To appear in Proceedings of HPCA-3 (February, 1997)

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

Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve
Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77005
$\{vijaypai \mid parthas \mid sarita\}$@rice.edu

Abstract

Current microprocessors exploit high levels of instruction-level parallelism (ILP) through techniques such as multiple issue, dynamic scheduling, and non-blocking reads. This paper presents the first detailed analysis of the impact of such processors on shared-memory multiprocessors using a detailed execution-driven simulator. Using this analysis, we also examine the validity of common direct-execution simulation techniques that employ previous-generation processor models to approximate ILP-based multiprocessors.

We find that ILP techniques substantially reduce CPU time in multiprocessors, but are less effective in reducing memory stall time. Consequently, despite the presence of inherent latency-tolerating techniques in ILP processors, memory stall time becomes a larger component of execution time and parallel efficiencies are generally poorer in ILP-based multiprocessors than in previous-generation multiprocessors.

Examining the validity of direct-execution simulators with previous-generation processor models, 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. For our applications, the errors in execution time with these simulators range from 26% to 192% for the most commonly used model, and from -8% to 73% for the most accurate model.

*This work is supported in part by the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, and CDA-9502791, and the Texas Advanced Technology Program under Grant No. 003604016. Vijay S. Pai is also supported by a Fannie and John Hertz Foundation Fellowship.

1 Introduction

Shared-memory multiprocessors built from commodity microprocessors 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 use direct-execution simulators, which typically assume a processor model with single issue, static (in-order) scheduling, and blocking reads.

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. This paper 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 and more application-dependent. This deficiency 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 execution time than in previous-generation systems. 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.

Our results on the validity of using current direct-execution simulation techniques to approximate ILP multiprocessors are as follows. For applications where our ILP multiprocessor fails to significantly overlap read miss latency, a direct-execution simulation using a simple previous-generation processor model with a higher clock speed for the processor and the L1 cache provides a reasonable approximation. However, when ILP techniques effectively overlap read miss latency, all of our direct-execution simulation models can show significant errors for important metrics. Overall, for total execution time, the most commonly used direct-execution technique gave 26% to 192% error, while the most accurate direct-execution technique gave -8% to 73% error.

The rest of the paper is organized as follows. Section 2 describes our experimental methodology. Sections 3-5 describe and analyze our results. Section 6 discusses related work. Section 7 concludes the paper.

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 aggressive memory system and interconnect suitable for `ILP` systems. 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*.

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 [17, 20]. 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. 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.

2.2 Simulated Architectures

We model 8-processor NUMA shared-memory systems with the system nodes connected by a two-dimensional mesh. Our systems use an invalidation coherence protocol and are release-consistent [7].

The following details the processors and memory hierarchy modeled. Figure 1 summarizes our system parameters. The extended version of this paper also includes results for 16 and 32 processor systems and a sensitivity analysis for several parameters [16].

Processor Models. Our `ILP` processor resembles the MIPS R10000 processor [12], with 4-way issue, dynamic scheduling, non-blocking reads, register renaming, and speculative execution. Unlike the MIPS R10000, however, our processor implements release

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	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 1: System parameters

consistency. 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 applications except Water; even with Water, 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 direct-execution simulators, we also investigate variants of the `Simple` model that reflect approximations for ILP-based multiprocessors made in recent literature. These are further described in Section 5.

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 [6] 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) [11], 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. [22]. 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. 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) [5, 18].

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. 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 op-

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: Application characteristics

tions `-O2 -funrollloop`.

2.4 Applications

We use six applications for this study – LU, FFT, and Radix from the SPLASH-2 suite [22], Mp3d and Water from the SPLASH suite [21], and Erlebacher from the Rice parallel compiler group [1]. We modified LU slightly to use flags instead of barriers for better load balance. Figure 2 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.

3 Impact of ILP on a Multiprocessor

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 Overall Results

Figures 3(a) and 3(b) illustrate our key overall results. For each application, Figure 3(a) 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[1], data memory stalls, and synchronization stalls. Figure 3(b) indicates the relative importance of the ILP speedups of the different components by showing the time spent on 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.

[1]We chose to combine the 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.

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 3.2.

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 3.3 discusses the factors affecting synchronization ILP speedup in our applications.

3.2 Data Memory ILP Speedup

We first discuss various factors that can contribute to data memory ILP speedup (Section 3.2.1), and then show how these factors interact in our applications (Section 3.2.2).

3.2.1 Contributing Factors

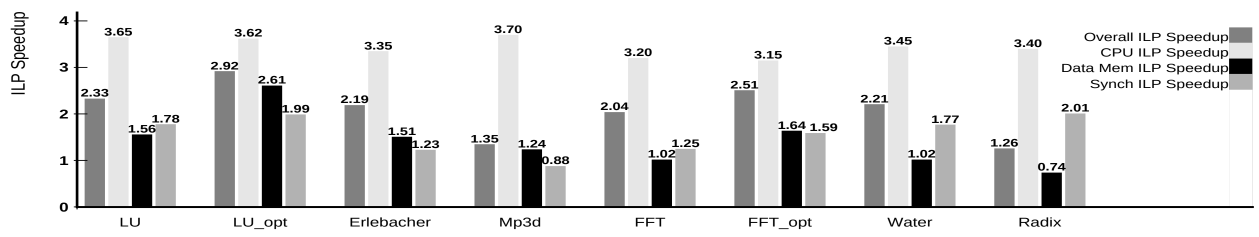
Figure 3(b) shows that memory time is dominated by read miss time in all of our applications. We therefore focus on factors influencing read miss ILP speedup. These factors are summarized in Figure 4.

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. Equation (1) in Figure 4 uses the above terms to express the read miss ILP speedup and isolates two contributing factors – the *miss factor* and the *unoverlapped factor*.

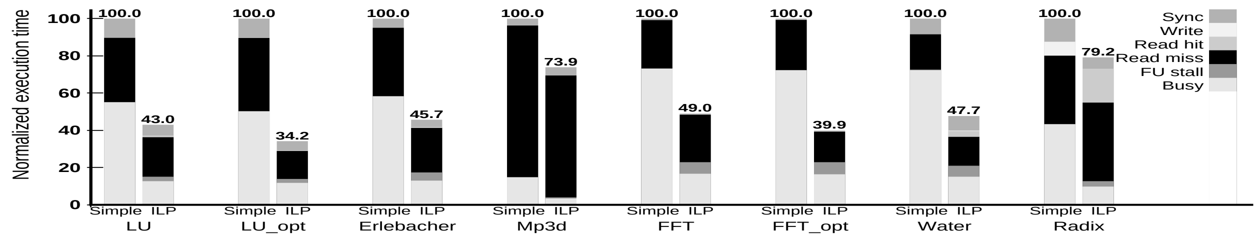
Miss factor. This is the first factor isolated in Equation (1). It specifies the ratio of the miss counts in the `Simple` and `ILP` systems. These miss counts can differ since reordering and speculation in the `ILP` processor can alter the cache miss behavior. A miss factor greater than 1 thus contributes positively to read miss ILP speedup, as the `ILP` system sees fewer misses than the `Simple` system.

Unoverlapped factor. This is the second factor isolated in Equation (1). It specifies the ratio of the exposed, or unoverlapped, miss latency in the `ILP` and `Simple` systems. The lower the unoverlapped factor, the higher the read miss ILP speedup.

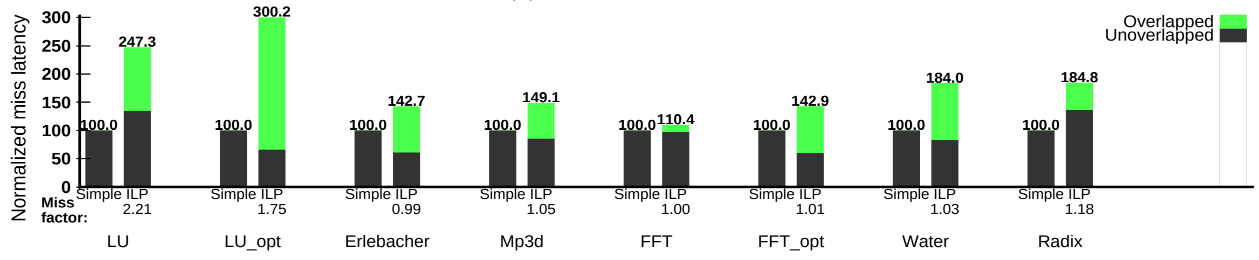
In the `Simple` system, the entire L1 miss latency is unoverlapped. To understand the factors contributing to unoverlapped latency in the `ILP` system, Equa-



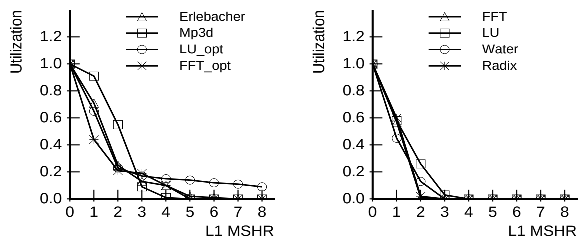
(a) ILP speedup and components



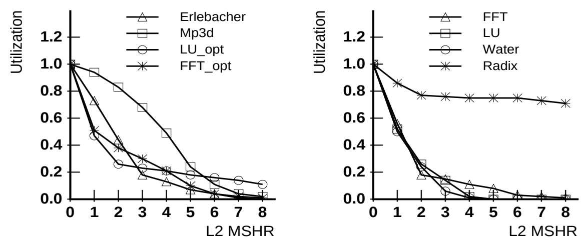
(b) Execution time components



(c) Effect of ILP on average miss latency



(d) L1 MSHR occupancy



(e) L2 MSHR occupancy

Figure 3: Effectiveness of ILP in a multiprocessor system

tion (2) first expresses the unoverlapped `ILP` miss latency as the difference between the total `ILP` miss latency and the overlapped miss latency. The total `ILP` miss latency can be expanded, as in Equation (3), 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). Finally, Equation (4) performs an algebraic simplification to express the unoverlapped factor in terms of two factors – 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. 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

$$\textbf{Read Miss ILP Speedup} = \underbrace{\frac{\#\text{ Simple L1 Misses}}{\#\text{ ILP L1 Misses}}}_{\textbf{Miss Factor}} \times \underbrace{\frac{\text{Simple Unoverlapped L1 Miss Latency}}{\text{ILP Unoverlapped L1 Miss Latency}}}_{\textbf{(1 / Unoverlapped Factor)}} \tag{1}$$

$$\textbf{Unoverlapped Factor} = \frac{\text{ILP L1 Miss Latency} - \text{ILP Overlapped L1 Miss Latency}}{\text{Simple L1 Miss Latency}} \tag{2}$$

$$= \frac{\text{Simple L1 Miss Latency} + \text{ILP Extra L1 Miss Latency} - \text{ILP Overlapped L1 Miss Latency}}{\text{Simple L1 Miss Latency}} \tag{3}$$

$$= 1 - \left(\underbrace{\frac{\text{ILP Overlapped L1 Miss Latency}}{\text{Simple L1 Miss Latency}}}_{\textbf{Overlapped Factor}} - \underbrace{\frac{\text{ILP Extra L1 Miss Latency}}{\text{Simple L1 Miss Latency}}}_{\textbf{Extra Factor}} \right) \tag{4}$$

Figure 4: Factors affecting read miss ILP speedup

forces misses to be resolved at more 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`.

3.2.2 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 next show how the factors discussed in Section 3.2.1 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 significantly 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(c) graphically represents the unoverlapped, overlapped, and extra latencies and factors described in Section 3.2.1. 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(c) incorporate time spent by a read miss in the processor memory unit and any overlap seen during that time.

Figures 3(d) and 3(e) provide additional data to indicate overlapped and extra latency after a read miss is issued to the memory system. These figures illustrate MSHR occupancy distributions at the L1 and L2 caches, respectively. They give 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. Our clustering optimizations in LU_opt and FFT_opt 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(c), 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(c) 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.

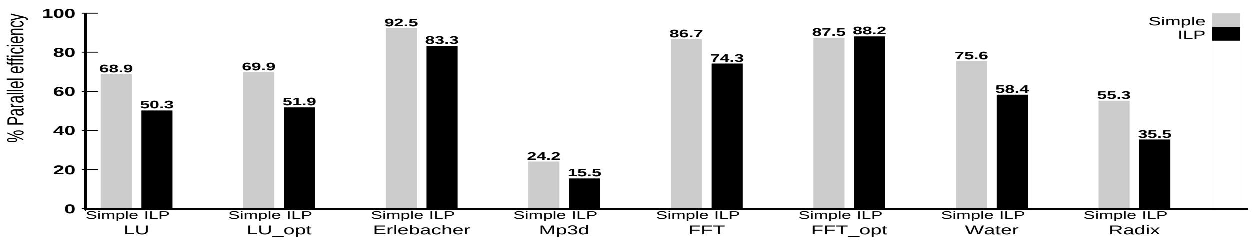


Figure 5: Parallel Efficiency with `Simple` and `ILP` systems

The high extra factor is primarily due to write traffic. Figure 3(e) shows that in Radix, L2 MSHRs are saturated for over 70% of the execution. Further 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(c), 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.

3.3 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 [7]. The third factor is a negative effect: increased contention in the memory system due to higher frequency of accesses can increase overall synchronization latency.

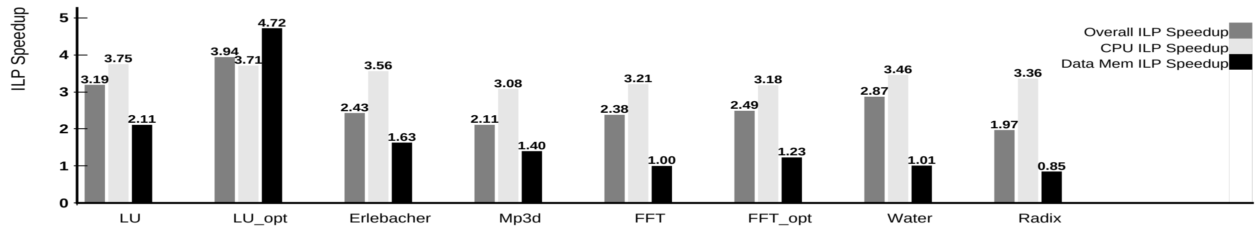
The above factors combine to produce a variety of synchronization speedups for our applications, ranging from 0.88 in Mp3d to 2.01 in Radix. However, synchronization accounts for only a small fraction of total execution time in all our applications; therefore, synchronization ILP speedup does not contribute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

4 Impact of ILP on Parallel Efficiency

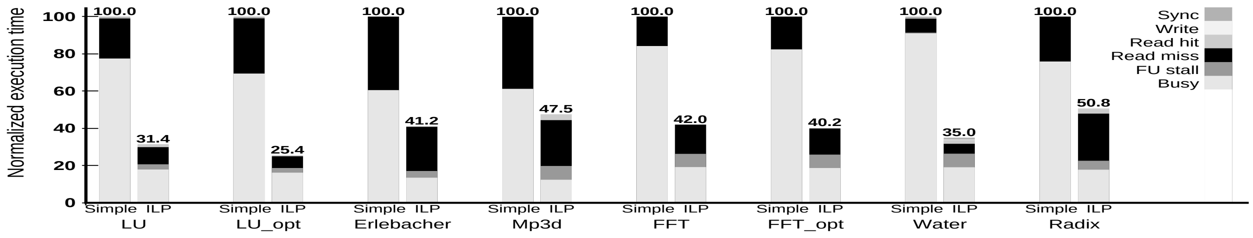
Figure 5 shows the parallel efficiency achieved by our 8-processor `ILP` and `Simple` systems for all our applications, expressed as a percentage.[2] Except for FFT_opt, parallel efficiency for `ILP` configurations is considerably less than that for `Simple` configurations. In the extended version of the paper, we show that this trend continues in 16 and 32 processor systems [16].

To understand the reasons for the difference in parallel efficiencies between `Simple` and `ILP` multiprocessors, Figure 6 presents data to illustrate the impact of ILP in uniprocessors, analogous to the data in Figure 3 for multiprocessors. As in multiprocessors, uniprocessor CPU ILP speedups are high while memory ILP

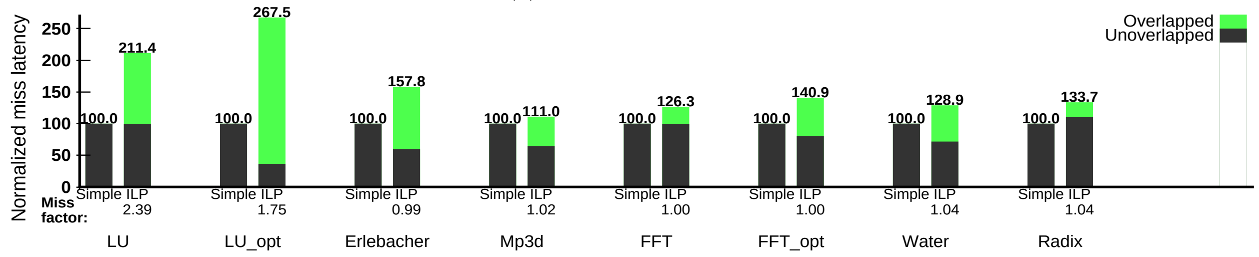
[2]Parallel efficiency of an application on an 8-processor `ILP` multiprocessor $= \left(\frac{\textit{Execution time on } \texttt{ILP} \textit{ uniprocessor}}{\textit{Execution time on } \texttt{ILP} \textit{ multiprocessor}}\right) \times \frac{1}{8}$. Parallel efficiency of an application on an 8-processor `Simple` multiprocessor is defined analogously.



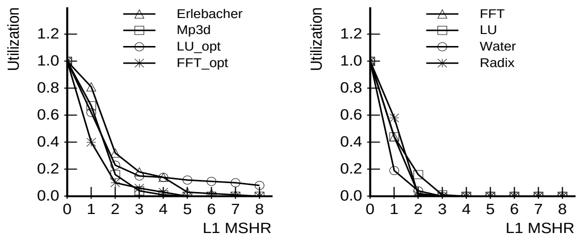
(a) ILP speedup and components



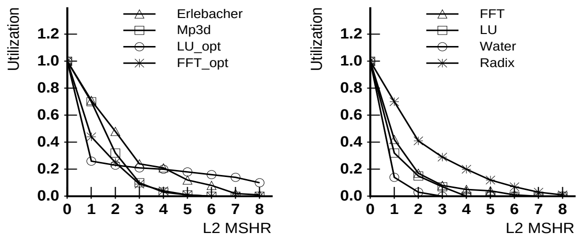
(b) Execution time components



(c) Effect of ILP on average miss latency



(d) L1 MSHR occupancy



(e) L2 MSHR occupancy

Figure 6: Effectiveness of ILP in a uniprocessor system

speedups are generally low. However, comparing Figure 6(a) with Figure 3(a) 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 6(b) with Figure 3(b) 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 `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 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.

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 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 execution 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.

5 Impact of ILP on Simulation Methodology

The previous sections use 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 direct-execution simulators based on the `Simple` processor model and its variants.

5.1 Models and Metrics

For the experiments in this section, we study three variants of the `Simple` model to approximate the `ILP` model based on recent literature [9, 10]. 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 1, 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 additionally 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. The extended version of this paper also examines other metrics [16]. Due to lack of space, we do not present the results for FFT and LU.

5.2 Execution Time and its Components

Figure 7 shows the total execution time (and its components) for each application and simulation model, normalized to the execution time for the `ILP` model for the specific application.

Section 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.[3] As a result, errors in total execution time range from

[3]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.

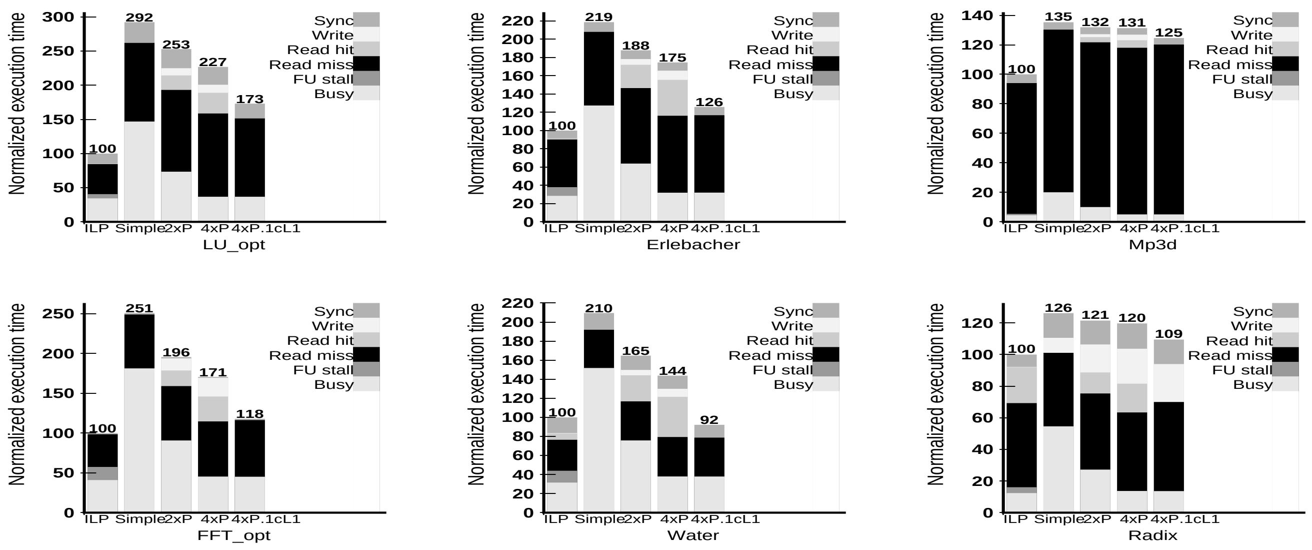


Figure 7: Predicting execution time and its components using simple simulation models

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.

5.3 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 specifically focus on the `Simple` and

	LU opt	Erle.	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 8: Relative importance of memory component

`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 8 tabulates this data. Similar information for the other components and models can be derived from the graphs of Figure 7.

As shown in Section 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.

5.4 Summary and Alternative Models

The `Simple`, `Simple.2xP`, and `Simple.4xP` models 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*.

6 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 [2]. 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, as well as other hardware factors [16], 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.

Nayfeh et al. considered design choices for a single-package multiprocessor [13], 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 [14]. There have also been a few studies of consistency models using ILP multiprocessors [8, 17, 23]. However, none of the above work details the benefits achieved by ILP in the multiprocessor.

Our variants of the `Simple` processor model in Section 5 are based on the works of Heinrich et al. [9] and Holt et al. [10]. 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 [19]) 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.

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 the presence of read clustering (e.g. [15]), coalescing (e.g. [4]), and contention (e.g. [3]).

7 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 direct-execution 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 lower read miss speedups 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.

The disparity in the impact of ILP on CPU time and 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 the reduced overlap 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 direct-execution 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 read miss ILP speedup. However, this model can show significant inaccuracy in cases of high or moderate read miss ILP speedup since it does not properly account for the effects of overlapping read misses.

Unfortunately, full ILP simulation will invariably take more simulation time than direct-execution simulation 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.

8 Acknowledgments

We would like to thank Vikram Adve, Shubu Mukherjee, Vivek Pai, and Ram Rajamony for valuable feedback on earlier drafts of this paper.

References

[1] V. S. Adve et al. An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs. In *Supercomputing '95*, 1995.

[2] D. H. Albonesi and I. Koren. An Analytical Model of High-Performance Superscalar-Based Multiprocessors. In *PACT'95*, 1995.

[3] D. Burger et al. Quantifying Memory Bandwidth Limitations of Current and Future Microprocessors. In *ISCA-23*, 1996.

[4] M. Butler and Y. Patt. The Effect of Real Data Cache Behavior on the Performance of a Microarchitecture that Supports Dynamic Scheduling. In *MICRO-24*, 1991.

[5] R. G. Covington et al. The Efficient Simulation of Parallel Computer Systems. *Intl. Journal in Computer Simulation*, 1, 1991.

[6] J. H. Edmondson et al. Internal Organization of the Alpha 21164, A 300-MHz 64-bit Quad-Issue CMOS RISC Microprocessor. *Digital Technical Journal*, 7(1), 1995.

[7] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA-17*, 1990.

[8] K. Gharachorloo et al. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *ISCA-19*, 1992.

[9] M. Heinrich et al. The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor. In *ASPLOS-VI*, 1994.

[10] C. Holt et al. Application and Architectural Bottlenecks in Large Scale Distributed Shared Memory Machines. In *ISCA-23*, 1996.

[11] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *ISCA-8*, 1981.

[12] MIPS Technologies, Inc. *R10000 Microprocessor User's Manual, Version 1.1*, January 1996.

[13] B. A. Nayfeh et al. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *ISCA-23*, May 1996.

[14] K. Olukotun et al. The Case for a Single-Chip Multiprocessor. In *ASPLOS-VII*, October 1996.

[15] K. Oner and M. Dubois. Effects of Memory Latencies on Non-Blocking Processor/Cache architectures. In *Supercomputing'93*, 1993.

[16] V. S. Pai et al. The Impact of Instruction-Level Parallelism on Multiprocessor Performance and Simulation Methodolgy. ECE TR-9606, July 1996, Revised December 1996.

[17] V. S. Pai et al. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *ASPLOS-VII*, October 1996.

[18] U. Rajagopalan. The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors. Master's thesis, Rice University, 1995.

[19] S. K. Reinhardt et al. Decoupled Hardware Support for Distributed Shared Memory. In *ISCA-23*, 1996.

[20] M. Rosenblum et al. The Impact of Architectural Trends on Operating System Performance. In *SOSP-15*, 1995.

[21] J. P. Singh et al. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1), 1992.

[22] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22*, 1995.

[23] R. N. Zucker and J.-L. Baer. A Performance Study of Memory Consistency Models. In *ISCA-19*, 1992.