To appear in Proceedings of ISCA-24 (June, 1997)

The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems *

Parthasarathy Ranganathan, Vijay S. Pai, Hazim Abdel-Shafi, Sarita V. Adve

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77005
{parthas|vijaypai|shafi|sarita}@rice.edu

Abstract

Current microprocessors aggressively exploit instruction-level parallelism (ILP) through techniques such as multiple issue, dynamic scheduling, and non-blocking reads. Recent work has shown that memory latency remains a significant performance bottleneck for shared-memory multiprocessor systems built of such processors.

This paper provides the first study of the effectiveness of software-controlled non-binding prefetching in shared-memory multiprocessors built of state-of-the-art ILP-based processors. We find that software prefetching results in significant reductions in execution time (12% to 31%) for three out of five applications on an ILP system. However, compared to previous-generation systems, software prefetching is significantly less effective in reducing the memory stall component of execution time on an ILP system. Consequently, even after adding software prefetching, memory stall time accounts for over 30% of the total execution time in four out of five applications on our ILP system.

This paper also investigates the interaction of software prefetching with memory consistency models on ILP-based multiprocessors. In particular, we seek to determine whether software prefetching can equalize the performance of sequential consistency (SC) and release consistency (RC). We find that even with software prefetching, for three out of five applications, RC provides a significant reduction in execution time (15% to 40%) compared to SC.

1 Introduction

Shared-memory multiprocessors are increasingly built of commodity microprocessors that exploit high levels of instruction-level parallelism (ILP) with techniques such as multiple issue, dynamic scheduling, and non-blocking reads. Such ILP features have the potential for greatly improving system performance. However, recent work has shown that while these features significantly improve the performance of computation, memory system performance remains a key bottleneck in multiprocessors [27].

To reduce memory stall time, many current processors support software-controlled non-binding prefetching. With this technique, the compiler or programmer schedules an explicit prefetch instruction for a location that will be accessed by the processor at a later time, with the goal of bringing the location into the processor's cache before it issues a demand memory access [4]. Previous studies have shown that software-controlled non-binding prefetching can eliminate a large fraction of memory stall time in shared-memory multiprocessors [24, 33]. However, all such studies used previous-generation processors with single-issue, static scheduling, and blocking reads. Consequently, such studies do not account for the interactions between software prefetching and the other latency-tolerating techniques already incorporated in ILP-based multiprocessors. An analysis of these interactions is required to assess the effectiveness of current software prefetching strategies for state-of-the-art shared-memory multiprocessors.

Memory system performance also depends on the consistency model of the system. Relaxed memory consistency models such as release consistency (RC) can potentially tolerate more memory latency than the simple model of sequential consistency (SC), but mandate a more complex programming model. Studies on previous-generation multiprocessors have shown that software prefetching can improve the performance of both SC and RC [14]. Systems with ILP processors, however, can incorporate hardware optimizations such as speculative reads and hardware prefetching from the instruction window to improve the performance of SC [10]. A recent study has shown that such techniques substantially narrow the performance gap between SC and RC, but a significant gap remains for some applications [28]. Since software prefetching targets the same latencies as RC, it is important to determine if software prefetching can eliminate the remaining gap between SC and RC on current multiprocessors, allowing the high performance of RC with the simple programming model of SC.

This paper is the first study of software-controlled non-binding prefetching in shared-memory multiprocessor systems using state-of-the-art ILP processors, and has two goals:

(1) To understand how software prefetching interacts with ILP in multiprocessors, and to identify its limitations.

(2) To understand if software prefetching can equalize the performance of SC and RC.

*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.

To achieve these goals, we study the impact of software prefetching on five applications from the SPLASH and SPLASH-2 suites. We insert prefetches in the applications by hand, following the currently best known compiler prefetching algorithm [23]. We run the applications on a detailed execution-driven simulator modeling shared-memory multiprocessors with state-of-the-art ILP processors.

For our first goal, we compare the performance benefits of software prefetching on two multiprocessor systems referred to as `Simple` and `ILP`. These systems are identical in every respect except that `Simple` uses previous-generation processors while `ILP` uses state-of-the-art ILP processors. For this part, we assume RC for both systems since RC has been shown to have the best performance (compared to SC or processor consistency) for these systems [9, 11, 28].

We find that for three out of our five applications, current software prefetching methods achieve a significant reduction in total execution time (12% to 31%) on `ILP`. However, even after applying software prefetching, data memory stall time constitutes a large fraction of the total execution time on `ILP`. Specifically, four out of five applications spend more than 30% of their time in data memory stalls. In contrast, on `Simple`, four out of five applications see less than 17% data memory stall time after applying software prefetching. Overall, we find current software prefetching methods to be significantly less effective in addressing memory stall time in `ILP` than in `Simple`, leaving most of our applications largely memory bound on `ILP`.

Compared to `Simple`, the two primary factors that contribute to the reduced effectiveness of prefetching on `ILP` are an increase in the number of late prefetches and an increase in contention for various resources. These factors occur because `ILP` speeds up computation and thus provides less work with which to overlap prefetch latency. Further, the increased frequency of misses in `ILP` also leads to increased resource contention. This in turn increases the latency of individual misses and further increases the number of late prefetches. Several straightforward modifications to the prefetching scheme to reduce late prefetches and resource contention did not show any appreciable benefits.

For our second goal, we compare the performance benefits of prefetching on ILP multiprocessors that implement SC and RC. For SC, we consider a straightforward implementation as well as an optimized implementation incorporating write buffering, speculative reads, and hardware prefetching from the instruction window [9, 10, 28]. We find that even with software prefetching, for three applications, RC provides significant reductions in execution time (15% to 40%) compared to the most optimized version of SC. The effectiveness of software prefetching in SC implementations is limited for reasons similar to the RC system. These limitations, however, have a larger impact on SC since SC exposes write latencies, which are hidden by RC. Thus, we find that software prefetching is unable to close the performance gap between SC and RC for our applications.

The rest of the paper is organized as follows. Section 2 gives background information on the software prefetching algorithm and the memory consistency implementations evaluated in this study. Section 3 presents our simulated architectures, performance metrics, simulation environment, and applications. Section 4 discusses our results on the interaction of software prefetching with ILP on RC. Section 5 re-examines various assumptions of our prefetching scheme and explores several modifications. Section 6 describes our results on the interaction of software prefetching with consistency models. Section 7 discusses related work. Section 8 concludes the paper.

2 Background

Sections 2.1 and 2.2 respectively describe the base software prefetching algorithm and memory consistency implementations we use in this study.

2.1 Software Prefetching Algorithm

The best known software prefetch insertion algorithm implemented in a compiler is by Mowry et al. [23, 24]. We describe this algorithm below. Section 3.4 describes how we use it to insert prefetches in our applications.

The algorithm is loop-based, and consists of an analysis phase and a scheduling phase. The *analysis phase* identifies the accesses that do not exhibit cache locality, and for which prefetches need to be inserted. When determining locality, this phase assumes a cache size reduced by an appropriate fraction to account for conflict misses. The analysis conservatively assumes that no locality is maintained across synchronization.

The *scheduling phase* follows the analysis phase and uses loop peeling, unrolling, and strip-mining to insert prefetches only for the accesses that are expected to cause cache misses. Accesses to the same cache line are grouped into equivalence classes, and an exclusive prefetch is used to obtain ownership along with the data for the line if at least one member of the equivalence class is a write. The inner-most loop corresponding to an access is software pipelined to schedule a prefetch a certain number of iterations ahead of the demand access. The number of iterations is computed by the formula $\lceil \frac{L}{W} \rceil$, where L is the expected miss latency in cycles and W is an estimate of the shortest possible path through an iteration in cycles. This number is referred to as the *prefetch distance*, and is expected to represent the number of iterations needed to completely overlap the latency of a prefetch. In this paper, we also use the term prefetch distance to refer to the number of instructions needed to overlap the prefetch latency.

The above core algorithm only handles affine accesses. It is further extended to handle indirect accesses when the indirection is through an affine reference. Two prefetches are inserted for each indirect access. The first prefetch is used to fetch the address of the indirect access, and the second uses the address to prefetch the indirect access. Since the second prefetch uses the value of the first, the first prefetch must be scheduled before the second according to the prefetch distance.

2.2 Memory Consistency Models

The memory consistency model of a system determines the extent to which the system can appear to overlap or reorder memory operations to hide memory latency.

Sequential consistency (SC) [18] guarantees that all memory operations appear to execute in program order and hence offers a simple and intuitive programming model. We examine two implementations of sequential consistency – SCplain and SCopt. SCplain is a naive implementation that enforces memory ordering by stalling the issue of a memory operation until the previous memory operation of that processor has completed.

SCopt is a more aggressive implementation that improves performance through three hardware techniques, hardware prefetching of writes from the instruction window [10, 28],

ILP Processor	
Processor speed	300MHz
Maximum fetch/decode/retire rate (instructions per cycle)	4
Instruction window	64 entries
Functional units	2 integer arithmetic 2 floating point 2 address generation
Simultaneous speculated branches	8
Maximum instructions in memory queue	32
Network parameters	
Network speed	150MHz
Network width	64 bits
Flit delay (per hop)	2 network cycles
Bus type	Split transaction
Bus width and speed	128 bits at 100 MHz
Cache parameters	
Cache line size	64 bytes
L1 cache (on-chip)	Direct mapped, 16 KB
L1 request ports	2
L1 hit time	1 cycle
Number of L1 MSHRs	8
L2 cache (off-chip)	4-way associative, 64 KB
L2 request ports	1
L2 hit time	8 cycles, pipelined
Number of L2 MSHRs	8
Memory parameters	
Memory access time	18 cycles (60 ns)
Memory transfer bandwidth	16 bytes/cycle
Memory interleaving	4-way

Figure 1: Default system parameters

speculative reads [10, 28], and write buffering [9]. The first two techniques, hardware prefetching and speculative reads, exploit the instruction-lookahead window and speculation support available in ILP processors. The hardware prefetching technique issues a non-binding prefetch for a decoded memory instruction in the instruction issue window once the address of the instruction is computed. The effectiveness of this technique is primarily constrained by the size of the instruction window and the time for the address to be computed. Speculative reads extend the benefits of hardware prefetching by speculatively using the values of reads brought into the cache, even while previous demand accesses are incomplete. If a possible violation of memory ordering is detected due to early use of such data, the system rolls back the speculative load and all subsequent instructions. The third optimization in SCopt, write buffering, allows writes to retire from the instruction window as soon as they reach the head of the window. Each write is buffered in the processor's memory queue, and is issued to the cache only after the previous write (by program order) has completed. Write buffering allows multiple writes to be retired before issue. A read following such writes, however, must wait for these writes to complete before the read can retire from the instruction window.

The release consistency model (RC) [12] allows more overlap and reordering of memory operations than SC, albeit at the cost of greater programming complexity. RC distinguishes between data and synchronization operations, allowing data operations to be reordered with respect to one another. We only study a straightforward implementation of RC. We do not consider a corresponding RCopt implementation, as previous work has shown that hardware prefetching and speculative reads do not significantly affect the performance of RC for our applications [28], and RC already implements a superset of the write buffering optimization. Writes in RC retire from the instruction window as soon as they reach the head, maintaining a memory queue entry until issue to the cache; unlike SCplain and SCopt, RC allows multiple writes to issue in parallel.

3 Methodology

This section describes the experimental methodology used in this paper. Sections 3.1, 3.2, and 3.3 respectively describe the architectures modeled, our performance metrics, and our simulation environment. Section 3.4 describes the applications used in this study and the prefetches inserted in these applications.

3.1 Simulated Architectures

The first part of this study compares two multiprocessor systems – `ILP` and `Simple`. These systems are equivalent in all respects except for the processor modeled. `ILP` uses state-of-the-art high-performance processors while `Simple` uses previous generation processors. We compare the `ILP` and `Simple` systems not to propose any architectural tradeoff, but rather, to understand how software prefetching interacts with ILP techniques. Therefore, the two systems have identical clock rates, and include identical aggressive memory and network systems suitable for `ILP`.

3.1.1 Processor Models

The `ILP` system uses state-of-the-art processors employing ILP features such as multiple-issue, out-of-order execution, speculative execution, non-blocking reads, and register renaming. Our implementation of the processor core resembles the MIPS R10000 microarchitecture [22], but also includes aggressive features from other architectures. Default parameters for the processor are listed in Figure 1. The `Simple` system uses previous-generation statically-scheduled, single-issue processors with blocking reads.

`Simple` processors rely on compilers to schedule instructions to hide functional unit latencies. In `ILP` processors, the hardware instruction window can serve the same purpose. Since we do not have access to a compiler that schedules instructions according to our `Simple` architecture, we use single-cycle functional unit latencies in both models for a fair comparison of the two systems. To study the impact of this assumption on `ILP` performance, we ran our simulations with realistic functional unit latencies. We found that, except on one application (Water), there was little impact on `ILP` execution time because `ILP` processors successfully overlapped functional unit latencies. Our key findings continue to hold even in Water.

Both processor models include support for software-controlled non-binding prefetching, with both exclusive-mode and shared-mode prefetches. Prefetch instructions retire as soon as they reach the top of the instruction window, but occupy a slot in the processor's memory queue until they are issued. Prefetches are not dropped even if resource constraints block their issue, and prefetched lines are brought into the highest level of the memory hierarchy. These two assumptions follow previous work [23], and are re-evaluated in Section 5.

Both processor models support RC, using the SPARC V9 `MEMBAR` fence instructions to impose ordering at synchronization points [32]. The `ILP` processor additionally supports the two implementations of sequential consistency described in Section 2.2 – SCplain and SCopt. As with software prefetching, we do not drop hardware prefetches in SCopt even if their issue is blocked due to resource constraints. The SCopt processor includes a speculative load buffer to monitor outstanding speculative reads [10] and employs a mechanism similar to that used in the MIPS R10000 [22] to recover when possible consistency violations are detected.

3.1.2 Memory Hierarchy and Multiprocessor Configuration

We simulate a hardware cache-coherent, non-uniform memory access (CC-NUMA) shared-memory multiprocessor using an invalidation-based, three-state directory coherence protocol. Each node in our simulated system includes a processor, two levels of caches, and a portion of the global shared-memory and directory. A split-transaction bus connects the network interface, directory controller, and the rest of the system node. The nodes are connected using a two-dimensional mesh network. Figure 1 summarizes the memory system parameters.

Both caches are non-blocking with 8 miss status holding registers (MSHRs) [17] each. The MSHRs store information about the misses and coalesce multiple requests to the same cache line. In the event of a write request being received for the same line as a pending read, the MSHR blocks the write request and issues an ownership request only after the read returns (we refer to such stalls as write-after-read stalls). This implementation avoids complex races at the MSHRs and directory that can arise from reordering in the network, and resembles several current systems.

The L1 cache is dual-ported, allowing two accesses to be handled simultaneously, and uses a write-allocate write-back policy. We assume a write-back policy for the L1 cache since previous work has shown that SC systems perform best with write-back caches and because several recent systems support write-back L1 caches. The L2 cache is a fully pipelined, write-allocate write-back cache. The L1 cache size is 16 KB and the L2 cache size is 64 KB. These sizes are chosen based on the input sizes of our applications (described in Section 3.4), following the methodology described by Woo et al. [34]. The primary working sets for our applications fit in the L1 cache, while the secondary working sets do not fit in the L2 cache.

The number of processors in our system varies by application and is either 8 or 16 depending on the scalability of the application, as summarized in Figure 3.

3.2 Performance Metrics

In addition to reporting execution times, we consider various components of execution time to enable identification of the performance bottlenecks in our systems. We divide the execution time into three components – CPU, data memory, and synchronization. We use the following convention to account for stall cycles [27, 28, 30]. All cycles where the processor retires the maximum number of instructions allowed by the architecture are considered busy time. Otherwise, we charge the cycle to the stall time component of the first instruction that could not be retired that cycle. We group together the busy time and functional unit stall time as CPU time. We subdivide the data memory stall time into the time spent on L1 hits, L2 hits, local memory accesses, and remote memory accesses, for both reads and writes. Henceforth, we use the term *memory stall time* to denote the data memory stall component of execution time.

We divide prefetches into various categories, as summarized in Figure 2. These categories are *useful*, in which a prefetched line arrives on time and is used by a demand access; *late*, in which a prefetched line arrives after the demand access; *early*, in which the prefetched line is replaced or invalidated before use, or is never used; and *unnecessary*, in which the line being prefetched is already present in either the cache or the MSHRs. We describe prefetches that result in the replacement of a line needed by a demand access as *damaging*; all prefetch types other than unnecessary prefetches can also be damaging.

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

Figure 2: Classification of prefetches

Application	Input Size	Processors
LU	256 by 256 matrix, block 8	8
FFT	65536 points	16
MP3D	50000 particles	8
Water	512 molecules	16
Radix	1024 radix, 512K keys, max 512K	8

Figure 3: Applications, input sizes, and system sizes

3.3 Simulation Environment

We use the Rice Simulator for ILP Multiprocessors (RSIM) [26] to model the `Simple` and `ILP` systems described in the previous sections. RSIM models the processors, memory system, and interconnection network in detail, including contention at all resources. Specifically, unlike current direct-execution simulators, we accurately model the details of the processor pipelines. Our simulator is execution-driven (as opposed to trace-driven) and hence allows the interactions between processors during the simulation to affect the course of the simulation. To speed up our simulations, we assume that all instructions hit in the instruction cache and that all private variables hit in the data cache. However, we do model contention due to private data accesses at various processor and cache resources.

3.4 Applications and Prefetching Methodology

We use five applications in this study – Radix, LU, and FFT from the SPLASH-2 suite [34], and Water and Mp3d from the SPLASH suite [31]. Since we do not have a compiler that implements software-prefetching for C programs, we insert prefetches by hand, following the algorithm by Mowry et al. (Section 2.1) for all applications except Water. Prefetching for Water is described further below.

We assume a default prefetch distance of 200 instructions to model the representative latency seen in the system. In Section 5, we vary the prefetch distance to study the impact of this assumption. To account for conflict misses, our locality analysis assumes a cache size equivalent to a fraction of the L1 cache. We choose a fraction that minimizes the number of unnecessary prefetches for all our applications. We perform loop transformations as required by the prefetching algorithm. Whenever loop-unrolling was used in the prefetch version of an application, we also used it for the base application, as loop-unrolling improves the performance of `ILP`. In addition, since our focus is on the prefetch algorithm and not on a particular compiler implementation, we assume aggressive compiler techniques (e.g., procedure inlining, symbolic loop-bound analysis) to perform locality analysis. For RC, since write latency is already hidden, we do not issue exclusive prefetches for cache lines that are only written; however, we use exclusive prefetches for reads of lines that will also be written. For SC, we additionally issue exclusive prefetches for cache lines that are only written, since SC can expose write latency. We next describe each application and the prefetches inserted.

We study two versions of **LU**. The first, **LUorig**, is the original SPLASH-2 application enhanced by using flags instead of barriers for better load balance. The second version, **LUopt**, additionally uses loop-interchange transformations to move read misses closer to each other. This increases the number of read misses that overlap with each other, better exploiting the support for non-blocking reads in ILP [27]. LUopt achieves higher performance than LUorig on ILP without prefetching, but is outperformed by LUorig with the addition of prefetching. We therefore report the results of both versions in Section 4; subsequent sections use only LUorig. Prefetching in LU covers all loop nests in the application. The writes in these loop nests follow reads to the same locations, so we insert exclusive prefetches for such read accesses.

We consider two versions of **FFT**, **FFTorig** and **FFTopt**. Analogous to LUopt, FFTopt clusters read misses (in the transpose phase) to increase overlap. We only report results for FFTopt because it outperforms FFTorig on the ILP system, both with and without prefetching. Prefetching is used for all read accesses expected to miss; some of these are exclusive prefetches. In the transpose phase, there are write misses for which there are no previous reads in the same phase; these writes are prefetched in the SC version.

In **Mp3d**, all reads to particles are prefetched. These particles are used to generate addresses for cells, which are prefetched using the method for indirect accesses described in Section 2.1. All prefetches are exclusive. Some cell accesses cannot be prefetched because their addresses are known only immediately before their demand accesses.

In **Radix**, prefetches are inserted for all reads that are expected to miss and whose addresses are known sufficiently before the demand access. Exclusive prefetches are used in the prefix-sum phase of this application. The key sort or "permutation" phase consists of writes to lines that are not previously read in the same phase. These writes cannot be prefetched for SC because their addresses are not known early enough before the demand access.

Water is the only application for which we do not strictly follow the algorithm in Section 2.1. The algorithm assumes no locality across synchronization, and thus does not schedule prefetches across synchronization accesses. However, since locks in Water do not have much contention and the critical sections are too small to schedule prefetches, we move prefetches corresponding to accesses within a critical section to before the critical section. This allows these prefetches to overlap with the lock acquire of the critical section, improving the benefits of prefetching. We issue exclusive prefetches in the UPDATE_FORCES procedure to prefetch the force data structures before entering the associated critical sections. We also prefetch molecule displacements. These prefetches occur within loops with large bodies, and consequently see prefetch distances much larger than the default 200 instructions in the other applications.

The input sizes for our applications are summarized in Figure 3. These are greater than or equal to the sizes used in the SPLASH and SPLASH-2 distributions for all applications except LU. In the case of LU, owing to higher simulation times with our detailed simulator, we use a problem size one step smaller than recommended, but also scale down the number of processors appropriately. We use 16 processors for applications that scaled well (FFT and Water) and 8 processors otherwise (LU, Mp3d, and Radix). Data layouts for these applications for LU, FFT, and Radix follow the recommendations of the SPLASH-2 distribution; data layouts for Mp3d and Water aim to maximize locality.

4 Interaction of Software Prefetching with ILP

This section evaluates how software prefetching interacts with ILP techniques in shared-memory multiprocessors by comparing its impact on the ILP system with that on the Simple system. This section uses RC for both ILP and Simple, since RC has been shown to provide better performance than SC for both types of systems [9, 11, 28].

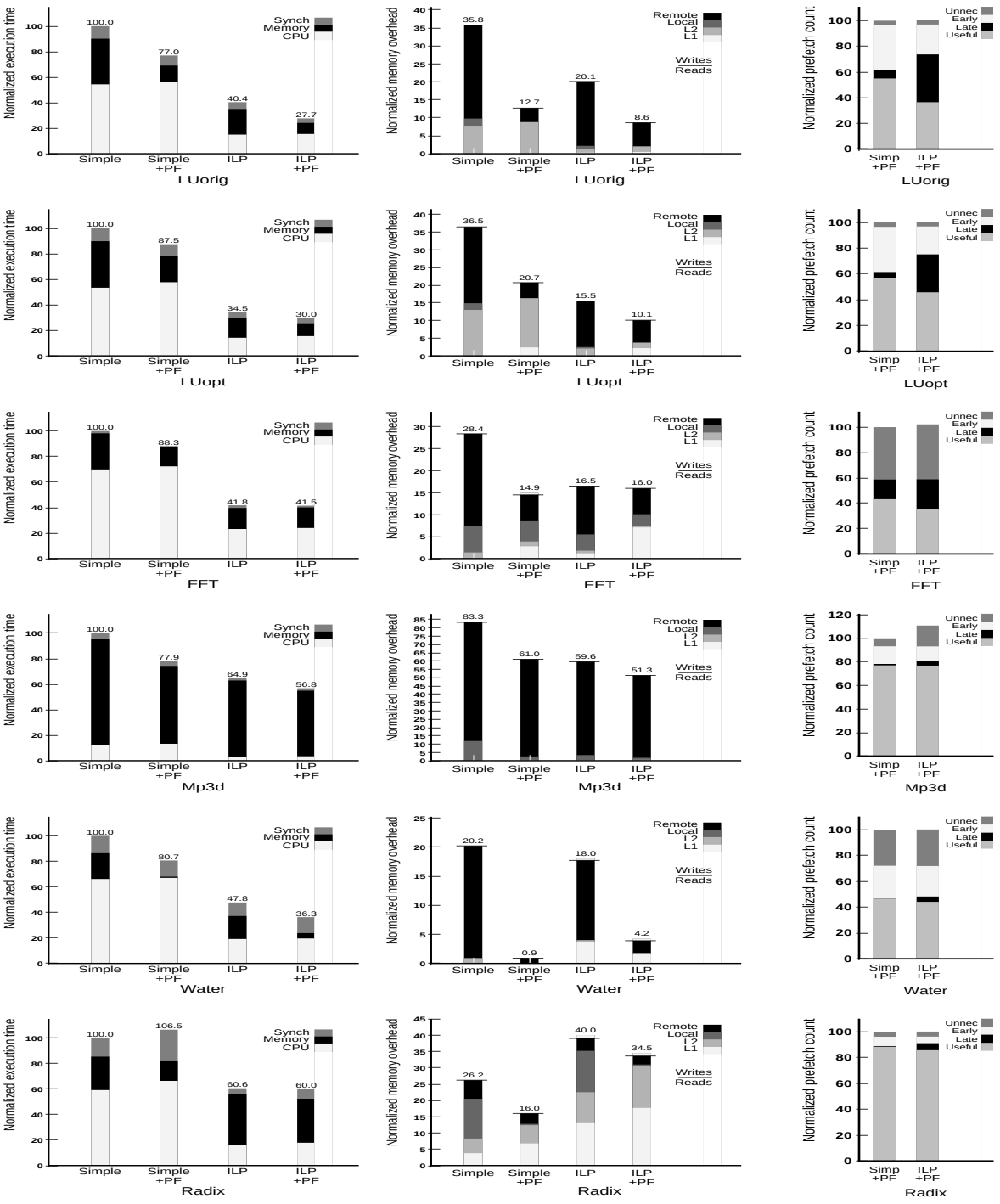
Figures 4 and 5 present the key results from our experiments. Throughout Figure 4, +PF indicates the addition of software prefetching. Figure 4(a) shows the execution times for each application on Simple and ILP, both with and without software prefetching. The execution times are normalized to the time for the application on Simple without prefetching, and are divided into three components – CPU, memory stall, and synchronization stall time. Figure 4(b) magnifies the memory region of Figure 4(a), providing a more detailed characterization of memory latency. Each bar showing memory stall time is separated into read and write components by a horizontal dividing line. With RC, no significant write component time is seen in any application (however, contention caused by writes can impact read time). Read and write stall times are further divided into time spent stalled on L1 cache hits, L2 cache hits, misses to local memory, and misses to remote memory. Figure 4(c) shows the prefetches issued to the L1 cache, and divides them into the categories *useful*, *late*, *early*, and *unnecessary* according to the definitions in Section 3.2. The total number of prefetches are normalized to the number in Simple.

Figure 5 summarizes the key statistics from Figure 4. The first two rows show the reductions in overall execution time and memory stall time due to software prefetching in both ILP and Simple. The third row shows the percentage of time an ILP or Simple execution is stalled for memory with software prefetching. The last row shows the percentage of prefetches that are late in ILP and Simple. For LU, Figure 5 reports only LUorig since LUorig with software prefetching performs better than LUopt with software prefetching for both Simple and ILP.

4.1 Overall Results

As shown in the first row of Figure 5, software prefetching achieves an overall reduction in execution time in ILP similar to or greater than that in Simple for three out of five applications (LU, Water, and Radix). The reductions for ILP are significant (12% to 31%) for three applications (LU, Mp3d, and Water). However, focusing on the memory stall time (the execution time component specifically targeted by prefetching), we find that software prefetching is less effective in reducing memory stalls on ILP than on Simple. Specifically, for our applications, prefetching reduces the absolute memory stall time by an average of 34% in ILP compared to 56% in Simple. The net effect is that even after prefetching is applied to ILP, memory stall time constitutes more than 30% of the total execution time for all applications except Water, and more than 40% of the time for three applications. In contrast, in Simple, all applications except Mp3d see less than 17% memory stall time. Thus, for our applications, the ILP system remains memory-bound even with software prefetching.

The reasons for the reduced effectiveness of software prefetching on memory stall time in ILP and its resulting impact on total execution time are identified in the following subsections.



(a) Execution time components (b) Memory stall time (c) Prefetch characterization

Figure 4: Impact of software prefetching on `Simple` and `ILP` performance

Performance Metric	LU		FFT		Mp3d		Water		Radix	
	ILP	Simple	ILP	Simple	ILP	Simple	ILP	Simple	ILP	Simple
% execution time reduced	31	23	2	13	12	22	24	19	1	-7
% memory stall time reduced	57	65	9	53	14	27	77	96	14	39
% memory stall time remaining	31	17	41	15	91	78	12	1	58	15
% prefetches that are late	37	7	23	16	4	1	4	0	5	0

Figure 5: Effectiveness of software prefetching

Factor	LUorig	LUopt	FFT	Mp3d	Water	Radix
Late prefetches	√	√	√	√	√	√
Resource contention		√	√		√	√
Speculative prefetches			√	√		
Overlapped accesses	√	√				
Early prefetches	√	√		√	√	√

Figure 6: Factors shaping the effectiveness of software prefetching for ILP

4.2 Memory Stall Time Reduction

We first identify three factors that make software prefetching less successful in reducing memory stall time in ILP than in Simple. We then identify two factors that enable ILP to achieve benefits in memory stall time reduction not available in Simple. Figure 6 summarizes these factors and the applications affected by each factor. Our focus here is on the difference between Simple and ILP, so we do not discuss issues that are common to both systems.

Increased late prefetches. Figures 4(c) and 5 show 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 dynamic scheduling speed up computation in ILP, decreasing the computation time with which each prefetch is overlapped. In addition, Simple stalls on any read misses that are not prefetched or that incur a late prefetch, thereby allowing other outstanding prefetches to complete. ILP does not provide similar leeway, as it does not immediately stall the processor on misses. The above reasons also lead to an increased frequency of memory accesses in the ILP case, which in turn increases contention for system resources. As a result, misses have longer latency in ILP, further increasing the number of late prefetches.

Our results also show that optimizations to cluster read misses for the ILP system (described in Section 3.4) can reduce the effectiveness of software prefetching. In the absence of prefetching, LUopt provides a 13% reduction in execution time compared to LUorig for ILP. With prefetching, however, LUopt sees 10% *more* execution time than LUorig. The clustering optimization used in LUopt decreases the amount of computation between successive misses. As a result, prefetches require a greater number of iterations in order to completely overlap the expected miss latency. However, the loops in LUopt do not have enough iterations to accommodate the needed prefetch distance, so the prefetches in LUopt are unable to successfully hide their targeted latencies.

Increased resource contention. Even without prefetching, ILP processors stress system resources more than Simple. These resources include processor functional units, cache ports, cache MSHRs, the memory bus, the memory and directory banks, and the network. Resource contention is less in Simple because Simple processors stall on read misses, allowing resources used by previous writes and prefetches to free up during these stalls.

Resource contention can result in an increase in each of the components of memory latency, and can be further exacerbated by prefetching. For prefetched accesses, an increase in latency after the L1 cache access can either be successfully overlapped by prefetching or appear as late prefetches, described above. However, prefetching cannot target increased latencies incurred at or before the L1 cache access. We focus on this type of resource contention next.

We recognize contention for resources at or before the L1 cache access in our applications by observing the presence of exposed L1 read hit time, as this time is otherwise hidden in both Simple and ILP. Additionally, the presence of an exposed L2 read hit component in ILP also generally indicates resource contention, since ILP techniques usually hide L2 read hit latencies otherwise. We identify the sources of these types of resource contention by examining MSHR occupancy at the caches and functional unit utilization (not shown here for lack of space).

In our applications, resource contention before the L1 cache particularly limits the ILP performance of LUopt, FFT, and Radix. These applications see significant MSHR saturation, caused by overlapped reads in LUopt and FFT and by writes in Radix. When the MSHRs of a cache saturate, subsequent misses stall at the cache ports, eventually blocking even cache hits. This effect is particularly large in FFT and Radix, as evidenced by a significant L1 read hit component. As discussed above, such a component cannot be targeted by software prefetching, and is actually exacerbated by the greater frequency of requests with prefetching. Additionally, for FFT, software prefetching with ILP also sees a shortage of ALUs and address generation units needed to calculate addresses for both prefetch and demand accesses.

Water stands as an exception to the above trends, as software prefetching actually relieves some resource contention in this application. Water sees resource contention for the processor's memory queue due to lock releases. Each release is marked with a release memory fence, using a SPARC V9 memory barrier. This fence prevents the issue of all subsequent writes until all previous reads and writes complete. As a result, later writes can fill up the memory queue if earlier accesses do not complete quickly enough. If the memory queue fills up, subsequent accesses (including cache hits and private accesses) are prevented from entering the memory queue or being processed in any fashion. Since prefetching can cause accesses before the release memory fence to complete more quickly, this source of resource contention is actually reduced by software prefetching.

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. Among our applications, only Mp3d experiences a significant number of speculative prefetches, with 11% more total prefetches in ILP than in Simple. However, most of the speculated prefetches are to lines also prefetched on the correct path, and the correct prefetches either coalesce with the speculative prefetches or hit because of them. Thus, speculative prefetches do not impact performance in our applications.

Overlapped accesses. In ILP, accesses that are difficult to prefetch may be overlapped because of non-blocking reads and dynamic scheduling. Thus, overall, prefetching may appear more effective in ILP than in Simple, as prefetching in ILP only needs to target those accesses that are not already overlapped by ILP. We see the benefit of overlapped accesses in LUorig and LUopt. In particular, the fraction of remote latency overlapped in these applications

is much smaller in ILP than in Simple for the reasons discussed above. Nevertheless, the fraction of overall memory stall time reduced in these applications is similar in both systems. In these applications, the lines to be prefetched suffer from repeated L1 cache conflicts; thus, prefetches can be replaced from the L1 cache before their corresponding demand accesses. In Simple, each of these conflicting accesses must incur at least an L2 access time penalty. In contrast, non-blocking reads and dynamic scheduling allow the ILP system to effectively overlap the latencies of these conflicting accesses, thereby removing a limitation to prefetching experienced by Simple.

Fewer early prefetches. In most of our applications, the number of early prefetches drops in ILP. This reduction occurs because the ILP system allows less time between a prefetch and its subsequent demand access, decreasing the likelihood of an intervening replacement or invalidation. Early prefetches can hinder demand accesses by replacing or invalidating needed data from the same or other caches without providing any benefits in latency reduction. Thus, the reductions in early prefetches seen in ILP can potentially help improve prefetching effectiveness for these applications. However, in our applications, such benefits are offset by an increase in late prefetches with ILP.

4.3 Impact on Total Execution Time

Despite its reduced effectiveness in addressing memory stall time, software prefetching achieves significant execution time reductions with ILP for three of our applications (see Figure 5) for several reasons.

Increased weight of memory stall time. ILP features like multiple issue and dynamic scheduling provide greater improvements for computation time than for the memory stall time on all our applications [27]. As a result, memory stall time contributes far more to execution time in ILP than in Simple. Thus, even a smaller fraction of memory stall time reduced in ILP can lead to a reduction in overall execution time similar to or greater than that seen in Simple.

Reduced prefetch overhead. For all of our applications, the ILP system sees less instruction overhead from prefetching (as a percentage of total execution time) than the Simple system. This is because dynamic scheduling and multiple issue allow many of the additional instructions associated with software prefetching to overlap with other computation or memory accesses.

Effect on synchronization stall time. Prefetching can also impact the synchronization component of execution time. The impact of prefetching with ILP on synchronization time varies among our applications, with both positive and negative interactions. However, the synchronization component contributes little to ILP execution time in all of our applications except Water. In Water, there is a negative interaction because prefetching increases synchronization acquire latency in ILP due to increased contention in the memory system. Thus, synchronization time expands to fill a large part of the reduction in memory stall time with prefetching in the ILP system.

5 Re-examining Assumptions of the Prefetching Scheme

The previous section shows that late prefetching and resource contention are the two key limitations to the effectiveness of prefetching on ILP. This section re-examines several assumptions in our prefetching scheme, with the goal of reducing late prefetches and resource contention in ILP.

5.1 Reducing Late Prefetches

Increasing the prefetch distance. A straightforward way to reduce the number of late prefetches is to increase the prefetch distance. We varied the prefetch distance from our default of 200 instructions to 400 and 800 instructions. Figure 7 shows the results for LU, FFT, Mp3d, and Radix (we henceforth only focus on LUorig since it gives better performance with prefetching). We do not apply this technique to Water because, in this application, prefetches either occur within loops where one iteration already has more than 800 instructions, or occur within control statements where prefetching before the branch can lead to an excessive number of prefetches of lines that are not used by the processor. In Figure 7, PF, PF2, and PF4 refer to prefetching distances of 200, 400, and 800 instructions respectively. Figure 7(a) shows the execution times for each version normalized to ILP without prefetching, and Figure 7(b) characterizes the types of prefetches seen in each version. The variation in the total number of prefetches in the three versions is due to speculative prefetching (Section 4.2).

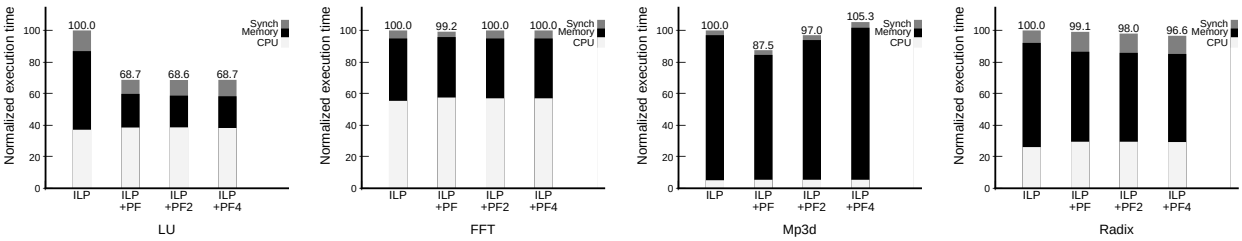
As Figure 7 shows, the PF2 and PF4 versions reduce the number of late prefetches in all applications. However, there is no significant improvement in execution time in any of the applications. On the contrary, increased prefetch distances have a negative impact on some applications.

There are three reasons why increased prefetch distances do not improve performance for our applications. First, in Mp3d and Radix, the decrease in late prefetches occurs at the expense of an increase in early prefetches. Many prefetches arrive at the cache much before the demand access. These are vulnerable to cache replacements or invalidations for a longer time, as also observed in studies of previous-generation multiprocessors [33]. In Mp3d, these early prefetches hurt performance because they prematurely invalidate other processors' cache lines (due to false and true sharing). In Radix, most early prefetches are replaced prefetches that do not adversely affect other processors.

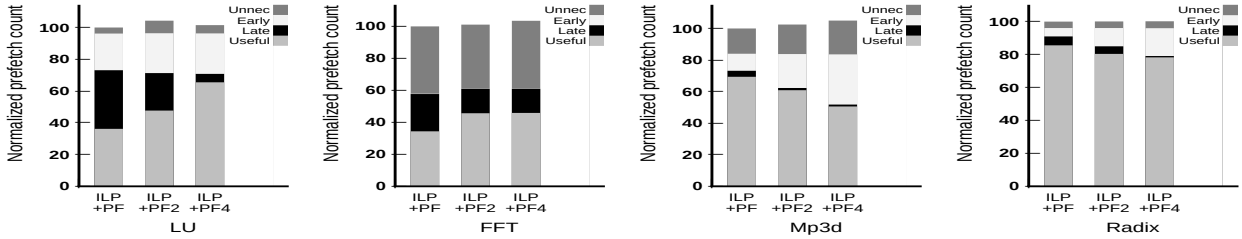
Second, on applications that are resource-bound, larger prefetch distances further stress system resources by keeping more prefetches outstanding at a time. Our detailed statistics show that the resultant MSHR saturation exposes additional L1 hit latencies (LU, FFT, and Radix) and L2 hit latencies (LU and Radix), which offset the performance benefits due to reducing late prefetches.

Third, in FFT, some loops containing late prefetches do not have an adequate amount of computation to increase the prefetch distance sufficiently in order to reduce late prefetches.

Latency-sensitive prefetching. When the prefetch distance is increased in LU, Mp3d, and Radix, early prefetches increase primarily because the algorithm makes the approximation that all misses incur a common latency. In NUMA multiprocessors, however, remote memory latencies are significantly larger than local memory latencies. Consequently, it may be beneficial to perform a form of *latency-sensitive* prefetching for such systems. The prefetching algorithm can be modified to determine the prefetch distance for each access based on the predicted latency of the access (e.g., by using information about the data layout). Such an approach has been discussed in other studies for software and hardware prefetching [8, 13, 21]. Our applications, however, did not show much benefit with this modification.



(a) Execution time components



(b) Prefetch characterization

Figure 7: Effect of increasing the prefetch distance

It was difficult to apply latency-sensitive prefetching to LU since different dynamic instances of the same memory instruction in LU can reference both local and remote memory. For Mp3d and Radix, we assumed a prefetch distance of 100 instructions for loops with only local prefetches and the best observed distance (from PF, PF2, and PF4) for loops with remote prefetches. Both Mp3d and Radix achieve negligible improvements (less than 1%). In Radix, there is a significant reduction in the number of early prefetches; however, the demand latencies overlapped by these prefetches contribute only a small part of the total execution time. In Mp3d, the high false and true sharing makes late prefetches preferable to early prefetches.

Outer-loop prefetching. Figure 7(b) shows that FFT continues to see a significant number of late prefetches even in the PF4 version (20% of the total number of prefetches). The prefetching algorithm described in Section 2.1 adds prefetching at the innermost level of the loop nest that causes cache misses. However, the transpose phase of FFT uses a blocked algorithm, and the inner loop of the transpose does not have enough computation or iterations to accommodate a larger prefetching distance. In this case, the outer loop could potentially be software pipelined to prefetch further in advance. Thus, we insert prefetches for a block that will be accessed in a later iteration, rather than for later lines of the same block in the current iteration[1]. We implemented this form of outer-loop prefetching in FFT. However, we observed a 16% slowdown for two reasons. First, the addition of outer-loop prefetching increases the number of prefetches outstanding at a time, further increasing the resource contention in the system (evidenced by a 13% increase in MSHR saturation). Second, adding outer-loop prefetching introduces inter-block cache conflicts.

5.2 Reducing Resource Contention

Although the base architecture used in Section 4 is aggressive in processor, network, and memory system resources, some applications still observe reduced benefits from prefetching on ILP due to increased resource contention. We next re-examine two assumptions in our prefetch strategy that can affect resource contention with ILP.

L2 prefetching. The prefetch strategy used in Section 4 prefetched data into the L1 cache. Prefetching only to the L2 cache can potentially reduce contention in two ways. First, L2 prefetching allows the prefetching algorithm to use the larger L2 cache size in its locality analysis, possibly resulting in fewer prefetches and consequently less contention for address generation units and cache ports. Second, bypassing the L1 cache can potentially reduce resource contention at the L1 cache. However, subsequent demand accesses now see the L2 cache latency, instead of the L1 cache latency. With ILP processors, such L2 cache latencies can be overlapped by non-blocking reads and dynamic scheduling, and thus may not hinder performance.

We find that, for all our applications, the execution time with L2 prefetching is within 1% of the execution time with L1 prefetching. L2 prefetching does not see any performance degradation because ILP techniques are successful in overlapping most of the L2 cache latencies. However, L2 prefetching provides no significant benefits, as the use of the L2 cache size in locality analysis does not reduce the number of prefetches for our applications and input sizes.

Prefetch drop strategy. Following previous work, our base prefetch strategy does not drop prefetches even when resource constraints block their issue [23]. We examined an alternate strategy that drops prefetches when the L1 cache MSHRs are saturated. Our results show less than 2% difference in execution time on all our applications. Dropping prefetches does not give greater performance improvements because any advantages in reducing resource contention are offset by greater latencies seen by subsequent demand misses, as in studies with previous-generation processors [23].

5.3 Summary and Implications

The results of this section show that straightforward modifications to the prefetching algorithm to reduce late prefetches do not necessarily translate to more effective prefetching in

[1] McIntosh has simultaneously developed a similar algorithm for a High Performance Fortran compiler [20].

our applications because of increased early prefetches, inadequate computation, and/or increased stress on system resources. Additionally, alternate prefetch strategies that target L1 resource contention, like L2 prefetching and a simple prefetch drop strategy, do not significantly impact the performance of prefetching for our applications.

These results show that further techniques are needed to alleviate the effects of late prefetches and resource contention. Latency-reducing techniques such as *producer-initiated* communication primitives [1, 15, 29] appear promising. A recent study with simple processors has shown that such primitives can interact positively with software prefetching to reduce the effects of both late prefetches and resource contention [1].

6 Interaction of Prefetching with Consistency Models on ILP Systems

This section evaluates the performance benefits of software prefetching with sequential consistency on ILP multiprocessors and also determines if software prefetching can equalize the performance of sequential consistency (SC) and release consistency (RC). We examine two implementations of sequential consistency – SCplain and SCopt – representing the straightforward and aggressive implementations of SC respectively, and the straightforward implementation of RC, as described in Section 2.2. Figure 8 summarizes the results for this section and is analogous to Figures 4(a) and 4(b).

6.1 Overall Results

We find that the performance benefits of software prefetching vary widely across the consistency implementations. The benefits seen on SCplain are consistently higher than those seen on RC. Consequently, software prefetching reduces the gap between SCplain and RC for all our applications. Nevertheless, RC with software prefetching still shows a substantial reduction in execution time relative to SCplain with software prefetching, ranging from 24% to 65% for our applications.

On SCopt, the addition of software prefetching leads to significant improvements for LU and Water, and small or no improvements for the other three applications; these are comparable to the improvements due to prefetching on RC. Software prefetching significantly decreases the performance gap between SCopt and RC only in LU. RC with software prefetching continues to give significant reductions in execution time compared to SCopt with software prefetching on FFT, Mp3d, and Radix (15%, 35% and 40% respectively).

The addition of software prefetching thus does not entirely eliminate the performance advantages of RC. The following sections respectively analyze the reasons for the remaining memory stall time in SC systems after adding software prefetching and the reasons for the remaining performance difference between SC and RC.

6.2 Memory Stall Time Reduction in SC Systems

Without software prefetching, the SC implementations see larger memory stall time than RC due to consistency constraints [28]. Software prefetching can address much of this additional time and thus can lead to larger reductions in memory stall time than in RC. However, even after adding software prefetching, a significant amount of memory stall time still remains with SC in all our applications except Water (an average of over 58% of total execution time for SCplain and over 43% for SCopt). We next discuss three reasons for the remaining memory stall time in SC systems, focusing on the differences between SC and RC with software prefetching.

Limitations of software prefetching. As in RC, much of the remaining memory stall time in SC is due to the limitations of prefetching stemming from late and early prefetches or resource contention. However, the impact of these limitations is higher on SCplain and SCopt, since they see these effects on writes as well as reads (RC hides the latency of writes). SCplain sees an even higher impact of these limitations since it does not overlap any part of the read latency (unlike SCopt and RC). Additionally, in SC, downgraded prefetches (exclusive prefetches transitioned to shared-state by an external coherence action before the demand access) can further reduce the benefits of prefetching for write accesses.

One or more of the above effects is seen with all our applications in the SC systems. The effect of late prefetches is seen in all applications. The effect of early prefetches is seen in Mp3d (caused by true and false sharing) and LU (caused by conflicts at the L1 and L2 caches). Only FFT sees significant resource contention, evidenced through MSHR saturation. In contrast to RC, Radix does not see resource contention in SC since the writes in the key permutation phase are no longer overlapped, and so do not saturate the MSHRs.

Consistency-related delays. SC with software prefetching also experiences memory stall time owing to the constraints of the consistency implementation.

In SCplain, demand accesses cannot be issued before reaching the head of the memory queue. Consequently, SCplain without software prefetching sees a significant amount of consistency-related read and write hit latency (2% to 17% of execution time). These latencies are not targeted by software prefetching.

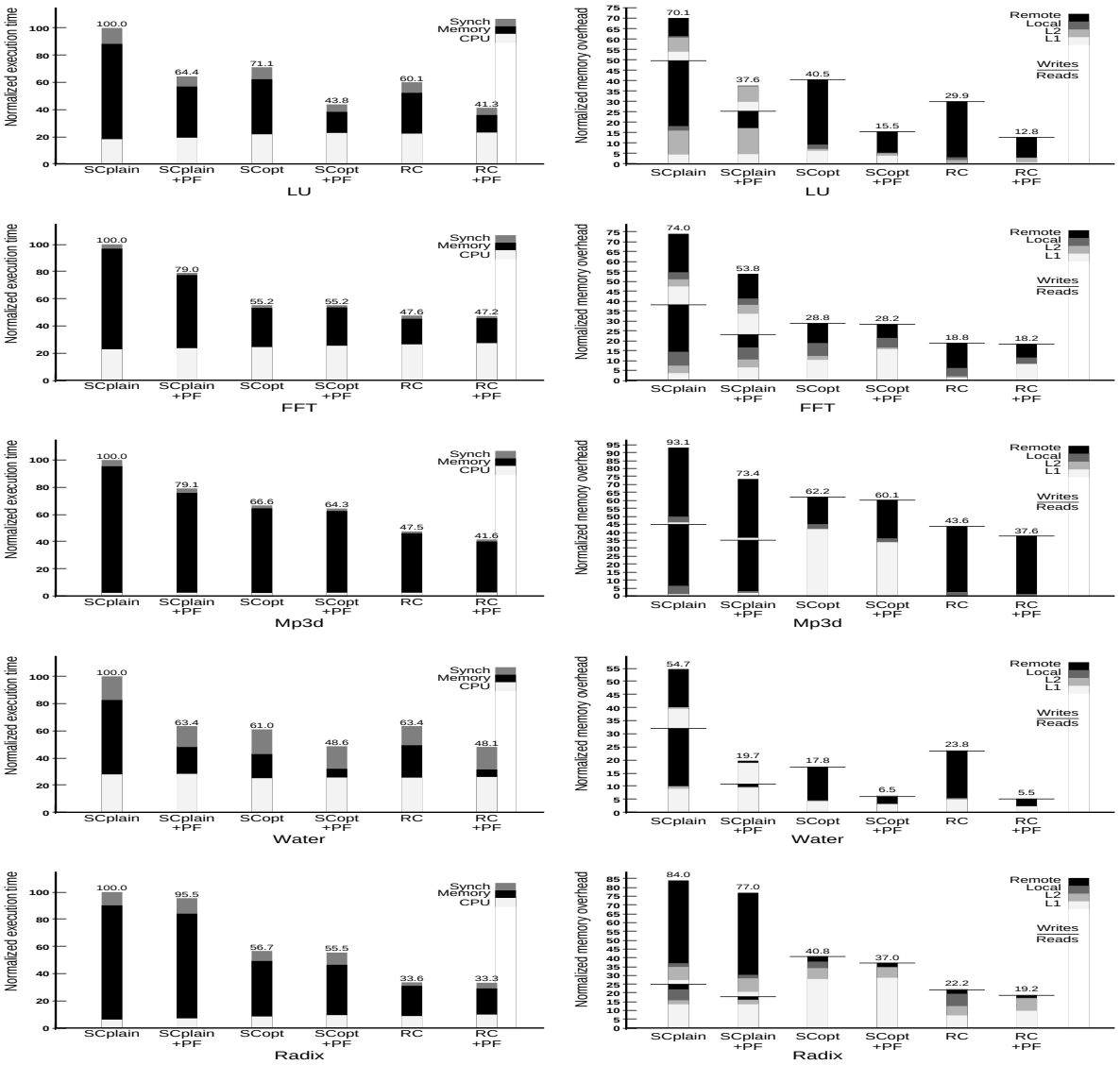
In SCopt, write buffering prevents write stall times from being directly exposed to the processor. However, read operations in SCopt cannot retire from the instruction window until all previous writes have completed. Thus, unlike RC, reads can result in stall time if previous writes were not successfully prefetched.

Unamenable accesses. Finally, a significant portion of the memory stall time can still remain because of memory accesses that are not amenable to prefetching. In Mp3d and Radix, addresses of some high-latency memory accesses are determined very close to their use, preventing the prefetching algorithm from issuing prefetches for these accesses. The SC systems expose the latency of both read and write operations that are unamenable to prefetching; RC, on the other hand, generally sees only read latencies.

6.3 Comparison of RC and SC

Comparing RC and SCplain, RC with software prefetching achieves lower execution time than SCplain with software prefetching by hiding the latency of write misses (whether late, early, or unprefetched), and by avoiding consistency-related delays. One or more of these components is substantial in the SCplain executions of all our applications. As a result, the gap in execution time between RC and SCplain RC remains significant, ranging from 24% to 65% for our applications.

Comparing RC and SCopt, RC with software prefetching achieves lower execution time than SCopt with software prefetching if reads in SCopt experience stall time waiting



(a) Execution time components

(b) Memory stall time components

Figure 8: Effect of software prefetching on consistency models

for previous write latencies (due to ineffective prefetching or unprefetched accesses). FFT, Mp3d, and Radix see this effect. Consequently, RC with software prefetching shows significantly lower execution times than SCopt with software prefetching for these applications (15%, 35% and 40% respectively). LU and Water have comparable performance with RC with software prefetching and SCopt with software prefetching.

Additionally, as mentioned in Section 4.3, software prefetching can contribute either positively or negatively to the synchronization component of execution time. However, memory stall time dominates the execution times of most of our applications, and subsequently determines the performance difference between SC and RC.

Thus, our results indicate that software prefetching is unable to close the gap between SC and RC. Write latencies that prefetching is unable to hide remain the most important reason for this performance gap. Our results also show that software prefetching alone does not obviate the need for hardware optimizations to SC, since SCopt continues to significantly outperform SCplain for all applications, even with the addition of software prefetching.

7 Related Work

To the best of our knowledge, there have not been any previous evaluations of either hardware or software controlled non-binding prefetching for state-of-the-art ILP multiprocessors.

Gornish evaluated binding prefetching for both single-issue and multiple-issue statically-scheduled multiprocessors with non-blocking reads [13]. The binding prefetches rely on software cache-coherence and require complete cache flushes before and after each parallel loop. Processor pipelines and functional-unit contention are not modeled. This work integrates hardware and software prefetching support, dynamically adapting hardware prefetching distance according to the latency of each reference. The study finds that soft-

ware prefetching provides execution time improvements on their multiple-issue system similar to or greater than those seen with their single-issue system, but does not analyze the interaction of ILP features with prefetching.

Three recent studies on prefetching in uniprocessors examine state-of-the-art uniprocessors with multiple issue, dynamic scheduling, and non-blocking reads. Luk and Mowry proposed software prefetching algorithms for pointer-based data structures and evaluated their algorithms for an ILP processor similar to the MIPS R10000 [19]. Bennett and Flynn compared stream buffers (a form of hardware-controlled prefetching), hardware stride-based prefetching, and victim caches for state-of-the-art ILP processors [2]. They found that stream buffers and stride-based prefetching do not have much impact on most SPEC92 programs, as these techniques stress bus bandwidth. In another study, Bennett and Flynn proposed a *prediction cache* that dynamically adapts between stream buffer and victim cache functionality based on program miss patterns [3]. The prediction cache dynamically adjusts the stream buffer prefetching distance based on the late prefetch pattern of the program. Although all of the above studies use aggressive ILP uniprocessors, they do not focus on the impact of ILP on prefetching, and so do not relate the results of their studies to the ILP features of the processor.

Other previous studies on prefetching with ILP uniprocessors examine statically-scheduled processors. Chen et al. investigated the use of software-controlled prefetching on statically-scheduled multiple-issue uniprocessors for non-numeric applications [6]. This work seems to allow only one outstanding read (but several outstanding prefetches). Further, they assume a small read latency of 10 cycles. Their results show prefetching to be effective with multiple issue uniprocessors. Chen and Baer compared the performance of non-blocking reads and hardware prefetching on a statically-scheduled single-issue uniprocessor [5]. They found that their hardware prefetching scheme generally performed better than non-blocking reads and is less sensitive to memory latency. They also studied the combination of non-blocking reads and hardware prefetching for two applications and found it to perform better than either technique alone. Farkas et al. studied the impact of non-blocking reads and stream buffers with statically-scheduled multiple-issue uniprocessors [7]. They assumed a small miss latency of 16 cycles. They found that both non-blocking reads and stream buffers improve performance; the best performance is achieved when both are combined.

In the context of previous-generation systems, several studies have proposed algorithms for software-controlled non-binding prefetching [4, 16, 23, 24, 25]. Of these, Mowry et al. developed the most sophisticated algorithm and compiler implementation, with a comprehensive evaluation for both uniprocessors and shared-memory multiprocessors [23, 24, 25]. This algorithm is described in Section 2.1. Tullsen and Eggers evaluated software-controlled non-binding prefetching on a bus-based system [33]. They characterized bandwidth needs of their applications, and found that the benefits of prefetching degrade as bandwidth needs increase. They also found that increasing the prefetch distance can reduce late prefetches but can also increase early prefetches, and thus does not significantly improve performance. We found this to be true for our ILP-based system as well (Section 5.1).

Gupta et al. evaluated the interaction between software-controlled prefetching and straightforward implementations of consistency models for systems with simple processors [14]. They found that prefetching significantly improved the performance of both SC and RC. However, for two out of their three applications, a significant performance gap remained between SC and RC even after adding prefetching.

8 Conclusions

Shared-memory multiprocessors are being increasingly built from commodity microprocessors that aggressively exploit instruction-level parallelism. This paper provides the first study of the effectiveness of software-controlled non-binding prefetching in multiprocessor systems built of state-of-the-art ILP processors. We compared two multiprocessors systems – `ILP` and `Simple` – that are equivalent in every respect except that the former uses state-of-the-art ILP processors while the latter uses previous-generation processors (with single issue, static scheduling, and blocking reads). We found that while software prefetching resulted in substantial reductions in execution time (12% to 31%) for three out of five applications on the `ILP` system, it is significantly less effective in reducing the data memory stall time on the `ILP` system than on the `Simple` system. Even after the addition of software prefetching, memory stall time constituted more than 30% of the execution time for four out of five applications on the `ILP` system. In contrast, on the `Simple` system, memory stall time constituted less than 17% of the execution time for four out of five applications. Thus, software prefetching does not significantly change the memory-bound nature of most of our applications on the `ILP` system.

The key factors that limit the effectiveness of software prefetching on the `ILP` system are an increase in late prefetches and an increase in contention at various resources. Our results thus motivate modifications to prefetching algorithms so that they are more sensitive to resource use and to variations in expected latency for different accesses. Several straightforward enhancements to the prefetching scheme, however, did not show any improvements. An alternative technique to improve memory system performance is the use of latency-reducing (rather than tolerating) techniques. For example, producer-initiated communication primitives may be able to reduce latency and resource contention, while interacting positively with prefetching.

This paper also examines whether software prefetching can equalize the performance of sequential consistency (SC) and release consistency (RC) on ILP multiprocessors. We found that although software prefetching reduces the gap between the straightforward implementations of SC and RC, a significant gap remained between these implementations (ranging from 24% to 65% for our applications). Comparing an optimized hardware implementation of SC to a straightforward implementation of RC, we found that a significant gap remains for three out of five applications (15% to 40%). Write latencies that prefetching does not effectively hide remain the most important reason for the performance gap between SC and RC. Thus, future efforts to provide SC programmability with RC performance need to more effectively address the limitations that write latencies impose on SC.

9 Acknowledgments

We would like to thank Kathi Fletcher, Nat McIntosh, Shubu Mukherjee, Ram Rajamony, and Willy Zwaenepoel for valuable feedback on earlier drafts of this paper.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. 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*, 1997.

[2] J. E. Bennett and M. J. Flynn. Latency Tolerance for Dynamic Processors. Stanford University, CSL-TR-96-687, 1996.

[3] J. E. Bennett and M. J. Flynn. Reducing Cache Miss Rates Using Prediction Caches. Stanford University, CSL-TR-96-707, 1996.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[5] T.-F. Chen and J.-L. Baer. Reducing Memory Latency via Non-Blocking and Prefetching Caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[6] W. Y. Chen et al. Data Access Microarchitectures for Superscalar Processors with Compiler-Assisted Data Prefetching. In *Proceedings of the 24th Annual International Symposisum on Microarchitecture*, 1991.

[7] K. Farkas, N. Jouppi, and P. Chow. How Useful are Non-Blocking Loads, Stream Buffers and Speculative Execution in Multiple Issue Processors? In *Proceedings of the 1st International Conference on High-Performance Computer Architecture*, 1995.

[8] K. Fletcher. Compiler-hardware cooperation in prefetching for shared-memory multiprocessors. Ph.D. Thesis Proposal, Rice University, September 1995.

[9] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 4th Inernational Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[10] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the International Conference on Parallel Processing*, 1991.

[11] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding Memory Latency Using Dynamic Scheduling in Shared-Memory Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

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

[14] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[15] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[16] A. C. Klaiber and H. M. Levy. An Architecture for Software-Controlled Data Prefetching. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991.

[17] D. Kroft. Lockup-Free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th International Symposium on Computer Architecture*, 1981.

[18] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[19] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[20] N. McIntosh. Private communication. *Rice University*, February 1997.

[21] N. McIntosh, K. Fletcher, K. Cooper, and K. Kennedy. Compiler Techniques for Software Prefetching on Cache-Coherent Shared-Memory Multiprocessors. Center for Research on Parallel Computation, Rice University, CRPC-TR96675-S, 1997.

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

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

[24] T. Mowry and A. Gupta. Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[25] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[26] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *Proceedings of the 3rd Workshop on Computer Architecture Education*, 1997.

[27] V. S. Pai, P. Ranganathan, and S. 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*, 1997.

[28] V. S. Pai, P. Ranganathan, S. V. Adve, and T. 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*, 1996.

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

[30] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The Impact of Architectural Trends on Operating System Performance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, 1995.

[31] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, 20(1):5–44, 1992.

[32] Sparc International. *The SPARC Architecture Manual, Version 9*, 1993.

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

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.