/divide	7 cycles
Most floating-point	3 cycles
Floating-point divide	16 cycles
Floating-point square root	33 cycles

Table 4.1 : Functional unit latencies for the base system configuration used in simulating impact of memory clustering transformations. All other system parameters are identical to the ILP system of Chapters 2 and 3, as given in Table 2.1.

a hypernode. Thus, we use the Exemplar as an SMP and do not consider data placement issues. We use the `pthreads` library for our explicitly parallel applications.

4.3.2 Evaluation Workload

We evaluate our clustering transformations using a latency-detection microbenchmark and five scientific applications. Table 4.2 summarizes the evaluation workload for the simulated and real systems. The number of processors used for the multiprocessor experiments is based on application scalability, with a limit of 16 on the simulated system and 8 on the real machine. Each code is compiled with the Sun SPARC SC4.2 compiler for the simulation and the Exemplar C compiler for the real machine, using the `-xO4` optimization level for the Sun compiler and `+O3` optimization level for the Exemplar compiler. We do not consider prefetching in this chapter; the interaction of clustering and software prefetching is described in Chapter 5. We incorporate miss clustering transformations by hand, following the algorithms presented in this chapter.

Latbench is based on the `lat_mem_rd` kernel of `lmbench` [MS96]. `lat_mem_rd` sees inner-loop address recurrences from pointer-chasing. Latbench wraps this loop in an outer loop that iterates over different pointer chains, with no locality in or across chains. The pseudocode, given below, shows code added for Latbench in sans-serif.

```
for (j=0;j<N;j++){        // j-loop added for latbench
  p = A[j];               // j-index added for latbench
  for(i=0;i<I;i++)
   p = p→next             // serialized misses
  USE(p)                  // keeps p live
```

Latbench is clustered with unroll-and-jam. As in `lat_mem_rd`, looping overhead is minimized by unrolling the innermost loops to include 1000 pointer dereferences in each loop body, for both the base and clustered versions.

Each of our regular codes (Erlebacher, FFT, LU, and Ocean) has only cache-line sharing dependences. These codes are clustered with unroll-and-jam and postlude interchanging.

Em3d has both cache-line and address dependences, but only cache-line recurrences. MST has address recurrences from its hash-table lookups. Both codes are clustered using unroll-and-jam. The dominant loop nests in both applications have variable inner-loop lengths, so only the minimum length seen in the unrolled copies is fused (jammed). Each copy completes its remaining length separately. We assumed that the outer loops were explicitly identified as parallel to enable transformation despite the pointer references.

Mp3d has no recurrences, but sees poor miss clustering because of large loop bodies. Thus, inner-loop unrolling and aggressive scheduling can provide clustering here, as discussed in Section 4.2.4. We assumed that the dominant `move` loop was explicitly marked parallel. Despite having been transformed to reduce true and false sharing (Section 2.1), Mp3d does not see substantial multiprocessor speedup on the Convex Exemplar. As a result, it is only run as a uniprocessor code on the real machine.

The other applications studied in Chapter 2 (Radix and Water) are not included in this study. The execution time of Radix is dominated by its key permutation phase, in which the MSHRs are quickly filled up by write accesses (Section 2.2.2). As a result, there are no MSHRs or system bandwidth available for read miss clustering. Water is dominated by a single loop that iterates over individual molecules being simulated. However, this loop cannot be unrolled and rescheduled in the fashion of Mp3d because each molecule access in Water is surrounded by a lock. Our transformations are not currently suited for the possible

Microbenchmark	**Input Size**	**Processors**
Latbench	6.4M data size	1

Application	**Input Size**	**Processors**
Em3d	32K nodes, deg. 20, 20% rem.	1,16
Erlebacher	64x64x64 cube, block 8	1,16
FFT	65536 points	1,16
LU	256x256 matrix, block 16	1,8
Mp3d	100K particles	1,8
MST	1024 nodes	1
Ocean	258x258 grid	1,8

(a) Simulated system

Microbenchmark	**Input Size**	**Processors**
Latbench	40M data size	1

Application	**Input Size**	**Processors**
Em3d	100K nodes, deg. 20, 20% rem.	1,8
Erlebacher	128x128x128 cube, block 8	1,8
FFT	4M points	1,8
LU	4224x4224 matrix, block 128	1,8
Mp3d	2M particles	1
MST	1024 nodes	1
Ocean	1026x1026 grid	1,8

(b) Convex Exemplar

Table 4.2 : Data set sizes and number of processors for experiments on simulated and real systems.

problems presented by synchronization.

For all of the applications, all the clustering transformations only target expected read misses. However, leading references for lines that were only written were also counted in order to account for their MSHR usage.

4.3.3 Evaluation metrics

The most important metric for the effectiveness of these techniques is their impact on overall execution time. Additionally, since we focus on memory parallelism, we will also consider the impact of read miss clustering on the data memory component, which is dominated by read miss time. The equations of Chapter 2 are not directly applicable to determining the read miss speedup of a specific optimization in an ILP based configuration because those equations assume that the entire read miss latency of the base configuration is exposed.

To adapt the equations of Chapter 2 to properly handle an ILP base system, we first replace the references to Simple and ILP systems with the more general base ($base$) and optimized (opt) systems, respectively. We then add in new terms such as $l_{base,overlapped}$ to represent the latency overlapped by ILP techniques in the base system. In this way the equations change as follows:

$$Read\ Miss\ Speedup = \frac{M_{base} \times l_{base,exposed}}{M_{opt} \times l_{opt,exposed}} \tag{4.5}$$

$$Read\ Miss\ Speedup = \frac{M_{base}}{M_{opt}} \times \frac{l_{base,exposed}}{l_{opt,exposed}} \tag{4.6}$$

$$Miss\ Factor = \frac{M_{base}}{M_{opt}} \tag{4.7}$$

$$Exposed\ Factor = \frac{l_{opt,exposed}}{l_{base,exposed}} \tag{4.8}$$

$$Exposed\ Factor = \frac{l_{opt} - l_{opt,overlapped}}{l_{base} - l_{base,overlapped}} \tag{4.9}$$

$$Exposed\ Factor = \frac{l_{base} + l_{opt,extra} - l_{opt,overlapped}}{l_{base} - l_{base,overlapped}} \tag{4.10}$$

$$Exposed\ Factor = \frac{1 - \left(\frac{l_{opt,overlapped}}{l_{base}} - \frac{l_{opt,extra}}{l_{base}}\right)}{1 - \frac{l_{base,overlapped}}{l_{base}}} \tag{4.11}$$

$$Overlapped\ Factor = \frac{l_{opt,overlapped}}{l_{base}} \tag{4.12}$$

$$Extra\ Factor = \frac{l_{opt,extra}}{l_{base}} \tag{4.13}$$

$$Self\text{-}overlapped\ Factor = \frac{l_{base,overlapped}}{l_{base}} \tag{4.14}$$

This set of equations continues to define the overlapped factor and extra factor in terms of the *total* base latency. Additionally, Equation 4.14 defines a *self-overlapped factor*, which accounts for the overlapping seen in the base. The self-overlapped factor has a value from 0 to 1 inclusive, with 0 representing no effective overlap and 1 representing complete latency hiding. The following equation summarizes the impact of each factor on overall read miss speedup.

$$Read\ Miss\ Speedup = \frac{Miss\ Factor \times (1 - Self\text{-}overlapped\ Factor)}{1 - (Overlapped\ Factor - Extra\ Factor)} \tag{4.15}$$

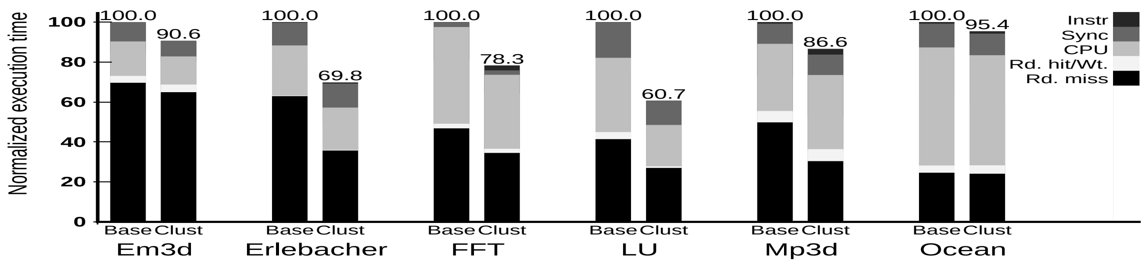
A higher self-overlapped factor increases the exposed factor, thereby decreasing the possible read miss speedup achieved by clustering. This suggests that applications with substantial overlap in the base ILP system (as judged from the overlapped factor of Chapter 2) will have more difficulty seeing benefits from an additional latency-tolerating optimization.

As in the equations of Chapter 2, these equations show that read miss speedup can be achieved either by reducing the number of misses or by decreasing the exposed latency. The code transformations that we consider can lead to a reduction in the miss count through scalar replacement of misses or by causing the hardware to reorder accesses and change cache behavior. Either way, a miss factor close to 1 implies that read miss speedup comes primarily from a decrease in exposed latency.

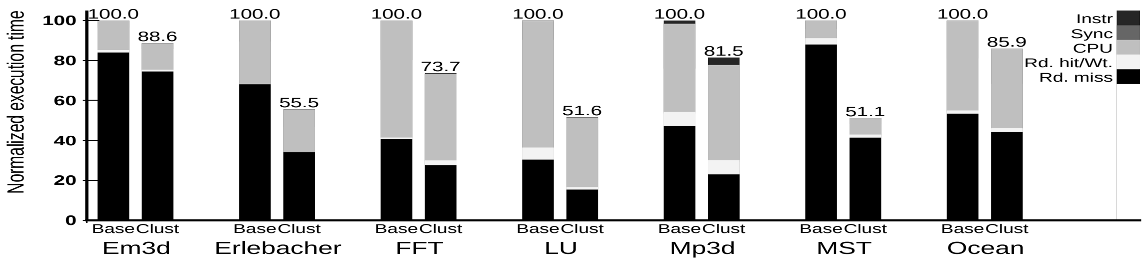
4.4 Performance of Latbench Microbenchmark

The base Latbench of Section 4.3.2 indicates an average read miss stall time of 171 ns on the simulated system (identical to `lat_mem_rd`). Clustering drops the average stall time caused by each read miss to 32 ns, a speedup of 5.34X. On the Convex Exemplar, clustering reduces the average stall time for each miss from 502 ns to 87 ns, providing a speedup of 5.77X.

These results indicate the potential gains from memory parallelism transformations, but also show some bottlenecks, since the speedups are less than 10 (the number of simulta-



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 4.3 : Impact of clustering transformations on application execution time.

neous misses supported by each processor). Our more detailed statistics for the simulated system show that clustering increases contention, increasing average *total* latency from 171 ns to 316 ns (with total latencies measured from address generation to completion). Further, bus and memory bank utilization for the simulated system both exceed 85% after clustering. Thus, a further increase in speedup would require greater bandwidth at both the bus and the memory. (Total latencies and utilizations are not directly measurable for the real machine.)

4.5 Simulated Application Performance

4.5.1 Impact of clustering on execution time

Figure 4.3 shows the impact of the clustering transformations on application execution time for the base simulated system. Figure 4.3(a) shows multiprocessor experiments, while Figure 4.3(b) shows uniprocessor experiments. The execution time of each application is shown both before and after clustering (Base/Clust), normalized to the given application and system size without clustering.

Overall, the clustering transformations studied provide from 5–39% reduction in multiprocessor execution time for these applications, averaging 20%. The multiprocessor benefits in Erlebacher and Mp3d come almost entirely from reducing the memory stall time. (Mp3d sees some degradation in CPU time because of no scalar replacement or pipeline improvement and slightly worse return-address prediction.) Em3d, FFT, and LU see benefits split between memory stall time and CPU time; unroll-and-jam helps the CPU component through better functional unit utilization (in all three) and through scalar replacement (in FFT and LU). By speeding up the data producers in LU, the clustering transformations also reduce the synchronization time for data consumers. Our more detailed statistics show that the L2 miss count is nearly unchanged in all applications, indicating that locality is preserved and that scalar replacement primarily affects cache hits. The multiprocessor version of Ocean sees the smallest overall benefits from the clustering transformations, and those benefits are almost entirely the result of scalar replacement on CPU time. The potential benefits of clustering transformations on Ocean are limited because the base version of this application already sees some memory parallelism. Additionally, clustering increases conflict misses in this application. All applications except Ocean see more multiprocessor execution time reduction from the newly exposed benefits in read miss clustering than the previously studied benefits in CPU time.

The uniprocessor sees slightly larger overall benefits from the clustering transformations, ranging from 11–49% (average 30%). The speedup of data memory stalls is greater in the uniprocessor than in the multiprocessor, as the uniform latency and bandwidth characteristics of the uniprocessor better facilitate overlap. (For perfect overlap, misses of the *same latency* must be clustered together. Our algorithms do not consider this issue.) However, since the uniprocessor spends a substantially smaller fraction of time in data memory stalls than the multiprocessor for FFT and LU, the clustering transformations' benefits for these codes are predominantly in the CPU component. The only uniprocessor-specific

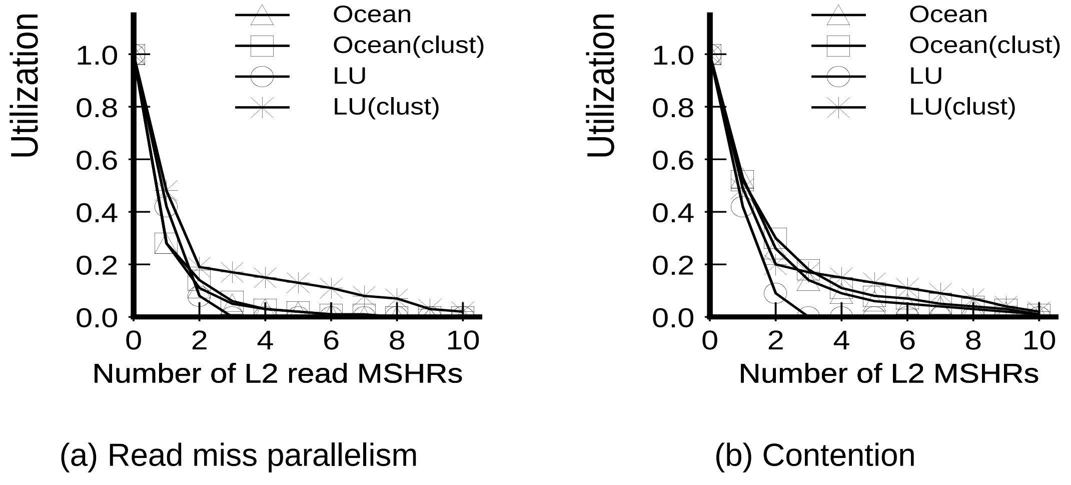


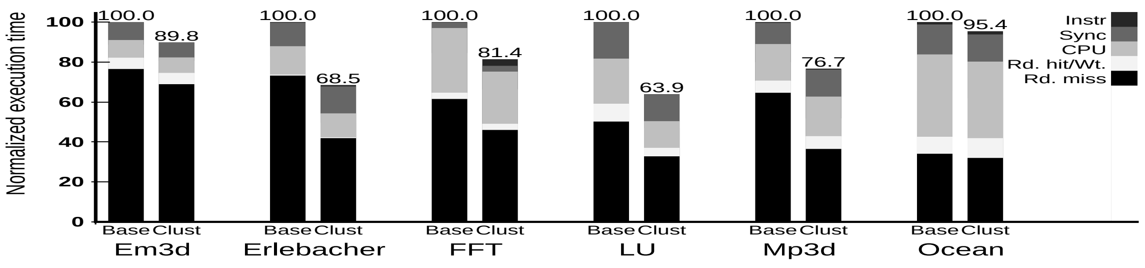
Figure 4.4 : Factors shaping memory parallelism (read L2 MSHR utilization) and contention (total L2 MSHR utilization).

application, MST, sees nearly all of its improvement in its dominant data memory stall component. The other applications respond to the clustering transformations qualitatively in the same way as in the multiprocessor system.

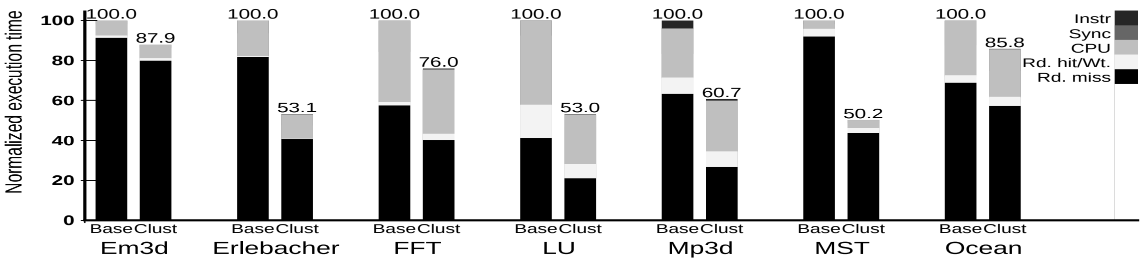
All simulation experiments show few instruction memory stalls. Thus, the code added by our transformations does not significantly impact I-cache locality for these loop-intensive codes.

4.5.2 Memory Parallelism and Contention

The MSHR utilization graphs of Figure 4.4 depict the sources of memory parallelism and contention for the simulated multiprocessor runs of Ocean and LU, the two extreme applications with regard to improvement from the transformations. (The corresponding data is not directly measurable on the real system.) Figure 4.4(a) indicates read miss parallelism, showing the fraction of total time for which at least N L2 MSHRs are occupied by read misses for each possible N on the X axis. The clustering transformations only slightly improve read miss parallelism for Ocean, since the stencil-type computations in Ocean give even the base version some clustering. In contrast, the transformations convert LU from



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 4.5 : Execution times on simulated system with faster processors.

a code that almost never had more than 1 outstanding read miss to one with 2 or more outstanding read misses 20% of the time and up to 9 outstanding read misses at times.

Figure 4.4(b) shows the total L2 MSHR utilization, including both reads and writes. This indicates contention, measuring how many requests use the memory system at once. In the unclustered version of the code, LU sees little additional contention from writes, while Ocean sees some write contention. By only targeting read misses, the clustering transformations do not further increase contention caused by writes. Any negative effects from increased read contention are largely offset by the performance benefits of read miss parallelism.

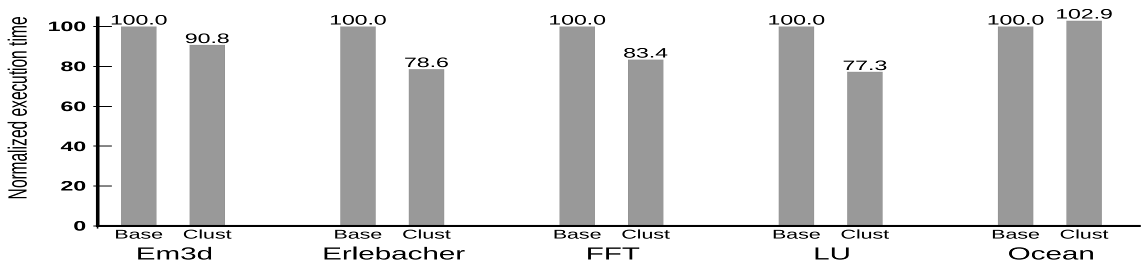
4.5.3 Sensitivity to system parameters

Processor speeds and external memory latencies diverge further for processors in the gigahertz frequency range. To model this trend, we also performed experiments that modified our base configuration to include 1 GHz processors without changing any absolute memory hierarchy times (in ns or MHz). These results, shown in Figure 4.5, are similar in overall execution time reduction to the base configuration. In particular, this configuration sees execution time reductions averaging 21% in the multiprocessor (ranging 5–36%) and averaging 33% in the uniprocessor (ranging 12–50%). However, the larger fraction of memory stall time in these systems allows memory parallelism to provide more of the total benefits than in the base. Thus, targeting memory parallelism becomes more important for configurations with gigahertz processors.

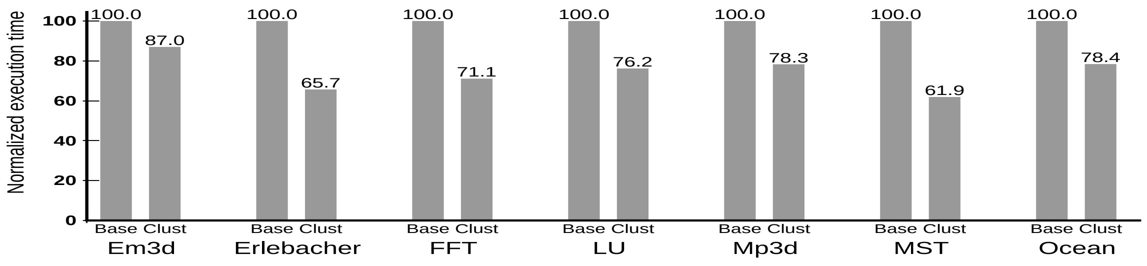
4.6 Exemplar Application Performance

This section repeats our experiments on a Convex Exemplar to confirm the benefits of clustering on a current system. There are numerous differences between the simulated system and the real machine, as described in Section 4.3.1. Figure 4.6 shows that clustering also provides significant benefits in the real system. The multiprocessor and uniprocessor execution time reductions from clustering range from 9–38% for all but the multiprocessor version of Ocean, which sees a 3% degradation. (More detailed execution time breakdowns are not available on the real system.)

As in the simulation, the uniprocessor sees larger benefits than the multiprocessor. Although the SMP hypernode does not distinguish local and remote misses, there are still latency and bandwidth differences between ordinary and cache-to-cache misses. The multiprocessor configuration, however, sees lower benefits in the real machine than in simulation. This discrepancy could arise for two reasons. First, the memory banks and address busses of an SMP are shared by all the processors, potentially increasing multiprocessor contention relative to the simulated CC-NUMA system. Second, as discussed in Section 4.5.1, clustering adds new conflict misses to the multiprocessor version of Ocean. Such conflict misses would further increase the multiprocessor contention, causing a slight performance degradation. The clustering transformations provide similar uniprocessor execution time benefits in both the simulation and the real machine for all applications except



(a) Multiprocessor



(b) Uniprocessor

Figure 4.6 : Impact of read miss clustering on Convex Exemplar execution time.

LU. The difference in LU may stem from the different interleaving policies of the simulated and real machines [Soh93, HJ87].

4.7 Summary and Discussion

This chapter shows that miss clustering can provide benefits in execution time on both the real and simulated machine. Chapter 5 extends this study by showing the interaction between clustering and software prefetching. Other promising avenues for miss clustering include a thorough compiler implementation based on the algorithms presented here, investigation of other application domains, and analysis of lower bandwidth systems to determine how a clustering algorithm should account for their resource constraints. Alternative approaches to clustering besides unroll-and-jam should also be considered in the future. For example, clustering can also take advantage of loop fusion to increase memory parallelism even for unnested loops.

At the most fundamental level, this study shows that previously-known compiler transformations can be targeted to improve data read miss parallelism and thus better exploit non-blocking cache hierarchies. In addition to the implications for compiler design, the benefits shown here also have implications for hardware design. The use of appropriate loop transformations can expose more data parallelism within an instruction window for a single thread. This potential for data parallelism to be exploited at the single-thread level by the compiler suggests that plans to greatly increase effective hardware instruction window size or to seek parallelism from multiple threads can be considered less pressing. Further, this work motivates hardware that supports large numbers of outstanding read misses, and shows that applications can be tailored to use a large number of MSHRs.

Other recent studies of locality-enhancing algorithms also have interesting implications for clustering. For example, dynamic data packing and memory forwarding can improve execution time by increasing cache locality and reducing bandwidth needs for irregular code [DK99, LM99]. At the same time, such transformations can reduce memory parallelism by bringing originally disparate memory locations together into a common cache line. Just as we have shown that miss clustering can improve performance by exploiting multiple instances of naturally self-spatial references, it seems plausible that clustering techniques could also be applied to references made to exhibit spatial locality through transformation. Alternatively, the locality transformations could themselves be extended to incorporate support for miss clustering; data that are likely to be accessed together can first be aggregated and then mutually spread over multiple cache lines.

Chapter 5

Comparing and Combining Read Miss Clustering and Software Prefetching

Chapters 3 and 4 consider two latency-tolerance techniques, software-prefetching and read miss clustering. Both read miss clustering and software prefetching can hide a significant fraction of miss latency in multiprocessor and uniprocessor systems. However, it is unclear which technique is superior or if these techniques can be combined profitably, since each targets the same type of latencies and uses the same system resources. This chapter shows that these techniques are actually mutually beneficial, each helping to overcome the performance limitations of the other.

Experimental results show that clustering alone outperforms prefetching alone for most of the applications and systems that we study, and that the combination of read miss clustering and prefetching yields better execution time benefits than either technique alone in most cases. The combination of clustering and prefetching yields a significant improvement in latency tolerance over prefetching alone (the state-of-the-art implemented in systems today), with an average of 21% reduction in execution time across all cases studied in simulation and an average of 16% reduction in execution time for 5 out of 10 cases on the Exemplar. The experimental results also show reductions in execution time relative to clustering alone averaging 15% for 6 out of 11 cases in simulation and 20% for 6 out of 10 cases on the Exemplar.

Section 5.1 describes limitations in each of read miss clustering and software prefetching, and Section 5.2 explains why these two techniques can combine in a mutually beneficial way. Section 5.3 describes the methodology used in this study. Section 5.4 compares and combines clustering and prefetching in simulation, and Section 5.5 confirms the bene-

	Read Miss Clustering	**Prefetching**
Reorders references?	Yes	No
Loop-level	Outer loop	Inner loop
Maximum latency hidden	Factor of $\frac{N-1}{N}$ with N simultaneous outstanding misses	All in steady-state
Legality issues?	Limited by dependences	None
Dynamic instruction overhead	Minimal	Address computation, prefetch instructions
Extraneous fetches	None	Possible for some schemes
Instruction cache footprint	Increased	Increased
Resources consumed	Miss buffers, bandwidth	Miss buffers, bandwidth

Table 5.1 : Key features of read miss clustering and software prefetching.

fits of these techniques on a real system. This chapter focuses on a comparison of the two techniques and their combination; the previous chapters have already covered each scheme in isolation.

5.1 Limitations of Read Miss Clustering and Software Prefetching

Table 5.1 lists the key features of read miss clustering and software prefetching that determine performance. This section discusses problems that limit the effectiveness of each of these latency-tolerance techniques, as well as limitations shared by both schemes.

5.1.1 Limitations of Read Miss Clustering

Read miss clustering achieves latency tolerance by restructuring the code to encourage parallelism among demand read misses. The following concerns can limit the applicability and effectiveness of clustering.

Incomplete latency hiding. A demand read miss that has not completed by the time it reaches the head of the instruction window will block retirement, incurring data memory stall time. Read miss clustering alone usually leaves some latencies exposed, since later misses are hidden behind the stall time of earlier misses.

Legality issues. Certain dependences can prevent the inner-loop fusion step required by unroll-and-jam [CCK88, CK94]. These constraints can limit the applicability of read miss clustering.

5.1.2 Limitations of Software Prefetching

Section 3.3.2 discussed several important limitations of current software prefetching algorithms. This chapter particularly focuses on the impact of prologue late prefetches, short steady-states, hard-to-prefetch references, and the instruction overhead of prefetching.

5.1.3 Limitations Shared by Clustering and Prefetching

Both read miss clustering and software prefetching can increase resource contention, increasing total system latencies. Both techniques can also introduce new conflict misses by increasing the number of active lines in the cache. Additionally, both techniques can increase the static code size and instruction-cache footprint of an application. These performance limitations may impede these latency-tolerance techniques in environments with low bandwidth or poor instruction memory.

5.2 Combining Clustering and Prefetching

Software prefetching and read miss clustering seem to target the same types of latencies, and both techniques require the same system resources (specifically, cache miss buffers and memory system bandwidth). However, their latency-tolerance methods are quite distinct, since prefetching pipelines inner loops to add new fetches ahead of their demand accesses and clustering restructures loop nests at an outer-loop level to extract parallelism among demand read misses (Table 5.1). This section qualitatively discusses how the two techniques can actually improve performance further when combined. Section 5.2.1 describes

```
for(j=0;j<$I_o$;j++)
 for(i=0;i<$I_i$;i++)
  ···A[j,i]
```

(a) Original code

```
for(j=0;j<$I_o$;j+=$N$)
 for(i=0;i<$I_i$;i++)
  ···A[j,i]
  ···A[j+1,i]
  ......
  ···A[j+N-1,i]
```

(b) After clustering alone

Figure 5.1 : Pseudocode of a 2-D matrix traversal, (a) as originally generated, and (b) after read miss clustering with unroll-and-jam (postlude not shown). All pseudocode uses row-major notation. Figure 5.2 shows the corresponding codes after the addition of prefetching.

the potential for combining these latency-tolerance techniques. Section 5.2.2 presents a limitation of the combined technique and a way to resolve it.

5.2.1 Potential Benefits of Combining Clustering and Prefetching

This section discusses how applying read miss clustering before software prefetching can address some of the specific limitations described in Chapter 3. Figure 5.1(a) gives pseudocode for a 2-D matrix traversal, with all pseudocode specified in row-major notation. For reference, Figures 5.1(b) and 5.2(a) show the matrix traversal after clustering alone and after prefetching alone, respectively. Figure 5.2(b) shows the matrix traversal after applying the combination of clustering followed by prefetching.

Incomplete latency hiding. Effective prefetching can tolerate all steady-state latencies, while read miss clustering alone leaves at least some miss latencies exposed. Thus, prefetching can potentially tolerate the latencies left behind after clustering (for references in the steady-state prefetching loop).

Prologue late prefetches. Read miss clustering can reduce the effect of prologue late prefetches by reducing the number of times an inner-loop is started. Consider a 2-level loop nest with I_o outer loop iterations, such as the code in Figure 5.1(a). With prefetching alone, the inner loop would be started I_o times, with late prefetches on the first steady-state iteration each time. If unroll-and-jam with a degree of N is applied before prefetching, the

```
for(j=0;j<$I_o$;j++)
 for(i=0;i<$d$;i+=$L$)
  PF(&A[j,i])

 for(i=0;i<$I_i - d$;i++)
  if(i$\bmod L \equiv 0$) PF(&A[j,i+$d$])

  $\cdots$A[j,i]

 for(;i<$I_i$;i++)
  $\cdots$A[j,i]
```

(a) After prefetching alone

```
for(j=0;j<$I_o$;j+=$N$)
 for(i=0;i<$d'$;i+=$L$)
  PF(&A[j,i])
  PF(&A[j+1,i])
  ......
  PF(&A[j+N-1,i])
 for(i=0;i<$I_i - d'$;i++)
  if(i$\bmod L \equiv 0$)PF(&A[j,i+$d'$])
                         PF(&A[j+1,i+$d'$])
                         ......
                         PF(&A[j+N-1,i+$d'$])
  $\cdots$A[j,i]
  $\cdots$A[j+1,i]
  ......
  $\cdots$A[j+N-1,i]
 for(;i<$I_i$;i++)
  $\cdots$A[j,i]
  $\cdots$A[j+1,i]
  ......
  $\cdots$A[j+N-1,i]
```

(b) After clustering and prefetching

Figure 5.2 : Pseudocode of the 2-D matrix traversals of Figure 5.1 after adding software prefetching. Figure 5.1(a) tolerates latencies with software prefetching alone, while Figure 5.1(b) uses the combination of clustering and prefetching.

outer loop will only have $\frac{I_o}{N}$ iterations, and the inner loop only starts $\frac{I_o}{N}$ times. This reduces the number of separate instances of prologue late prefetches (or other exposed latencies at the beginning of each steady-state) by a factor of N.

Short steady-states. Read miss clustering increases the computation in an inner-loop iteration (the C term of Equation 3.1) without changing the number of inner-loop iterations (I_i), as is evident by comparing Figures 5.1(a) and 5.1(b). Since the prefetch distance (in iterations) is inversely proportional to C, the clustering technique can reduce the prefetch distance to fewer iterations and can increase the length of the steady-state, increasing the effectiveness of prefetching. (This implies that d', the prefetch distance of Figure 5.2(b), is smaller than d, the prefetch distance of Figure 5.2(a).)

Hard-to-prefetch references. The memory parallelism benefits of read miss clustering also apply to references prefetched with a prefetch distance insufficient to overlap their latency fully. Read miss clustering restructures the code to improve parallelism of demand read misses, and this is reflected after applying prefetching through increased parallelism among both prefetches and unprefetched misses.

Instruction overhead. Read miss clustering through unroll-and-jam can exploit *scalar replacement*, by which redundant memory references are replaced with register operations and the total number of instructions is reduced [AC72, Car96, CK94]. If the redundant references tended to hit in the cache (as seen in Chapter 4), scalar replacement can reduce both the unnecessary prefetches resulting from these references and their address-generation overhead.

Other limitations. The memory parallelism provided by read miss clustering can help to tolerate any latencies exposed by early or damaging prefetches. (However, in some cases, clustering may also increase early or damaging prefetches by keeping more lines active in the cache at once.)

Trends. We expect D (the miss latency in cycles) to increase with successive generations of systems, as processor clock speeds improve faster than memory latencies. The number of cycles per iteration, C, is likely to decrease with more aggressive processor architectures. Both hardware trends increase the prefetch distance. Additionally, software that more aggressively uses locality transformations such as tiling sees shorter inner loops with each inner loop initiated more times [AKL81, Por89, WL91]. These hardware and software trends increase the impact of prologue late prefetches, short steady-states, and hard-to-prefetch references, all of which can be addressed by read miss clustering.

5.2.2 Addressing a Limitation of Clustered Prefetching

Unroll-and-jam produces a postlude of leftover iterations if the outer loop's iteration count is not divisible by the unrolling degree. Chapter 4 noted that clustering could be improved in the postlude through loop interchange. This interchange should not degrade locality, since the original number of outer-loop iterations in the postlude is less than the degree of unroll-and-jam. However, the new inner loop has few iterations, increasing the likelihood of late prefetches. Thus, applying the standard inner-loop prefetching algorithm may only add instruction overhead without actually tolerating any latencies.

The small number of inner-loop iterations in the interchanged postlude suggests the use of *outer-loop prefetching*: applying software pipelining at an outer-loop and scheduling prefetches ahead by a certain number of outer-loop iterations [McI98]. In general, outer-loop prefetching algorithms can be ineffective because of increased conflict and capacity misses, requiring code reorganization through strip-mining to facilitate a reasonable prefetch distance and reduce the likelihood of early prefetches. However, the short inner loops in the interchanged postlude make such potential negative effects less likely and such strip-mining unnecessary. Even though we may not know the exact inner-loop iteration count, we can consider this value to be as low as 1 iteration when calculating the outer-loop prefetch distance. Any resulting early prefetches will only affect the postlude, and read miss clustering will help overlap their latencies.

We thus propose the following outer-loop prefetching algorithm for the postlude whenever possible. This algorithm, illustrated with an example in Figure 5.3, starts with the postlude produced by unroll-and-jam (before interchanging), and combines inner-loop prefetching and loop interchange. If certain dependences prevent this algorithm, then the potential for negative interactions (e.g., short or nonexistent steady-states because of the short loops) suggests that the interchanged postlude is best left unprefetched.

1. Figure 5.3(a) represents the postlude resulting from applying unroll-and-jam of degree N on the 2-D matrix traversal of Figure 5.1(a). Perform the prefetching algorithm of Mowry et al. only on the innermost loop of such a postlude loop nest [MLG92]. In the prefetching steady-state, do not strip-mine or unroll for selective prefetching. Instead, use an `if` conditional at the beginning of each loop iteration to ensure that prefetches are sent only at the correct iterations. (Mowry et al. suggest this technique where code expansion due to strip-mining or unrolling is

unacceptable [MLG92].)

This stage produces a prefetching prologue, steady-state, and epilogue, each at the innermost level only, as illustrated in Figure 5.3(b).

2. Separate the loop nest into a prologue nest, a steady-state nest, and an epilogue nest, as shown in Figure 5.3(c). Such separation is not acceptable in general, since the prologue prefetches could move too far from the demand references for an outer loop with many iterations. However, the postlude has few outer-loop iterations.

3. Separately interchange each of the three loop nests by moving the outermost loop to the innermost position (Figure 5.3(d)). The steady-state and epilogue interchanges increase the clustering of unprefetched misses and late prefetches and decrease the likelihood of early prefetches for those references that can be prefetched sufficiently in advance. The prologue is interchanged so that prefetches follow the same order as the demand references.

4. Invoke loop-invariant code motion for data and prefetches found to be invariant to the new inner loop. Note that references that were invariant with respect to the original innermost loop remain unprefetched. However, these should account for less data than the prefetched references, and clustering will help to hide their miss latencies.

After these steps, the postlude has prefetches scheduled according to future iterations of the new outer loop, while the inner loop clusters prefetches and demand references to increase memory parallelism.

5.3 Experimental Methodology

Our evaluation environments in this chapter are identical to those used in Chapter 4, for both the simulated and real systems. Our evaluation workload consists of all the same applications and data sets except for Mp3d. Mp3d is dominated by a single-level loop nest and thus would provide no opportunity to improve prefetching through techniques that

```
for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
 for(i=0;i<$I_i$;i++)
  $\cdots$A[j,i]
```

(a) Original postlude

```
for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
 for(i=0;i<$d$;i+=$L$)
  PF(&A[j,i])
 for(i=0;i<$I_i - d$;i++)
  if($i \bmod L \equiv 0$) PF(&A[j,i+$d$])
  $\cdots$A[j,i]
 for(;i<$I_i$;i++)
  $\cdots$A[j,i]
```

(b) After prefetching

```
for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
 for(i=0;i<$d$;i+=$L$)
  PF(&A[j,i])
for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
 for(i=0;i<$I_i - d$;i++)
  if($i \bmod L \equiv 0$) PF(&A[j,i+$d$])
  $\cdots$A[j,i]
for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
 for(I=$I_i - d$;i<$I_i$;i++)
  $\cdots$A[j,i]
```

(c) After loop-nest separation

```
for(i=0;i<$d$;i+=$L$)
 for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
  PF(&A[j,i])
for(i=0;i<$I_i - d$;i++)
 for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
  if($i \bmod L \equiv 0$) PF(&A[j,i+$d$])
  $\cdots$A[j,i]
for(;i<$I_i$;i++)
 for(j=$N \times \lfloor\frac{I_o}{N}\rfloor$;j<$I_o$;j++)
  $\cdots$A[j,i]
```

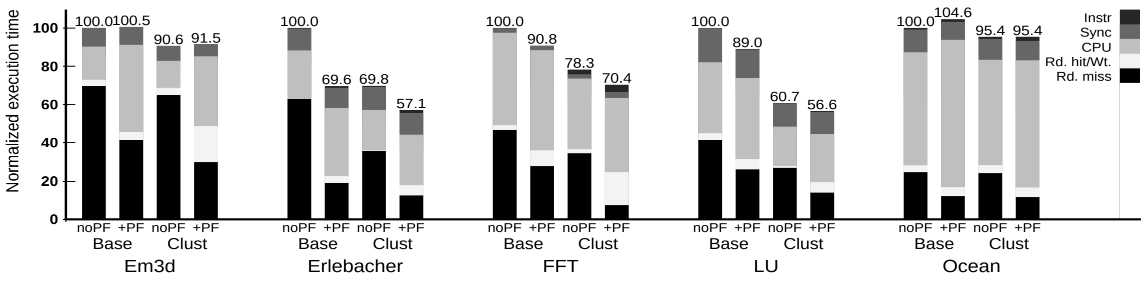
(d) After interchange

Figure 5.3 : Applying both software prefetching and loop interchange to provide clustered prefetching in the postlude left by unroll-and-jam.

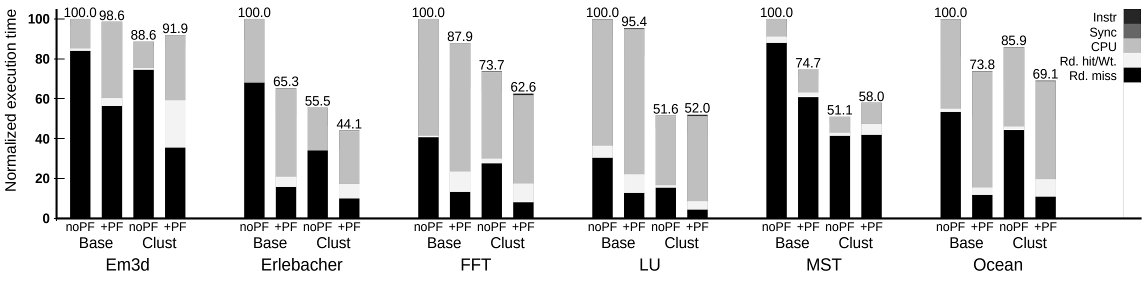
extract memory parallelism at an outer-loop scope. We have also omitted the Latbench microbenchmark, since it does not provide additional insights into clustered prefetching.

As in Chapter 4, we apply clustering by hand to our codes, following the algorithms of Section 4.2. For the simulated system, we apply prefetching by hand following the algorithms described in Chapter 3. Since the Exemplar C compiler supports prefetching, we use compiler-generated prefetching for the real machine [SGH97]. The Exemplar compiler could prefetch all test programs except MST.

Our key metrics used in this study are total execution time and its components. We also gain insights by counting those late prefetches for which a demand reference exposes data read miss stall time.



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 5.4 : Execution times on base simulated system with software prefetching, clustering, and their combination. All times are shown normalized to the execution time with neither prefetching nor clustering.

5.4 Simulation Results

Figure 5.4(a) shows the multiprocessor execution times of the applications running on the base simulated system, while Figure 5.4(b) shows simulated uniprocessor execution times. The execution time bars show the original code (Base/noPF), the code after prefetching alone (Base/+PF), after clustering alone (Clust/noPF), and after the combination of the two (Clust/+PF). All execution-time bars are split as described in Section 4.3.3 and normalized to the execution time with neither prefetching nor clustering. (For MST, this chart only shows prefetch arrays. Greedy prefetching is described later.)

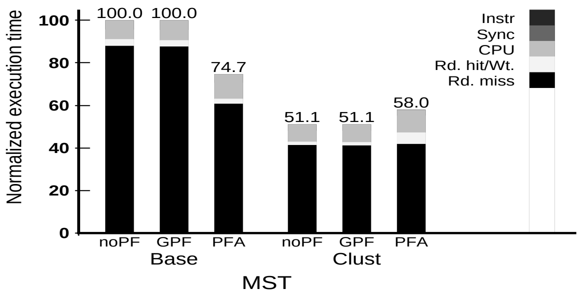


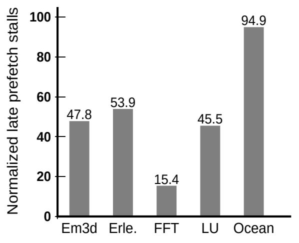
Figure 5.5 : Comparing greedy prefetching and prefetch arrays for MST on base simulated uniprocessor system. Execution times are normalized to the unclustered code without prefetching.

5.4.1 Comparing Clustering and Prefetching

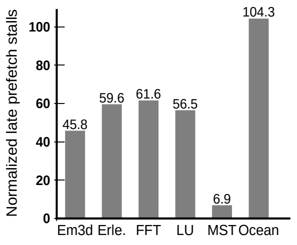
We focus here on the first three bars of Figure 5.4 for each application and system configuration (i.e., base code, prefetching alone, and clustering alone). Section 5.4.2 discusses the fourth bar, which combines clustering and prefetching.

Comparing clustering alone to prefetching alone, we see that clustering gives comparable or better overall execution times than prefetching for all applications except the uniprocessor Ocean. In the multiprocessor, clustering reduces execution time an average of 20% (ranging 5–39%), while prefetching reduces execution time an average of 17% for 3 out of 5 applications (ranging 9–30%, with less than 5% degradation on the other 2 codes). In the uniprocessor, clustering reduces execution time an average of 30% (ranging 5–39%), while prefetching reduces execution time an average of 17% (ranging 1–35%).

To understand the differences between clustering and prefetching, we consider the individual components of execution time. Prefetching alone actually has a greater impact on data memory stall time for all applications except LU and MST. However, the instruction overhead of prefetching (discussed in Section 5.1) increases CPU time and offsets the greater memory stall time benefits relative to the clustered code for all cases except the uniprocessor Ocean. This CPU overhead actually leads to slight performance degradations *relative to the base code* in the multiprocessor Em3d and Ocean. Additionally, cluster-



(a) Multiprocessor



(b) Uniprocessor

Figure 5.6 : Number of late prefetch stalls after clustered prefetching represented as a percentage of the late prefetch stalls seen with prefetching alone.

ing alone actually sees reductions in the CPU component of execution time for many of the applications because of the scalar replacement benefits of unroll-and-jam (discussed in Section 5). Both latency tolerance techniques have little negative impact on instruction memory stalls, since these loop-based applications still tend to hit in the I-cache.

Figure 5.5 includes results for MST with both greedy prefetching (GPF) and prefetch arrays (PFA). As discussed in Chapter 3, prefetch arrays increase the needed working set and cause new misses and extraneous prefetches, while greedy prefetching suffers from hard-to-prefetch references and prefetch distances limited to 1 iteration. Schemes that use longer artificial jump pointers would be inapplicable, since the linked-lists in MST are very short. On the other hand, clustering alone tolerates latencies more effectively by restructuring the demand references at an outer-loop level so that multiple lists are traversed in parallel.

5.4.2 Combination of Clustering and Prefetching

Overall results. The fourth bar of each application and system configuration in Figure 5.4 represents the combination of clustering and prefetching. Even though each scheme in isolation realizes substantial opportunities for tolerating latencies, the combination allows additional opportunities in many cases. Except for the uniprocessor Em3d and MST, this combination either performs the best or sees less than 1% degradation from the best. Compared to clustering alone, the combination reduces execution time an average of 12% for 3 out of 5 applications in the multiprocessor (ranging 7–18%, with a 1% degradation in 1 code and no impact on another) and 18% for 3 of 6 uniprocessor codes (ranging 15–21%, with less than 5% degradation in 2 codes and 14% degradation in MST). Compared to prefetching alone, the combination reduces execution time an average of 18% in the multiprocessor (ranging 9–36%) and 24% in the uniprocessor (ranging 6–49%). We compute these averages conservatively by comparing clustered prefetching against the best of either the base code or the code with one optimization alone. Specifically, for the multiprocessor Em3d and Ocean, we compare clustered prefetching here against the unprefetched code instead of the actual degraded performance seen with prefetching alone.

To specifically show how the benefits of clustered prefetching derive from more effective prefetching, Figure 5.6 shows the impact of clustering on the number of late prefetches that lead to stalls. All bars show the number of late prefetch stalls for clustered prefetching as a percentage of the number for prefetching alone. Figures 5.6(a) and 5.6(b) show multiprocessor and uniprocessor systems, respectively. Read miss clustering reduces the number of late prefetch stalls by an average of 49% on the multiprocessor and 44% on the uniprocessor, with dramatic improvements in all cases except Ocean. The scalar replacement benefits of clustering also reduce the CPU overhead of prefetching and the number of unnecessary prefetches for some applications.

Compared to clustering alone, some of the improvements in read miss latency tolerance seen with clustered prefetching are negated by an increase in CPU overhead from prefetching. Additionally, an increase in read hit and write time (from increased contention) also degrades the performance of some applications. Clustered prefetching sees less benefits compared with clustering alone than with prefetching alone simply because prefetching alone is not as effective as clustering alone for most of these codes.

Application-specific details. In Em3d, clustering improves the impact of prefetching on miss stall time by overlapping prologue late prefetches and by increasing the otherwise small steady-state sizes (from the short traversals of the `from_nodes` structures and the

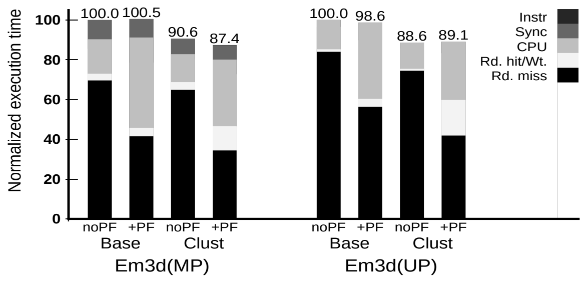


Figure 5.7 : Execution times of Em3d on base system when less clustering is used in combination with prefetching.

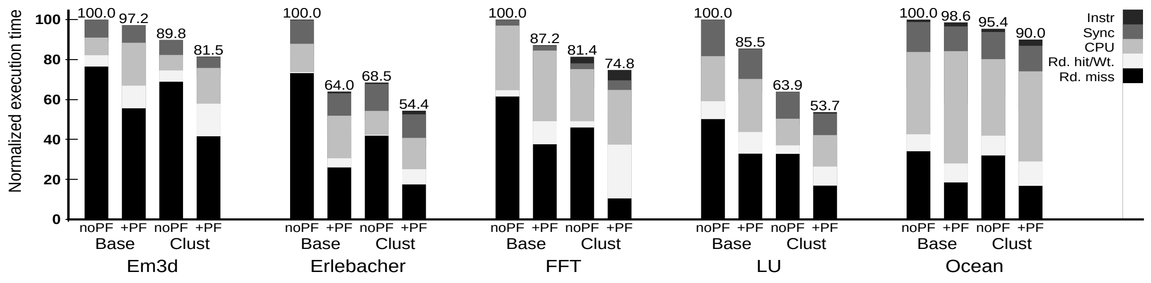
indirect prefetching), reducing late prefetch stalls over 50%. Prefetching helps to tolerate read miss latencies remaining after clustering alone. However, clustered prefetching sees stalls from exposed read hits. A closer examination shows register spilling when clustering and prefetching are applied together, though neither transformation alone causes spills. These spills tend to hit in the cache, but increase contention for cache ports. Previous work on unroll-and-jam suggests limiting the unrolling degree based on register pressure [CK94]. An unroll-and-jam algorithm suitable for combining with prefetching may profit from register pressure heuristics extended to estimate the additional register pressure caused by prefetching. To demonstrate the potential effectiveness of using less clustering in combination with prefetching, Figure 5.7 shows the normalized execution times for Em3d with a smaller degree of unroll-and-jam in the Clust/+PF run (4, instead of the 6 used in Clust/noPF). The resulting reduction in register spills and read hit stall time improves total execution time and allows benefits for clustered prefetching in the multiprocessor.

Erlebacher, FFT, and LU all have important phases blocked for cache locality and/or load balance: the fine-grained wavefront pipeline in Erlebacher, the transpose in FFT, and the entire code of LU. As discussed in Chapter 3, none of these blocked portions achieves a steady-state with prefetching alone. Clustering actually enables a steady state for the transpose of FFT, reducing the number of late prefetch stalls by 85% for the multiprocessor. Clustering also benefits all three applications through overlapped prologue late prefetches. Erlebacher also sees some benefits from scalar replacement of references that tend to cause unnecessary prefetches, while scalar replacement in LU substantially reduces the total instruction count. In Erlebacher and FFT, prefetching helps to tolerate steady-state latencies left behind by clustering in other phases of the application. These three applications benefit significantly from clustered prefetching, with substantial benefits relative to prefetching alone in all cases and relative to clustering alone in all but the uniprocessor LU. (Clustering alone has the greatest impact on latency tolerance in LU; the incremental latency tolerance provided by prefetching is not sufficient to offset its CPU overhead, leading to a slight degradation.)

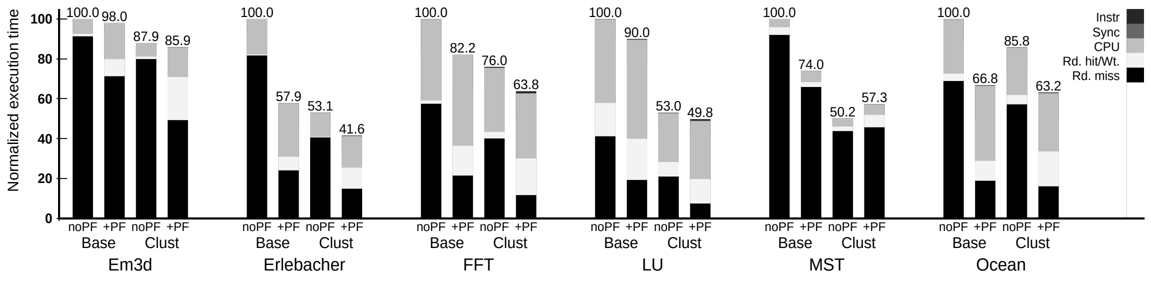
For MST, prefetch arrays were seen to improve the unclustered version, but actually degrade the performance of the clustered code. In particular, clustering leaves less available bandwidth for the extra fetches added by the prefetch array scheme. As a result, clustering exposes the negative effects of this scheme's increased working set and new misses. (This is analogous to the degradations previously seen with prefetch arrays in low-bandwidth systems [KDS00].) Greedy prefetching avoids such degradations, but also provides no benefits over clustering alone since its prefetch distance is too limited to provide the needed latency tolerance. Thus, clustering alone provides the best performance in MST.

Ocean sees an increase in conflict misses from clustering. Combined with prefetching, this causes additional contention-related stalls and early prefetches. Additionally, these conflicts also turn some unnecessary prefetches into necessary ones, increasing the number of late prefetch stalls in the uniprocessor and causing clustered prefetching to see more data memory stall time than prefetching alone. However, scalar replacement provides some benefits in Ocean by reducing unnecessary prefetches and the CPU overhead of prefetching. The net effect is that clustered prefetching provides the best overall execution time.

Sensitivity to system parameters. Processor speeds and external memory latencies diverge further for processors in the gigahertz frequency range. To model this trend, we also performed experiments that modified our base configuration to include 1 GHz processors without changing any absolute memory hierarchy times (in ns or MHz). The results in Figure 5.8 show behavior qualitatively similar to Figure 5.4. (Em3d is shown with reduced



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 5.8 : Execution times on simulated system with faster processors.

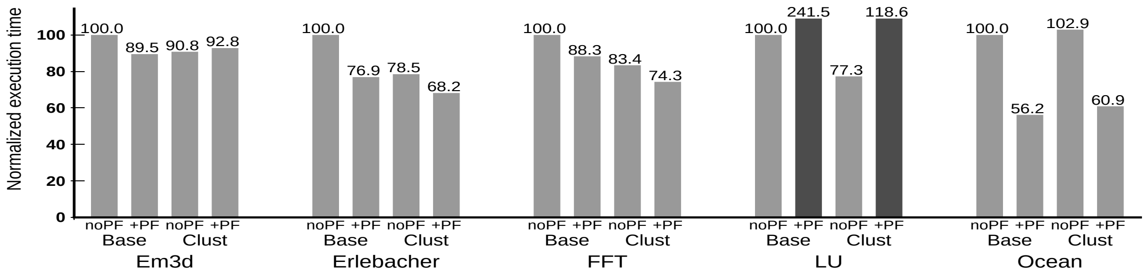
clustering in combination with prefetching. MST is still shown with prefetch arrays; our detailed statistics show that greedy prefetching still has negligible impact on performance.) The only performance difference of consequence relative to the base system is that CPU overhead becomes less important in this configuration. This improves the effectiveness of prefetching in both the base and clustered versions. Prefetching alone now outperforms clustering alone in the multiprocessor Erlebacher and the uniprocessor Ocean, and clustered prefetching is better than or equal to either scheme in isolation for all but the uniprocessor MST.

5.5 Results on Real Machine

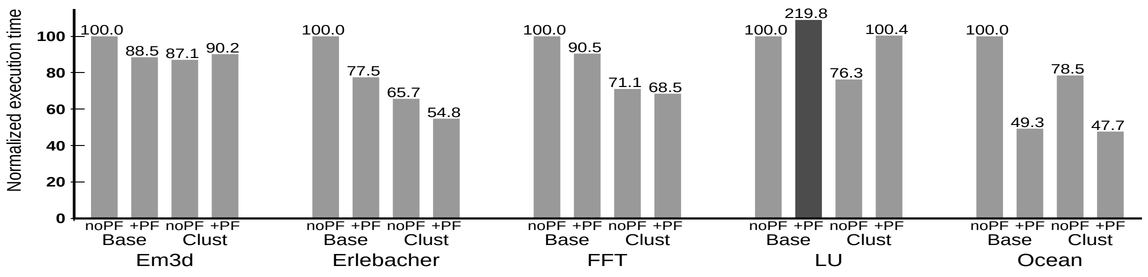
Figure 5.9 gives the impact of prefetching, clustering, and their combination for applications run on the Convex Exemplar. Only total execution times are shown, since detailed splitups are not available on the real machine. Both multiprocessor and uniprocessor runs are shown, and execution times are normalized to the code with the same number of processors with neither prefetching nor clustering. Here, clustering alone outperforms prefetching alone for 6 out of 10 applications and systems shown. Clustered prefetching performs better than both clustering alone and prefetching alone for 5 out of 10 cases (multiprocessor and uniprocessor Erlebacher, multiprocessor and uniprocessor FFT, and uniprocessor Ocean). Compared to prefetching alone, clustered prefetching reduces execution time an average of 14% for 2 out of 5 multiprocessor codes (ranging 11–16%) and 19% for 3 out of 5 uniprocessor codes (ranging 3–29%). Compared to clustering alone, clustered prefetching reduces execution time an average of 21% for 3 out of 5 multiprocessor codes (ranging 11–40%) in the multiprocessor and 20% for 3 out of 5 uniprocessor codes (ranging 4–40%). As in Section 5.4.2, these averages are computed conservatively by comparing against the base code when a code is degraded by its sole optimization. Additionally, we do not count LU since both prefetching and clustered prefetching are greatly degraded here. All the observed degradations are discussed below.

The multiprocessor version of Ocean sees the best performance with prefetching alone, with 8% degradation from clustered prefetching. The simulation results of Section 5.4 had suggested an increase in conflicts with clustering or clustered prefetching. These may be further exacerbated with the HP PA-8000 processor, as its cache is direct-mapped and requires serialization of conflicting references [SGH97].

Em3d sees only minor performance differences between prefetching alone, clustering alone, and clustered prefetching. Prefetching alone gives the best performance in the multiprocessor, while clustering alone performs best in the uniprocessor. Em3d sees an anomalous benefit from prefetching in the unclustered version due to a specific implementation choice in the Exemplar compiler. Although not explicitly specified in the technical paper



(a) Multiprocessor execution time



(b) Uniprocessor execution time

Figure 5.9 : Execution times on Convex Exemplar with software prefetching, clustering, and their combination. All times are shown normalized to the execution time with the same number of processors and neither prefetching nor clustering. (Darker bars represent degradations beyond the size of the charts.)

on the algorithm used [SGH97], we observe from assembly-code analysis that the Exemplar C compiler does not generate a prologue or epilogue for prefetched loops. The lack of a prologue may not affect performance, since the prologue prefetches are likely to be late (Section 5.2.1). Without an epilogue, however, the algorithm always fetches extra data beyond the end of the inner loop. In Em3d, these items are actually used in later outer-loop iterations. With clustered prefetching, most of these prefetches become unnecessary and simply add instruction overhead, since misses from later iterations of the original outer-loop would have already been executed by the time the code reaches the last iterations of the inner loop.

Prefetching algorithms in general cannot rely on such benefits by omitting the epilogue, since transformations such as array padding can prevent extra fetches from touching useful data [RT98]. Worse yet, such extraneous fetches could increase conflict and capacity misses (especially for tiled codes), as well as coherence traffic and apparent data sharing for fine-grained shared-memory multiprocessor codes. Such problems arise in LU, where prefetching dramatically degrades performance with or without clustering. Clustered prefetching can mitigate these negative effects by reducing the prefetch distance, which limits the extraneous data fetched, and by overlapping the resulting capacity or communication misses (reducing the prefetching degradation from 142% to 54% on the multiprocessor and from 120% to 32% on the uniprocessor). Nevertheless, clustering alone significantly outperforms clustered prefetching and prefetching alone for LU. We consider the Em3d and LU results more indicative of specific implementation issues in the Exemplar compiler rather than underlying problems in clustered prefetching.

5.6 Summary and Discussion

This chapter compares and combines two latency-hiding techniques, read miss clustering and software prefetching. We show that the two techniques can combine to overcome performance limitations in either technique alone. Prefetching can tolerate latencies left behind after read miss clustering alone, while read miss clustering can reduce the impact of prologue late prefetches, excessive prefetch distances, hard-to-prefetch references, and prefetch instruction overhead.

Experimental results show that clustering alone outperforms prefetching alone for most of the applications and systems that we study, but that their combination most often gives the best performance. Clustered prefetching reduces execution time relative to prefetching alone (the currently implemented state-of-the-art) by an average of 21% across all cases studied in simulation and an average of 16% for 5 out of 10 cases on the Exemplar. Compared to miss clustering alone, the combination sees reductions in execution time averaging 15% for 6 out of 11 cases in simulation and 20% for 6 out of 10 cases on a real machine. Further, those cases where clustered prefetching falls short can be attributed to a small number of causes that are not fundamental limitations of the technique itself: cache conflicts (Ocean), register pressure (Em3d), compiler anomalies (lack of epilogue in Exemplar versions of Em3d and LU), or insufficient bandwidth (MST with prefetch arrays). Compiler and hardware solutions for these problems can further improve the performance of clustered prefetching.

Chapter 6

Analyzing and Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors

The previous chapters have performed their simulation experiments with the Rice Simulator for ILP Multiprocessors (RSIM). RSIM simulates the individual processors of a multiprocessor system in detail, including the processor pipelines and the functionality of each instruction. In contrast, most other shared-memory multiprocessor simulation studies use a much simpler model of the processor, assuming single-issue, in-order issue, blocking reads, and no speculative execution. Unfortunately, the more detailed ILP-processor based simulators are much slower: we find an average slowdown of 9.7X compared to simple-processor based simulators.

The higher speed of simple-processor based simulators comes from the inherent benefits of simulating a less complex processor, as well as from several speed enhancing techniques developed for such simulators. Direct execution is one such widely used technique that has previously relied on simple-processor features such as blocking reads, in-order issue, and no speculation [CDJ$^+$91, DGH91, MRF$^+$97]. This chapter presents a novel adaptation of direct execution to substantially speed up simulation of shared-memory multiprocessors with ILP processors, without much loss of accuracy. We have developed a new simulator, DirectRSIM, based on our new technique. We evaluate the accuracy and speed of DirectRSIM by comparing it with RSIM, a state-of-the-art detailed ILP-processor based shared-memory simulator, as well as two representative simple-processor based direct execution simulators. For a variety of system configurations and applications, and using RSIM as the baseline for accuracy, we find:

- DirectRSIM, on average, is 3.6X faster than RSIM with an error in simulated execution time of only 1.3% (range of -3.9% to 2.2%).

- Simple-processor based simulators remain an average of 2.7X faster than DirectRSIM. However, this additional speed comes at a high cost, with average error in execution time of 46% (range of 0% to 128%) with the best simple-processor model, and average error of 137% (range of 9% to 438%) with the most common model.

Our results suggest a reconsideration of the appropriate simulation methodology for shared-memory systems. Earlier, the order-of-magnitude speed advantage of the simple-processor based simulators over RSIM made a compelling argument for their use in spite of their potential for large errors. It is not clear that those errors are still justifiable given only a 2.7X speed advantage relative to DirectRSIM.

The remainder of this chapter proceeds as follows. Section 6.1 gives background information on the previous state-of-the-art in simulation methodologies. Section 6.2 describes the difficulties in integrating direct-execution simulation with ILP processor simulation and how these are resolved for the DirectRSIM simulator. Section 6.3 describes in detail the implementation of our direct-execution simulation methodology in DirectRSIM. Section 6.4 describes how we evaluate the simulators used in this study. Sections 6.5 and 6.6 analyze the accuracy and speed of the simulators studied and show how DirectRSIM improves the tradeoff between accuracy and speed in ILP multiprocessor simulation. Section 6.7 summarizes the results of this study.

6.1 Background

This section explains the high-speed direct execution technique used for simple-processor simulators, as well as the methodologies used for simulating ILP-based systems.

6.1.1 Direct execution with simple processors

Direct execution is a widely used form of execution-driven simulation, and has been shown to be accurate and fast for modeling shared-memory systems with simple processors [CDJ+91, DGH91, MRF+97].

Direct execution decouples functional and timing simulation. Functional simulation generates values (for registers and memory) and control flow, while timing simulation determines the number of cycles taken by the simulated execution. Direct execution achieves high speed in two ways. First, for functional simulation, it directly executes the application on the host. Second, timing for non-memory instructions is determined mostly by static analysis. The application is instrumented to convey this analysis to the memory timing simulator. Previous direct execution shared-memory simulators assume in-order issue and no speculation since they cannot model the effects of out-of-order issue statically and they view only one basic block at a time. With the exception of the Wisconsin Wind Tunnel-II [MRF+97], these simulators also assume single-issue processors. Timing for memory references is modeled in detail, and is the most expensive part of the simulation.

For memory simulation, the application is usually instrumented to invoke the timing simulator on each memory reference, as these are the only points of interaction between the processors. When an application process invokes the timing simulator on a read, its functional simulation is suspended until the timing simulator completes the entire simulation of the read, thereby modeling only blocking reads. Stores are either modeled as blocking or non-blocking. For non-blocking writes, direct execution of the write's process may be resumed as soon as the appropriate simulation events for the write are scheduled (but not necessarily completed). The timing simulator can process these events asynchronously with respect to the write's process because, unlike a read, later instructions of the process do not depend on the completion of the write.

6.1.2 Simulators for ILP shared-memory systems

RSIM [PRA97a], SimOS with the MXS processor simulator [RBDH97], and Armadillo [GC98] are shared-memory simulators that model ILP processors explicitly and in detail. They use detailed execution-driven simulation, interpreting every instruction and simulating its effects on the complete processor pipeline and memory system in software. Neither MXS nor Armadillo are currently publicly distributed.

Researchers have also used simple-processor based simulators to model ILP-processor based shared-memory systems using certain approximations. The most common approximation is to simply simulate a system with a simple processor to approximate a system with an ILP processor with the same clock speed. We refer to this sort of simulation model as *Simple*. Other studies have sped up the clock rate of the simulated simple processor and the primary cache access time to model the benefits of ILP [HKO$^+$94, HSH96]. We call such variants *Simple*-Nx, where N is the multiple by which the simulated clock rate is increased. However, such simulators still do not model the processor's ability to tolerate read miss latency by overlapping multiple read misses with each other. As a result, they are not likely to accurately model data memory system performance for applications with high read miss clustering.

6.2 Direct execution with ILP shared-memory multiprocessors

There are two problems with using previous direct execution techniques for ILP-processor based shared-memory systems:

Values for non-blocking reads. After a non-blocking read invokes the timing simulator, its direct execution process must be allowed to proceed before its timing simulation completes. This is required so that the direct execution process can generate later instructions for the timing simulator to execute in parallel with the read. However, the value that the read will return in the simulated architecture is unknown until the read's timing simulation is complete; this value depends on writes to the same location by other processors before the read reaches memory in the simulated architecture. Thus, the first problem is that the simulator must decide what value to return when a read occurs in the direct execution while it is incomplete in the timing simulation, and what action to take when the direct execution reaches a later instruction dependent on such a read.

Timing simulation of ILP features. The second problem is that a simple static analysis is insufficient to determine the impact of ILP features (such as out-of-order issue, speculative execution, and non-blocking reads) on the execution time of CPU instructions and on the time at which a memory instruction can be issued (or when it stalls the processor). Previous direct execution techniques do not directly provide a way to account for these features.

Sections 6.2.1 and 6.2.2 discuss our solutions to the above problems.

6.2.1 Values for non-blocking reads

We focus on a release-consistent architecture [GLL$^+$90]. For ease of explanation, we assume that synchronization accesses are identified to the simulator.

When a synchronization read invokes the timing simulator, it is treated as a blocking read as in previous direct execution simulators. When invoked by a data read, the timing simulator starts processing all instructions executed since its last invocation as described in Section 6.2.2. The timing simulator may return control to the direct execution before the read completes at (or even issues to) the memory hierarchy. The read returns the current value for the accessed memory location at the time of the direct execution, based on the following insight.

If the read does not form a data race with a write from another process in the simulated execution, the read and write will be executed in the same order in the direct execution as in the simulated execution. A read that is not part of a data race (a non-race read) must be separated from any conflicting write by a chain of synchronization releases and acquires; these synchronization accesses are ordered as in previous direct execution simulators and enforce the necessary orderings among non-race accesses. Thus, for a non-race read, the value at the time of the direct execution can be safely returned and used by dependent instructions.

For a race read, the value returned may be different from the one that would be returned in the simulated architecture. This value would be legal for release consistency, but may not be possible on the simulated architecture. Since data races are generally rare in parallel programs, we expect this issue to not have a significant impact. Further, a system that obeys the data-race-free consistency model [AH90] (which requires identifying data races for a guarantee of sequential consistency) and blocks on race reads can naturally be simulated with our technique without any error (by simply blocking on the race reads).

6.2.2 Timing simulation of ILP features

Like previous direct execution simulators, our technique performs the functional simulation directly on the host machine and invokes the timing simulator only on memory references. Unlike previous uses of direct execution, the application is instrumented to record the path taken by the direct execution since the previous invocation of the timing simulator by the same process. The timing simulator simulates the timing for this path with the goal of providing the best accuracy and speed possible.

A naive timing simulator would simply replicate the features of detailed simulators such as RSIM, modeling the register state, pipeline stages, and all instruction effects in detail. Instead, our timing simulator improves speed relative to RSIM in three ways. First, direct execution allows the timing simulator not only to avoid instruction emulation, but also to make use of the values determined in direct execution to speed up several parts of simulation (for example, register renaming and memory disambiguation).

Second, we approximate some parts of the processor simulation, motivated by the results in Chapter 2 that show that the key characteristic in determining shared-memory multiprocessor performance is the behavior of the memory system and its interaction with the processor. Our most significant approximation is that we do not simulate speculated execution paths that are mispredicted. This approximation does not preclude modeling other effects of speculation; for example, we keep track of branch prediction tables and stall instruction fetch on a mispredicted branch as the processor waits for the branch to be resolved. The simulation speed benefits of this approximation cannot be exploited by detailed simulators such as RSIM since RSIM does not know if a prediction is correct until the prediction is actually resolved in the simulated execution. The timing simulator of DirectRSIM has this information at the time the prediction is made, based on the values generated by the direction execution.

Third, with direct execution, the different application processes execute asynchronously in the simulation. In contrast, RSIM's processor and cache simulation, due to its detailed nature, is inherently a cycle-by-cycle simulation in which all processors and caches proceed in lockstep (Section 6.4.3). We improve the speed of our timing simulation by further increasing the asynchrony in our system, partly by exploiting the features described above. The next section provides further details.

6.3 Implementation of DirectRSIM

DirectRSIM implements the direct execution methodology described in Section 6.2. It consists of an application instrumentation mechanism (Section 6.3.1) and a timing simulator (Section 6.3.2).

6.3.1 Application code instrumentation

The instrumentation code calls the timing simulator on each memory reference and provides it with the execution path to be processed. The path is represented as ranges of contiguous program-counter values traversed by the direct execution since the last invocation of the timing simulator. For this purpose, the instrumentation code marks each unconditional branch or taken path of a conditional branch as ending a program-counter range and starting a new range. Each contiguous program-counter range can span multiple basic blocks, and multiple ranges can be passed to the timing simulator on each call. We currently instrument the application assembly code, but could also use the more general methods of executable-editing or dynamic binary translation.

6.3.2 Timing simulator

The timing simulator consists of three main parts: the event-driven simulation engine, the multiprocessor memory system simulator, and the processor simulator. The event-driven simulation engine and multiprocessor memory system simulator are common to all our simulators, and are described in more detail in Section 6.4. The processor simulator is the key feature that sets DirectRSIM apart. Upon entry, the DirectRSIM processor simulator processes the execution path provided by the instrumentation code, attempting to bring each instruction from the path into its instruction window.

Key functionality, data structures, and simulation clocks. The key work done by the processor simulator is: (1) keeping track of true dependences and structural hazards, and determining when instructions complete or when reads and writes can be issued based on these dependences, (2) retiring instructions from the instruction window at appropriate times based on the above completion times, (3) maintaining branch prediction tables, and (4) memory forwarding (i.e., if a read is ready to issue while a previous write to the same location is pending, then the write's value is forwarded to the read). Of these, the key motivation for the DirectRSIM processor model is the need to maintain data dependences properly; we cannot simply use the processor model of Simple-ix (Section 6.1.2) because it has no mechanism to prevent multiple non-blocking reads from being issued even if there were a dependence from one to the next.

The key data structures in the processor simulator are (1) a structure analogous to the reorder buffer or instruction window of an ILP processor, (2) a read queue and a write queue to track memory accesses that need to be issued, (3) a structure to track outstanding writes, hashed on their addresses for efficient forwarding (4) the branch prediction table, and (5) a structure for tracking structural hazards for functional units.

The memory system and event-driven simulation engine of DirectRSIM provide a global view of time in the system. However, unlike RSIM, the processors are not required to be in lockstep with the global clock when performing internal actions. Each processor is allowed to maintain local views of the clock that run ahead of the global clock, as long as it synchronizes with the global clock before issuing any instruction to the memory system. The completion timestamps of individual instructions are one type of localized clock. Additionally, each processor simulator has two other views of time: a fetch time and a retire time. Instructions are marked with the value of the fetch time when they are fetched into the window, and the processor retires instructions from the head of its instruction window according to the value of the retire time (as further explained below).

Instruction issue and completion. As the processor simulator brings instructions into its simulated instruction window, it tags non-memory instructions with their completion times, if known. The completion time for an instruction is known as long as it is not directly or indirectly data dependent on any incomplete reads. For such an instruction, the completion timestamp depends on its latency and the availability of a functional unit. The latter is approximated by tracking the future use of functional units by instructions whose completion times are already known; it is possible that some instructions with unknown completion times may interfere with the current instruction, but this effect is not modeled. If an instruction's completion time is not immediately known, it is attached to the instructions on which it is dependent; its completion timestamp will be set upon completion of these instructions.

For a read instruction, the processor simulator calculates a timestamp for the time when the read is ready to issue (if known), and inserts it in the read queue in issue time order. If the issue time is not known (due to dependencies on incomplete reads), then the read is attached to the instructions on which it is dependent and inserted into the read queue on completion of these instructions. When the global simulation time catches up with the issue time of a read, the processor simulator checks to see if the read can be forwarded from a previous write. This check is efficient since addresses for all previous writes are immediately known through direct execution, and can be matched through a hash table. If there is no forwarding, an event is scheduled for issuing the read to the memory system.

On forwarding, a completion time is marked for the read.

As with most current processor simulators, to ensure precise interrupts, a write instruction is marked ready for issue only when it reaches the top of the instruction window. At this time, the write will be inserted in the write queue with an issue timestamp equal to the current retire time. When the global time catches up with the issue time, an event for the issue of the write is scheduled.

Instruction fetch and retirement. Instruction fetching continues until either the instruction window or the read queue or the write queue fills up, or all instructions executed by the functional simulator since the last timing simulator invocation are processed, or there is a misspeculation. On a misspeculation, instruction fetching continues once the misspeculation penalty is determined. When the instruction window is full, the processor simulator tries to retire the first set of instructions. Retirement is an entirely local action; the head of the instruction window can always be retired unless it is an incomplete read. The processor's retire clock is possibly updated based on the completion time of the retiring instruction and the number of instructions that have already retired at that time relative to the processor's peak retire rate.

Suspending and resuming processor simulation. A processor's simulation (and its corresponding direct execution process) is suspended when its instruction window is full, it cannot retire any further instructions, and no other reads or writes can be issued (either because they are dependent on other reads, or because the cache ports are full, or because the global time has not caught up with their issue time yet). At this point, the processor stalls in a state waiting for an action that will allow progress on any of the above situations. A processor's direct execution may be resumed once all of its directly executed instructions so far have been entered in its instruction window.

The timing simulator of DirectRSIM effectively acts as a trace-driven simulator operating on the trace of instructions executed since its last invocation by the same process. DirectRSIM, however, is still execution-driven because the simulated application's execution path is affected by the dynamic ordering of synchronization accesses and contention. For uniprocessor simulation, however, DirectRSIM would effectively become a trace-driven simulator.

6.4 Evaluation Methodology

We compare the accuracy and speed of four shared-memory simulators. The following sections describe the systems modeled, the applications studied, the simulators evaluated, and the metrics for evaluation.

6.4.1 Simulated architectures

The base system architectures simulated in this chapter are very similar to the base systems simulated in the other chapters. However, there are a few important differences: this chapter uses a MESI cache-coherence protocol instead of MSI, a SPARC v8 architecture instead of SPARC v9, 4 functional units of each kind instead of 2, 8 MSHRs per cache instead of 10, round-robin instead of permutation-based memory interleaving, single-cycle functional units, and a perfect instruction cache model. These changes are not expected to have much relative impact on either accuracy or simulation speed among the various simulators. Table 6.1 summarizes the key system parameters for our base system. Results for five variations of the base system are also reported, as described in Section 6.5.

The L1 and L2 cache sizes follow the methodology of Woo et al. [WOT+95] for our application input sizes (described in Section 6.4.2). All primary working sets in these applications fit in the L1 cache, while the secondary working sets do not fit in the L2 cache. Currently, a perfect instruction cache and TLB are modeled since the application suite is known to have a small instruction cache and TLB miss ratio.

6.4.2 Applications

We study 5 applications – Erlebacher, FFT, LU, MP3D, and Radix. The LU and FFT versions studied are clustered using uncontrolled interchange (as discussed in Section 4.1): these are different from the base and clustered versions studied in previous chapters. No

ILP processor parameters	
Processor speed	500MHz
Fetch/decode/retire rate	4 per cycle
Instruction window	64 instructions in-flight
Memory queue size	32
Outstanding branches	8
Number of functional units	4 ALUs, 4 FPUs, 4 address generation units
Functional unit latencies	1 cycle
Memory hierarchy and network parameters	
L1 cache	16 KB, direct-mapped, 2 ports, 8 MSHRs, 64-byte line
L2 cache	64 KB, 4-way associative, 1 port, 10 MSHRs, 64-byte line, pipelined
Memory banks	4-way, round-robin interleaving
Bus	167 MHz, 256 bits, split transaction
Network	2D mesh, 250MHz, 64 bits, flit delay of 2 network cycles per hop
System latencies in absence of contention	
L1 hit	1 processor cycle
L2 hit	10 processor cycles
Local memory	85 processor cycles
Remote memory	180-260 processor cycles
Cache-to-cache transfer	210-310 processor cycles

Table 6.1 : Base system parameters. The number of processors varies by application, as described in Section 6.4.2 and Table 6.2.

read miss clustering transformations have been applied to the other codes. (The use of read miss clustering would most likely cause even larger errors in the Simple simulators, since these transformations increase the amount of memory parallelism and the Simple simulators do not model this ILP performance feature.) Additionally, Mp3d has not been optimized for locality as in the previous chapters.

Table 6.2 lists the input data sizes (chosen so that the simulations complete in reasonable time) and the number of processors simulated for each application (based on the scalability of the application for the input size used).

Application	Input Size	Processors
Erlebacher	64x64x64 cube, block 8	16
FFT	65536 points	16
LU	256x256 matrix, block 8	8
Radix	512K keys, max: 512K, 1024	8
Mp3d	50000 particles	8

Table 6.2 : Application input sizes and number of simulated processors.

6.4.3 Simulators

We compare DirectRSIM with RSIM (the only publicly available detailed ILP-processor based shared-memory simulator), and Simple and Simple-*i*x (two representative simple-processor based direct execution simulators). RSIM and DirectRSIM directly model the ILP processor described in Section 6.4.1. Simple and Simple-*i*x use a simple-processor model to approximate the ILP processor, using previous direct execution methodology (Section 6.1.1). We chose these two simple-processor approximations since they are the most widely used and the best reported such approximations respectively. Recall that to model the 500 MHz 4-way base ILP processor, Simple-4x models a 2 GHz single-issue processor. To ensure that the speed of our simple-processor based simulators is representative of the state-of-the-art, we also compared Simple to the recently released Wisconsin Wind Tunnel-II (a simple-processor based direct execution simulator) and found the speed of the two simulators to be comparable (Section 6.6.1).

The differences between our four simulators are limited to the processor model and its interaction with the cache hierarchy. The memory system simulation in all simulators uses nearly identical code. It is based on an event-driven simulation engine [CDJ$^+$91], where events for the simulated system modules are scheduled by inserting them on a central event queue, and are triggered by a central driver routine.

A few differences between RSIM and the other simulators arise because of the inherent differences between detailed and direct execution based simulation. The direct execution simulators use user-level lightweight processes to provide the register and stack state needed by each simulated processor for direct execution. Each activation of a process incurs the overhead of a lightweight context-switch. RSIM does not use lightweight processes, as it simulates all register and stack state in software. Instead, it uses a special event that occurs every cycle and examines the state of each processor, L1 cache, and L2 cache, scheduling any external events triggered by these parts of the system in the event queue. Effectively, RSIM simulates the processors and caches on a cycle-by-cycle basis, since in a detailed ILP processor simulation it can be expected that some processor or cache will have some event scheduled every cycle.

Additionally, the direct execution simulators optimize L1 cache hits whenever the cache is guaranteed to have ports available and not be stalled for resources such as MSHRs. In these cases, the processor simulator itself accounts for the impact of the hit on execution time without forwarding the request to the L1 cache module. The processor may, however, still have to stall to allow the global simulation clock to catch up with the issue time of such an access. RSIM issues all hits to the caches, consistent with its cycle-by-cycle detailed simulation policy.

6.4.4 Metrics

The accuracy of a direct execution simulator is evaluated based on the execution time it reports for the simulated application (excluding initialization), relative to the time reported by RSIM. Since all simulators use nearly identical code for the memory system, the discrepancy in simulated execution times occurs solely due to the level of detail in the processor models. To gain further insight, we also report three components of the execution time – CPU time, memory stall time, and synchronization stall time – calculated as described in Chapter 2.

To determine simulator speed, the elapsed (wall-clock) time is measured for each simulation when run on an unloaded single 250MHz UltraSPARC-II processor of a Sun Ultra Enterprise 4000 server with 1GB memory and 1MB L2 cache. The simulators were all compiled using the Sun C 4.2 compiler with the highest practical level of optimization.

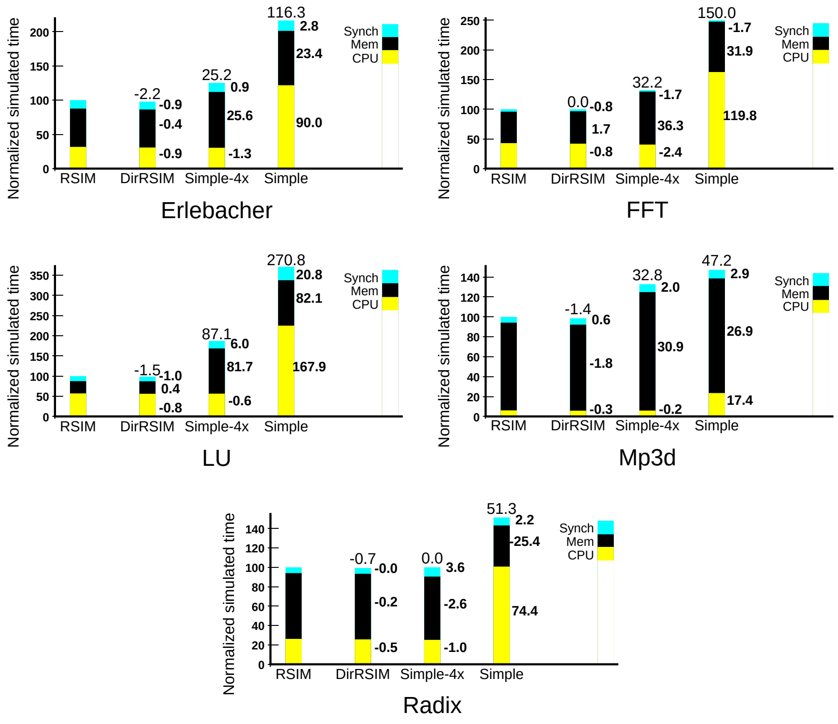


Figure 6.1 : Simulator accuracy for the base system.

The time spent in the initialization phase of the application is not included, since this time is not reported in the simulated execution time and can be sped up in various ways orthogonal to the rest of the simulation methodology.

6.5 Results on simulator accuracy

6.5.1 Base system configuration

Figure 6.1 shows the simulated execution time and its components reported by each simulator for each application on the base system configuration, normalized to that for RSIM. The number above each bar in the figure gives the percentage error in total execution time relative to RSIM. Numbers shown at the side of a bar represent the breakup of the total error among the three components of execution time.

Figure 6.1 shows that DirectRSIM reports overall simulated execution time very close to RSIM on all our applications, with a maximum error of 2.2%. This is a striking improvement over the best previous approximation of Simple-4x, which sees an execution time error of 87% for LU and 25% to 33% on three other applications studied. The Simple simulator sees much larger errors, ranging from 47% to 271%.

The differences between the four simulators arise from their abilities to capture the benefits that ILP provides to the various components of execution time. As discussed in Chapter 2, ILP reduces the CPU component of execution time by issuing multiple instructions at a time and by issuing instructions out of order. ILP can sometimes reduces the memory component primarily by overlapping multiple long latency memory operations with each other, or also by overlapping memory latency with CPU instructions. ILP can also increase the memory component by increasing contention for resources or by changing an access pattern. Synchronization time is negligible for all our applications, and is not discussed further.

The Simple model cannot capture the effects of ILP on either the CPU or memory stall component of execution time. Simple-4x models much of the benefit for the CPU component (because its clock speed is increased by a factor equal to the issue width of the processor). Most of the errors seen by Simple-4x are in the memory stall component, primarily because Simple-4x does not allow multiple read misses to overlap with each other. Thus, this method cannot properly capture ILP-specific improvements in the memory stall component of execution time.

DirectRSIM models the impact of ILP in both CPU and memory stall components of execution time, and provides a closer and more consistent approximation to the functionality of detailed execution-driven simulators.

6.5.2 Other system configurations

Table 6.3 summarizes the variations on the base system configuration studied in this section. These configurations are intended to capture trends towards higher processor clock

Configuration	**Difference from the base configuration**
Lat. x2	Roughly twice the local and remote memory latencies.
Lat. x3	Three times the local memory latency, and a minimum contentionless remote-to-local latency ratio of 3:1.
ILP+	Processor is twice as aggressive, with double the instruction issue width, instruction window size, processor memory unit size, functional units, branch-prediction hardware, cache ports, and MSHRs.
ILP++	Same as ILP+, but with four times the instruction window size, memory unit size, and MSHRs as the base.
C. net	Constant-latency 50-cycle network instead of a 2-D mesh network.

Table 6.3 : Variations on base configuration.

speeds, larger remote to local memory latency ratios, aggressive processor microarchitectures, and aggressive network configurations. In the ILP+ and ILP++ configurations, Simple-8x is used rather than Simple-4x.

Tables 6.4(a), (b), and (c) show the percentage errors in total execution time relative to RSIM as seen by DirectRSIM, Simple-ix, and Simple respectively for the various system configurations (the first row in the tables repeats the data of the base configuration shown in Figure 6.1). DirectRSIM continues to see very low errors, with an average of 1.3% and a maximum of 3.9%. In contrast, the errors with Simple-ix remain high for most of the applications, and continue to vary widely, ranging from 0% to 128%, with an average of 46%. The errors seen with Simple are even higher, ranging from 9% to 438%, averaging 137%. As with the base configuration, most of the error with Simple-ix comes from the memory component, while the error with Simple comes from both the CPU and the memory component; these component discrepancies are shown in Tables 6.5 and 6.6. As expected, the errors are greatest in the applications with the most read miss overlap. This application characteristic becomes even more important for systems with future aggressive processors (e.g., ILP+ and ILP++), as seen by the increase in error with Simple and Simple-ix for these configurations.

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	-2.2	0.0	-1.5	-1.4	-0.7	1.2
Lat. x2	-1.6	-1.5	-2.0	-0.7	0.0	1.2
Lat. x3	-0.7	2.2	-0.7	0.0	-0.3	0.8
ILP+	-2.8	0.2	-3.5	-3.9	-0.8	2.2
ILP++	0.7	-1.2	-2.4	-0.9	-0.8	1.2
C. net	-1.5	-0.8	-1.6	-0.5	-0.5	1.0
Avg.	1.6	1.0	1.9	1.2	0.5	1.3

(a) % error in execution time for DirectRSIM relative to RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	25.2	32.2	87.1	32.8	0.0	35.5
Lat. x2	27.5	35.0	109.0	31.9	3.6	41.4
Lat. x3	27.6	38.4	90.5	23.3	2.0	36.4
ILP+	31.4	50.0	122.4	58.6	4.0	53.3
ILP++	69.8	58.8	127.8	98.2	10.1	72.9
C. net	23.1	29.7	84.7	28.1	3.6	33.8
Avg.	34.1	40.7	103.6	45.5	3.9	45.5

(b) % error in execution time for Simple-ix relative to RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	116.3	150.0	270.8	47.2	51.3	127.1
Lat. x2	77.7	99.5	232.4	38.8	22.7	94.2
Lat. x3	54.5	73.4	147.6	25.8	9.1	62.1
ILP+	156.3	227.8	425.1	78.2	68.0	191.1
ILP++	231.2	247.1	437.8	122.8	77.8	223.3
C. net	110.6	145.8	264.9	38.3	55.9	123.1
Avg.	124.4	157.3	296.4	58.5	47.5	136.8

(c) % error in execution time for Simple relative to RSIM

Table 6.4 : Simulator accuracy for all configurations. (Averages are over absolute values of the errors.)

In conclusion, DirectRSIM achieves significantly greater and more reliable accuracy than Simple-ix or Simple in a variety of current and future multiprocessor configurations.

6.6 Results on simulator speed

6.6.1 Comparing Simple with Wisconsin Wind Tunnel-II

Before we compare the speed of our simulators, we seek to insure that our baseline Simple simulator has speed representative of the state-of-the-art. We compare our Simple simulator with the Wisconsin Wind Tunnel-II (WWT-2). (We could not perform an analogous experiment for RSIM, since it is the only publicly available detailed ILP-processor based shared-memory simulator.)

We choose WWT-2 for comparison since it has been used in many architectural studies and represents the state-of-the-art, it is publicly available, and it simulates SPARC executables similar to Simple. WWT-2 supports parallel simulation, but we use it in sequential mode (similar to Simple) since parallel performance is an orthogonal issue. Parallel simulation technology could be applied to all our simulators but is beyond the scope of this work.

The comparison is based on three applications – LU, FFT, and Radix. These are the only common applications among those used in this study and in the application suite distributed with WWT-2. The application input sizes and the hardware used to run the simulators in this comparison are the same as in the rest of this chapter. For the simulated system, similar parameters were used for both simulators to the extent possible. For example, a constant latency network configuration (100 cycle latency) and direct mapped caches were used in both simulators as these are the only options supported by the released version of WWT-2. Similarly, since WWT-2 only supports a single level cache, both cache levels in Simple were set to be the same size; all caches in both simulators are the L2 cache size used in the rest of this chapter.

Nevertheless, a completely fair comparison among the two simulators is difficult because of differences in the modeled architectures. The most important difference for our purposes is that the released version of WWT-2 does not support a CC-NUMA protocol. The closest protocol to CC-NUMA in WWT-2 is S-COMA. We used this protocol in WWT-2 with a hardware stache size of 320K based on a WWT-2 based study of coherence protocols [FW97]. That study showed that S-COMA is comparable to CC-NUMA for FFT, CC-NUMA performs worse than S-COMA for LU, and S-COMA performs worse than CC-NUMA for Radix.

Our simulator speed measurements, summarized in Table 6.7, closely follow the above trend – WWT-2 and Simple show similar speed for FFT, Simple is significantly slower for LU, and WWT-2 is significantly slower for Radix. There is less than 15% difference between the simulators when comparing the sum of the simulation times for the three applications.

Undoubtedly, there are many factors that contribute to the above results. Our goal is simply to show that the Simple simulator is in the same class as other widely used simulators. We believe the results in this section provide such evidence, and confirm that the experimental infrastructure of this study is representative of the state-of-the-art.

6.6.2 Comparing RSIM, DirectRSIM, and Simple

Figure 6.2 graphically depicts the elapsed times for the four simulators in the base configuration for each application, normalized to the time for RSIM. The number above the bars for DirectRSIM, Simple-ix, and Simple are the speedups achieved by those simulators over RSIM. Since the elapsed times for Simple and Simple-ix are always similar and since Simple gives much larger errors, we do not discuss the speed of Simple any further. Table 6.8 tabulates speedup for each pair of simulators, for all configurations in Table 6.3. As a reference for absolute speed, RSIM simulates an average of roughly 20,000 instructions per second for the base configuration (more data on absolute speed is provided in Table 6.9).

Simple-ix gives the best elapsed time, with an average speedup of 9.7 over RSIM. DirectRSIM has some additional overheads from processor simulation, but still sees an

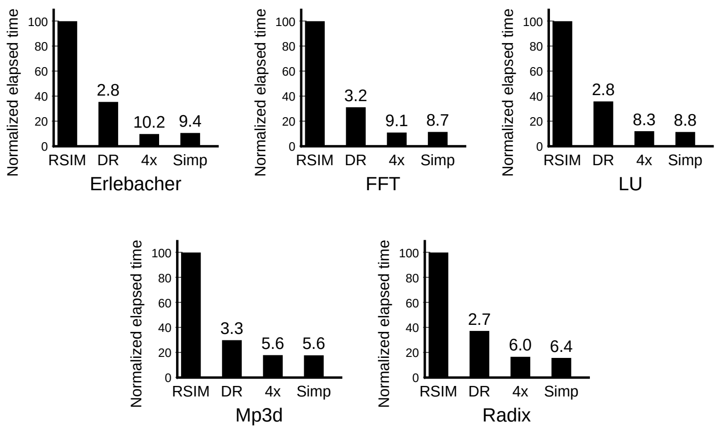


Figure 6.2 : Simulator speed for the base system. DR=DirectRSIM, 4x=Simple-4x, Simp=Simple.

average speedup of 3.6 over RSIM. Of particular interest are the increases in DirectRSIM speedup for the longer-latency configurations, which represent configurations with faster processor speeds. DirectRSIM profits by switching from a largely cycle-driven simulator to a purely event-driven simulator, and so will be less sensitive to future increases in system latencies than RSIM. DirectRSIM also sees higher speedups in ILP+ and ILP++ by effectively targeting the more expensive processor simulation component seen by these aggressive microarchitectures.

Most notably, the speed advantage of Simple-ix is reduced to an average of 2.7X compared to DirectRSIM. The competitive speed of DirectRSIM indicates that the speed benefits of simple-processor based simulators may no longer be enough to justify their large inaccuracies in modeling current and future multiprocessor systems.

6.6.3 Detailed analysis of DirectRSIM speed

To further understand the reasons for the speed differences among the simulators, Figure 6.3 depicts their execution profiles for LU on the base configuration as reported by `prof` (with monitoring turned on only during the parallel phase of the application). The

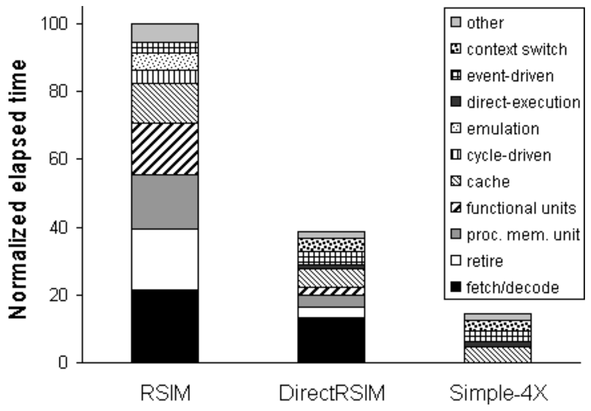


Figure 6.3 : Components of elapsed time for various simulators.

other applications show similar profiles. The function calls of the simulators are divided according to the logical tasks they perform. From the bottom to the top of each bar, these tasks are instruction fetch and decode (including dependence checking), instruction retirement, processor memory unit simulation, functional unit management, cache simulation, cycle-driven simulation management, instruction emulation, direct-execution, event-driven simulation management, context switching among lightweight processes, and other tasks (e.g., memory and network simulation, and branch speculation). Not all tasks are present in all simulators.

DirectRSIM vs. RSIM. DirectRSIM improves speed relative to RSIM primarily by reducing the time spent simulating instruction fetch and decode, instruction retirement, the processor memory unit, and functional unit management. DirectRSIM's knowledge of values and addresses through direct execution enables more efficient register renaming and management of write-to-read forwarding, respectively. The provision to allow internal processor actions to proceed ahead of the global clock enables more efficient instruction fetching and retirement. The instruction dependence checking for issue is sped by the use of timestamps. Functional unit management is sped by the structure to approximately track future functional unit utilizations.

DirectRSIM also spends less time than RSIM in cache simulation by not simulating accesses that are known to hit in the L1 cache without contention. Among the remaining components of elapsed time (accounting for less than 20% of RSIM's total time), DirectRSIM eliminates the cycle-driven controller, but adds a component to handle context-switching and also increases event-driven simulation overhead. DirectRSIM avoids the overhead of instruction emulation (about 4% of RSIM time) and replaces it with a smaller component in direct execution. In the "other" category, DirectRSIM also uses values computed in direct execution to reduce the cost of mispredicted branches.

DirectRSIM vs. Simple-4x. As expected, most of DirectRSIM's overhead relative to Simple-4x stems from its processor simulation features. It also sees slightly more overhead in memory hierarchy simulation (due to increased resource contention from non-blocking reads).

6.7 Summary and Implications

This section presents a new simulation technique for shared-memory multiprocessors with ILP processors that combines the speed advantages of simple-processor based simulators with the accuracy of detailed ILP-processor based simulators. Our technique is based on a novel adaptation of direct execution. First, it allows a data read to proceed in direct execution even before its simulation has completed at the memory system. Second, it provides an efficient timing simulator that accounts for aggressive ILP features such as multiple issue, out-of-order issue, and non-blocking reads.

DirectRSIM, our implementation of the new technique, sees an average of 1.3% error (maximum of only 3.9%) in simulated execution time relative to RSIM for all studied applications and configurations. At the same time, DirectRSIM sees a speedup of 3.6 over RSIM. In contrast, the best current simple-processor based simulation methodology sees large and variable errors in execution time, ranging from 0% to 128%, and averaging 46%. The most commonly used simple-processor based simulation methodology sees errors ranging from 9% to 438%, averaging 137%. Despite its superior accuracy, Direct-

RSIM sees only a factor of 2.7X slowdown compared to current simple-processor based simulators. Although the speed advantage of simple-processor based simulators is still significant, it may no longer be enough to justify their high errors or their inapplicability to important classes of ILP-specific architectural studies. Our results, therefore, suggest a reconsideration of simulation methodology for evaluating shared-memory systems.

In the future, several features supported in other simulators can be added to DirectRSIM to further improve its speed and/or functionality. Examples include parallelization, sampling, instrumentation through executable editing or binary translation, instruction cache, TLB, and full simulation of system calls. We are not aware of any fundamental problems in incorporating such support for DirectRSIM. Finally, if desired, we believe that support for simulating mispredicted paths could also be incorporated.

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	-0.4	1.7	0.4	-1.8	-0.2	0.9
Lat. x2	-0.7	0.6	-0.4	-1.7	0.3	0.7
Lat. x3	-1.2	1.4	0.3	-1.8	0.2	1.0
ILP+	-0.3	1.0	1.2	-5.4	-0.2	1.6
ILP++	-5.5	0.8	1.8	-5.2	0.2	2.7
C.net	-0.8	1.2	0.2	-0.8	-0.0	0.6
Avg.	1.5	1.1	0.7	2.8	0.2	1.3

(a) % error due to the memory component for DirectRSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	25.6	36.3	81.7	30.9	-2.6	35.4
Lat. x2	26.3	36.7	100.7	29.5	2.5	39.1
Lat. x3	23.9	41.3	83.9	21.5	1.0	34.3
ILP+	33.1	56.7	117.1	55.1	3.8	53.2
ILP++	73.0	66.5	122.7	94.1	16.1	74.5
C.net	24.2	34.3	79.8	26.3	3.4	33.6
Avg.	34.4	45.3	97.6	42.9	4.9	45.0

(b) % error due to the memory component for Simple-$\imath$x

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	23.4	31.9	82.1	26.9	-25.4	37.9
Lat. x2	24.3	30.8	102.1	27.0	-18.2	40.5
Lat. x3	23.1	36.9	82.7	20.2	-6.8	33.9
ILP+	29.8	47.8	117.4	49.1	-28.7	54.6
ILP++	68.7	57.1	123.0	86.6	-18.3	70.7
C.net	21.8	28.5	80.6	21.1	-23.7	35.1
Avg.	31.9	38.8	98.0	38.5	20.2	45.5

(c) % error due to the memory component for Simple

Table 6.5 : Simulator errors due to the memory component. (Averages are over absolute values of the errors.)

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	-0.9	-0.8	-0.8	-0.3	-0.5	0.7
Lat. x2	-0.4	-0.6	-0.7	-0.1	-0.3	0.4
Lat. x3	-0.2	-0.3	-0.3	-0.1	-0.2	0.2
ILP+	-2.3	0.3	-3.2	-0.6	-0.6	1.4
ILP++	1.2	0.4	-2.1	-0.5	-0.8	1.0
C.net	-0.6	-1.2	-0.8	-0.3	-0.5	0.7
Avg.	0.9	0.6	1.3	0.3	0.5	0.7

(a) % error due to the CPU component for DirectRSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	-1.3	-2.4	-0.6	-0.2	-1.0	1.1
Lat. x2	-0.6	-1.3	-0.7	-0.1	-0.6	0.7
Lat. x3	-0.4	-0.7	-0.3	-0.0	-0.3	0.3
ILP+	-3.8	-5.4	-2.7	-0.7	-1.7	2.9
ILP++	-4.6	-5.6	-2.7	-0.7	-1.8	3.1
C.net	-1.3	-2.1	-0.6	-0.2	-1.1	1.1
Avg.	2.0	2.9	1.3	0.3	1.1	1.5

(b) % error due to the CPU component for Simple-ix

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	90.0	119.8	167.9	17.4	74.4	93.9
Lat. x2	50.7	69.7	112.9	9.0	39.1	56.3
Lat. x3	28.6	38.7	54.8	3.7	14.5	28.1
ILP+	122.4	181.2	275.8	24.2	96.2	140.0
ILP++	158.5	192.0	282.5	30.5	101.8	153.1
C.net	87.9	119.8	164.5	15.0	77.9	93.0
Avg.	89.7	120.2	176.4	16.6	67.3	94.0

(c) % error due to the CPU component for Simple

Table 6.6 : Simulator errors due to the CPU component. (Averages are over absolute values of the errors.)

Application	Simple	WWT-2
FFT	248 sec	263 sec
LU	394 sec	112 sec
Radix	395 sec	797 sec
Total	1037 sec	1172 sec

Table 6.7 : Simulation time for Simple and WWT-2.

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	2.8	3.2	2.8	3.3	2.7	3.0
Lat. x2	3.3	3.9	3.0	4.8	3.2	3.6
Lat. x3	3.7	3.8	3.5	5.9	4.8	4.3
ILP+	3.8	4.4	3.1	3.7	3.2	3.6
ILP++	2.9	4.2	3.2	5.0	3.7	3.8
C. net	3.1	3.3	3.0	4.4	2.7	3.3
Avg.	3.3	3.8	3.1	4.5	3.4	3.6

(a) Speedup of DirectRSIM over RSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	3.6	2.8	3.0	1.7	2.2	2.7
Lat. x2	3.3	2.8	2.8	1.6	2.5	2.6
Lat. x3	3.0	3.8	2.6	2.1	2.7	2.8
ILP+	3.2	2.7	2.9	1.6	2.2	2.5
ILP++	4.0	2.8	3.0	1.6	2.9	2.9
C. net	3.6	3.1	3.3	2.0	3.0	3.0
Avg.	3.4	3.0	2.9	1.8	2.6	2.7

(b) Speedup of Simple-ix over DirectRSIM

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	10.2	9.1	8.3	5.6	6.0	7.8
Lat. x2	10.9	10.9	8.3	7.8	8.0	9.2
Lat. x3	11.2	14.5	9.2	12.6	12.8	12.1
ILP+	12.1	11.6	9.1	6.0	7.3	9.2
ILP++	11.7	11.7	9.5	8.2	10.8	10.4
C. net	11.0	10.1	9.8	8.9	8.0	9.6
Avg.	11.2	11.3	9.0	8.2	8.8	9.7

(c) Speedup of Simple-ix over RSIM

Table 6.8 : Simulator speed for all configurations.

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	23.8	23.5	27.6	10.4	15.7	20.2
Lat. x2	20.7	20.4	25.5	8.2	13.9	17.7
Lat. x3	18.3	18.1	23.4	8.4	10.0	15.6
ILP+	18.6	18.6	23.2	9.0	13.3	16.5
ILP++	16.0	16.1	18.9	7.2	11.5	13.9
C.net	23.5	24.9	27.0	10.2	17.6	20.6
Avg.	20.1	20.3	24.3	8.9	13.7	17.4

(a) RSIM - Simulated kilo-instructions per second

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	66.4	75.6	70.0	32.8	42.4	57.5
Lat. x2	68.7	76.7	68.7	38.8	45.2	59.6
Lat. x3	76.2	76.5	74.5	50.3	50.5	65.6
ILP+	74.1	83.0	70.4	33.5	48.6	61.9
ILP++	71.3	79.0	69.4	29.6	47.0	59.2
C.net	72.8	80.9	73.4	42.4	48.0	63.5
Avg.	71.6	78.6	71.1	37.9	46.9	61.2

(b) DirectRSIM - Simulated kilo-instructions per second

	Erle.	FFT	LU	Mp3d	Radix	Avg.
Base	247.5	183.8	240.1	48.1	97.7	163.4
Lat. x2	263.8	190.2	245.9	56.8	99.0	171.1
Lat. x3	276.1	218.2	255.4	74.7	101.8	185.2
ILP+	200.7	224.1	190.8	56.9	131.2	160.7
ILP++	213.3	213.3	244.2	50.3	115.8	167.3
C.net	291.6	236.1	259.9	69.2	136.6	198.7
Avg.	248.8	210.9	239.38	59.3	113.7	174.4

(c) Simple-$\imath$x - Simulated kilo-instructions per second

Table 6.9 : Absolute speed of the simulators in thousands of instructions simulated per second.

Chapter 7

Fast Characterization of ILP Memory System Parameters

The previous chapter focused on improving evaluation methodology through fast and accurate simulation. Simulation is an essential part of the architectural design process, but is encumbered by its slow speed. Analytical modeling, on the other hand, is a much faster alternative for system performance evaluation. However, analytical system models often fail to represent real system performance accurately because they are driven by synthetic parameters rather than real workloads. This chapter presents FastILP, a fast high-level simulator that focuses on characterizing the fundamental ILP data memory system parameters of an application rather than directly trying to measure the timing of an application running on a specific simulated system. These parameters are then used to drive a fast an accurate analytical model of ILP multiprocessor system performance [SPA+98]. The parameters to be measured are largely independent of underlying system latencies, and thus apply across a broad range of systems to be studied.

This chapter only focuses on the parameters of the analytical model and how FastILP generates the ILP-specific parameters. More detailed information on the analytical model is available in the paper in which this methodology was presented [SPA+98].

7.1 Fundamental ILP Memory System Parameters

The key parameters required for characterizing the memory system behavior of a given application are as follows:

τ	Average time between read, write, or upgrade requests to main memory, *not counting the time when the processor is completely stalled or is spin-waiting on a synchronization event*
CV_τ	Coefficient of Variation of τ
$f_{synch-write}$	Fraction of write requests that are generated by atomic read-modify-write instructions or that coalesce with at least one later read
f_M	Fraction of processor stalls that find M MSHRs with outstanding read requests ($M \in [0, \#\mathrm{MSHRs}]$)
P_x	Probability that a memory request is of type x ($x \in \{\mathrm{read}, \mathrm{write}, \mathrm{upgrade}, \mathrm{writeback}\}$)
P_{wb}	Probability that a read or write request causes a writeback of a cache block
$P_{L\|x}$	Probability that directory is local for a type x transaction ($x \in \{\mathrm{read}, \mathrm{write}, \mathrm{upgrade}, \mathrm{writeback}\}$)
$P_{M\|x,y}$	Probability that home memory can supply the data for a type x, y request ($x \in \{\mathrm{read}, \mathrm{write}\}$, $y \in \{\mathrm{localhome}, \mathrm{remotehome}\}$)
$P_{3hop\|x\&not-mem}$	Probability that a request of type x to a remote home is forwarded to a cache at a third node ($x \in \{\mathrm{read}, \mathrm{write}\}$)
H	Average number of network switches traversed by a packet
X	Average number of invalidates caused by a write or upgrade to a clean line

Application parameters other than τ, CV_τ, f_M, and the part of $f_{synch-write}$ that is due to writes coalescing with later memory read requests do not depend directly on ILP features and can thus be measured using current fast simulators for multiprocessors with simple single-issue processors (e.g., [CDJ$^+$91, WR96]).

7.2 Fast High-Level Simulation

The remainder of this section provides an overview of FastILP, a fast high-level simulator for quickly estimating the key ILP parameters τ, CV_τ, f_M, and $f_{synch-write}$. Since FastILP does not need to measure the exact cycle count for an execution, it can achieve very high performance by abstracting out the details of both the ILP processor and the memory system, and modeling only enough state to generate the required ILP parameters. FastILP differs from conventional cycle by cycle ILP-based multiprocessor simulators in three key ways.

First, FastILP speeds up processor simulation using techniques from DirectRSIM. Each instruction in FastILP sets the timestamp of its destination register based on the completion time for that instruction. For non-memory instructions, the completion time is determined by the timestamps of the source registers of the instruction, and the availability of the appropriate functional unit. For memory instructions, the processor keeps enough state information to simulate memory disambiguation. The completion timestamp calculation for a memory instruction is unique to FastILP, as described below.

Second, FastILP speeds up memory system simulation by taking advantage of two observations: the ILP parameters do not depend on the exact latencies or configuration of the memory system, and L2 cache misses have high latencies that can be overlapped effectively only with other memory misses (as shown in the previous chapters). Using these observations, FastILP does not explicitly simulate any part of the memory system beyond the cache hierarchy. FastILP divides simulated time into distinct "eras," which start when one or more memory replies unblock the processor and end when the processor blocks again waiting for a memory reply. No memory replies return *during* an era. One or more replies return together at the beginning of each era, depending on whether the processor has enough work to completely overlap the time between incoming replies. For our results in Section 7.3, we assume that a 64-element instruction window does not provide enough computation to overlap this time between multiple replies, while a 128-element instruction window does. The use of eras allows FastILP to extend the concept of timestamps into the cache hierarchy as well, with each timestamp including both the era in which the data was ready, along with the cycle within the era. The parameters τ and CV_τ are calculated according to the points within each era at which misses occur; f_M is measured by counting the read requests outstanding at the end of each era. In this fashion, FastILP can process all instructions in-order, while still simulating an out-of-order processor.

Third, FastILP further speeds up simulation time by using trace-driven (as opposed to execution-driven) simulation, and by simulating the trace of only one processor. The use of trace-driven simulation is possible because FastILP does not need to account for synchronization spin time as this time is not measured in τ. Further, FastILP makes an approximation that mispredicted execution paths do not have a significant impact on the ILP parameters; this assumption is valid for the applications validated in Section 7.3. FastILP assumes homogeneous applications, allowing it to use the trace of only a single processor, if the trace provides information about memory accesses known to be communication misses. As communication misses generally stem from application and data set characteristics rather than processor microarchitecture or system latencies, such traces can be quickly generated by an appropriately instrumented fast simulator for multiprocessors with simple processors or a multiprocessor trace-generation utility.

Using the above optimizations, FastILP achieves two orders of magnitude speedup over RSIM, and over an order of magnitude speedup over DirectRSIM.

7.3 Validation of FastILP Parameters

The system architectures simulated in this chapter differ slightly from those in the previous chapter; the main differences are the use of an MSI cache coherence protocol (instead of MESI) and a slightly longer L2 cache access time (13 cycles instead of 10). All other representative contentionless system latencies are within 2% of those reported in the previous chapter, and the cache size, associativity, and MSHR parameters are all the same.

Table 7.1 gives the values of the ILP workload parameters as measured using RSIM, and Table 7.2 gives the values as generated by FastILP. The parameter $f_{synch-write}$ is omitted from the table since it is very small for most applications. The W column specifies the number of instructions in the instruction window in each case. The final column of Table 7.2, labeled *% err*, indicates the amount of difference seen in the throughput values reported by the analytical model using the RSIM and FastILP parameters. The differences are small for all the applications studied except for Water. FastILP underpredicts τ for Water because it does not yet model rollbacks of mispredicted reads triggered by write disambiguation. However, this problem is typically low in most applications. Additional experiments also indicate that FastILP can generate accurate parameters for codes that have

app.	W	τ	CV_τ	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\Sigma f_{i>8}$
Erle.	64	39.0	10.9	.65	.17	.09	.08	0	.01	0	0	0
	128	26.9	5.0	.64	.11	.10	.10	.02	.01	.01	.01	.01
FFT	64	63.9	12.8	.53	.47	0	0	0	0	0	0	0
	128	37.0	12.0	.12	.48	.39	0	0	0	0	0	0
LU	64	109.1	8.1	.51	.49	0	0	0	0	0	0	0
	128	74.0	3.6	.10	.19	.70	.01	0	0	0	0	0
Water	64	593.2	2.5	.73	.25	.01	0	0	0	0	0	0
	128	487.7	2.5	.49	.48	.01	.01	0	0	0	0	0

Table 7.1 : ILP workload parameters as measured by RSIM

app.	W	τ	CV_τ	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	$\Sigma f_{i>8}$	% err
Erle.	64	42.1	10.7	.61	.18	.11	.08	0	.01	0	0	0	-1.1
	128	23.2	9.6	.82	.02	0	.13	.02	0	0	.01	0	-0.5
FFT	64	56.6	12.5	.50	.50	0	0	0	0	0	0	0	9.1
	128	30.8	12.6	.14	.44	.43	0	0	0	0	0	0	-11.9
LU	64	104.6	3.8	.53	.47	0	0	0	0	0	0	0	-0.1
	128	78.6	3.4	.20	.06	.74	0	0	0	0	0	0	-8.6
Water	64	418.8	3.1	.71	.26	.01	0	0	0	0	0	0	20.0
	128	270.4	3.8	.05	.93	0	.01	0	0	0	0	0	59.5

Table 7.2 : ILP workload parameters as generated by FastILP

been clustered with loop interchange [SPA+98].

Additionally, the resulting analytical model was found to match the performance reported by RSIM within 12% across a variety of applications and configurations [SPA+98]. Thse configurations included systems with different bus speeds, directory access times, and memory access times. The same workload parameters can be used across many different simulated configurations because of the largely latency-independent definition of τ: this parameter explicitly excludes stall times.

7.4 Summary

The combination of fast high-level simulation and analytical modeling allow this evaluation methodology to provide fast and accurate timing estimates for ILP multiprocessor applications and configurations. FastILP addresses the workload characterization problem faced by many analytical models, while the analytical model provides the speed necessary to enable study of many independent configurations. FastILP may also be useful for high-level application characterization even without the analytical model, since the parameters it generates directly relate to memory parallelism (f_M) and contention (τ, CV_τ). These two types of concerns have been shown in the previous chapters to be important factors in determining ILP data memory system performance.

Because of the approximations and assumptions inherent to both FastILP and the analytical model, this combination cannot simply replace detailed simulation. Rather, this process can guide simulation by quickly narrowing the architectural design space to only those systems that seem most promising. The slower process of detailed simulation would then need to consider only a small subset of the architectural choices.

Chapter 8

Related Work

8.1 Work related to ILP memory system performance

There have been very few multiprocessor studies that model the effects of ILP. Albonesi and Koren provide a mean-value analysis model of bus-based ILP multiprocessors that offers a high degree of parametric flexibility [AK95]. However, the ILP parameters for their experiments (e.g., overlapped latency and percentage of requests coalesced) are not derived from any specific workload or system. Our simulation study shows that these parameters vary significantly with the application and hardware factors, and provides insight into the impact and behavior of the parameters. Furthermore, their model assumes a uniform distribution of misses and does not properly account for read clustering, which we have shown to be a key factor in providing read miss overlap and exploiting ILP features.

Nayfeh et al. considered design choices for a single-package multiprocessor [NHO96], with a few simulation results that used an ILP multiprocessor. Olukotun et al. compared a complex ILP uniprocessor with a one-chip multiprocessor composed of less complex ILP processors [ONH$^+$96]. There have also been a few studies of memory consistency models using ILP multiprocessors [GGH92, PRAH96, ZB92]. However, none of the above work details the benefits achieved by ILP in the multiprocessor.

There exists a large body of work on the impact of ILP on uniprocessor systems. Several of these studies also identify and/or investigate one or more of the factors we study to determine read miss ILP speedup, such as read clustering, coalescing, and contention. Oner and Dubois evaluate several applications on a uniprocessor system with non-blocking caches [OD93]. This work identifies a *critical latency* for each program, defined as the maximum cache miss latency that the system can perfectly tolerate. They find that for greater cache miss latencies, some latency tolerance is still possible if the program can overlap multiple misses together. Farkas and Jouppi found that numerical codes were able to see benefits in CPI from multiple outstanding read misses if the hardware provided the needed resources [FJ94]. Seznec and Lloansi consider effective cache miss penalties for out-of-order superscalar processors with non-blocking caches [SL95]. They briefly discuss the possibility of multiple outstanding cache misses, but do not give any information about which applications or to what extent this technique (as opposed to merely overlapping cache misses with other computation) actually provides benefits. Our work finds that read clustering is a key optimization that enables multiprocessors to exploit the features of ILP processors. We additionally find that the read misses clustered must have similar latency in order to achieve effective overlap, due to the dichotomy between local and remote miss latencies in a multiprocessor configuration. Our work is also the first to provide specific code transformations to improve clustering.

Butler and Patt investigate the impact of data cache misses on ILP processors [BP91]. Their study mentions the effect of coalesced requests in a non-blocking cache, but compares performance only among ILP configurations, rather than with a base processor.

Bacon et al. identified a problem they call *cache miss jamming*: if an architecture allows non-blocking reads but holds a critical resource on a memory system reply, two subsequent misses can take more than twice a single miss [BCJ$^+$94]. This problem would seem to make read miss clustering ineffective and potentially harmful. However, the architecture they studied had a latency of only 8 cycles for the critical word of a cache miss, followed by 7 cycles in which the data response exclusively held the cache data port. Modern systems see much greater initial memory latency (in cycles) for an external cache miss [LL97], and also hold critical resources for fewer cycles on a reply [MIP96]. As a result, modern systems provide opportunities for memory parallelism, as confirmed by our results on the Convex Exemplar.

Burger and Goodman evaluate several ILP processor configurations, finding that techniques used in such systems to tolerate latency can lead to increased contention for system bandwidth [BGK96]. Their study shows that this contention can contribute significantly to execution time in some SPEC95 applications. Our study finds that although bandwidth is an important factor in the performance of ILP-based multiprocessors, even systems with high bandwidth are limited in their ability to exploit ILP if applications do not allow sufficient clustering of read misses.

We have described previous applications of unroll-and-jam primarily in terms of their benefits for scalar replacement. Unroll-and-jam can also be used to increase floating-point unit parallelism in certain loops with recurrences carried on the inner loop but not on an outer loop [CCK88]. Such use of unroll-and-jam can also improve the interaction of unroll-and-jam with software-pipelining in the presence of inner-loop recurrences [CDS96]. More recent work has extended the heuristics used in unroll-and-jam by incorporating the effects of cache misses and prefetching into the balance calculation used to determine the degree of unrolling [Car96]. However, previous work has not sought to exploit unroll-and-jam either to encourage memory parallelism or to improve the effectiveness of prefetching.

In a sense, the clustering transformations of Chapter 4 improve performance for cache-line sharing recurrences by disrupting spatial locality just enough to increase memory parallelism. Although the use of single-word cache lines would naturally eliminate cache-line sharing, this approach seems unlikely for general-purpose processors. In particular, many important codes can effectively exploit the benefits of spatial locality, and a variety of static and dynamic transformations can improve spatial locality further [Tha82, DK99, LM99]. Where spatial locality is beneficial, the use of multi-word cache lines helps to amortize the fixed overhead costs of memory transfer, such as bus and interconnect arbitration, request and reply control messages, initial DRAM access latency, and request pin bandwidth. Eliminating spatial locality would also have no impact on address dependences, which are important in many codes.

8.2 Work related to software prefetching

Chapters 3 and 5 discusses the previous software prefetching material that our study uses most directly.

Gornish evaluated prefetching for both single-issue and multiple-issue statically-scheduled multiprocessors with non-blocking reads [Gor95]. The prefetches rely on software cache-coherence and require complete cache flushes before and after each parallel loop. Processor pipelines and functional-unit contention are not modeled. Gornish's study integrates hardware and software prefetching support, dynamically adapting hardware prefetching distance according to the latency of each reference. The study finds that software prefetching provides execution time improvements on their multiple-issue system similar to or greater than those seen with their single-issue system, but does not analyze the interaction of ILP features with prefetching.

Tullsen and Eggers evaluated software-controlled non-binding prefetching on a bus-based multiprocessor with simple processors [TE95]. They characterized bandwidth needs of their applications, and found that the benefits of prefetching degrade as bandwidth needs increase. They additionally found that increasing the prefetch distance can reduce late prefetches but can also increase early prefetches, and thus does not significantly improve performance. We discuss this same behavior in terms of short steady-states; techniques to increase the prefetch distance can negatively impact the length of the steady-state and the effectiveness of steady-state prefetching.

Previous prefetching techniques have also provided some memory parallelism among prefetches, but their main focus has been on fetching sufficiently in advance. Roth and Sohi discuss memory parallelism among prefetches, but they use parallelism largely to facilitate fetching ahead and make no attempt to fully utilize the resources for parallelism [RS99]. In contrast, read miss clustering restructures the code aiming to fully utilize the outstanding miss buffers of the nonblocking cache, increasing the parallelism achieved by the subsequent application of prefetching.

Saavedra et al. also observed that tiled codes were more difficult to prefetch [SMP+96].

They suggest optimizations to improve cache performance for tiled codes, but do not target the fundamental problem of short steady-states. As a result, their analysis generally favors prefetching alone over the combination of prefetching and tiling. We show that using unroll-and-jam for read miss clustering can help to address short steady-states by decreasing the needed prefetch distance. The use of read miss clustering thus allows us to maintain the bandwidth conservation of tiling while also improving the effectiveness of prefetching.

Just as prefetching can be seen as the general-purpose processor analogy to vector computing, clustered prefetching relates closely to the Multi Streaming Processor model of the more recent Cray SV series of vector architectures [BBC$^+$00]. In particular, both models seek to extract parallelism at outer-loop levels in order to avoid performance degradation from short inner loops.

This work has also focused on software latency tolerance techniques. Hardware techniques such as hardware prefetching or multithreading also provide latency tolerance [CB94, GHG$^+$91, Jou90, TEE$^+$96]. The interaction of clustered prefetching with such hardware latency tolerance techniques in ILP systems remains an open question.

8.3 Work related to ILP multiprocessor simulation

Chapter 6 reviewed the previous shared-memory simulation techniques most relevant to this work.

Like RSIM, SimOS with the MXS processor simulator and Armadillo also model ILP multiprocessors explicitly and in detail [GC98, RBDH97]. They use detailed execution-driven simulation, interpreting every instruction and simulating its effects on the complete processor pipeline and memory system in software. SimOS with MXS was developed concurrently with RSIM, while Armadillo was developed after RSIM. Unlike RSIM, neither is currently released for public distribution.

Simulation models based on direct-execution have been widely used for fast and accurate modeling of shared-memory systems with simple processors [CDJ$^+$91, DGH91, MRF$^+$97]. The use of variants of simple-processor-based models to approximate ILP multiprocessor system performance draws from publications by Heinrich et al. [HKO$^+$94] and Holt et al. [HSH96]. Both studies aim to model ILP processor behavior with faster simple processors, but neither work validates these approximations.

The Wisconsin Wind Tunnel-II uses a more detailed analysis at the basic-block level that accounts for pipeline latencies and functional unit resource constraints to model a superscalar HyperSPARC processor [FW97, MRF$^+$97, RPW96]. However, this model does not account for memory overlap, which, as our results show, is an important factor in determining the behavior of more aggressive ILP processors.

Brooks et al. describe the Cerberus Multiprocessor Simulator, a parallelized instruction-driven simulator for single-issue statically-scheduled processors with non-blocking reads in a cacheless “dance-hall” memory system [BAD89]. This is the earliest execution-driven multiprocessor simulator of which we know that modeled some degree of ILP. It was also used in a study of relaxed consistency models [ZB92].

Dynamic binary translation is sometimes used to speed up simulation [RBDH97] (as an alternative to direct execution). For our purposes, this technique can also be seen as a form of direct execution as it also decouples functional and timing simulation and executes most of the translated application directly on the host. Hence, the techniques presented in Chapter 6 can also be applied to dynamic binary translation.

In addition to direct-execution, sampling [RBDH97] and parallelization [MRF$^+$97] are used to speed up shared-memory simulation. Both techniques are orthogonal to ours and can be used in conjunction with DirectRSIM.

Concurrently, Krishnan and Torrellas have proposed a method similar to ours for direct-execution for ILP multiprocessors [KT98]. They do not discuss the potential for error (or solutions) when using values of non-blocking reads in direct execution. They also do not assess the accuracy of their simulator or compare performance with detailed simulation. Their performance comparison with a previous simple-processor simulator is done without memory system simulation, and shows slowdowns of 24–29X.

Schnarr and Larus concurrently developed a direct execution simulator for uniprocessors with ILP processors [SL98]. They simulate mispredicted paths and also propose instruction-window memoization. The applicability and/or benefits of some of their techniques for shared-memory multiprocessors are currently unclear (e.g., speculative stores and memoization). Further, their approach focuses on accurate microarchitectural simulation. DirectRSIM allows approximations, since we focus on accurate memory simulation in a multiprocessor with only as much emphasis on microarchitectural simulation as needed for correct memory simulation. It would be promising to consider the combination of fast multiprocessor simulation and speculative memoized simulation for future work.

Chapter 9

Conclusions and Future Directions

This chapter summarizes the analysis and insights of this dissertation and discusses future directions for research motivated by the results of this dissertation.

9.1 Conclusions

This study finds that for the applications and systems that we study, ILP hardware techniques effectively address the CPU component of execution time, but are less effective in reducing the data memory component of execution time, which is dominated by read misses to main memory. As a result, data read stall time becomes a larger bottleneck than in previous-generation systems. These deficiencies in the impact of ILP techniques on memory stall time arise primarily because of insufficient potential in our applications to overlap multiple read misses, as well as system contention from more frequent memory accesses. While software prefetching improves memory system performance with ILP processors, it does not change the memory-bound nature of these systems for most of the applications. We particularly show some of the factors that can limit prefetching and show that ILP systems can exacerbate these limitations.

The above observations motivate novel techniques to tolerate or reduce memory latency. We specifically propose ILP-specific code transformations to improve memory system parallelism by clustering multiple read misses together within the same instruction window (read miss clustering). We also show that read miss clustering and software prefetching can be profitably combined to address limitations in either technique alone. These new software latency-tolerance techniques substantially reduce application execution time by exploiting memory parallelism.

The performance benefits of memory parallelism also impact evaluation methodologies for ILP-based systems. In particular, simulators that model non-ILP processors can see large and highly application-dependent errors when used to approximate the performance of ILP processors running applications that can exploit memory parallelism. However, we found that these non-ILP simulators were as much as an order of magnitude faster than the detailed ILP simulator RSIM, making them attractive despite their shortcomings. We then presented a new simulation methodology that extends the fast direct-execution simulation methodology to support ILP. The resulting simulator, DirectRSIM, maintains accuracy close to RSIM while significantly narrowing the performance gap between the Simple and ILP simulation models. We also present a new high-level simulator called FastILP, which exploits analytical modeling in order to improve evaluation speed. In particular, FastILP simply generates the ILP-specific workload parameters needed for use by an underlying analytical model of ILP multiprocessor performance. As a result, FastILP can abstract out most processor and memory system details and achieve a simulation speed up to 100X that of RSIM.

9.2 Future directions

This dissertation prompts several new directions for achieving high performance by exploiting instruction-level and memory system parallelism. At a high level, our insights show that current and future aggressive hardware provides high peak performance, but current software tools do not take full advantage of many of the performance-enhancing features of aggressive architectures.

Chapter 4 focuses primarily on transformations at the loop-nest level, but also discusses the possible interaction between clustering and basic-block scheduling. We have not yet dealt with clustered codes that are limited by basic-block size and not amenable to previously-understood local scheduling techniques such as balanced scheduling [KE93, LE95]. In such situations, all of the independent misses exposed by the transformation will not actually issue to the memory system together, limiting the system's latency-tolerance ability. To improve latency tolerance, the instruction scheduler can reschedule independent misses to insure that they are indeed grouped together in a single out-of-order execution window. *Typed fusion* seems to be an appropriate candidate for these sorts of transformation, as that technique essentially allows a dependence graph to be partitioned and restructured according to relevant characteristics of the nodes [KM93].

Looking more broadly, compiler optimizations that extract knowledge dynamically also seem attractive for modern and future microarchitectures that extract performance through various dynamic mechanisms. Current compiler optimizations depend on knowing the amount of computational and storage resources provided by their targeted architectures. For example, instruction scheduling packs instructions together based on the number of functional units in the processor, while cache tiling chooses loop bounds for tiled code based on cache sizes. However, several problems limit static knowledge of resource availability. First, architecture-neutral formats such as Java bytecodes or heterogeneous environments such as grid computing cannot statically provide any reliable resource information about the eventual target. Second, microarchitectures that share resources among separate threads may dynamically vary the resources available to any thread. Examples include the functional-unit and cache sharing of multithreaded architectures and the second-level cache sharing of single-chip multiprocessors. Additionally, unpredictable cache conflicts or coherence activity can reduce the effective cache size available to a program.

Each of the above problems can be addressed by conservatively assuming a low amount of resources, but such assumptions lead to underutilization of system features when more resources are available. A potentially more attractive solution would be the use of *code parametrization*, allowing the parameters to change dynamically to make better use of the available resources. For example, hardware performance counters can provide fine-grained feedback on resource availability, guiding the code to change its resource utilization accordingly. Such use of parametrized static code allows resource sensitivity without the overhead of dynamic recompilation, but requires identifying code structures that benefit from resource sensitivity and determining how much flexibility can be provided without excessive overhead.

The new optimizations presented in this work have addressed some specific ways in which code can more effectively exploit the performance features of modern microarchitectures. As microarchitectures continue to progress, targeted code transformations should continue to provide opportunities for real codes to exploit the ever increasing and ever more complicated resources provided by the underlying system for high performance.

Bibliography

[AC72] Frances E. Allen and John Cocke. A Catalogue of Optimizing Transformations. In Randall Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[AH90] Sarita V. Adve and Mark D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AHAA97] Hazim Abdel-Shafi, Jonathan Hall, Sarita V. Adve, and Vikram S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 204–215, February 1997.

[AK95] David H. Albonesi and Israel Koren. An Analytical Model of High-Performance Superscalar-Based Multiprocessors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '95*, pages 194–203, June 1995.

[AKL81] Walid Abu-Sufah, David J. Kuck, and Duncan H. Lawrie. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[BAD89] Eugene D. Brooks III, Timothy S. Axelrod, and Gregory A. Darmohray. The Cerberus Multiprocessor Simulator. In Garry Rodrigue, editor, *Parallel Processing for Scientific Computing: Proceedings of the 3rd SIAM Conference on Parallel Processing for Scientific Computing* (December 1987), chapter 58, pages 384–390. SIAM, 1989.

[BBC$^+$00] Maynard Brandt, Jeff Brooks, Margaret Cahir, Tom Hewitt, Enrique Lopez-Pineda, and Dick Sandness. *The Benchmarker's Guide for CRAY SV1 Systems*. Cray Inc., July 2000.

[BCJ$^+$94] David F. Bacon, Jyh-Herng Chow, Dz-ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In *Proceedings of CASCON '94*, pages 270–282, October 1994.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[BP91] Michael Butler and Yale Patt. The Effect of Real Data Cache Behavior on the Performance of a Microarchitecture that Supports Dynamic Scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 34–41, November 1991.

[Car96] Steve Carr. Combining Optimization for Cache and Instruction-Level Parallelism. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '96*, pages 238–247, October 1996.

[CB94] T.-F. Chen and J.-L. Baer. A Performance Study of Hardware and Software Data Prefetching Schemes. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 223–232, April 1994.

[CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating Interlock and Improving Balance for Pipelined Machines. *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[CDG$^+$93] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[CDJ$^+$91] R. G. Covington, S. Dwarkadas, J. R. Jump, S. Madala, and J. B. Sinclair. The Efficient Simulation of Parallel Computer Systems. *International Journal of Computer Simulation*, 1:31–58, January 1991.

[CDS96] Steve Carr, Chen Ding, and Philip Sweany. Improving Software Pipelining with Unroll-and-Jam. In *Proceedings of 29th Hawaii International Conference on System Sciences*, January 1996.

[CK94] Steve Carr and Ken Kennedy. Improving the Ratio of Memory Operations to Floating-Point Operations in Loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, November 1994.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software Prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[DGH91] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the International Conference on Parallel Processing*, pages II–99–II107, August 1991.

[DK99] Chen Ding and Ken Kennedy. Improving Cache Performance in Dynamic Applications through Data and Computation Reorganization at Run Time. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, May 1999.

[DPA98] Murthy Durbhakula, Vijay S. Pai, and Sarita Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. Technical Report 9802, Department of Electrical and Computer Engineering, Rice University, 1998.

[DPA99] Murthy Durbhakula, Vijay S. Pai, and Sarita Adve. Improving the Speed vs. Accuracy Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, pages 23–32, January 1999.

[ERB$^+$95] John H. Edmondson, Paul I. Rubinfeld, Peter J. Bannon, Bradley J. Benschneider, Debra Bernstein, Ruben W. Castelino, Elizabeth M. Cooper, Daniel E. Dever, Dale R. Donchin, Timothy C. Fischer, Anil K. Jain, Shekhar Mehta, Jeanne E. Meyer, Ronald P. Preston, Vidya Rajagopalan, Chandrasekhara Somanathan, Scott A. Taylor, and Gilbert M. Wolrich. Internal Organization of the Alpha 21164, a 300-MHz 64-bit Quad-issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1):119–132, 1995.

[FJ94] K.I. Farkas and Norman P. Jouppi. Complexity/Performance Tradeoffs with Non-Blocking Loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[FW97] Babak Falsafi and David A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[GC98] Brian Grayson and Craig Chase. Characterizing Instruction Latency for Speculative Issue SMPs: A Case Study of Varying Memory System Performance on the SPLASH-2 Benchmarks. Workshop on Workload Characterization, held with MICRO-31, November 1998.

[GGH91] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, pages I355–I364, August 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[GHG$^+$91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[GLL$^+$90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Gor95] Edward H. Gornish. *Adaptive and Integrated Data Cache Prefetching for Shared-Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, 1995.

[Hew97] Hewlett-Packard Company. *Exemplar Architecture (S and X-Class Servers)*, January 1997.

[HJ87] David T. Harper III and J. Robert Jump. Vector access performance in parallel memories using a skewed storage scheme. *IEEE Transactions on Computers*, C-36(12):1440–1449, December 1987.

[HKO$^+$94] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, October 1994.

[HSH96] Chris Holt, Jaswinder Pal Singh, and John Hennessy. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 134–145, May 1996.

[Hun95] Doug Hunt. Advanced Features of the 64-bit PA-8000. In *Proceedings of IEEE Compcon*, pages 123–128, March 1995.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings* *of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[KDS00] Magnus Karlsson, Fredrik Dahlgren, and Per Stenström. A Prefetching Technique for Irregular Accesses to Linked Data Structures. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, pages 206–217, January 2000.

[KE93] Daniel R. Kerns and Susan J. Eggers. Balanced Scheduling: Instruction Scheduling When Memory Latency is Uncertain. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 278–289, June 1993.

[Kel96] Jim Keller. The 21264: A Superscalar Alpha Processor with Out-of-Order Execution. 9th Annual Microprocessor Forum, October 1996.

[KM93] Ken Kennedy and Kathryn S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 301–321. Springer-Verlag Lecture Notes in Computer Science, Portland, August 1993.

[Kro81] David Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[KT98] Venkata Krishnan and Josep Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '98*, October 1998.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[LE95] Jack L. Lo and Susan J. Eggers. Improving Balanced Scheduling with Compiler Optimizations that Increase Instruction-Level Parallelism. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 151–162, July 1995.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin 2000: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[LM96] Chi-Keung Luk and Todd C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[LM99] Chi-Keung Luk and Todd C. Mowry. Memory Forwarding: Enabling Aggressive Layout Optimizations by Guaranteeing the Safety of Data Relocation. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[McI98] Nathaniel McIntosh. *Compiler Support for Software Prefetching*. PhD thesis, Rice University, May 1998.

[MG91] Todd Mowry and Anoop Gupta. Tolerating Latency Through Software-Controlled Prefetching. *Journal on Parallel and Distributed Computing*, pages 87–106, June 1991.

[MIP96] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual, Version 2.0*, December 1996.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[Mow94] Todd Mowry. *Tolerating Latency through Software-controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

[MRF$^+$97] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design*, June 1997. Held in conjunction with ISCA.

[MS96] Larry McVoy and Carl Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 USENIX Technical Conference*, pages 279–295, January 1996.

[MWK99] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving Memory Hierarchy Performance for Irregular Applications. In *Proceedings of the 13th ACM-SIGARCH International Conference on Supercomputing*, pages 425–433, June 1999.

[NHO96] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the*

23rd Annual International Symposium on Computer Architecture, pages 67–77, May 1996.

[Nic87] Alexandru Nicolau. Loop Quantization or Unwinding Done Right. In *Proceedings of the 1st International Conference on Supercomputing*, pages 294–308, June 1987.

[OD93] K. Oner and M. Dubois. Effects of Memory Latencies on Non-Blocking Processor/Cache Architectures. In *Proceedings of the International Conference on Supercomputing*, 1993.

[ONH$^+$96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[PA99] Vijay S. Pai and Sarita Adve. Code Transformations to Improve Memory Parallelism. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 147–155, November 1999.

[PA00] Vijay S. Pai and Sarita Adve. Code Transformations to Improve Memory Parallelism. *Journal of Instruction Level Parallelism*, 2, May 2000.

[Por89] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, April 1989.

[PRA96] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. Technical Report 9606, Electrical and Computer Engineering Department, Rice University, July 1996.

[PRA97a] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. *RSIM Reference Manual, Version 1.0*. Electrical and Computer Engineering Department, Rice University, August 1997. Technical Report 9705.

[PRA97b] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, pages 72–83, February 1997.

[PRAA99] Vijay S. Pai, Parthasarathy Ranganathan, Hazim Abdel-Shafi, and Sarita Adve. The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 48(2):218–226, February 1999.

[PRAH96] Vijay S. Pai, Parthasarathy Ranganathan, Sarita V. Adve, and Tracy Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[RBDH97] Mendel Rosenblum, Edouard Bugnion, Scott Devine, and Stephen Herrod. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 1997. Special issue on Computer Simulation.

[RCRH95] Anne Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[RPAA97] Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, and Sarita V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 144–156, June 1997.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[RS99] Amir Roth and Gurindar S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[RT98] Gabriel Rivera and Chau-Wen Tseng. Eliminating Conflict Misses for High-Performance Architectures. In *Proceedings of the International Conference on Supercomputing*, pages 353–360, July 1998.

[SGH97] Vatsa Santhanam, Edward H. Gornish, and Wei-Chung Hsu. Data Prefetching on the HP PA-8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[SL95] André Seznec and Fabien Lloansi. About effective cache miss penalty on out-of-order superscalar processors. Technical Report PI-970, IRISA, November 1995.

[SL98] Eric Schnarr and Jim Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, October 1998.

[SMP$^+$96] Rafael H. Saavedra, Weihua Mao, Daeyeon Park, Jacqueline Chame, and Sungdo Moon. The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 39–45, April 1996.

[Soh93] G. S. Sohi. High-Bandwidth Interleaved Memories for Vector Processors – A Simulation Study. *IEEE Transactions on Computers*, 42(1):34–44, January 1993.

[SP88] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, C-37(5):562–573, May 1988.

[SPA$^+$98] Daniel J. Sorin, Vijay S. Pai, Sarita V. Adve, Mary K. Vernon, and David A. Wood. Analytic Evaluation of Shared-Memory Systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 380–391, June 1998.

[Sun97] Sun Microelectronics. *UltraSPARC-II: Second Generation SPARC v9 64-Bit Microprocessor With VIS*, July 1997.

[SWG92] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[TE95] D.M. Tullsen and S.J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems*, 13(1):57–88, February 1995.

[TEE$^+$96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[Tha82] Khalid O. Thabit. *Cache Management by the Compiler*. PhD thesis, Rice University, 1982.

[WL91] Michael E. Wolf and Monica S. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[WR96] Emmett Witchel and Mendel Rosenblum. Embra: Fast and Flexible Machine Simulation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 68–79, May 1996.

[ZB92] Richard N. Zucker and Jean-Loup Baer. A Performance Study of Memory Consistency Models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.