RICE UNIVERSITY

The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology

by

Vijay S. Pai

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Sarita V. Adve, Chair
Assistant Professor in Electrical and
Computer Engineering

J. Robert Jump
Professor of Electrical and Computer
Engineering

Alan L. Cox
Associate Professor of Computer Science

Houston, Texas

April, 1997

The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodology

Vijay S. Pai

Abstract

Current microprocessors exploit high levels of instruction-level parallelism (ILP). This thesis presents the first detailed analysis of the impact of such processors on shared-memory multiprocessors.

We find that ILP techniques substantially reduce CPU time in multiprocessors, but are less effective in reducing memory stall time for our applications. Consequently, despite the latency-tolerating techniques incorporated in ILP processors, memory stall time becomes a larger component of execution time and parallel efficiencies are generally poorer in our ILP-based multiprocessor than in an otherwise equivalent previous-generation multiprocessor. We identify clustering independent read misses together in the processor instruction window as a key optimization to exploit the ILP features of current processors.

We also use the above analysis to examine the validity of direct-execution simulators with previous-generation processor models to approximate ILP-based multiprocessors. We find that, with appropriate approximations, such simulators can reasonably characterize the behavior of applications with poor overlap of read misses. However, they can be highly inaccurate for applications with high overlap of read misses.

Acknowledgments

I would like to thank my advisor, Sarita Adve, for research direction and challenges over the past three years. I also thank my officemate and perennial co-author, Parthasarathy Ranganathan, for a terrific exchange of ideas ever since we started working together. It has been a great pleasure to work with these two people for the past three years, and I look forward to more. I also thank my thesis committee, Alan Cox and Bob Jump, for valuable feedback.

I cannot thank my family enough for all kinds of guidance over the years. I particularly thank my parents, Sadananda and Sharda Pai, for all the challenges over the years, including convincing me to go to graduate school. I also owe a great deal to my grandmother, Janabai Kamath, for support and for lots of nutritious meals along the way.

My graduate career has been supported by a Fannie and John Hertz Foundation Fellowship.

Contents

Illustrations

Chapter 1

Introduction

1.1 Motivation

Shared-memory multiprocessors built from commodity microprocessors (e.g., Convex Exemplar, Sequent STiNG, SGI Origin series) are expected to provide high performance for a variety of scientific and commercial applications. Current commodity microprocessors improve performance with aggressive techniques to exploit high levels of instruction-level parallelism (*ILP*). For example, the HP PA-8000, Intel Pentium Pro, and MIPS R10000 processors use multiple instruction issue, dynamic (out-of-order) scheduling, multiple non-blocking reads, and speculative execution. However, most recent architecture studies of shared-memory systems assume a processor model with single issue, static (in-order) scheduling, and blocking reads. 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.

Although researchers have shown the benefits of aggressive ILP techniques for uniprocessors, there has not yet been a detailed or realistic analysis of the impact of such ILP techniques on the performance of shared-memory multiprocessors. Such an analysis is required to fully exploit advances in uniprocessor technology for multiprocessors. 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.

1.2 Contributions

This thesis makes two contributions.

(1) This is the first detailed study of the effectiveness of state-of-the-art ILP processors in a shared-memory multiprocessor, using a detailed simulator driven by real applications.

(2) This is the first study on the validity of using current direct-execution simulation techniques to model shared-memory multiprocessors built from ILP processors.

Our experiments for assessing the impact of ILP on shared-memory multiprocessor performance show that all 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. This deficiency in the impact of ILP techniques on memory stall time arises primarily because of insufficient potential in our applications to overlap multiple read misses, as well as system contention from more frequent memory accesses.

The discrepancy in the impact of ILP techniques on the CPU and read stall components leads to two key effects for our applications. First, read stall time becomes a larger component of simulated execution time than in previous-generation multiprocessors. Second, parallel efficiencies for ILP multiprocessors are lower than with previous-generation multiprocessors for all but one application. Thus, 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 as a key optimization for exploiting the ILP features of current processors.

Our results on the validity of using simulators based on simple processor models to approximate the behavior of ILP processors in multiprocessor systems are as follows. 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 26% to 192% error, while the most accurate technique gave -8% to 73% error. 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.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 describes our experimental methodology. Chapters 3 and 4 describe and analyze our results. Chapter 5 discusses related work. Chapter 6 provides concluding remarks and discusses future work motivated by this thesis.

Chapter 2

Experimental Methodology

The following sections describe the metrics used in our evaluation, the architectures simulated, the simulation environment, and the applications.

2.1 Measuring the Impact of ILP

To determine the impact of ILP techniques in multiprocessors, we compare two multiprocessor systems – `ILP` and `Simple` – equivalent in every respect except the processor used. The `ILP` system uses state-of-the-art high-performance microprocessors with multiple issue, dynamic scheduling, and non-blocking reads. We refer to such processors as `ILP` processors. The `Simple` system uses previous-generation microprocessors with single issue, static scheduling, and blocking reads, matching the processor model used in many current direct-execution simulators. We refer to such processors as `Simple` processors. We compare the `ILP` and `Simple` systems to determine how multiprocessors benefit from ILP techniques, rather than to propose any architectural tradeoff between the `ILP` and `Simple` architectures. Therefore, both systems have the same clock rate and feature an identical state-of-the-art aggressive memory system and interconnect. Section 2.2 provides more detail on these systems.

The key metric we use to evaluate the impact of ILP is the speedup in execution time achieved by the `ILP` system over the `Simple` system, which we call the *ILP speedup*. ILP speedup is defined in Equation 2.1,

$$ILP\ speedup = \frac{t_{\texttt{Simple}}}{t_{\texttt{ILP}}} \tag{2.1}$$

where $t_{\texttt{Simple}}$ represents the execution time of the application on the `Simple` system and $t_{\texttt{ILP}}$ represents the execution time on the `ILP` system.

To study the factors affecting ILP speedup, we study the components of execution time – busy, functional unit stall, synchronization stall, and data memory stall. However, these components are difficult to distinguish with `ILP` processors, as each instruction can potentially overlap its execution with both previous and following instructions. We hence adopt the following convention, also used in other studies [PRAH96, RBH+95]. If, in a given cycle, the processor retires the maximum allowable number of instructions, we count that cycle as part of busy time. Otherwise, we charge that cycle to the stall time component corresponding to the first instruction that could not be retired. All instructions retire from the instruction window in program order in order to guarantee precise interrupts. Thus, the stall time for a class of instructions represents the number of cycles that instructions of that class spend at the head of the instruction window (also known as the reorder buffer or active list) before retiring.

We analyze the effect of each component of execution time by examining the ILP speedup of that component, which is the ratio of the times spent on the component with the `Simple` and `ILP` systems, as well as the *weight* of that component, which is the fraction of total execution time spent in that component. For example, the data memory ILP speedup is defined as

$$Data\ memory\ ILP\ speedup = \frac{t_{mem,\texttt{Simple}}}{t_{mem,\texttt{ILP}}} \tag{2.2}$$

where $t_{mem,\texttt{Simple}}$ and $t_{mem,\texttt{ILP}}$ represent the data memory component of execution time on the `Simple` and `ILP` systems, respectively. Similarly, the weight of the data memory component on the `Simple` system is defined as

$$Data\ memory\ weight = \frac{t_{mem,\texttt{Simple}}}{t_{\texttt{Simple}}} \tag{2.3}$$

with $t_{mem,\texttt{Simple}}$ representing the time spent on data memory stalls and $t_{\texttt{Simple}}$ representing total execution time on the `Simple` system.

2.2 Simulated Architectures

This study models cache-coherent non-uniform memory access (CC-NUMA) shared-memory multiprocessor systems with the system nodes connected by a two-dimensional mesh. The multiprocessors in this study use a 3-state invalidation-based directory coherence protocol and are release-consistent [GLL+90]. The cache coherence protocol used is illustrated in Figure 2.1.

Figure 2.2 shows the primary blocks in the multiprocessor configuration modeled in this study. Figure 2.3 summarizes our default multiprocessor system parameters. Most of the analysis in this paper focuses on 8-processor systems; however, this study also considers uniprocessor, 16-processor, and 32-processor systems. Although more richly connected network topologies than the mesh, such as the crossbar, can be used in some of these medium-scale multiprocessor configurations, we evaluate systems with the mesh network in order to capture the behavior of CC-NUMA multiprocessors, and with an aim to present insights that can be extended to larger systems as well.

The following details the processors and memory hierarchy incorporated in the simulated systems.

Processor Models. Our `ILP` processor resembles the MIPS R10000 processor [MIP96], with 4-way issue, dynamic scheduling, non-blocking reads, register renaming, and speculative execution. Unlike the MIPS R10000, however, our processor implements release consistency, as our previous work has shown this consistency model to achieve higher performance on ILP multiprocessor systems [PRAH96]. The `ILP` processor has a memory unit which holds up to 32 entries. Reads remain in the memory unit until they have completed; writes leave the memory unit as soon as they issue to the cache, as they do not impose any constraint on subsequent data accesses in the release consistency memory model. Further, the processor is assumed

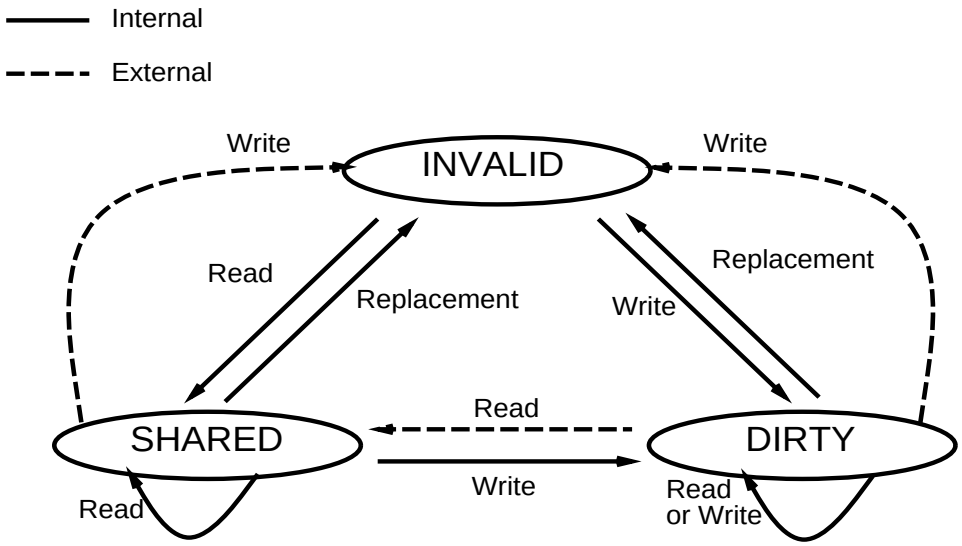


Figure 2.1 Cache coherence protocol diagram

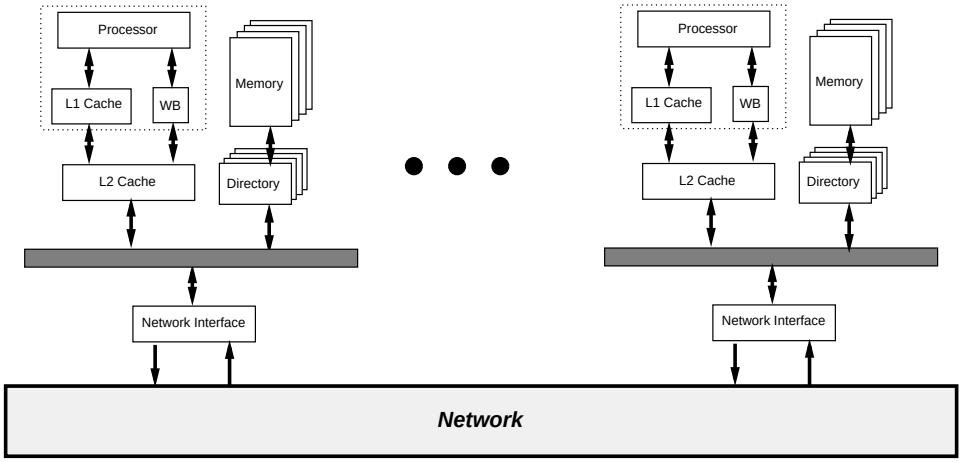


Figure 2.2 Multiprocessor system modeled

ILP Processor	
Processor speed	300MHz
Maximum fetch/retire rate (instructions per cycle)	4
Instruction issue window	64 entries
Functional units	2 integer arithmetic 2 floating point 2 address generation
Branch speculation depth	8
Memory unit size (maximum number of outstanding loads and unissued stores)	32 entries
Network parameters	
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles
Cache parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped, 16 K
L1 request ports	2
L1 hit time	1 cycle
Number of L1 MSHRs	8
L2 cache (off-chip)	4-way associative, 64 K
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of L2 MSHRs	8
Write buffer entries	8 cache lines
Memory parameters	
Memory access time	18 cycles (60 ns)
Memory transfer bandwidth	16 bytes/cycle
Memory Interleaving	4-way

Figure 2.3 System parameters

to have sufficient physical registers available so as never to stall for a shortage of renaming registers. The `Simple` processor uses single-issue, static scheduling, and blocking reads, and has the same clock speed as the `ILP` processor.

Most recent direct-execution simulation studies assume single-cycle latencies for all processor functional units. We choose to continue with this approximation for our `Simple` model to represent currently used simulation models. To minimize sources of difference between the `Simple` and `ILP` models, we also use single-cycle functional unit latencies for `ILP` processors. Nevertheless, to investigate the impact of this approximation, we simulated all our applications on an 8-processor `ILP` system with functional unit latencies similar to the UltraSPARC processor. We found that the approximation has negligible effect on all but one of our applications (Water); even in that application, our overall results continue to hold. This approximation has little impact because, in multiprocessors, memory time dominates, and `ILP` processors can easily overlap functional unit latency.

For the experiments related to the validity of simulators based on a simple processor model, we also investigate variants of the `Simple` model that reflect approximations for ILP-based multiprocessors made in recent literature. These are further described in Chapter 4.

Memory Hierarchy. The `ILP` and `Simple` systems have an identical memory hierarchy with identical parameters. Each system node includes a processor with two levels of caching, a merging write buffer [ERB+95] between the caches, and a portion of the distributed memory and directory. A split-transaction system bus connects the memory, the network interface, and the rest of the system node.

The L1 cache has 2 request ports, allowing it to serve up to 2 data requests per cycle, and is write-through with a no-write-allocate policy. The L2 cache has 1 request port and is a fully-pipelined write-back cache with a write-allocate policy. Each cache also has an additional port for incoming coherence messages and replies. Both the L1 and L2 caches have 8 miss status holding registers (MSHRs) [Kro81],

which reserve space for outstanding cache misses (the L1 cache allocates MSHRs only for read misses as it is no-write-allocate). The MSHRs support coalescing so that multiple misses to the same line do not initiate multiple requests to lower levels of the memory hierarchy. We do not include such coalesced requests when calculating miss counts for our analysis.

We choose cache sizes commensurate with the input sizes of our applications, based on the methodology of Woo et al. [WOT+95]. Primary working sets of all our applications fit in the L1 cache, and secondary working sets of most applications do not fit in the L2 cache.

2.3 Simulation Environment

We use the Rice Simulator for ILP Multiprocessors (RSIM) to simulate the `ILP` and `Simple` architectures described in Section 2.2 [PRA97a]. RSIM models the processors, memory system, and network in detail, including contention at all resources. It is driven by application executables rather than traces, allowing interactions between the processors to affect the course of the simulation. The code for the processor and cache subsystem performs cycle-by-cycle simulation and interfaces with an event-driven simulator for the network and memory system. The latter is derived from the Rice Parallel Processing Testbed (RPPT) [CDJ+91, Raj95].

Since we simulate the processor in detail, our simulation times are five to ten times higher than those for an otherwise equivalent direct-execution simulator. To speed up simulation, we assume that all instructions hit in the instruction cache (with a 1 cycle hit time) and that all accesses to private data hit in the L1 data cache. These assumptions have also been made by many previous multiprocessor studies using direct-execution, and are not likely to significantly affect our results, since our applications do not have much private data. We do, however, model contention for processor resources and L1 cache ports due to private data accesses.

The applications are compiled with a version of the SPARC V9 gcc compiler modified to eliminate branch delay slots and restricted to 32 bit code, with the options `-O2 -funrollloop`.

2.4 Applications

We use six applications for this study – LU, FFT, and Radix from the SPLASH-2 suite [WOT+95], Mp3d and Water from the SPLASH suite [SWG92], and Erlebacher from the Rice parallel compiler group [AWMC+95]. We modified LU slightly to use flags instead of barriers for better load balance. Figure 2.4 gives input sizes for the applications and their execution times on a `Simple` uniprocessor.

We also study versions of LU and FFT that include ILP-specific optimizations that can be implemented in a compiler. Specifically, we use function inlining and loop interchange to schedule read misses closer to each other so that they can be overlapped in the `ILP` processor. We refer to these optimized applications as LU_opt and FFT_opt.

Application	Input Size	Cycles
LU	256 by 256 matrix, block 8	1.03×10^8
FFT	65536 points	3.67×10^7
Radix	1K radix, 512K keys, max: 512K	3.15×10^7
Mp3d	50000 particles	8.82×10^6
Water	512 molecules	2.68×10^8
Erlebacher	64 by 64 by 64 cube, block 8	7.62×10^7

Figure 2.4 Application characteristics

Chapter 3

Impact of ILP on Multiprocessor Performance

This chapter evaluates the impact of instruction-level parallelism on multiprocessor performance. Section 3.1 focuses on eight-processor systems, comparing the performance of `Simple` and `ILP` systems as described in Section 2.2. Section 3.2 shows the impact of ILP on parallel efficiency, which indicates system scalability. Section 3.3 identifies several limitations in the performance improvements given by ILP and evaluates the extent to which these limitations are artifacts of current technological constraints. Finally, Section 3.4 summarizes the key findings of these studies and describes additional issues in the impact of ILP on shared-memory multiprocessor performance.

3.1 Impact of ILP on an Eight-processor System

This section describes the impact of ILP on multiprocessors by comparing the 8-processor `Simple` and `ILP` systems described in Section 2.2.

3.1.1 Overall Results

Figures 3.1 and 3.2 illustrate our key overall results. For each application, Figure 3.1 shows the total ILP speedup as well as the ILP speedup of the different components of execution time. The execution time components include CPU time, data memory stalls, and synchronization stalls. We combine both busy time and functional unit (FU) stalls together into CPU time when computing ILP speedups, because the Simple processor does not see any FU stalls. Figure 3.2 indicates the relative importance of the ILP speedups of the different components by showing the time spent on

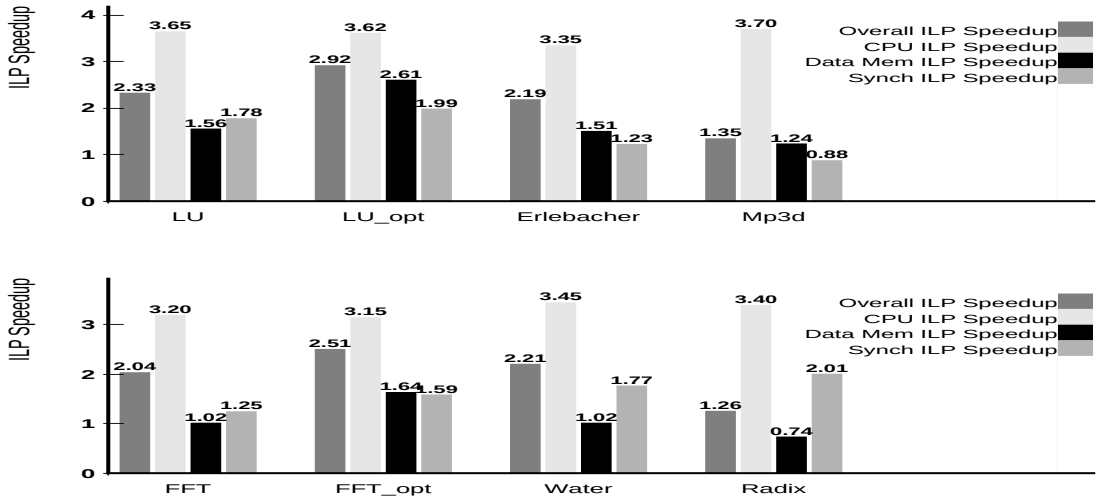


Figure 3.1 ILP speedup and components in an 8-processor system

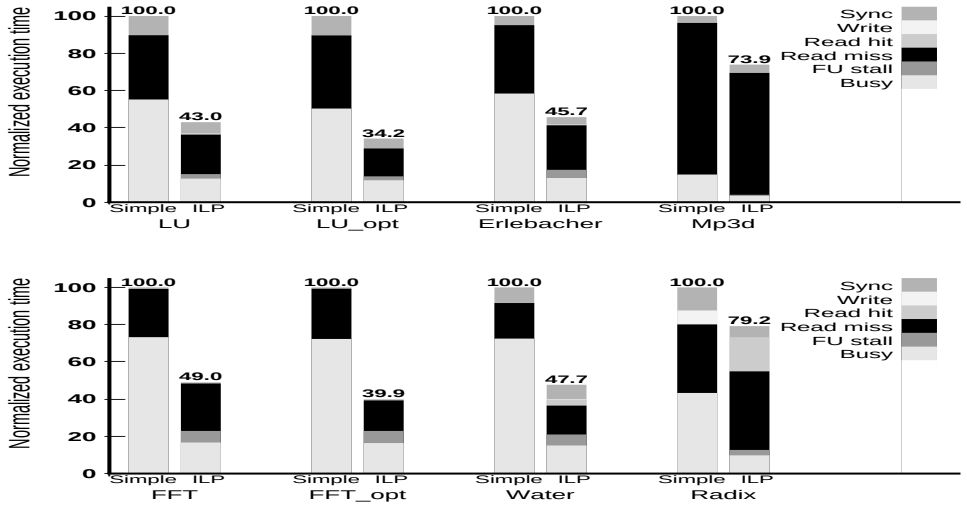


Figure 3.2 Execution time components in an 8-processor system

each component (normalized to the total time on the `Simple` system). The busy and stall times are calculated as explained in Section 2.1.

All of our applications exhibit speedup with `ILP` processors, but the specific speedup seen varies greatly, from 1.26 in Radix to 2.92 in LU_opt. All the applications achieve similar and significant CPU ILP speedup (3.15 to 3.70). In contrast, the data memory ILP speedup is lower and varies greatly across the applications, from 0.74 (a *slowdown!*) in Radix to 2.61 in LU_opt.

The key effect of the high CPU ILP speedups and low data memory ILP speedups is that data memory time becomes more dominant in `ILP` multiprocessors than in `Simple` multiprocessors. Further, since CPU ILP speedups are fairly consistent across all applications, and data memory time is the only other dominant component of execution time, the data memory ILP speedup primarily shapes the overall ILP speedups of our applications. We therefore analyze the factors that influence data memory ILP speedup in greater detail in Section 3.1.2.

Synchronization ILP speedup is also low and varies widely across applications. However, since synchronization does not account for a large portion of the execution time, it does not greatly influence the overall ILP speedup. Section 3.1.3 discusses the factors affecting synchronization ILP speedup in our applications.

3.1.2 Data Memory ILP Speedup

We first discuss various factors that can contribute to data memory ILP speedup, and then show how these factors interact in our applications.

Contributing Factors

Figure 3.2 shows that memory time is dominated by read miss time in all of our applications. We therefore focus on factors influencing read miss ILP speedup.

The read miss ILP speedup is the ratio of the total stall time due to read misses in the `Simple` and `ILP` systems. The total stall time due to read misses in a given

system is simply the product of the average number of L1 misses and the average exposed, or unoverlapped, L1 cache miss latency, as expressed in Equation 3.1.

$$Read\ Miss\ ILP\ Speedup = \frac{M_{\texttt{Simple}} \times l_{\texttt{Simple}}}{M_{\texttt{ILP}} \times l_{\texttt{ILP},unoverlapped}} \tag{3.1}$$

In Equation 3.1, $M_{\texttt{Simple}}$ and $M_{\texttt{ILP}}$ represent the number of L1 misses seen in the `Simple` and `ILP` systems respectively. $l_{\texttt{Simple}}$ represents the average L1 miss latency in the `Simple` system, while $l_{\texttt{ILP},unoverlapped}$ is the average unoverlapped L1 cache miss latency in the `ILP` system.

Equation 3.1 can be rewritten as follows:

$$Read\ Miss\ ILP\ Speedup = \frac{M_{\texttt{Simple}}}{M_{\texttt{ILP}}} \times \frac{l_{\texttt{Simple}}}{l_{\texttt{ILP},unoverlapped}} \tag{3.2}$$

Thus, Equation 3.2 isolates two contributing factors to overall read miss ILP speedup – the ratio of misses in the `Simple` and `ILP` systems, and the ratio of the average miss latency in the `Simple` system to the average unoverlapped miss latency in the `ILP` system.

The first factor is called the *miss factor*, and is defined as

$$Miss\ Factor = \frac{M_{\texttt{Simple}}}{M_{\texttt{ILP}}} \tag{3.3}$$

The miss counts seen in the `Simple` and `ILP` systems can differ since reordering and speculation in the `ILP` processor can alter the cache miss behavior. A miss factor greater than 1 thus contributes positively to read miss ILP speedup, as the `ILP` system sees fewer misses than the `Simple` system.

The second factor in Equation 3.2 is the reciprocal of the *unoverlapped factor*. Unoverlapped factor is defined as the ratio of the exposed, or unoverlapped, miss latency in the `ILP` and `Simple` systems, and is expressed as:

$$Unoverlapped\ Factor = \frac{l_{\texttt{ILP},unoverlapped}}{l_{\texttt{Simple}}} \tag{3.4}$$

A lower unoverlapped factor leads to a higher read miss ILP speedup.

In the `Simple` system, the entire L1 miss latency is unoverlapped. To understand the factors contributing to unoverlapped latency in the `ILP` system, Equation 3.5 expresses the average unoverlapped `ILP` miss latency as the difference between the average total `ILP` miss latency (denoted $l_{\texttt{ILP}}$) and the average overlapped miss latency (denoted $l_{\texttt{ILP},overlapped}$).

$$Unoverlapped\ Factor = \frac{l_{\texttt{ILP}} - l_{\texttt{ILP},overlapped}}{l_{\texttt{Simple}}} \tag{3.5}$$

The total `ILP` miss latency can be expanded, as in Equation 3.6, as the sum of the miss latency incurred by the `Simple` system and an extra latency component added by the `ILP` system (for example, due to increased contention). In Equation 3.6, $l_{\texttt{ILP},extra}$ represents the average extra latency component seen in the `ILP` system.

$$Unoverlapped\ Factor = \frac{l_{\texttt{Simple}} + l_{\texttt{ILP},extra} - l_{\texttt{ILP},overlapped}}{l_{\texttt{Simple}}} \tag{3.6}$$

Equation 3.7 then performs an algebraic simplification to express the overlapped and extra latencies seen by the `ILP` system relative to the miss latency in the `Simple` system.

$$Unoverlapped\ Factor = 1 - \left(\frac{l_{\texttt{ILP},overlapped}}{l_{\texttt{Simple}}} - \frac{l_{\texttt{ILP},extra}}{l_{\texttt{Simple}}} \right) \tag{3.7}$$

Equation 3.7 can be used to isolate two factors that shape the unoverlapped factor – the *overlapped factor* and the *extra factor* – which are, respectively, the `ILP` overlapped and extra latencies expressed as a fraction of the `Simple` miss latency. These factors are defined below:

$$Overlapped\ Factor = \frac{l_{\texttt{ILP},overlapped}}{l_{\texttt{Simple}}} \tag{3.8}$$

$$Extra\ Factor = \frac{l_{\texttt{ILP},extra}}{l_{\texttt{Simple}}} \tag{3.9}$$

Read miss ILP speedup is higher with a higher overlapped factor and a lower extra factor.

The *overlapped factor* increases with increased overlap of misses with other useful work. The number of instructions behind which a read miss can overlap is limited

by the instruction window size. Further, read misses have longer latencies than other operations that occupy the instruction window. Therefore, read miss latency can normally be completely hidden only behind other read misses. Thus, for a high overlapped factor (and high read miss ILP speedup), applications should exhibit read misses that appear clustered together within the instruction window.

On the other hand, the *extra factor* must be low for a high read miss ILP speedup. Extra miss latencies can arise from contention for system resources, as the ILP techniques allow `ILP` processors to issue memory references more frequently than `Simple` processors. Extra miss latency can also arise from a change in miss behavior if the miss pattern in `ILP` processors forces more misses to be resolved at remote levels of the memory hierarchy.

In summary, the unoverlapped factor contributes positively to read miss ILP speedup if the `ILP` unoverlapped miss latency is less than the `Simple` miss latency. This factor depends on how much potential for read miss overlap is exploited (overlap factor) and on how much is lost due to contention (extra factor). A positive contribution results if the latency overlapped by `ILP` exceeds any extra latency added by `ILP`. On the other hand, the read miss component of execution time can incur a slowdown relative to the `Simple` processor if the extra factor of `ILP` exceeds the overlapped factor.

Analysis of Applications

Read miss ILP speedup (not shown separately) is low (less than 1.6) in all our applications except LU, LU_opt, and FFT_opt; Radix actually exhibits a slowdown. We now show how the factors described above contribute to read miss ILP speedup for our applications.

Miss factor. Most of our applications have miss factors close to 1, implying a negligible contribution from this factor to read miss ILP speedup. LU and LU_opt, however, have high miss factors (2.21 and 1.75 respectively), which contribute signif-

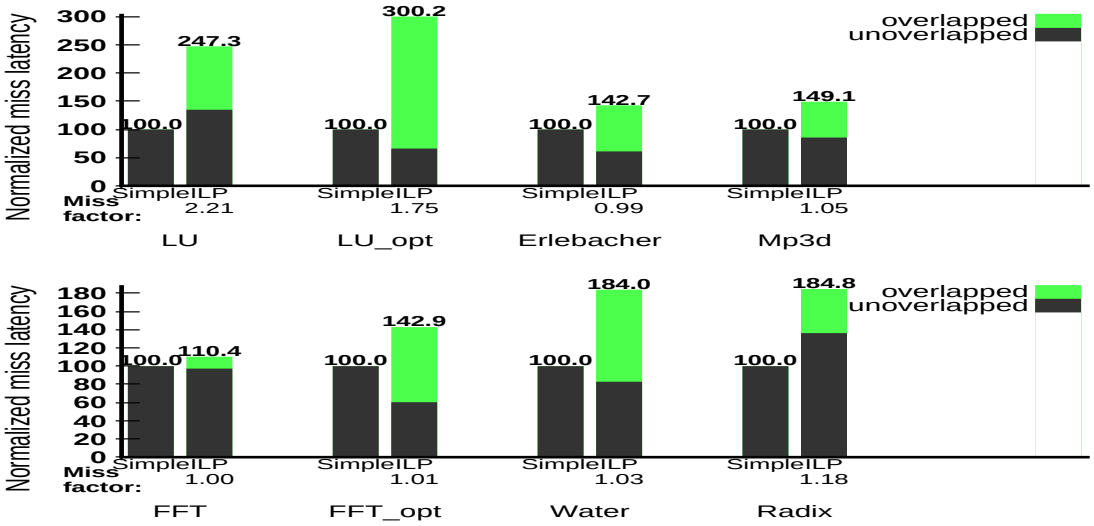


Figure 3.3 Effect of ILP on average miss latency in an 8-processor system

icantly to the read miss ILP speedup. These high miss factors arise because the `ILP` system reorders certain accesses that induce repeated conflict misses in the `Simple` system. In the `ILP` system, the first two conflicting requests overlap, while subsequent requests to the conflicting lines coalesce with earlier pending misses, thus reducing the number of misses seen by the system.

Unoverlapped factor. Figure 3.3 graphically represents the unoverlapped, overlapped, and extra latencies and factors. The two bars for each application show the average L1 read miss latency in `Simple` and `ILP` systems, normalized to the `Simple` system latency. The light part of the `ILP` bar shows the average overlapped latency while the dark part shows the unoverlapped latency. Because of the normalization, the dark and the light parts of the `ILP` bar also represent the unoverlapped and overlapped factors as percentages, respectively. The difference between the full `ILP` and `Simple` bars represents the extra factor. Below each `ILP` bar, we also show the miss factor for reference – the read miss ILP speedup is the miss factor divided by the unoverlapped factor.

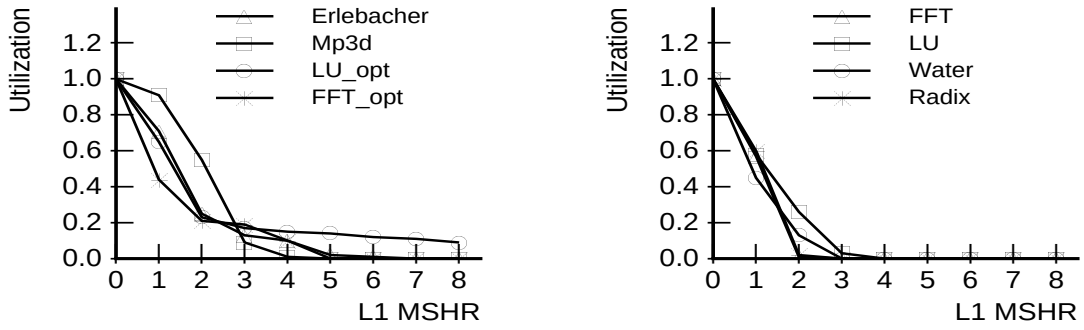
We measure the latency of a read miss from the time the address for the miss is generated to the time the value arrives at the processor; therefore, the extra and

overlapped factors in Figure 3.3 incorporate time spent by a read miss in the processor memory unit and any overlap seen during that time.

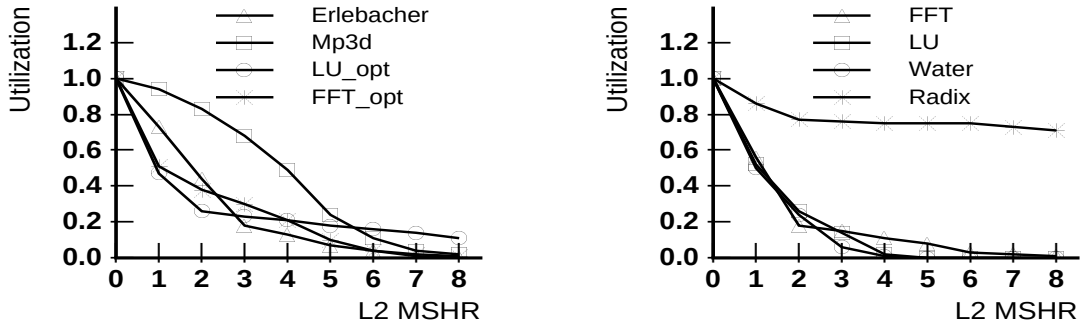
Figure 3.4 provides additional data to indicate overlapped and extra latency after a read miss is issued to the memory system. This figure illustrates MSHR occupancy distributions at the L1 and L2 caches. Each plot gives the fraction of total time (on the vertical axis) for which at least N MSHRs are occupied by misses, where N is the number on the horizontal axis. Recall that only read misses reserve L1 MSHRs, as the L1 cache is no-write-allocate. Thus, the L1 MSHR occupancy graph indicates L1 read miss overlap in the system. Since the L2 MSHR occupancy graph includes both read and write misses, an L2 MSHR occupancy greater than the corresponding L1 MSHR occupancy indicates resource contention seen by reads due to interference from writes. We next use the above data to understand the reasons for the unoverlapped factor seen in each application.

LU_opt, FFT_opt, Erlebacher, and Mp3d have moderate to high overlapped factors due to their moderate to high L1 MSHR occupancies. The optimizations we use in LU_opt and FFT_opt to cluster read misses together in the instruction window are responsible for their higher overlap relative to LU and FFT respectively. The increased frequency of reads due to the high read overlap in these four applications leads to an extra latency due to contention effects, primarily in the main memory system. Write traffic additionally increases this extra factor, though not significantly. However, as shown in Figure 3.3, on all these applications, the positive effects of the overlapped factor outweigh the negative effects of the extra factor, subsequently leading to a low unoverlapped factor and, hence, higher read miss ILP speedups.

Radix, on the other hand, illustrates the opposite extreme. Figure 3.3 shows that in Radix, the negative effects of extra latency due to increased contention significantly outweigh the positive effects due to overlap, leading to a high unoverlapped factor of 1.36. The high extra factor is primarily due to write traffic. Figure 3.4 shows that in Radix, L2 MSHRs are saturated for over 70% of the execution. Further



(a) L1 MSHR occupancy



(b) L2 MSHR occupancy

Figure 3.4 MSHR occupancy in an 8-processor `ILP` system

misses now stall at the L2 cache, preventing other accesses from issuing to that cache; eventually, this backup reaches the primary cache ports and the processor's memory units, causing misses to experience a high extra latency. This backup also causes Radix to see a large read hit component. Further, the low L1 MSHR occupancy, seen in Figure 3.4, shows that Radix has little potential to overlap multiple read misses.

FFT is the only application to see neither overlap effects nor contention effects, as indicated by the low L1 and L2 MSHR occupancies. This leads to an unoverlapped factor close to 1 and consequently a read miss ILP speedup close to 1.

Finally, we discuss two applications – LU and Water – which show relatively high overlapped and extra factors, despite low MSHR occupancies. In LU (and LU_opt, to a lesser extent), the `ILP` processor coalesces accesses that cause L1 cache misses in the

`Simple` case. Our detailed statistics show that these misses are primarily L2 cache hits in the `Simple` case. Thus, the `Simple` miss latency includes these L2 cache hits and remote misses while the `ILP` miss latency includes only the remaining remote misses. This change in miss pattern leads to a higher average miss latency in the `ILP` system than in the `Simple` system, leading to a high extra factor. The extra factor further increases from a greater frequency of memory accesses, which leads to increased network and memory contention in the `ILP` system. LU can overlap only a portion of this extra latency, leading to an unoverlapped factor greater than 1. However, LU still achieves a read miss ILP speedup because of its miss factor.

Water stands apart from the other applications because of its synchronization characteristics. Its extra latency arises because reads must often wait on a pending acquire operation to complete before issuing. The latency contribution caused by this waiting, however, is overlapped by the lock acquire itself. As a result, Water has a large apparent overlap. Nevertheless, Water's poor MSHR occupancy prevents it from getting a low unoverlapped factor, and its read miss ILP speedup is close to 1.

In summary, the key reasons for the low read miss ILP speedup in most of our applications are a lack of opportunity in the applications for overlapping read misses and/or increased contention in the system.

As discussed above, our analysis primarily focuses on the effectiveness of ILP on the data memory component of execution time since this is the component which determines the overall effectiveness of ILP in our systems. Any deficiency in overlap among data read misses is particularly damaging since these operations are the only particularly high-latency instructions in the models we study and in current microprocessors. If future microprocessors include high-latency computational instructions, then those instructions must also be effectively overlapped in order to achieve significant benefits from ILP.

3.1.3 Synchronization ILP Speedup

In general, `ILP` processors can affect synchronization time in the following ways. First, ILP reduces synchronization waiting times through reduced computation time and overlapped data read misses. Second, acquire latency can be overlapped with previous operations of its processor, as allowed by release consistency [GLL+90]. The third factor is a negative effect: increased contention in the memory system due to higher frequency of accesses can increase overall synchronization latency.

The above factors combine to produce a variety of synchronization speedups for our applications, ranging from 0.88 in Mp3d to 2.01 in Radix. However, synchronization accounts for only a small fraction of total execution time in all our applications; therefore, synchronization ILP speedup does not contribute much to overall ILP speedup for our applications and system.

3.2 Impact of ILP on Parallel Efficiency

This section details the impact of ILP on parallel efficiency. The parallel efficiency (PE) of a multiprocessor application running on a system with N processors is defined as:

$$PE_N = \frac{Execution\ time\ on\ uniprocessor}{Execution\ time\ on\ multiprocessor} \times \frac{1}{N} \tag{3.10}$$

Section 3.2.1 analyzes the parallel efficiency of 8-processor ILP configurations, comparing ILP effectiveness on uniprocessors and multiprocessors. Section 3.2.2 builds upon this analysis to show the impact of ILP on the parallel efficiency of larger configurations.

3.2.1 Parallel Efficiency on an Eight-Processor Configuration

Figure 3.5 shows the parallel efficiency achieved by our 8-processor `ILP` and `Simple` systems for all our applications, expressed as a percentage. Except for FFT_opt,

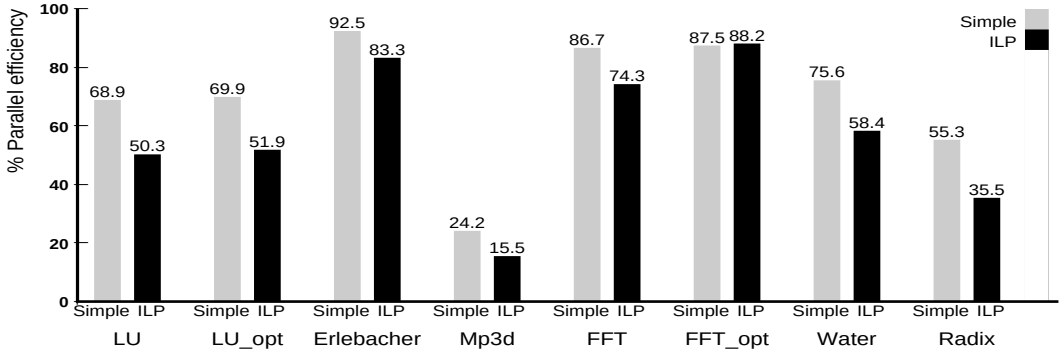


Figure 3.5 Parallel Efficiency with `Simple` and `ILP` systems

parallel efficiency for `ILP` configurations is considerably less than that for `Simple` configurations.

The parallel efficiency of an `ILP` system can be related to the ILP speedup of the `ILP` system by extending Equation 3.10 as follows:

$$PE_{N,ILP} = \frac{\textit{Execution time on } \texttt{ILP} \textit{ uniprocessor}}{\textit{Execution time on } \texttt{ILP} \textit{ multiprocessor}} \times \frac{1}{N} \tag{3.11}$$

$$= \frac{\frac{\textit{Execution time on } \texttt{Simple} \textit{ uniprocessor}}{\textit{Uniprocessor ILP speedup}}}{\frac{\textit{Execution time on } \texttt{ILP} \textit{ multiprocessor}}{\textit{Multiprocessor ILP speedup}}} \times \frac{1}{N} \tag{3.12}$$

$$= \frac{\textit{Multiprocessor ILP speedup}}{\textit{Uniprocessor ILP speedup}} \times \frac{\textit{Execution time on } \texttt{Simple} \textit{ uniprocessor}}{\textit{Execution time on } \texttt{Simple} \textit{ multiprocessor}} \times \frac{1}{N} \tag{3.13}$$

$$= \frac{\textit{Multiprocessor ILP speedup}}{\textit{Uniprocessor ILP speedup}} \times PE_{N,Simple} \tag{3.14}$$

Thus, the parallel efficiency seen by the `ILP` system is greater than that seen by the `Simple` system if the multiprocessor ILP speedup is greater than the uniprocessor ILP speedup; if the multiprocessor ILP speedup is lower, the `ILP` system has less parallel efficiency than the `Simple` system. Thus, to understand the difference seen in the parallel efficiencies between the `Simple` and `ILP` multiprocessors, Figures 3.6–3.9 presents data to illustrate the impact of ILP in uniprocessors, analogous to the data in Figures 3.1–3.4 for multiprocessors. As in multiprocessors, uniprocessor CPU ILP

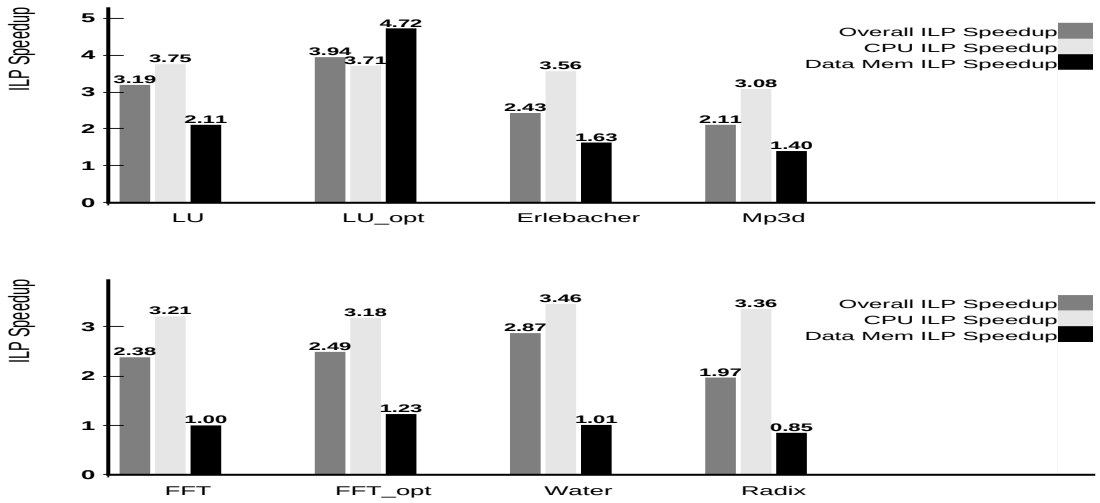


Figure 3.6 ILP speedup and components in a uniprocessor system

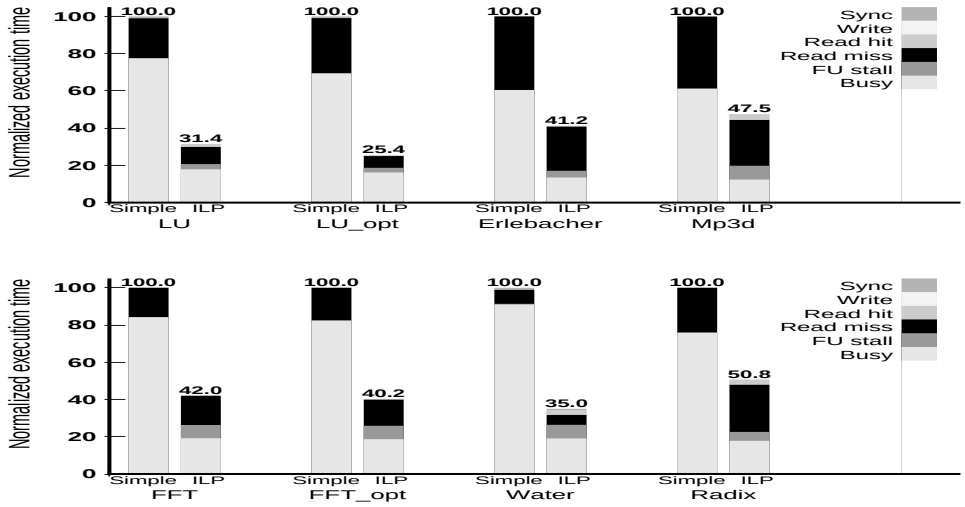


Figure 3.7 Execution time components in a uniprocessor system

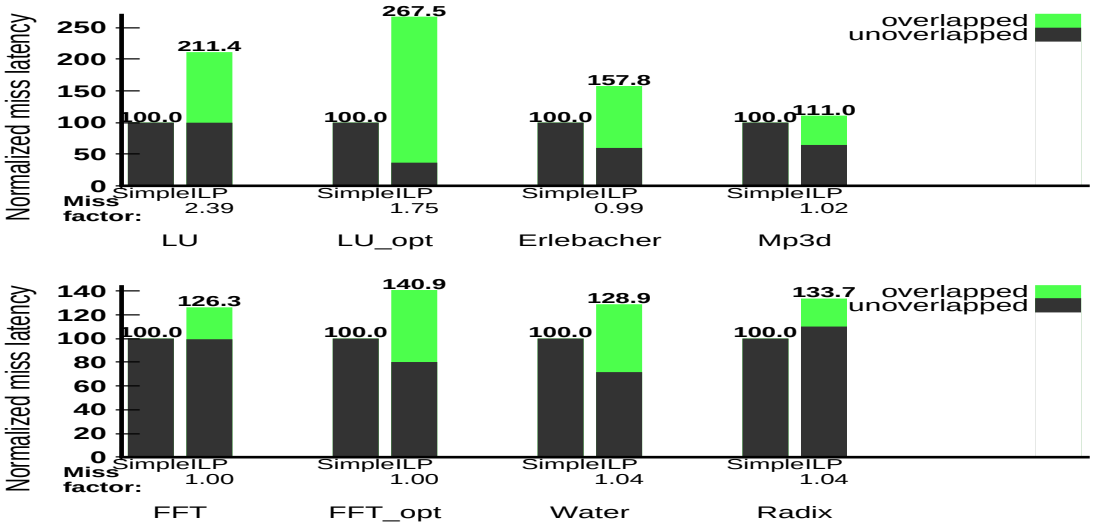
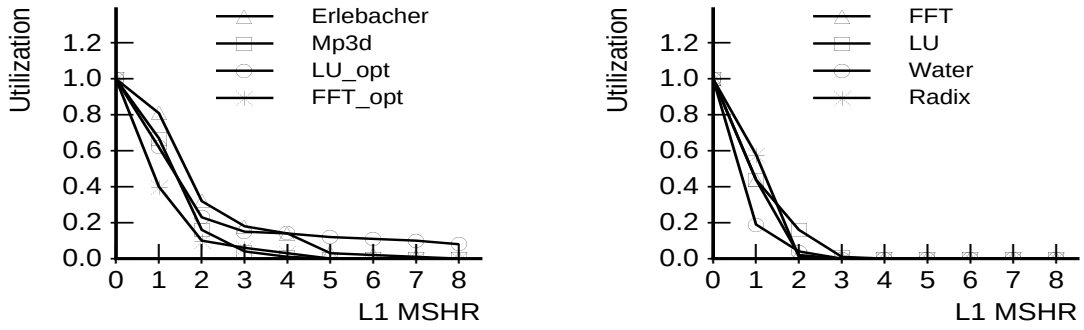


Figure 3.8 Effect of ILP on average miss latency in a uniprocessor system

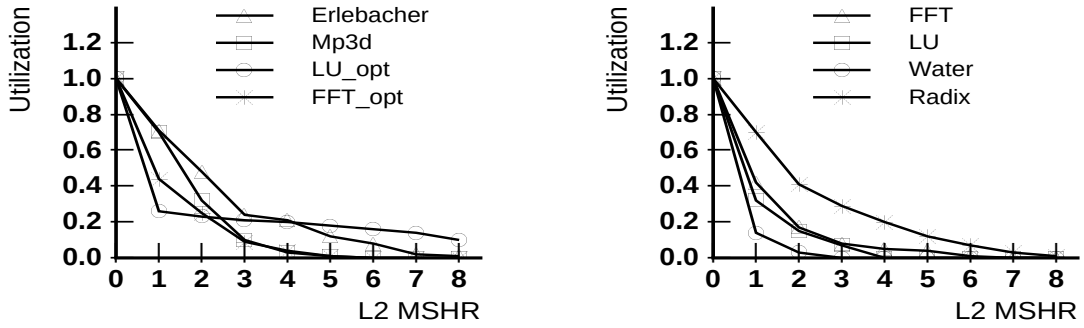
speedups are high while memory ILP speedups are generally low. However, comparing Figure 3.6 with Figure 3.1 shows that for all applications other than FFT_opt, the overall ILP speedup is less in the multiprocessor than in the uniprocessor. This degradation directly implies lower parallel efficiency for the `ILP` multiprocessor than the `Simple` multiprocessor. We next describe several reasons for the lower ILP speedup in the multiprocessor and then describe why FFT_opt does not follow this trend.

First, comparing Figure 3.7 with Figure 3.2 shows that, for most applications, the read miss component of execution time is more significant in the multiprocessor because these applications see a large number of remote misses. Consequently, read miss ILP speedup plays a larger role in determining overall ILP speedup in the multiprocessor than in the uniprocessor. Since read miss ILP speedup is lower than CPU ILP speedup, and since read miss ILP speedup is not higher in the multiprocessor than in the uniprocessor, the larger role of read misses results in an overall ILP speedup degradation on the multiprocessor for these applications.

Second, for some applications, our CC-NUMA `ILP` multiprocessor may see less read miss overlap because of the dichotomy between local and remote misses in multiprocessor configurations; multiprocessors not only need a clustering of misses for



(a) L1 MSHR occupancy



(b) L2 MSHR occupancy

Figure 3.9 MSHR occupancy in a uniprocessor `ILP` system

effective overlap, but also require remote misses to be clustered with other remote misses in order to fully hide their latencies*. All applications other than FFT_opt that achieve significant overlap in the uniprocessor see less overlap (and, consequently, less read miss ILP speedup) in the multiprocessor because their data layouts do not provide similar latencies for each of the misses overlapped in the instruction window. As a result, read misses in the multiprocessor configuration are not overlapped to the same extent as in the uniprocessor case, leading to a reduction in read miss ILP speedup in the multiprocessor case.

*This effect would not apply to a uniform-memory-access (UMA) machine. However, the other effects in this section are applicable to both NUMA and UMA multiprocessors.

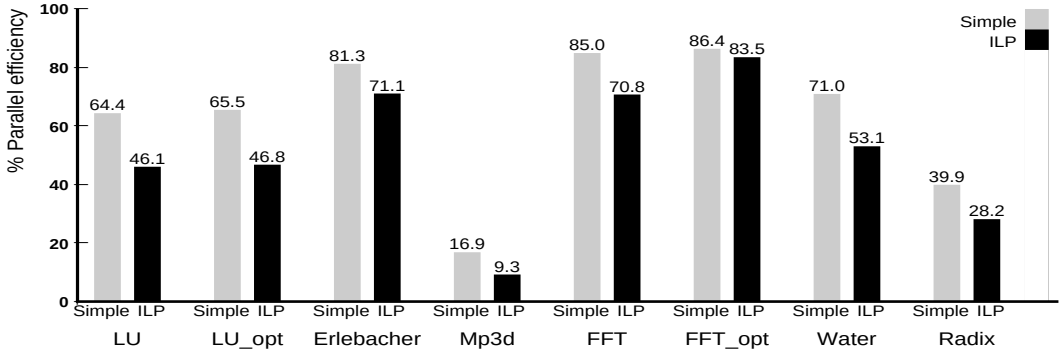
Third, the read miss ILP speedups of most applications degrade from increased contention in the multiprocessor. Radix is an extreme example where L2 MSHR saturation occurs in the multiprocessor case but not in the uniprocessor. This MSHR saturation arises because extensive false-sharing in the multiprocessor causes writes to take longer to complete; therefore, writes occupy the MSHRs for longer, increasing the MSHR contention seen by reads.

Finally, synchronization presents additional overhead for multiprocessor systems, and in most cases sees less ILP speedup than the overall application.

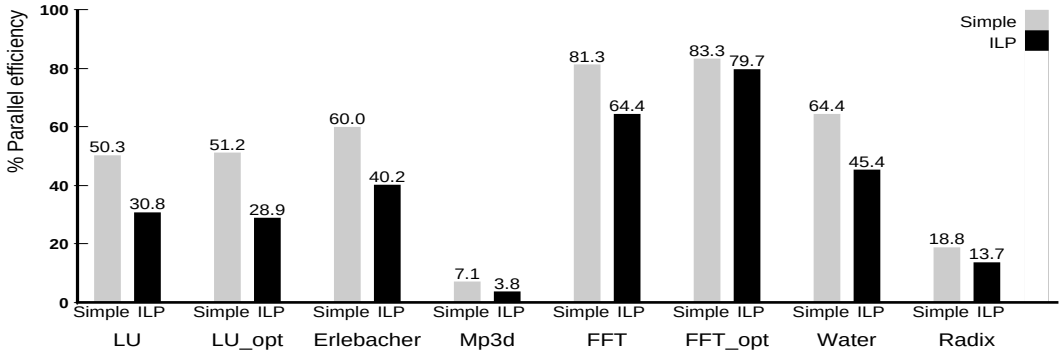
FFT_opt stands apart from the other applications for two key reasons. First, FFT_opt avoids a reduction in read miss overlap in the multiprocessor since reads that cluster together in the instruction window in the blocked transpose phase of the algorithm are usually from the same block, with the same home node and sharing pattern. Therefore, these reads do not suffer from the effects of the dichotomy between local and remote misses described above. Second, the introduction of remote misses causes the blocked transpose phase of the algorithm to contribute more to the total memory stall time, as this is the section with the most communication. As this is also the only phase that sees significant read miss ILP speedup, total read miss ILP speedup increases, preventing degradation in overall ILP speedup.

3.2.2 Parallel Efficiency in Larger Configurations

Figure 3.10 shows the parallel efficiencies seen in our applications with 16 and 32 processor `Simple` and `ILP` systems. Following the trends of Section 3.2.1, Figure 3.10 shows that `ILP` systems tend to scale less effectively than `Simple` systems. To analyze the sources of this trend, Figure 3.11 shows ILP speedup components for our applications in 1, 8, 16, and 32 processor configurations. For all the applications (except Radix), ILP effectiveness decreases in 16 and 32 processor configurations; however, that decrease is generally not as substantial as the initial decrease from uniprocessor to 8-processor systems. The reasons for the decrease in the applications other than



(a) 16 Processor System



(b) 32 Processor System

Figure 3.10 Parallel Efficiency in Larger Configurations

FFT_opt and Radix, however, are analogous to the reasons for the decrease from 1 to 8 processors described in Section 3.2.1.

In FFT_opt, a reduction in memory ILP speedup with 16 and 32-processor configurations leads to a subsequent reduction in overall ILP speedup. This reduction in memory ILP speedup stems from two sources. First, the read misses that cluster together in the instruction window experience memory bank conflicts in larger configurations. This effect arises because the padding added to the matrix used in this application must amount to at least one page (4096 bytes) per processor. As a result, 4 cache lines must be added to each row in a 16-processor configuration, and 8 cache lines must be added to each row in 32-processor configurations. As our system has 4-way interleaving, accesses in the same column of different rows will con-

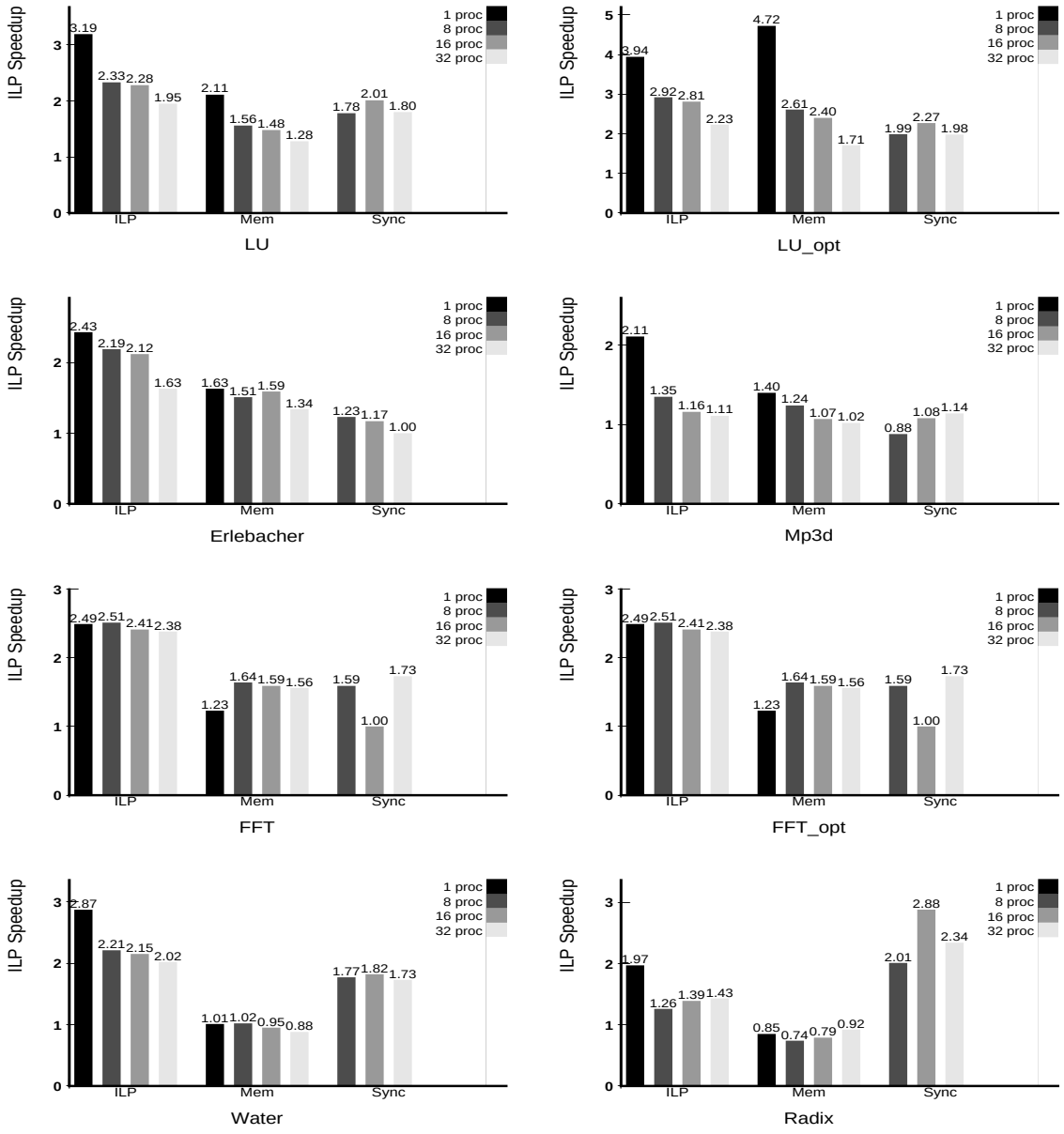


Figure 3.11 Scalability of ILP speedup

tend for the same memory module. As the accesses clustered together in FFT_opt are of this variety, contention increases and read miss ILP speedup consequently degrades. Additionally, some degradation in read miss ILP speedup arises because the 16-processor and 32-processor configurations allow more of each processor's working set to fit into the L2 cache. As a result, some of the overlapping accesses are L2 hits, while others must go to local or remote memory. The dichotomy between the latencies seen in these overlapping accesses can thus reduce the benefits of overlap in the same fashion as the dichotomy between local and remote memory accesses described in Section 3.2.1.

In Radix, ILP speedup actually starts increasing with increasing number of processors for two reasons. The primary reason is that synchronization takes up a progressively larger time as more processors are added. With Radix, the `ILP` processor provides better load balance in a relatively unbalanced prefix sum phase, by making the computation go faster. Thus, synchronization speedup is high, and an increased contribution of this component increases the overall speedup. Second, the memory miss ILP slowdown improves slightly because of a higher miss factor.

3.3 Alleviating Limitations to ILP Speedup

Section 3.1 showed that multiprocessor ILP speedup for our applications and architecture has considerable scope for improvement. Section 3.2 further showed that the multiprocessor lags behind the uniprocessor in ILP speedup for all but one of our applications. We see that both deficiencies come from three primary limitations:

1. insufficient overlap
2. increased resource contention
3. large memory component of execution time, even in `Simple`

This section examines the extent to which these limitations are artifacts of current technological constraints. The three subsequent sections respectively examine a

system with a much larger ILP instruction window to address limitation (1), a system with much greater network and memory system bandwidth to address limitation (2), and a system with larger cache sizes to address limitation (3). Each section focuses on four representative applications, (LU_opt, FFT_opt, Erlebacher, and Radix) on both uniprocessor and 8-processor configurations. These applications are chosen to represent high, moderate, and low ILP speedups.

3.3.1 Effect of Larger Instruction Window

This section evaluates an ILP system with a 256-entry instruction window. The register file needed by such a large instruction window may lead to a negative impact on clock cycle time given current technological constraints [FJC96]; however, it is necessary to choose such a size since our goal is to determine the extent to which current instruction window sizes limit ILP performance. Additionally, the memory unit is scaled to 128 entries.

Figure 3.12 shows the effects of the larger window size on uniprocessor and 8-processor systems for our representative applications. As expected, the larger instruction window improves the performance of both multiprocessor and uniprocessor configurations by increasing memory miss ILP speedup.

The improvements with the large instruction window decrease the gap between uniprocessor and multiprocessor ILP speedup significantly for LU_opt and Erlebacher. In LU_opt, there is a large difference in the memory speedups in the uniprocessor and multiprocessor configurations; however, enough overlap is exposed in both configurations that memory speedup reaches a point of diminishing significance and CPU ILP speedup starts to dominate the calculation of overall ILP speedup. Since CPU speedups are similar, overall ILP speedups in the uniprocessor and multiprocessor versions do not differ as greatly as with a smaller instruction window. In Erlebacher, the larger instruction window increases data memory ILP speedup greatly in both the uniprocessor case and multiprocessor case; in fact, multiprocessor memory speedup

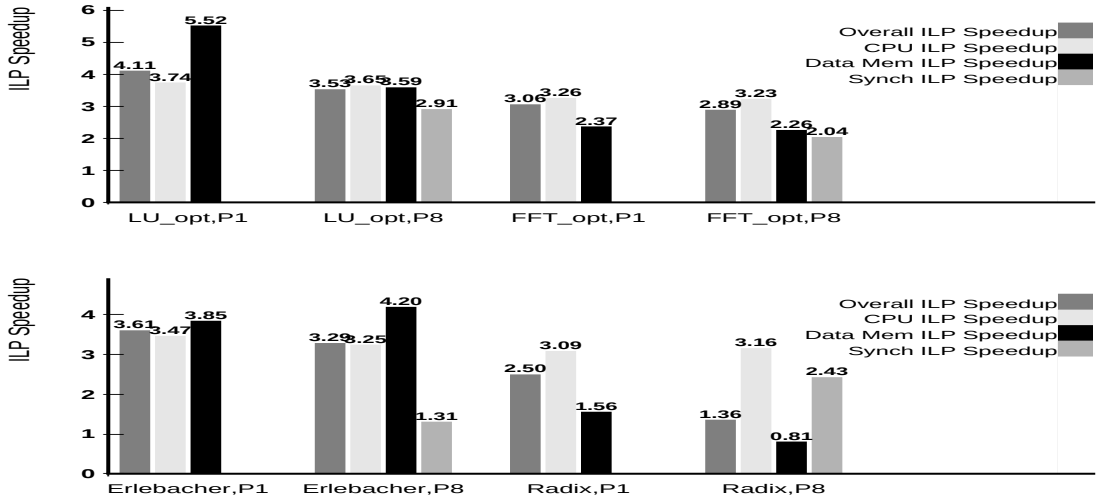


Figure 3.12 Effectiveness of ILP with larger instruction window

actually exceeds uniprocessor memory speedup because the multiprocessor avoids some of the memory bank conflicts seen in the uniprocessor case. As a result, the difference between uniprocessor and multiprocessor ILP speedups shrinks in this application.

In contrast, the ILP speedup improvements from the larger instruction window increase the gap between uniprocessor and multiprocessor ILP speedups in FFT_opt and Radix. In these applications, the increased gap occurs because of resource contention: specifically, high write traffic in the multiprocessor Radix saturates the write-buffer and secondary cache MSHRs, and high read miss traffic in the multiprocessor FFT saturates the primary cache MSHRs. This saturation leads to blockage and backup, eventually preventing the processor from issuing memory accesses until the saturation clears up. In both applications, shorter latencies prevent the uniprocessor from experiencing saturation effects. Thus, these applications have become limited by the resource needs imposed by longer latencies rather than overlap alone; thus, increasing the size of the instruction window cannot solve the multiprocessor degradation in ILP speedup for these applications.

3.3.2 Effect of a High-Bandwidth System

This section evaluates the performance of the representative applications with a system aimed to greatly reduce resource contention. The systems for this section use 64 L1 and L2 MSHRs, 16 L1 cache ports, 512 write buffer entries, 256-way interleaved memory, and a 128 byte wide interconnection network (wider than the largest single packet in our system). The high-bandwidth system aims to reduce the effects of contention and extra miss latency, so it may be able to more thoroughly exploit available potential for overlap.

As seen in Figure 3.13, the high bandwidth system improves memory ILP speedup and overall speedup for each of our applications. The reduction in contention latency provided by this system leads to lower unoverlapped latency factors and, consequently, to better memory ILP speedups. This improvement helps reduce the gap between uniprocessor and multiprocessor ILP speedup in Radix, since the multiprocessor case can more thoroughly utilize the additional resources (particularly L2 MSHRs) provided by this configuration. However, a gap between uniprocessor ILP speedup and multiprocessor ILP speedup still remains for LU_opt and Erlebacher. This is because the reasons for the ILP speedup degradation in the multiprocessor configuration for the original system still hold: (i) longer, more variable (e.g., local vs. remote) latencies in the multiprocessor make overlap more difficult and increase the weight of the memory component, and (ii) degradation from synchronization persists for Erlebacher.

3.3.3 Effect of Larger Caches

In this section we determine ILP effectiveness for systems with a 32 KB two-way associative primary cache and a 2 MB eight-way associative secondary cache. This second-level cache holds all important data sets for our representative applications [WOT+95].

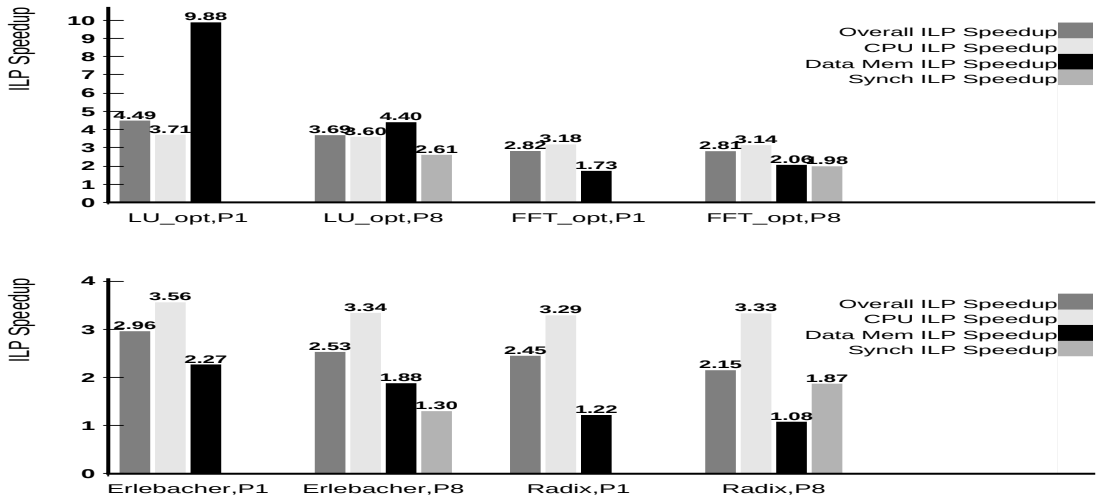


Figure 3.13 Effectiveness of ILP in very high bandwidth system

Larger caches have three primary effects on our systems. First, by decreasing effective memory latencies, they reduce the relative execution time contribution of the memory component. This effect makes CPU ILP speedup more important in determining overall ILP speedup. Second, by reducing the number of misses, larger caches reduce network and MSHR contention and so give better speedup for contention-bound applications. Finally, larger caches reduce the potential for miss overlap since they provide fewer misses to overlap. This effect reduces memory ILP speedup in applications which had high overlap because of conflict or capacity misses in the smaller cache configurations.

Figure 3.14 shows the effectiveness of `ILP` systems for our representative applications, in uniprocessor and multiprocessor configurations. The cache size increase improves overall multiprocessor ILP speedup for all the representative applications. However, in LU_opt and Radix, this is a smaller improvement than that brought about by providing higher bandwidth, and in LU_opt, FFT_opt, and Erlebacher, this is a smaller improvement than that provided by a bigger instruction window. Larger caches reduce memory ILP speedup in the case of LU_opt by turning many previously

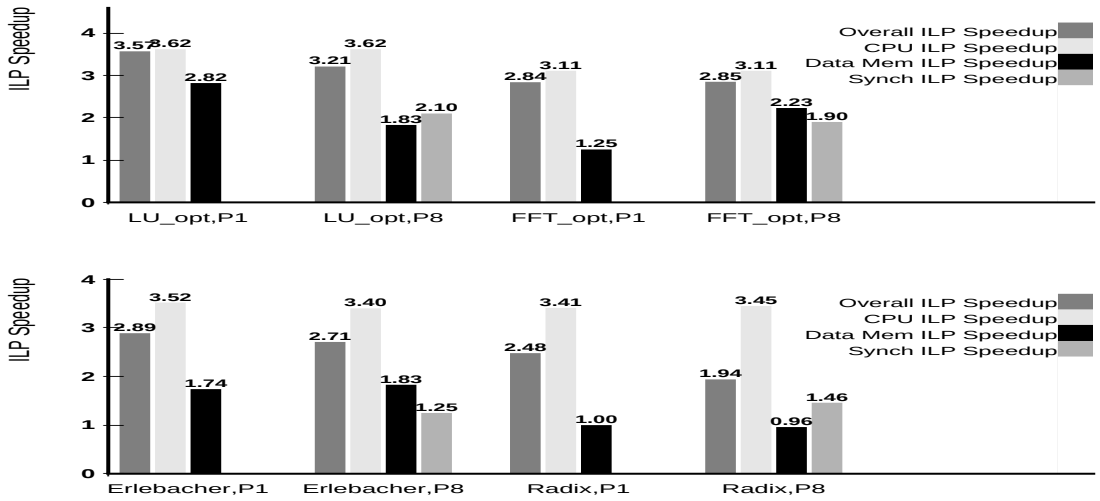


Figure 3.14 Effects of larger cache configuration

overlapped misses into primary cache hits; additionally, larger caches do not address coherence misses in our applications.

Although memory ILP speedup remains lower than CPU ILP speedup, the weight of memory time decreases, making CPU speedup more important in determining overall ILP speedup. As a result, the ILP speedups of the various applications fall into a smaller range than with the default configuration.

Comparing uniprocessor and multiprocessor systems, we find that larger cache sizes have narrowed the gap for LU_opt, Erlebacher, and Radix, while having little effect on the already narrow gap for FFT_opt. In each of these cases, we see that the large caches prevent multiprocessors from exacerbating the negative effects of ILP, since these large caches reduce much of the latency and resource occupancy that plague our default system.

3.3.4 Summary

We find that each of the architectural enhancements we examined effectively handles the issue it addresses. However, each technique also has a negative point that prevents it from reaching our goal of high ILP speedup on multiprocessor systems,

at least equal to that achieved in uniprocessor systems. Larger instruction windows increase overlap, but contention-bound applications still experience degradation in multiprocessor systems. Higher bandwidth systems reduce contention, but fail to address the underlying low potential for overlap or to eliminate the dichotomy of local and remote accesses found in multiprocessors. Thus, while ILP speedups improve, a gap between uniprocessor and multiprocessor ILP remains. Finally, large caches can reduce both latency and contention, but also remove potential for overlap. However, a larger cache solves this latter negative aspect (of low overlap) by itself, since it decreases the weight of the memory execution component, and thus reduces the significance of memory ILP speedup. In this way large caches bring uniprocessor and multiprocessor speedups close together for all of our applications, while improving ILP speedup for three of them. Nevertheless, large caches generally provided less improvements in ILP speedup, since they also substantially improve the performance of `Simple`.

3.4 Summary and Additional Issues

This study finds that for our applications, in current systems, 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 because of insufficient clustering of read misses in our application and/or system contention from more frequent misses in the ILP system, and leads to two key consequences. First, read miss latency actually appears as a greater relative performance bottleneck in ILP multiprocessors than in previous-generation multiprocessors, despite the latency-tolerating techniques incorporated in ILP processors. Second, ILP processors generally achieve less parallel efficiency and scalability than previous-generation systems.

Our experiments also consider three hardware modifications that aim to increase ILP multiprocessor performance. These experiments show that each of these tech-

niques may solve one of the problems in obtaining high ILP speedup, but each technique also experiences certain limitations. Further, the levels of hardware support needed for some of these techniques are not yet technologically or economically feasible. Thus, it seems appropriate to consider other modifications that can more easily target the remaining read miss latency in ILP multiprocessor systems. Such support may include novel latency-tolerating techniques such as compiler optimizations to more aggressively cluster read misses, while also accounting for the disparity in latencies between local and remote misses. Previously investigated latency-tolerating techniques such as software-controlled non-binding prefetching may also help to reduce the memory latency bottleneck in ILP multiprocessors [RPASA97]. Additionally, techniques to reduce latency (rather than tolerating latency) or to decrease bandwidth requirements, such as producer-initiated communication (e.g. [ASHAA97, HLRW92, Pou94]), may address any remaining deficiencies in the potential of ILP multiprocessors to hide miss latency.

Chapter 4

Impact of ILP on Simulation Methodology

The previous chapter uses a detailed cycle-by-cycle simulator to understand the impact of ILP on a multiprocessor. We next explore the validity of modeling `ILP` systems with simulators based on the `Simple` processor model and its variants, which are commonly used in multiprocessor architectural studies. Such simulator models enable the use of current direct-execution simulation techniques and thus can complete multiprocessor simulations much faster than a more detailed `ILP`-based simulation model.

4.1 Models and Metrics

For the experiments in this chapter, we study three variants of the `Simple` model to approximate the `ILP` model based on recent literature [HKO+94, HSH96]. The first two, `Simple.2xP` and `Simple.4xP`, model the `Simple` processor sped up by factors of two and four, respectively. `Simple.2xP` seeks to set peak IPC equal to the IPC achieved by the target `ILP` system (Our `ILP` system generally obtains an IPC of approximately 2 for our applications). `Simple.4xP` seeks to achieve an instruction issue rate equal to `ILP`, which is 4 in our system. For the memory hierarchy and interconnect, both models use the same latencies (in terms of absolute time) as the `ILP` system. Thus the latencies in Figure 2.3, which are given in terms of processor cycles, need to be appropriately scaled for these models. The final approximation, `Simple.4xP.1cL1`, not only speeds up the processor by a factor of 4, but further recognizes that L1 cache hits should not stall the processor. Hence, this model ad-

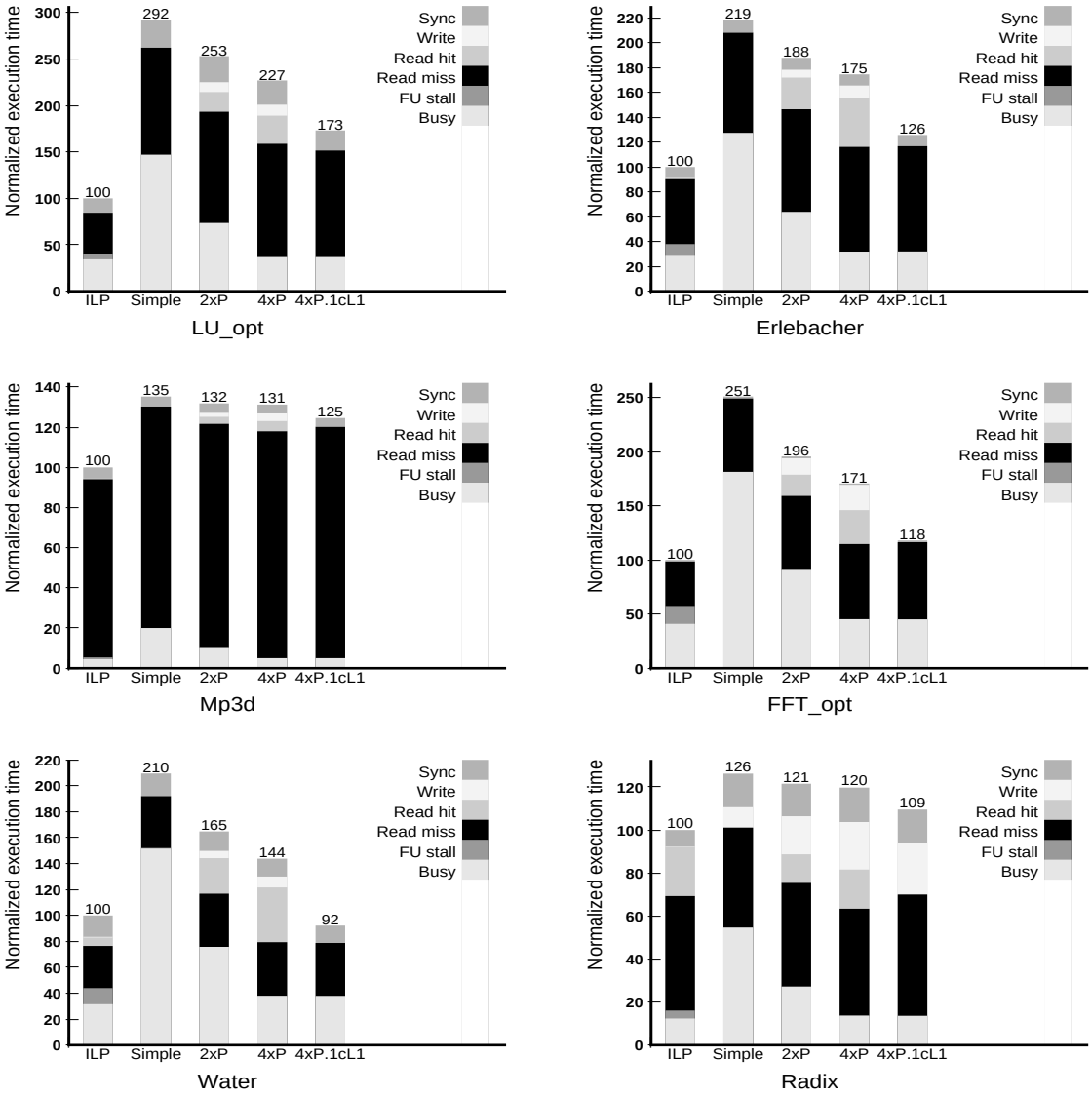


Figure 4.1 Predicting execution time and its components using simple simulation models

ditionally speeds L1 cache and write buffer access time to one processor cycle of this model. The rest of the memory hierarchy and the interconnect remain unchanged.

We use the total execution time and the relative importance of various components of execution time as the primary metrics to describe the effectiveness of these simulation models. We additionally investigate the accuracy of these simulation models in determining the multiprocessor speedups of representative applications. This study does not present results for the unoptimized versions of FFT and LU, as the optimized versions of these applications give better performance on the ILP system.

4.2 Execution Time and its Components

Figure 4.1 shows the total simulated execution time (and its components) for each application and simulation model, normalized to the simulated execution time for the `ILP` model for the specific application.

Chapter 3 already compares the `Simple` and `ILP` models. The ILP speedup of an application indicates the factor by which the total execution time of the `Simple` model deviates from the actual time with `ILP`. As a result, the error in predicted total execution time increases for applications that are better at exploiting ILP. This error occurs from mispredicting the time spent in each component of execution time in proportion to the ILP speedup of that component. Errors in the total execution time with the `Simple` model range from 26% to 192% for our applications.

`Simple.2xP` and `Simple.4xP` reduce the errors in total execution time by reducing busy time compared to `Simple`. Busy time falls by factors of roughly 2 and 4, respectively, in these models, and actually resembles `ILP` busy time in the case of `Simple.4xP`. However, absolute read miss time stays nearly unchanged compared to `Simple`, and actually increases in some cases due to added contention. Synchronization time also remains mostly unchanged. Further, these two models add extraneous read hit stall components, since every L1 cache access now takes more than one processor cycle; one of these cycles is considered busy, but the remaining

processor cycles are considered stall time because of blocking reads. Similarly, each of these models also incurs an unwanted write component.† As a result, errors in total execution time range from 21% to 153% for `Simple.2xP` and 20% to 127% for `Simple.4xP`, for our applications.

`Simple.4xP.1cL1` removes the extraneous read hit and write components of `Simple.4xP`. This model is more accurate than the other `Simple`-based models in predicting the total execution time, giving approximately 25% or less error on five applications. However, in the presence of high read miss ILP speedup, the inaccuracies in predicting read miss time still persist, giving an error of 73% in predicting the execution time for LU_opt. `Simple.4xP.1cL1` also significantly overestimates read miss time in FFT_opt and Erlebacher, bloating this component by 72% and 59% respectively. However, FFT_opt and Erlebacher do not see corresponding errors in total execution time because `Simple.4xP.1cL1` does not account for the functional unit stall component of `ILP`. This underestimate of CPU time offsets some of the overestimate in read miss time prediction, but does not solve the `Simple`-based models' fundamental inability to account for the effects of high or moderate read miss ILP speedup. Overall, as with the other `Simple`-based models, the errors seen with this model are also highly application-dependent, ranging from -8% to 73%, depending on how well the application exploits ILP.

4.3 Error in Component Weights

For certain studies, accurate prediction of the relative weights of the various components of execution may be more important than an accurate prediction of total execution time. We therefore next examine how well the `Simple`-based models predict the relative importance of the various components of execution time. We specifi-

†Neither the `Simple` nor the `ILP` processor stall for the completion of writes; however, the `Simple` processor must wait for a write to access the write-buffer before retiring that write, whereas `ILP` can retire a write before it is issued to the memory system, as long as a slot in the memory unit is available.

	LU opt	Erlebacher	Mp3d	FFT opt	Water	Radix
ILP	44.1	53.3	89.0	41.6	39.5	76.1
Simple	39.4	36.9	81.5	27.0	19.1	44.4
4xP.1cL1	66.3	67.4	92.4	60.4	44.3	73.4

Figure 4.2 Relative importance of memory component

cally focus on the `Simple` and `Simple.4xP.1cL1` models, since the former is the most widely used, and the latter is the most accurate in predicting total execution time for our applications. We also focus only on the percentage of execution time spent on the memory component with these simulation models; Figure 4.2 tabulates this data. Similar information for the other components and models can be derived from the graphs of Figure 4.1.

As shown in Chapter 3, the memory component is a greater portion of execution time on `ILP` systems than on `Simple` systems. `Simple` thus underestimates the importance of memory time in all of our applications (by more than 30% on four of them). In contrast, `Simple.4xP.1cL1` tends to overestimate the relative weight of the memory component, as this model fails to account for read miss overlap and also generally underestimates CPU time. These errors are highest in applications with moderate to high memory ILP speedup, with overestimates of 50%, 45%, and 27% in LU_opt, FFT_opt, and Erlebacher respectively.

4.4 Error in Multiprocessor Speedup

Figure 4.3 illustrates the multiprocessor speedups seen by the ILP-based system and as predicted by the `Simple` and `Simple.4xP.1cL1` simulation models. We only show graphs for applications that scale reasonably with at least one of the models. If the uniprocessor configuration has a higher ILP speedup than the multiprocessor configuration, then `Simple` predicts a higher speedup, and vice versa. The former

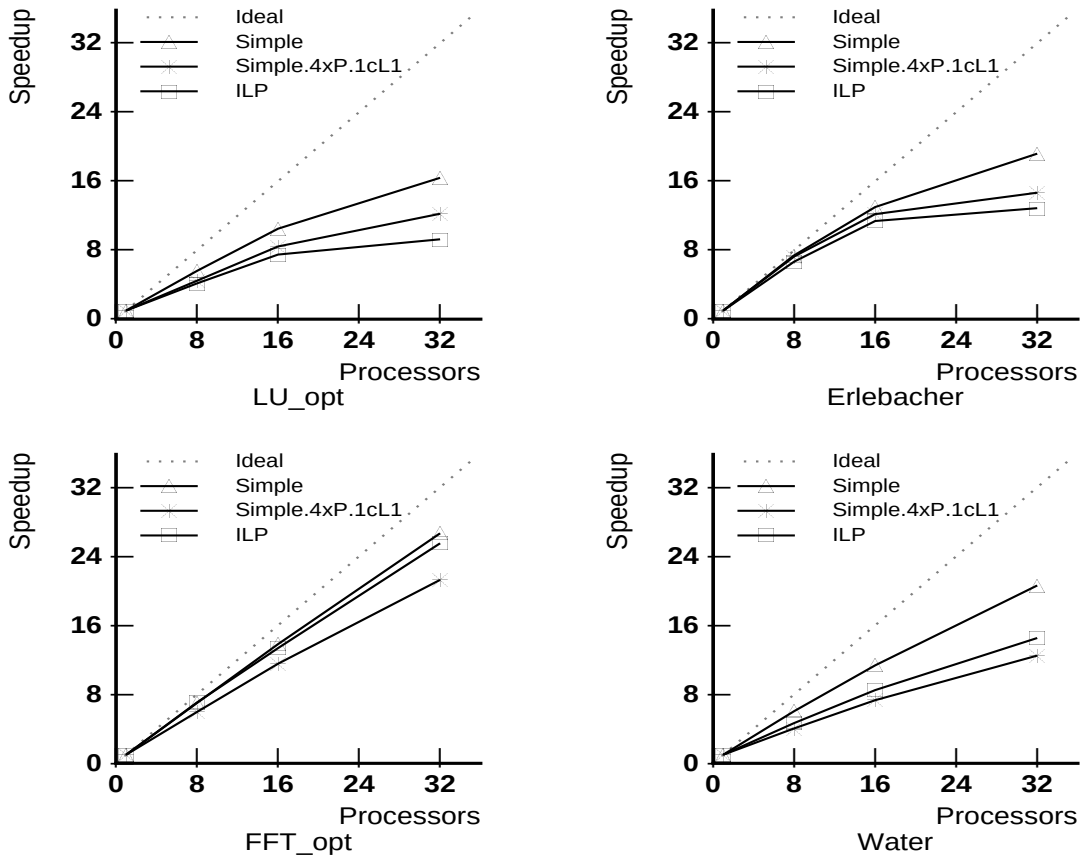


Figure 4.3 Speedups for Simple, ILP, Simple.4xP.1cL1 models

is the case for all the applications we consider (Section 4), although the error in prediction varies by application. Thus, FFT_opt has a very slight error while Water shows a large error.

`Simple.4xP.1cL1` is a good estimator of overall execution time for applications that do not exhibit significant memory ILP speedup; therefore, it follows that the model predicts multiprocessor speedups fairly well for Water. For the other applications, the model predicts higher speedup when its uniprocessor time prediction has a higher error than its multiprocessor time prediction. This is the case when uniprocessor ILP speedup is higher than the multiprocessor ILP speedup, as in Erlebacher and LU_opt. The reverse is true for FFT_opt, where the model predicts a lower speedup

compared to the ILP model. For all but FFT_opt, Simple.4xP.1cL1 gives a better approximation than Simple.

4.5 Summary and Alternative Models

The Simple, Simple.2xP, and Simple.4xP models studied in this chapter see a large range of errors across all our applications. In contrast, the Simple.4xP.1cL1 model provides a more reasonable approximation to ILP on many of our applications. However, although this model predicts the behavior of the busy and L1 cache hit components of the execution time reasonably well, it does not model the possibility of read miss speedup. Consequently, this model reasonably approximates ILP behavior for applications with low read miss ILP speedup, but can show high inaccuracies in predicting the performance of ILP on applications with high read miss ILP speedup.

A key insight for Simple.4xP.1cL1 is that ILP processors hide nearly all of the L1 cache hit latency. However, our detailed statistics (not shown here) show that ILP also overlaps most of the L2 cache hit latency. Thus, a reasonable extension to Simple.4xP.1cL1 would be to speed up L2 cache hit time to a single processor cycle. However, this model would remain inadequate in predicting the performance of applications which overlap portions of their local and remote memory accesses. Extending the above model further to account for local and remote memory accesses seems impractical, as overlap in these components of memory is highly application-specific and hardware-dependent, and is not known *a priori*. Thus, our results indicate the need for detailed simulators that employ an ILP processor model for multiprocessor systems.

Unfortunately, current multiprocessor simulators that account for ILP processor characteristics are much slower than simulators that do not model ILP processor characteristics, as such simulators generally employ direct-execution simulation techniques. For example, our experiments show the detailed execution-driven simulator RSIM, which handles ILP multiprocessors, to be at least 7 times slower in elapsed

time than the direct-execution shared-memory multiprocessor simulator on which it is based, RPPT [Raj95]. As fast simulators are needed to characterize the behavior of larger applications and data sets, our results motivate the need for new techniques to allow more efficient ILP processor simulation. For example, it may be possible to add support for non-blocking reads to direct-execution simulation techniques, thus allowing fast simulators to capture the effects of memory overlap.

Chapter 5

Related Work

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 consistency models using ILP multiprocessors [GGH92, PRAH96, ZB92]. However, none of the above work details the benefits achieved by ILP in the multiprocessor.

Our variants of the `Simple` processor model in Chapter 4 are based on the works of 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 (used in [RPW96]) 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. 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 [BIAD89]. 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].

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. This work identifies a *critical latency* for each program, defined as the maximum cache miss latency which can be perfectly tolerated by the system. They find that for greater cache miss latencies, some latency tolerance is still possible if the program can overlap multiple misses together. Our work finds that read clustering is a key optimization that enables multiprocessors to exploit the features of ILP processors, additionally finding 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. 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. 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. 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.

Other studies on the impact of ILP on uniprocessor configurations include the following. Jouppi and Wall look at ILP speedup of various processor configurations [JW89]. However, their main results do not consider cache misses. When they do consider cache misses, it is with regard to the impact of cache misses on peak IPC, rather than its effect on ILP speedup. Sohi and Franklin look at a variety of techniques for improving the bandwidth capacity of the first level cache [SF91]. Their work specifically addresses the extra bandwidth needs of superscalar processors. This paper proposes using multiported non-blocking L1 caches to increase the peak cache bandwidth. Conte also addresses the benefits of non-blocking caches, and does so in the context of superscalar processors [Con92]. This paper specifically uses a trace-driven simulation with sampling, so speculation effects may be lost. This paper also uses IPC, rather than ILP speedup, as a metric. Farkas and Jouppi evaluate various hardware tradeoffs in non-blocking cache design and provide miss CPI results for codes compiled with different values of "load latency" [FJ94]. They briefly touch on dual-issue machines, but they do so primarily in the context of their miss CPI characteristics, rather than their ILP speedups. Finally, Bennett and Flynn provide a detailed analysis of issues in the performance of ILP-based processors in [BF95], focusing on issues related to instruction fetch bandwidth, instruction issue rate, and finite instruction window effects.

Chapter 6

Conclusions

This paper first analyzes the impact of state-of-the-art ILP processors on the performance of shared-memory multiprocessors. It then examines the validity of evaluating such systems using commonly employed simulation techniques based on previous-generation processors.

To determine the effectiveness of ILP techniques, we compare the execution times for a multiprocessor built of state-of-the-art processors with those for a multiprocessor built of previous-generation processors. We use this comparison not to suggest an architectural tradeoff, but rather to understand where current multiprocessors have succeeded in exploiting ILP and where they need improvement.

We find that, for our applications, ILP techniques effectively address the CPU component of execution time, but are less successful in improving the data read stall component of execution time in multiprocessors. The primary reasons for less read miss time speedup than CPU time speedup with ILP techniques are an insufficient potential in our applications to have multiple read misses outstanding simultaneously and/or system contention from more frequent memory accesses in the ILP-based multiprocessor.

The disparity between the impact of ILP on CPU time and on read miss time has two implications. First, read stall time becomes a much larger component of execution time in ILP multiprocessors than in previous-generation multiprocessors. Second, most of our applications show lower parallel efficiency on an ILP multiprocessor than on a previous-generation multiprocessor. The key reasons for the reduced parallel efficiency on most of our applications are the greater impact of read stall time in the

multiprocessor than in the uniprocessor, increased contention in the multiprocessor, and reduced overlap in the multiprocessor due to the dichotomy between local and remote memory accesses. However, these do not appear to be fundamental problems; one of our applications exploits enough overlap in the ILP multiprocessor to see an increase in parallel efficiency.

Overall, our results indicate that despite the latency-tolerating techniques integrated within ILP processors, multiprocessors built from ILP processors have a *greater* need for additional memory latency reducing and hiding techniques than previous-generation multiprocessors. These techniques include conventional hardware and software techniques, and aggressive compiler techniques to enhance the read miss overlap in applications, while accounting for the dichotomy between local and remote memory accesses.

When addressing the validity of using current simple-processor-based simulation models to approximate an ILP multiprocessor, we find that a model that increases the speed of both the CPU and the L1 cache is a reasonable approximation for applications with low overlap of read misses. However, this model can show significant inaccuracy in cases of high or moderate read miss overlap since it does not properly account for the effects of overlapping read misses.

Unfortunately, full ILP multiprocessor simulation will invariably take more simulation time than the direct-execution simulation models allowed by using simple processor models. Therefore, in the absence of an alternative, we expect that direct-execution-based simulators will continue to be used, particularly for large applications and large data sets. This study provides insights on the inaccuracies that can be generated and suggests that the results of such simulations should be interpreted with care. For more accurate analysis of large applications, parallelization may serve as the enabling technology for high-performance ILP simulations. Additionally, it may be possible to add support for non-blocking reads to direct-execution simulation techniques, thus allowing fast simulators to capture the effects of memory overlap.

Bibliography

[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*, pages 194–203, June 1995.

[ASHAA97] 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.

[AWMC+95] Vikram S. Adve, Jhy-Chun Wang, J. Mellor-Crummey, Daniel Reed, Mark Anderson, and K. Kennedy. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Proceedings of Supercomputing '95*, December 1995.

[BF95] James E. Bennett and Michael J. Flynn. Performance Factors for Superscalar Processors. Technical Report CSL-TR-95-661, Stanford University, February 1995.

[BIAD89] 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.

[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.

[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.

[Con92] Thomas M. Conte. Tradeoffs in Processor/Memory Interfaces for Superscalar Processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 202–205, December 1992.

[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, June 1994.

[FJC96] K. Farkas, N. Jouppi, and P. Chow. Register File Design Considerations in Dynamically Scheduled Processors. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture*, pages 40–51, February 1996.

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

[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 International Symposium on Computer Architecture*, pages 15–26, May 1990.

[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 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–285, 1994.

[HLRW92] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors. In *Proceedings 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 262–273, October 1992.

[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.

[JW89] Norman P. Jouppi and David W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, April 1989.

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

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

[NHO96] Basem A. Nayfeh, Lance Hammond, and Kunle Olukotun. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 67–77, May 1996.

[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 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, October 1996.

[Pou94] D.K. Poulsen. *Memory Latency Reduction via Data Prefetching and Data Forwarding in Shared-Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, 1994.

[PRA97a] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[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.

[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 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, October 1996.

[Raj95] Usha Rajagopalan. The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors. Master's thesis, Department of Electrical and Computer Engineering, Rice University, January 1995.

[RBH+95] Mendel Rosenblum, Edouard Bugnion, Stephen Alan Herrod, Emmet Witchel, and Anoop Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 285–298, December 1995.

[RPASA97] 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*, 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.

[SF91] Gurindar Sohi and Manoj Franklin. High-Bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, April 1991.

[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.

[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 International Symposium on Computer Architecture*, pages 24–36, June 1995.

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