The Impact of Exploiting Instruction-Level Parallelism on Shared-Memory Multiprocessors

Vijay S. Pai, Parthasarathy Ranganathan, Hazim Abdel-Shafi, and Sarita Adve
Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77251-1892
{*vijaypai*|*parthas*|*shafi*|*sarita*}@*rice.edu*

Abstract— **Current microprocessors incorporate techniques to aggressively exploit instruction-level parallelism (ILP). This paper evaluates the impact of such processors on the performance of shared-memory multiprocessors, both without and with the latency-hiding optimization of software prefetching.**

Our results show that while ILP techniques substantially reduce CPU time in multiprocessors, they are less effective in removing memory stall time. Consequently, despite the inherent latency tolerance features of ILP processors, we find memory system performance to be a larger bottleneck and parallel efficiencies to be generally poorer in ILP-based multiprocessors than in previous-generation multiprocessors. The main reasons for these deficiencies are insufficient opportunities in the applications to overlap multiple load misses and increased contention for resources in the system. We also find that software prefetching does not change the memory bound nature of most of our applications on our ILP multiprocessor, mainly due to a large number of late prefetches and resource contention.

Our results suggest the need for additional latency hiding or reducing techniques for ILP systems, such as software clustering of load misses and producer-initiated communication.[1]

I. Introduction

Shared-memory multiprocessors built from commodity microprocessors are being increasingly used to provide high performance for a variety of scientific and commercial applications. Current commodity microprocessors improve performance by using aggressive techniques to exploit high levels of instruction-level parallelism (ILP). These techniques include multiple instruction issue, out-of-order (dynamic) scheduling, non-blocking loads, and speculative execution. We refer to these techniques collectively as *ILP techniques* and to processors that exploit these techniques as *ILP processors.*

Most previous studies of shared-memory multiprocessors, however, have assumed a *simple processor* with single-issue, in-order scheduling, blocking loads, and no speculation. A few multiprocessor architecture studies model state-of-the-art ILP processors [2], [7], [8], [9], but do not analyze the impact of ILP techniques.

To fully exploit recent advances in uniprocessor technology for shared-memory multiprocessors, a detailed analysis of how ILP techniques affect the performance of such systems and how they interact with previous optimizations for such systems is required. This paper evaluates the impact of exploiting ILP on the performance of shared-memory multiprocessors, both without and with the latency-hiding optimization of software prefetching.

For our evaluations, we study five applications using detailed simulation, described in Section II.

Section III analyzes the impact of ILP techniques on the performance of shared-memory multiprocessors without the use of software prefetching. All our applications see performance improvements from the use of current ILP techniques, but the improvements vary widely. In particular, ILP techniques successfully and consistently reduce the CPU component of execution time, but their impact on the memory stall time is lower and more application-dependent. Consequently, despite the inherent latency tolerance features integrated within ILP processors, we find memory system performance to be a larger bottleneck and parallel efficiencies to be generally poorer in ILP-based multiprocessors than in previous-generation multiprocessors. These deficiencies are caused by insufficient opportunities in the application to overlap multiple load misses and increased contention for system resources from more frequent memory accesses.

Software-controlled non-binding prefetching has been shown to be an effective technique for hiding memory latency in simple processor-based shared memory systems [6]. Section IV analyzes the interaction between software prefetching and ILP techniques in shared-memory multiprocessors. We find that, compared to previous-generation systems, increased late prefetches and increased contention for resources cause software prefetching to be less effective in reducing memory stall time in ILP-based systems. Thus, even after adding software prefetching, most of our applications remain largely memory bound on the ILP-based system.

Overall, our results suggest that, compared to previous-

This work is supported in part by an IBM Partnership award, Intel Corporation, the National Science Foundation under Grant No. CCR-9410457, CCR-9502500, CDA-9502791, and CDA-9617383, and the Texas Advanced Technology Program under Grant No. 003604-025. Sarita Adve is also supported by an Alfred P. Sloan Research Fellowship, Vijay S. Pai by a Fannie and John Hertz Foundation Fellowship, and Parthasarathy Ranganathan by a Lodieska Stockbridge Vaughan Fellowship.

[1] This paper combines results from two previous conference papers [11], [12], using a common set of system parameters, a more aggressive MESI (versus MSI) cache-coherence protocol, a more aggressive compiler (the better of SPARC SC 4.2 and gcc 2.7.2 for each application, rather than gcc 2.5.8), and full simulation of private memory references.

generation shared-memory systems, ILP-based systems have a greater need for additional techniques to tolerate or reduce memory latency. Specific techniques motivated by our results include clustering of load misses in the applications to increase opportunities for load misses to overlap with each other, and techniques such as producer-initiated communication that reduce latency to make prefetching more effective (Section V).

II. Methodology

A. Simulated Architectures

To determine the impact of ILP techniques on multiprocessor performance, we compare two systems – `ILP` and `Simple` – equivalent in every respect except the processor used. The `ILP` system uses state-of-the-art ILP processors while the `Simple` system uses simple processors (Section II-A.1). We compare the `ILP` and `Simple` systems not to suggest any architectural tradeoffs, but rather, to understand how aggressive ILP techniques impact multiprocessor performance. Therefore, the two systems have identical clock rates, and include identical aggressive memory and network configurations suitable for the `ILP` system (Section II-A.2). Figure 1 summarizes all the system parameters.

Processor parameters	
Clock rate	300 MHz
Fetch/decode/retire rate	4 per cycle
Instruction window (reorder buffer) size	64
Memory queue size	32
Outstanding branches	8
Functional units	2 ALUs, 2 FPUs, 2 address generation units; all 1 cycle latency
Memory hierarchy and network parameters	
L1 cache	16 KB, direct-mapped, 2 ports, 8 MSHRs, 64-byte line
L2 cache	64 KB, 4-way associative, 1 port, 8 MSHRs, 64-byte line, pipelined
Memory	4-way interleaved, 60 ns access time, 16 bytes/cycle
Bus	100 MHz, 128 bits, split transaction
Network	2D mesh, 150MHz, 64 bits, per hop flit delay of 2 network cycles
Nodes in multiprocessor	8
Resulting contentionless latencies (in processor cycles)	
L1 hit	1 cycle
L2 hit	10 cycles
Local memory	45 cycles
Remote memory	140-220 cycles
Cache-to-cache transfer	170-270 cycles

Fig. 1. System parameters.

A.1 Processor models

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

A.2 Memory Hierarchy and Multiprocessor Configuration

We simulate a hardware cache-coherent, non-uniform memory access (CC-NUMA) shared-memory multiprocessor using an invalidation-based, four-state MESI directory coherence protocol [4]. We model release consistency because previous studies have shown that it achieves the best performance [9].

The processing nodes are connected using a two-dimensional mesh network. Each node includes a processor, two levels of caches, a portion of the global shared-memory and directory, and a network interface. A split-transaction bus connects the network interface, directory controller, and the rest of the system node. Both caches use a write-allocate, write-back policy. The cache sizes are chosen commensurate with the input sizes of our applications, following the methodology described by Woo et al. [14]. The primary working sets for our applications fit in the L1 cache, while the secondary working sets do not fit in the L2 cache. Both caches are non-blocking and use miss status holding registers (MSHRs) [3] to store information on outstanding misses and to coalesce multiple requests to the same cache line. All multiprocessor results reported in this paper use a configuration with 8 nodes.

B. Simulation Environment

We use RSIM, the Rice Simulator for ILP Multiprocessors, to model the systems studied [10]. RSIM is an execution-driven simulator that models the processor pipelines, memory system, and interconnection network in detail, including contention at all resources. It takes SPARC application executables as input. To speed up our simulations, we assume that all instructions hit in the instruction cache. This assumption is reasonable since all our applications have very small instruction footprints.

C. Performance Metrics

In addition to comparing execution times, we also report the individual components of execution time – CPU, data memory stall, and synchronization stall times – to characterize the performance bottlenecks in our systems. With ILP processors, it is unclear how to assign stall time to specific instructions since each instruction's execution may be overlapped with both preceding and following instructions. We use the following convention, similar to previous work (e.g., [5]), to account for stall cycles. At every cycle, we calculate the ratio of the instructions retired from the instruction window in that cycle to the maximum retire rate of the processor and attribute this fraction of the cycle to the busy time. The remaining fraction of the cycle is attributed as stall time to the first instruction that could not be retired that cycle. We group the busy time and functional unit (non-memory) stall time together as CPU time. Henceforth, we use the term *memory stall time* to denote the data memory stall component of execution time.

In the first part of the study, the key metric used to

Application	Input Size
LU, LUopt	256x256 matrix, block 8
FFT, FFTopt	65536 points
Mp3d	50000 particles
Water	512 molecules
Radix	1024 radix, 512K keys, max 512K

Fig. 2. Applications and input sizes.

evaluate the impact of ILP is the ratio of the execution time with the `Simple` system relative to that achieved by the `ILP` system, which we call the *ILP speedup*. For detailed analysis, we analogously define an ILP speedup for each component of execution time.

D. Applications

Figure 2 lists the applications and the input sets used in this study. Radix, LU, and FFT are from the SPLASH-2 suite [14], and Water and Mp3d are from the SPLASH suite [13]. These five applications and their input sizes were chosen to ensure reasonable simulation times. (Since RSIM models aggressive ILP processors in detail, it is about 10 times slower than simple-processor-based shared-memory simulators.) LUopt and FFTopt are versions of LU and FFT that include ILP-specific optimizations that can potentially be implemented in a compiler. Specifically, we use function inlining and loop interchange to move load misses closer to each other so that they can be overlapped in the `ILP` processor. The impact of these optimizations is discussed in Sections III and V. Both versions of LU are also modified slightly to use flags instead of barriers for better load balance.

Since a SPARC compiler for our `ILP` system does not exist, we compiled our applications with the commercial Sun SC 4.2 or the gcc 2.7.2 compiler (based on better simulated `ILP` system performance) with full optimization turned on. The compilers' deficiencies in addressing the specific instruction grouping rules of our `ILP` system are partly hidden by the out-of-order scheduling in the `ILP` processor.[2]

III. Impact of ILP Techniques on Performance

This section analyzes the impact of ILP techniques on multiprocessor performance by comparing the `Simple` and `ILP` systems, without software prefetching.

A. Overall Results

Figures 3 and 4 illustrate our key overall results. For each application, Figure 3 shows the total execution time and its three components for the `Simple` and `ILP` systems (normalized to the total time on the `Simple` system). Additionally, at the bottom, the figure also shows the ILP speedup for each application. Figure 4 shows the parallel efficiency[3] of the `ILP` and `Simple` systems expressed as a percentage. These figures show three key trends:

- ILP techniques improve the execution time of all our applications. However, the ILP speedup shows a wide variation (from 1.29 in Mp3d to 3.54 in LUopt). The average ILP speedup for the original applications (i.e., not including LUopt and FFTopt) is only 2.05.
- The memory stall component is generally a larger part of the overall execution time in the `ILP` system than in the `Simple` system.
- Parallel efficiency for the `ILP` system is less than that for the `Simple` system for all applications.

We next investigate the reasons for the above trends.

B. Factors Contributing to ILP Speedup

Figure 3 indicates that the most important components of execution time are CPU time and memory stalls. Thus, ILP speedup will be shaped primarily by CPU ILP speedup and memory ILP speedup. Figure 5 summarizes these speedups (along with the total ILP speedup). The figure shows that the low and variable ILP speedup for our applications can be attributed largely to insufficient and variable memory ILP speedup; the CPU ILP speedup is similar and significant among all applications (ranging from 2.94 to 3.80). More detailed data shows that for most of our applications, memory stall time is dominated by stalls due to loads that miss in the L1 cache. We therefore focus on the impact of ILP on (L1) load misses below.

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

- **Load miss overlap**. Since the `Simple` system has blocking loads, the entire load miss latency is exposed as stall time. In `ILP`, load misses can be overlapped with other useful work, reducing stall time and increasing the ILP load miss speedup. The number of instructions behind which a load miss can overlap is, however, limited by the instruction window size; further, load misses have longer latencies than other instructions in the instruction window. Therefore, load miss latency can normally be completely hidden only behind other load misses. Thus, for significant load miss ILP speedup, applications should have multiple load misses clustered together within the instruction window to enable these load misses to overlap with each other.
- **Contention**. Compared to the `Simple` system, the `ILP` system can see longer latencies from increased contention due to the higher frequency of misses, thereby negatively affecting load miss ILP speedup.
- **Change in the number of misses**. The `ILP` system may see fewer or more misses than the `Simple` system because of speculation or reordering of memory accesses, thereby positively or negatively affecting load miss ILP speedup.

All of our applications except LU see a similar number of cache misses in both the `Simple` and `ILP` case. LU sees 2.5X fewer misses in `ILP` because of a reordering of accesses that otherwise conflict. When the number of misses does not change, the `ILP` system sees (>1) load miss ILP speedup if the load miss overlap exploited by `ILP` outweighs any additional latency from contention. We illustrate the

[2] To the best of our knowledge, the key compiler optimization identified in this paper (clustering of load misses) is not implemented in any current superscalar compiler.

[3] The parallel efficiency for an application on a system with N processors is defined as $\frac{\mathit{Execution\ time\ on\ uniprocessor}}{\mathit{Execution\ time\ on\ multiprocessor}} \times \frac{1}{N}$.

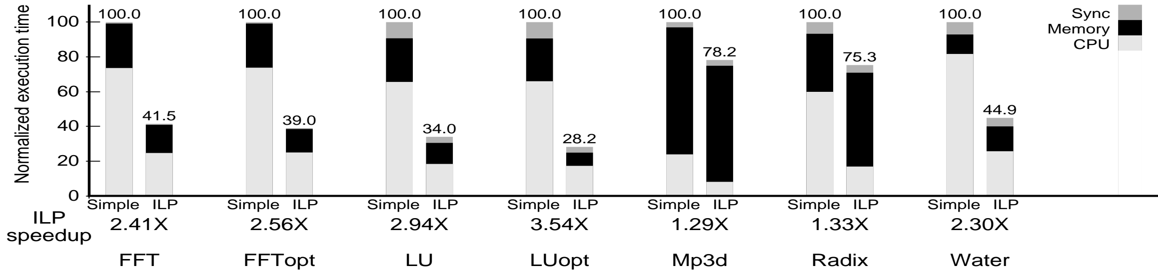


Fig. 3. Impact of ILP on multiprocessor performance.

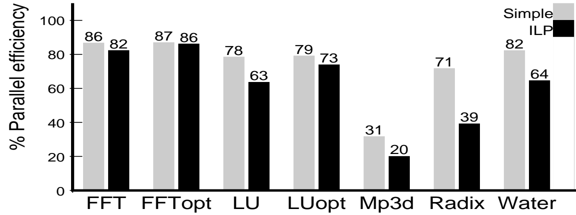


Fig. 4. Impact of ILP on parallel efficiency.

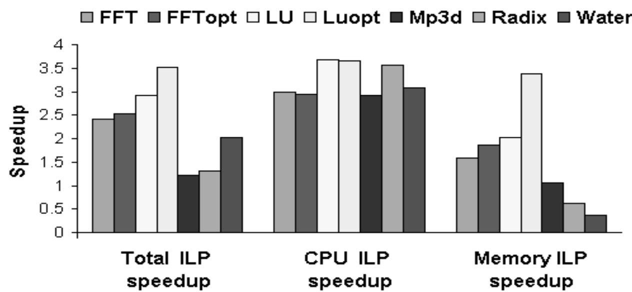


Fig. 5. ILP speedup for total execution time, CPU time, and memory stall time in the multiprocessor system.

effects of load miss overlap and contention using the two applications that best characterize them: LUopt and Radix.

Figure 6(a) provides the average load miss latencies for LUopt and Radix in the `Simple` and `ILP` systems, normalized to the `Simple` system latency. The latency shown is the total miss latency, measured from address generation to data arrival, including the overlapped part (in `ILP`) and the exposed part that contributes to stall time. The difference in the bar lengths of `Simple` and `ILP` indicates the additional latency added due to contention in `ILP`. Both of these applications see a significant latency increase from resource contention in `ILP`. However, LUopt can overlap all its additional latency, as well as a large portion of the base (`Simple`) latency, thus leading to a high memory ILP speedup. On the other hand, Radix cannot overlap its additional latency; thus, it sees a load miss *slowdown* in the `ILP` configuration.

We use the data in Figures 6(b) and (c) to further investigate the causes for the load miss overlap and contention-related latencies in these applications.

Causes for load miss overlap. Figure 6(b) shows the `ILP` system's L1 MSHR occupancy due to load misses for LUopt and Radix. Each curve shows the fraction of total time for which at least N MSHRs are occupied by load misses, for each possible N (on the X axis). This figure shows that LUopt achieves significant overlap of load misses, with up to 8 load miss requests outstanding simultaneously at various times. In contrast, Radix almost never has more than 1 outstanding load miss at any time. This difference arises because load misses are clustered together in the instruction window in LUopt, but typically separated by too many instructions in Radix.

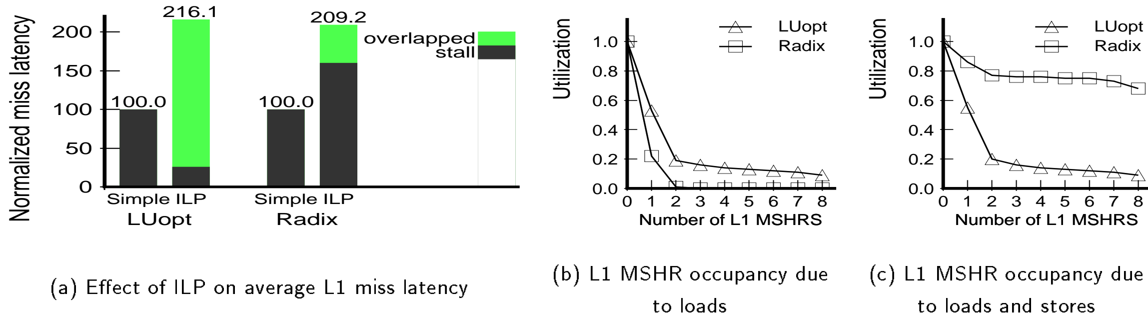
Causes for contention. Figure 6(c) extends the data of Figure 6(b) by displaying the total MSHR occupancy for both load and store misses. The figure indicates that Radix has a large amount of store miss overlap. This overlap does not contribute to an increase in memory ILP speedup since store latencies are already hidden in both the `Simple` and `ILP` systems due to release consistency. The store miss overlap, however, increases contention in the memory hierarchy, resulting in the ILP memory slowdown in Radix. In LUopt, the contention-related latency comes primarily from load misses, but its effect is mitigated since overlapped load misses contribute to reducing memory stall time.

C. Memory stall component and parallel efficiency

Using the above analysis, we can see why the `ILP` system generally sees a larger relative memory stall time component (Figure 3) and a generally poorer parallel efficiency (Figure 4) than the `Simple` system.

Since memory ILP speedup is generally less than CPU ILP speedup, the memory component becomes a greater fraction of total execution time in the `ILP` system than in the `Simple` system. To understand the reduced parallel efficiency, Figure 7 provides the ILP speedups for the uniprocessor configuration for reference. The uniprocessor also generally sees lower memory ILP speedups than CPU ILP speedups. However, the impact of the lower memory ILP speedup is higher in the multiprocessor because the longer latencies of remote misses and increased contention result in a larger relative memory component in the execution time (relative to the uniprocessor). Additionally, the dichotomy between local and remote miss latencies in a multiprocessor often tends to decrease memory ILP speedup (relative to the uniprocessor), because load misses must be overlapped not only with other load misses but with load misses with similar latencies[4]. Thus, overall, the multiprocessor system is less able to exploit ILP features than the corresponding uniprocessor system for most applications.

[4] FFT and FFTopt see better memory ILP speedups in the multiprocessor than in the uniprocessor because they overlap multiple load misses with similar multiprocessor (remote) latencies. The section of the code that exhibits overlap has a greater impact in the multiprocessor because of the longer remote latencies incurred in this section.



(a) Effect of ILP on average L1 miss latency

(b) L1 MSHR occupancy due to loads

(c) L1 MSHR occupancy due to loads and stores

Fig. 6. Load miss overlap and contention in the ILP system.

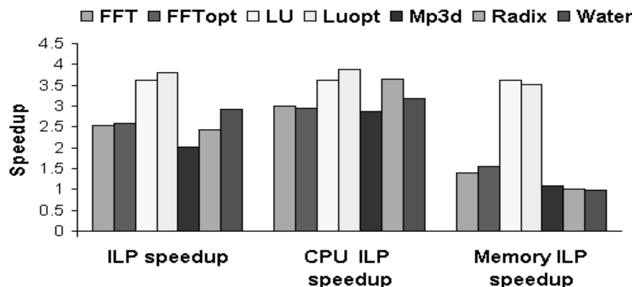


Fig. 7. ILP speedup for total execution time, CPU time, and memory stall time in the uniprocessor system.

Consequently, the `ILP` multiprocessor generally sees lower parallel efficiency than the `Simple` multiprocessor.

IV. Interaction of ILP Techniques with Software Prefetching

The previous section shows that the `ILP` system sees a greater bottleneck from memory latency than the `Simple` system. Software-controlled non-binding prefetching has been shown to effectively hide memory latency in shared-memory multiprocessors with simple processors. This section evaluates how software prefetching interacts with ILP techniques in shared-memory multiprocessors. We followed the software prefetch algorithm developed by Mowry et al.[6] to insert prefetches in our applications by hand, with one exception. The algorithm in [6] assumes that locality is not maintained across synchronization, and so does not schedule prefetches across synchronization accesses. We removed this restriction when beneficial. For a consistent comparison, the experiments reported are with prefetches scheduled identically for both `Simple` and `ILP`; the prefetches are scheduled at least 200 dynamic instructions before their corresponding demand accesses. The impact of this scheduling decision is discussed below, including the impact of varying this prefetch distance.

A. Overall Results

Figure 8 graphically presents the key results from our experiments (FFT and FFTopt have similar performance, so only FFTopt appears in the figure). The figure shows the execution time (and its components) for each application on `Simple` and `ILP`, both without and with software prefetching (`+PF` indicates the addition of software prefetching). Execution times are normalized to the time for the application on `Simple` without prefetching. Figure 9 summarizes some key data.

Software prefetching achieves significant reductions in execution time on `ILP` (13% to 43%) for three cases (LU, Mp3d, and Water). These reductions are similar to or greater than those in `Simple` for these applications. However, software prefetching is less effective at reducing memory stalls on `ILP` than on `Simple` (average reduction of 32% in `ILP`, ranging from 7% to 72%, vs. average 59% and range of 21% to 88% in `Simple`). The net effect is that even after prefetching is applied to `ILP`, the average memory stall time is 39% on `ILP` with a range of 11% to 65% (vs. average of 16% and range of 1% to 29% for `Simple`). For most applications, the `ILP` system remains largely memory-bound even with software prefetching.

B. Factors Contributing to the Effectiveness of Software Prefetching

We next identify three factors that make software prefetching less successful in reducing memory stall time in `ILP` than in `Simple`, two factors that allow `ILP` additional benefits in memory stall reduction not available in `Simple`, and one factor that can either help or hurt `ILP`. We focus on issues that are specific to `ILP` systems; previous work has discussed non-ILP specific issues [6]. Figure 10 summarizes the effects that were exhibited by the applications we studied. Of the negative effects, the first two are the most important for our applications.

Increased late prefetches. The last column of Figure 9 shows that the number of prefetches that are too late to completely hide the miss latency increases in all our applications when moving from `Simple` to `ILP`. One reason for this increase is that multiple-issue and out-of-order scheduling speed up computation in `ILP`, decreasing the computation time with which each prefetch is overlapped. `Simple` also stalls on any load misses that are not prefetched or that incur a late prefetch, thereby allowing other outstanding prefetched data to arrive at the cache. `ILP` does not provide similar leeway.

Increased resource contention. As shown in Section III, `ILP` processors stress system resources more than `Simple`. Prefetches further increase demand for resources, resulting in more contention and greater memory latencies. The resources most stressed in our configuration were cache ports, MSHRs, ALUs, and address generation units.

Negative interaction with clustered misses. Optimizations to cluster load misses for the `ILP` system, as in LUopt, can potentially reduce the effectiveness of software prefetching. For example, the addition of prefetching re-

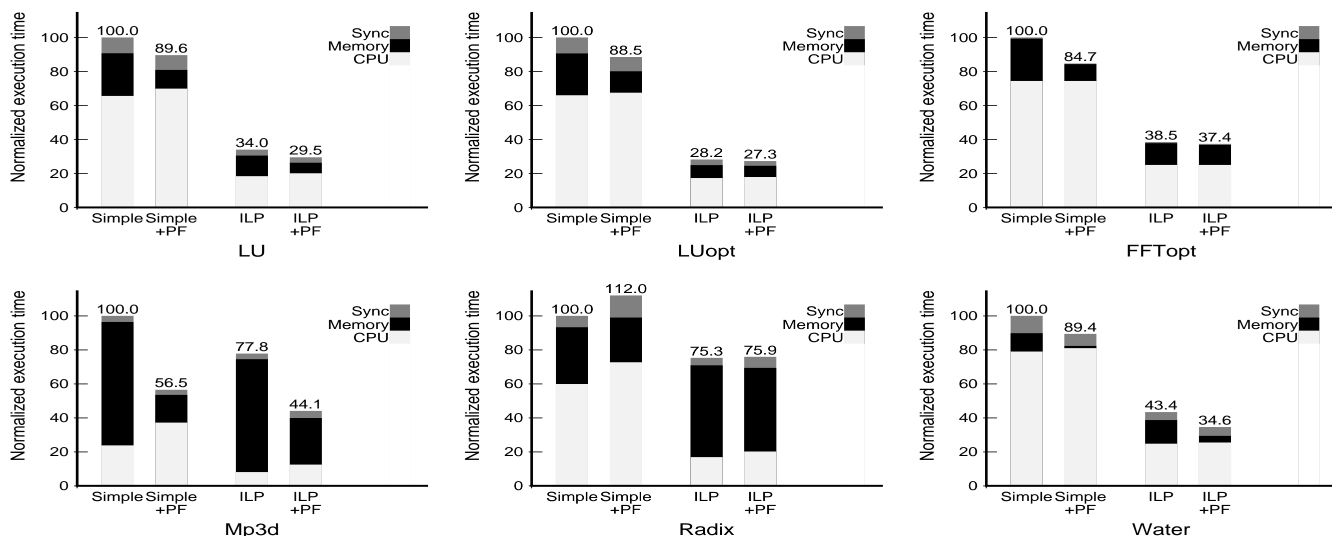


Fig. 8. Interaction between software prefetching and ILP.

duces the execution time of LU by 13% on the `ILP` system; in contrast, LUopt improves by only 3%. (On the `Simple` system, both LU and LUopt improve by about 10% with prefetching.) LUopt with prefetching is slightly better than LU with prefetching on `ILP` (by 3%). The clustering optimization used in LUopt reduces the computation between successive misses, contributing to a high number of late prefetches and increased contention with prefetching.

Overlapped accesses. In `ILP`, accesses that are difficult to prefetch may be overlapped because of non-blocking loads and out-of-order scheduling. Prefetched lines in LU and LUopt often suffer from L1 cache conflicts, resulting in these lines being replaced to the L2 cache before being used by the demand accesses. This L2 cache latency results in stall time in `Simple`, but can be overlapped by the processor in `ILP`. Since prefetching in `ILP` only needs to target those accesses that are not already overlapped by `ILP`, it can appear more effective in `ILP` than in `Simple`.

Fewer early prefetches. Early prefetches are those where the prefetched lines are either invalidated or replaced before their corresponding demand accesses. Early prefetches can hinder demand accesses by invalidating or replacing needed data from the same or other caches without providing any benefits in latency reduction. In many of our applications, the number of early prefetches drops in `ILP`, improving the effectiveness of prefetching for these applications. This reduction occurs because the `ILP` system allows less time between a prefetch and its subsequent demand access, decreasing the likelihood of an intervening invalidation or replacement.

Speculative prefetches. In `ILP`, prefetch instructions can be speculatively issued past a mispredicted branch. Speculative prefetches can potentially hurt performance by bringing unnecessary lines into the cache, or by bringing needed lines into the cache too early. Speculative prefetches can also help performance by initiating a prefetch for a needed line early enough to hide its latency. In our applications, most prefetches issued past mispredicted branches were to lines also accessed on the correct path.

App.	% reduction in execution time		% reduction in memory stall time		% remaining memory stall time		% prefetches that are late	
	Simple	ILP	Simple	ILP	Simple	ILP	Simple	ILP
LU	10	13	57	49	12	21	4	40
LUopt	11	3	49	14	14	24	12	35
FFTopt	15	3	60	7	12	32	13	29
Mp3d	43	43	78	59	29	62	1	12
Radix	-12	-1	21	9	23	65	0	2
Water	11	20	88	72	1	11	0	8
Average	14	14	59	32	16	39	5	17

Fig. 9. Detailed data on effectiveness of software prefetching. For the average, from LU and LUopt, only LUopt is considered since it provides better performance than LU with prefetching and `ILP`.

Factor	LU	LU opt	FFT opt	Mp3d	Water	Radix
Late prefetches	√	√	√	√	√	√
Resource contention		√	√		√	√
Clustered load misses		√	√			
Overlapped accesses	√	√				
Early prefetches	√	√		√	√	√
Speculative prefetches			√	√		

Fig. 10. Factors affecting the performance of prefetching for ILP.

C. Impact of Software Prefetching on Execution Time

Despite its reduced effectiveness in addressing memory stall time, software prefetching achieves significant execution time reductions with `ILP` in three cases (LU, Mp3d, and Water) for two main reasons. First, memory stall time contributes a larger portion of total execution time in `ILP`. Thus, even a reduction of a small fraction of memory stall time can imply a reduction in overall execution time similar to or greater than that seen in `Simple`. Second, `ILP` systems see less instruction overhead from prefetching compared to `Simple` systems, because ILP techniques allow the overlap of these instructions with other computation.

D. Alleviating Late Prefetches and Contention

Our results show that late prefetches and resource contention are the two key limitations to the effectiveness of prefetching on `ILP`. We tried several straightforward modifications to the prefetching algorithm and the system to address these limitations [12]. Specifically, we doubled and quadrupled the prefetch distance (i.e., the distance between a prefetch and the corresponding demand access), and increased the number of MSHRs. However, these modifications traded off benefits among late prefetches, early prefetches, and contention, without improving the combination of these factors enough to improve overall performance. We also tried varying the prefetch distance for each access according to the expected latency of that access (versus a common distance for all accesses), and prefetching only to the L2 cache. These modifications achieved their purpose, but did not provide a significant performance benefit for our applications [12].

V. Discussion

Our results show that shared-memory systems are limited in their effectiveness in exploiting ILP processors due to limited benefits of ILP techniques for the memory system. The analysis of Section III implies that the key reasons for the limited benefits are the lack of opportunity for overlapping load misses and/or increased contention in the system. Compiler optimizations akin to the loop interchanges used to generate LUopt and FFTopt may be able to expose more potential for load miss overlap in an application. The simple loop interchange used in LUopt provides a 13% reduction in execution time compared to LU on an `ILP` multiprocessor. Hardware enhancements can also increase load miss overlap; e.g., through a larger instruction window. Targeting contention requires increased hardware resources, or other latency reduction techniques.

The results of Section IV show that while software prefetching improves memory system performance with ILP processors, it does not change the memory-bound nature of these systems for most of the applications because the latencies are too long to hide with prefetching and/or because of increased contention. Our results motivate prefetching algorithms that are sensitive to increases in resource usage. They also motivate latency-reducing (rather than tolerating) techniques such as producer-initiated communication, which can improve the effectiveness of prefetching [1].

VI. Conclusions

This paper evaluates the impact of ILP techniques supported by state-of-the-art processors on the performance of shared-memory multiprocessors. All our applications see performance improvements from current ILP techniques. However, while ILP techniques effectively address the CPU component of execution time, they are less successful in improving data memory stall time. These applications do not see the full benefit of the latency-tolerating features of ILP processors because of insufficient opportunities to overlap multiple load misses and increased contention for system resources from more frequent memory accesses. Thus, ILP-based multiprocessors see a larger bottleneck from memory system performance and generally poorer parallel efficiencies than previous-generation multiprocessors.

Software-controlled non-binding prefetching is a latency hiding technique widely recommended for previous-generation shared-memory multiprocessors. We find that while software prefetching results in substantial reductions in execution time for some cases on the `ILP` system, increased late prefetches and increased contention for resources cause software prefetching to be less effective in reducing memory stall time in ILP-based systems. Even after the addition of software prefetching, most of our applications remain largely memory bound.

Thus, despite the latency-tolerating techniques integrated within ILP processors, multiprocessors built from ILP processors have a greater need for additional techniques to hide or reduce memory latency than previous-generation multiprocessors. One ILP-specific technique discussed in this paper is the software clustering of load misses. Additionally, latency-reducing techniques such as producer-initiated communication that can improve the effectiveness of prefetching appear promising.

References

[1] H. Abdel-Shafi et al. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proc. 3rd Intl. Symp. on High-Performance Computer Architecture*, pages 204–215, 1997.

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

[3] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proc. 8th Intl Symp. on Computer Architecture*, pages 81–87, 1981.

[4] J. Laudon and D. Lenoski. The SGI Origin 2000: A ccNUMA Highly Scalable Server. In *Proc. 24th Intl Symp. on Computer Architecture*, pages 241–251, 1997.

[5] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. 7th Intl Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 222–234, 1996.

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

[7] B. A. Nayfeh et al. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. In *Proc. 23rd Intl Symp. on Computer Architecture*, pages 67–77, 1996.

[8] K. Olukotun et al. The Case for a Single-Chip Multiprocessor. In *Proc. 7th Intl Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[9] V. S. Pai et al. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. 7th Intl Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 72–83, 1996.

[10] V. S. Pai et al. *RSIM Reference Manual, Version 1.0*. ECE TR 9705, Rice University, 1997.

[11] V. S. Pai et al. The Impact of Instruction Level Parallelism on Multiprocessor Performance and Simulation Methodology. In *Proc. 3rd Intl Symp. on High Performance Computer Architecture*, pages 72–83, 1997.

[12] P. Ranganathan et al. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In *Proc. 24th Intl Symp. on Computer Architecture*, pages 144–156, 1997.

[13] J. P. Singh et al. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, pages 5–44, Mar 1992.

[14] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Intl. Symp. on Computer Architecture*, pages 24–36, 1995.