Comparative Evaluation of Latency Reducing and Tolerating Techniques

Anoop Gupta, John Hennessy,
Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Techniques that can cope with the large latency of memory accesses are essential for achieving high processor utilization in large-scale shared-memory multiprocessors. In this paper, we consider four architectural techniques that address the latency problem: (i) hardware coherent caches, (ii) relaxed memory consistency, (iii) software-controlled prefetching, and (iv) multiple-context support. While some studies of benefits of the individual techniques have been done, no study evaluates all of the techniques within a consistent framework. This paper attempts to remedy this by providing a comprehensive evaluation of the benefits of the four techniques, both individually and in combinations, using a consistent set of architectural assumptions. The results in this paper have been obtained using detailed simulations of a large-scale shared-memory multiprocessor. Our results show that caches and relaxed consistency uniformly improve performance. The improvements due to prefetching and multiple contexts are sizeable, but are much more application-dependent. Combinations of the various techniques generally attain better performance than each one on its own. Overall, we show that using suitable combinations of the techniques, performance can be improved by 4 to 7 times.

1 Introduction

Large-scale shared-memory multiprocessors are expected to have remote memory reference latencies of several tens to hundreds of processor cycles [18, 22, 25, 30]. The large latencies arise partly due to the increased physical dimensions of the parallel machine and partly due to the ever increasing clock rates at which the individual processors operate. These large memory latencies can quickly offset any performance gains expected from the use of parallelism. Techniques that can help to reduce or hide these latencies are essential for achieving high processor utilization.

To cope with the large latencies, several different architectural techniques have been proposed. *Coherent caches* [3, 4, 18, 30] allow shared read-write data to be cached and significantly reduce the memory latency seen by the processors. *Relaxed memory consistency models* [1, 5, 8] hide latency by allowing buffering and pipelining of memory references. *Prefetching* techniques [11, 16, 21, 23] hide the latency by bringing data close to the processor before it is actually needed. *Multiple contexts* [3, 12, 13, 26, 29] allow a processor to hide latency by switching from one context to another when a high-latency operation is encountered.

Our primary objective in this paper is to characterize the benefits and costs of these four latency hiding techniques in a systematic and consistent manner. Although one can find papers that focus on the performance of the individual techniques [7, 11, 29], it is not possible to use these papers to perform a comparative evaluation, since the benchmark programs differ, or the architectural assumptions differ, or both. We believe that a consistent comparative evaluation is essential to understanding the tradeoffs in the use of the different techniques. Furthermore, since several of the techniques can be incorporated in a multiprocessor, we have also examined the interactions and gains from the combined use of these techniques.

The results presented in this paper are obtained from detailed architectural simulations performed for three parallel applications. The architecture used is based on the Stanford DASH multiprocessor [18], a large-scale shared-memory multiprocessor that provides coherent caches, a relaxed memory consistency model, and support for software-controlled prefetching. The applications we study are a particle-based simulator used in aeronautics (MP3D) [20], an LU-decomposition program (LU), and a digital logic simulation program (PTHOR) [27]. The applications are typical of those that may be found in an engineering environment.

Our results show that the provision of coherent caches leads to significant performance benefits. For this reason, all remaining experiments in the paper were done assuming that coherent caches are provided. Our studies of the sequential consistency model versus relaxed memory consistency models show that relaxing memory consistency uniformly improves performance. Prefetching and multiple-context processors also provide performance improvements, but the magnitude varies considerably depending on the application. Combining relaxed consistency with prefetching, or relaxed consistency with multiple contexts works well. However, combining prefetching and multiple contexts is less successful and often results in lower performance than each technique achieves individually. Overall, a suitable combination of the latency reducing and tolerating techniques discussed in this paper boost performance by a factor of 4 to 7 for the applications studied.

The paper is organized into seven sections. Section 2 describes the architectural assumptions, the benchmark applications, and the simulator used in this study. Simulation results for the performance of each of the techniques are presented in Sections 3–6. Finally, we conclude in Section 7.

2 Multiprocessor Architecture, Benchmark Applications, and Simulator

To enable meaningful performance comparisons between the different techniques it is necessary to focus on a specific class of multiprocessor architectures, since the tradeoffs may vary depending on the architecture chosen. For example, the tradeoffs for a small bus-based multiprocessor where broadcast is possible and miss latencies are ten to twenty cycles are quite different from the tradeoffs for a large-scale multiprocessor where broadcast is not possible and miss latencies may be a hundred cycles. This section presents the

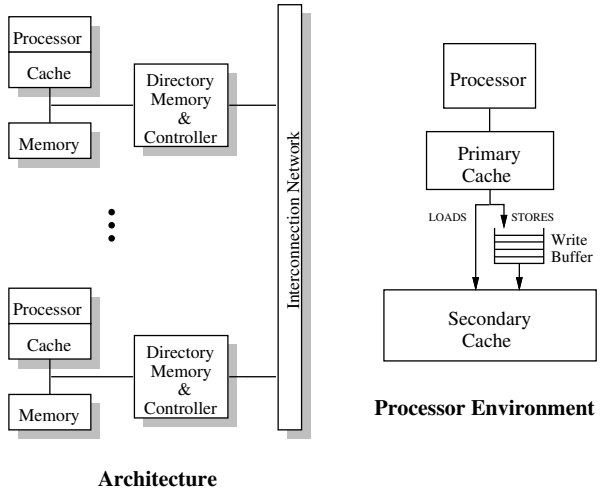

Figure 1: Architecture and processor environment.

architectural assumptions, the benchmark applications, and the simulation environment used to get the performance results.

2.1 Architectural Assumptions

For this study, we have chosen an architecture that resembles the DASH multiprocessor [18], a large-scale cache-coherent machine currently being built at Stanford. Figure 1 shows the high-level organization of the simulated architecture. The architecture consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes. Cache coherence is maintained using an invalidating, distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching it. When a write occurs, point-to-point messages are sent to invalidate remote copies of the block. Acknowledgement messages are used to inform the originating processing node when an invalidation has been completed.

For our simulations, we use the actual parameters from the DASH prototype wherever possible, but have removed some of the limitations that were imposed on the DASH prototype due to design effort constraints. Figure 1 also shows the organization of the processor environment. Each node in the system contains a 33MHz MIPS R3000/R3010 processor connected to a 64 Kbyte write-through primary data cache. The primary data cache interfaces to a 256 Kbyte secondary write-back cache. The interface consists of a read buffer and a write buffer. The write buffer is 16 entries deep. We allow reads to bypass writes in the write buffer if permitted by the memory consistency model. Both the primary and secondary caches are lockup-free [14], direct-mapped, and use 16 byte lines. The bus bandwidth of the node bus is 133 Mbytes/sec, and the peak network bandwidth is approximately 150 Mbytes/sec into and out of each node.[1]

The latency of a memory access in the simulated architecture depends on where in the memory hierarchy the access is serviced. Table 1 shows the latencies for servicing accesses at different levels of the hierarchy, in the absence of contention. The *local node* is the node that contains the processor originating a given request, while the *home node* is the node that contains the main memory and directory for the given physical memory address. A *remote node* is any node, other than the local or home node. The latency

[1]The architectural parameters and benchmark data sets in this paper differ from those in previous papers [7, 21], and therefore results cannot be directly compared.

Table 1: Latency for various memory system operations in processor clock cycles (1 pclock = 30 ns).

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	14 pclock
Fill from Local Node	26 pclock
Fill from Home Node (Home $\neq$ Local)	72 pclock
Fill from Remote Node (Remote $\neq$ Home $\neq$ Local)	90 pclock
Write Operations	
Owned by Secondary Cache	2 pclock
Owned by Local Node	18 pclock
Owned in Home Node (Home $\neq$ Local)	64 pclock
Owned in Remote Node (Remote $\neq$ Home $\neq$ Local)	82 pclock

shown for writes is the time for retiring the request from the write buffer. This latency is the time for acquiring exclusive ownership of the line, which does not necessarily include the time for receiving acknowledgement messages from invalidations.

2.2 Benchmark Programs

In this subsection we describe the computational structure of the three benchmark applications used in this paper. This information will be useful in later sections for understanding the performance results. The selected applications are representative of algorithms used in an engineering computing environment. All of the applications are written in C. The Argonne National Laboratory macro package [19] is used to provide synchronization and sharing primitives. Some general statistics for the benchmarks are shown in Table 2.

MP3D [20] is a 3-dimensional particle simulator. It is used to study the pressure and temperature profiles created as an object flies at high speed through the upper atmosphere. The primary data objects in MP3D are the particles (representing the air molecules), and the space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of particles over a sequence of time steps. During each time step, the particles are picked up one at a time and moved according to their velocity vectors. If two particles come close to each other, they may undergo a collision based on a probabilistic model. Collisions with the object and the boundaries are also modeled. The simulator is well suited to parallelization because each particle can be treated independently at each time step. The program is parallelized by statically dividing the particles equally among the processors. To minimize cache miss penalties, the particles assigned to a processor are allocated from shared memory in that processor's node. The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 10,000 particles, a 14x24x7 space array, and simulated 5 time steps.

LU performs LU-decomposition of dense matrices. The primary data structure in LU is the matrix being decomposed. Working from left to right, a column is used to modify all columns to its right. Once a columns has been modified by all columns to its left, it can be used to modify the remaining columns. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits until a column has been produced, and then uses that column to modify all columns that the processor owns. The main memory for storing columns that are owned by a processor is allocated from shared memory in that processor's node to reduce cache miss penalties. Once a processor completes a column, it releases any processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

PTHOR [27] is a parallel logic simulator based on the Chandy-Misra distributed-time simulation algorithm. The primary data

Table 2: General statistics for the benchmarks.

Program	Useful Cycles *(K)*	Shared Reads *(K)*	Shared Writes *(K)*	Locks	Barriers	Shared Data Size *(Kbytes)*
MP3D	5,774	1170	530	0	448	401
LU	27,861	5543	2727	3184	29	653
PTHOR	19,031	3774	454	75,878	2016	2925

structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then schedules the newly activated elements onto task queues. In the case that a processor runs out of tasks, it spins on the task queues until a new task is scheduled. This time shows up as busy time in our experiments, even though it should rightfully be counted as synchronization time. This fact leads to variations in busy time from experiment to experiment, even though the amount of useful work being done remains approximately the same. For our experiments we simulated five clock cycles of a small RISC processor consisting of the equivalent of 11,000 two-input gates.

2.3 Simulation Environment

An event-driven simulator is used to simulate the architecture at the behavioral level. The caches, the cache coherence protocol, the contention and arbitration for buses, are all modeled in detail. The simulations are based on a 16 processor configuration. We do not go beyond 16 processors since the concurrency requirements are very large for multiple context simulations. When modeling 4 hardware contexts per processor, 16 processors require the application to support 64 concurrent processes. However, some of our applications do not scale well to that many processes given the small data sets that we can simulate. The architecture simulator is tightly coupled to the Tango reference generator [9] to assure a correct interleaving of accesses. For example, a process doing a read operation is blocked until that read completes, where the latency of the read is determined by the architecture simulator. Unless specific directives are given by an application, main memory is distributed uniformly across all nodes using a round-robin page allocation scheme.

We now come to a difficult methodological problem that shows up when simulating large multiprocessors. Given that detailed simulators are enormously slower than the real machines being simulated, one can only afford to simulate much smaller problems than those that would be run on the real machine. The question arises of how to scale the machine parameters to get realistic performance estimates. Consider the MP3D application. In real life, the application is run with enough particles to fill the complete main memory of a machine. Since at each time step in the application all the particles are moved (i.e., the complete memory is swept through), the caches are expected to miss on each particle. If we use the 64 Kbyte primary and 256 Kbyte secondary caches in the simulator, then we would have to run MP3D with at least 125,000 particles to achieve realistic cache behavior. This would take extremely long to run.

We see no easy answer to this scaling question. For this study, we have chosen to scale down the cache sizes to obtain a more realistic ratio between problem size and cache size. We scale down the processor caches to 2 Kbyte primary and 4 Kbyte secondary caches.[2] For MP3D, we get miss ratios approximating a large problem with full-size caches by using only 10,000 particles and thus reduce the simulation time substantially. For LU, the data set size is chosen such that the data starts fitting into the combined caches of the processors only when the bottom third of the matrix remains to be factored. As a result, the processors get poor cache hit ratio in the beginning, and high hit ratios towards the end. This kind of behavior is typical of many numerical applications. For PTHOR, our experiments use a circuit with 11,000 gates. This gives us miss ratios comparable to a real machine simulating circuits with hundreds of thousands of gates. To explore the impact of cache size variations, we performed all of the experiments reported in this paper with the full-sized caches. We are unable to present these results due to lack of space. However, the results were similar to those obtained with the scaled caches. While the absolute execution times decreased with the larger caches, the relative gains from the various techniques were similar.

[2]These caches are only used for shared data. Instruction and private data references are not sent to the cache simulator and are implicitly assumed to hit in the cache.

3 Coherent Caches

The first of the four techniques that we study is caching of shared data. The use of processor caches is a well accepted technique for reducing latencies in uniprocessors. Their use in multiprocessors, however, is complicated by the fact that the caches need to be kept coherent. While the coherence problem is easily solved for small bus-based multiprocessors through the use of snoopy cache-coherence protocols [4], the problem is much more complicated for large-scale multiprocessors that use general interconnection networks. As a result, some existing large-scale multiprocessors do not provide caches (e.g., BBN Butterfly [25]), others provide caches that must be kept coherent by software (e.g., IBM RP3 [22]), and still others provide full hardware support for coherent caches (e.g., Stanford DASH [18]). In this section we evaluate the performance benefits when both private and shared read-write data are cacheable as allowed by hardware coherent caches versus the case when only private data are cacheable. An alternative to hardware coherence is software cache coherence. Due to lack of appropriate compiler technology, we could not evaluate the effectiveness of software schemes.

Figure 2 presents a breakdown of the normalized execution times with and without caching of shared data for each of the applications. Private data are cached in both cases. The latencies for non-cached shared data are five to ten cycles less than those in Table 1. This is because the caches are bypassed and there is no fill overhead on such accesses. The experiments assume the sequential consistency model, so that no buffering or pipelining of cache misses is allowed. The execution time of each application is normalized to the execution time of the case where shared data is not cached. The bottom section of each bar represents the busy time or useful cycles executed by the processor. The section above it represents the time that the processor is stalled waiting for reads. The section above that is the amount of time the processor is stalled waiting for writes to be completed. The top section, labeled synchronization, accounts for the time the processor is stalled due to locks and barriers.

As expected, the caching of shared read-write data provides substantial gains in performance, with benefits ranging from 2.2 to 2.7 fold improvement for the three programs. The largest benefit comes from a reduction in the number of cycles wasted due to read misses. The cycles wasted due to write misses are also reduced, although the magnitude of the benefits varies across the three programs due to different write hit rates. The cache hit rates achieved by MP3D, LU, and PTHOR are 80%, 66%, and 77% respectively for shared-read references, and 75%, 97%, and 47% for shared-write references.[3] It is interesting to note that these hit rates are substantially lower than the usual uniprocessor hit rates. The low hit rates arise from sev-

[3]The results for full caches achieved by MP3D, LU, and PTHOR are 82%, 76%, and 86% for shared reads, and 75%, 99%, and 52% for shared writes. MP3D shows the least gain from the larger caches since the majority of misses are inherent communication misses.

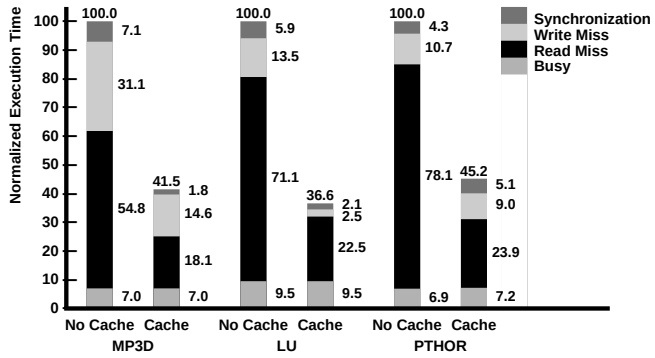

Figure 2: Effect of caching shared data.

eral factors: the data set size for engineering applications is large, parallelism decreases spatial locality in the application, and communication among processors results in invalidation misses. Still, hardware cache coherence is an effective technique for substantially increasing the performance with no assistance from the compiler or programmer.

Although caching shared data improves the performance substantially, the large number of cache misses and the large latency of each miss still keep the processor utilizations low (about 17% for MP3D, 26% for LU, and 16% for PTHOR). The next three sections study the effect of three different techniques for dealing with the large latency of cache misses by overlapping them with other computation and memory accesses. We assume hardware coherent caches for the rest of this study.

4 Relaxing the Memory Consistency Model

One way to remedy the large latency of cache misses is to hide the latency of accesses by buffering and pipelining the misses. Unfortunately, as a result of the combination of distributed memory, caches, and general interconnection networks used by large-scale multiprocessors [3, 18, 22], multiple requests issued by a processor may execute out of order. This may result in incorrect program behavior if the program depends on accesses to complete in order. Consequently, restrictions have to be placed on the types of buffering and pipelining allowed. These restrictions are determined by the memory consistency model supported by the multiprocessor.

Several memory consistency models have been proposed. The strictest model is that of *sequential consistency* (SC) [15]. It requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. Unfortunately, SC imposes severe restrictions on the outstanding accesses that a process may have, and thus limits the buffering and pipelining allowed. One of the most relaxed models is the *release consistency* (RC) model [8]. Release consistency requires that synchronization accesses in the program be identified and classified as either *acquires* (e.g., locks) or *releases* (e.g., unlocks). An acquire is a read operation (which can be part of a read-modify-write) that gains permission to access a set of data, while a release is a write operation that gives away such permission. This information is used to provide flexibility in buffering and pipelining of accesses between synchronization points. The main advantage of the relaxed models is the potential for increased performance. The main disadvantage is increased hardware complexity and a more complex programming model.

Other relaxed models that have been discussed in the literature are *processor consistency* [8, 10], *weak consistency* [5], and *DRF0* [1]. These models fall between sequential and release consistency models in terms of flexibility and are not considered further in this study. For a detailed performance evaluation of relaxed memory consistency models, we refer the reader to a previous study [7].

4.1 Implementation of Consistency Schemes

Sequential consistency is satisfied in our implementation by ensuring that the memory accesses from each process complete in the order that they appear in the program. This is achieved by delaying the issue of an access until the previous access completes. The processors used in this study already stall on reads until the read access is satisfied. In addition, under SC, we explicitly stall the processor after every write until the write completes.

Release consistency can be satisfied by (i) stalling the processor on an acquire access until it completes and (ii) delaying the completion of a release access until all previous memory accesses complete. In the implementation assumed in this paper, the first condition is automatically satisfied because the processor stalls on all read accesses (including acquires) until the read is complete. To satisfy the second condition for RC, the write buffer is stalled on a release access until previously issued writes complete. To fully realize the benefits of RC, we allow reads to bypass the write buffer and provide lockup-free caches that allow reads to be serviced while there are write misses outstanding [7]. This ensures that reads are not stalled due to previous writes. The lockup-free cache also allows multiple write accesses to be pipelined.

Although the conditions for satisfying RC allow accesses and computation following a read to be overlapped and pipelined with the read, the implementation we study does not allow such overlap since reads are blocking. The design of processors that allow multiple outstanding reads and out-of-order execution of instructions is a current topic of research. However, the feasibility of such processors in addition to their effectiveness in hiding the latency of reads is still an open question.

The cost of implementing RC over SC arises from the extra hardware cost of providing a lockup-free cache and keeping track of multiple outstanding requests. Although this cost is not negligible, the same hardware features are also required to support prefetching and multiple contexts.

4.2 Comparison of SC versus RC

Figure 3 presents the breakdown of execution times under SC and RC for the three applications. The execution times are normalized to those shown in Figure 2 with shared data cached. As can be seen from the results, RC removes all idle time due to write miss latency. The gains are large in MP3D and PTHOR since the write-miss time constitutes a large portion of the execution time under SC (35% and 20%, respectively), while the gain is small in LU due to the relatively small write-miss time under SC (7%).

The pipelining of writes under RC provides another way in which RC can outperform SC. If there is a release operation (e.g., unlock) behind several writes in the write buffer, then a remote processor waiting at an acquire (e.g., lock on the same variable) can observe the release sooner under RC, thus spinning for a shorter amount of time. Indeed, Figure 3 shows that synchronization times do decrease under RC. Overall, the release consistency model provides a speedup over sequential consistency of about 1.5 for MP3D, 1.1 for LU, and 1.4 for PTHOR.

While relaxing the memory consistency model effectively hides the latency of write accesses, the latency of read misses still remains. This is mostly due to the fact that the processors we assume in our study stall on reads and do not allow read misses to be overlapped with future computation and memory accesses. Because read miss times constitute a large portion of the execution time (especially when the write miss time is removed), there is an opportunity for

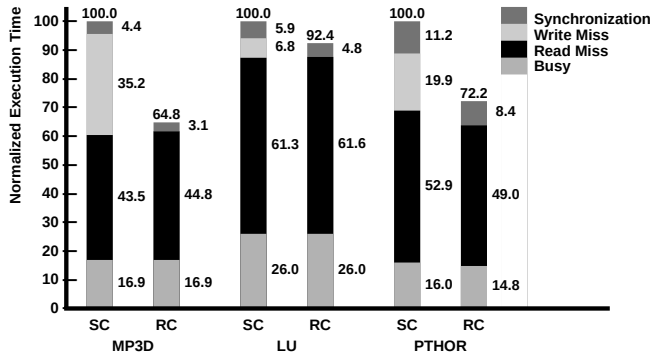

Figure 3: Effect of relaxing the consistency model.

large performance gains from techniques that can hide the read miss latency. Indeed, the prefetching and multiple context techniques discussed in the next two sections attain most of their benefit by hiding the latency of reads.

5 Prefetching

Prefetching uses knowledge about the expected misses in a program to move the corresponding data close to the processor before it is actually needed. Prefetching can be classified based on whether it is *binding* or *non-binding*, and whether it is controlled by *hardware* or *software*. With binding prefetching, the value of a later reference (e.g., a register load) is bound at the time when the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the same location during the interval between prefetch and reference. Binding prefetching studies done by Lee [17] reported significant performance loss due to such limitations. In contrast, non-binding prefetching also brings the data close to the processor, but the data remains visible to the cache coherence protocol and is thus kept consistent until the processor actually reads the value. Hardware-controlled prefetching includes schemes such as long cache lines and instruction look-ahead [16]. The effectiveness of long cache lines is limited by the reduced spatial locality in multiprocessor applications [6, 28], while instruction look-ahead is limited by branches and the finite look-ahead buffer size. With software-controlled prefetching, explicit prefetch instructions are issued. Software control allows the prefetching to be done selectively (thus reducing bandwidth requirements) and extends the possible interval between prefetch issue and actual reference, which is very important when latencies are large. The disadvantages of software control include the extra instruction overhead to generate the prefetches as well as the need for sophisticated software intervention. In this study, we consider *non-binding software-controlled prefetching* [21].

The benefits due to prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough in the code, so that the line is already in the cache by the time it is referenced. However, prefetching can improve performance even when this is not possible (e.g., when the address of a data structure cannot be determined until immediately before it is referenced). If multiple prefetches are issued back-to-back to fetch the data structure, the latency of all but the first prefetched reference can be hidden due to the pipelining of the memory accesses. Prefetching offers another benefit in multiprocessors that use an ownership-based cache coherence protocol [4]. If a line is to be modified, prefetching it directly with ownership can significantly reduce the write latencies and the ensuing network traffic for obtaining ownership. Network traffic is reduced in read-modify-write situations, since prefetching with ownership avoids first fetching a read-shared copy.

5.1 Prefetching Implementation and Assumptions

In our model, a prefetch instruction is similar to a write in that it is issued to a prefetch buffer (which is identical to a write buffer, except that it only handles prefetch requests) and does not block the processor. The reason for having a separate prefetch buffer is to avoid delaying prefetch requests unnecessarily behind writes in the write buffer [21]. We model a prefetch buffer that is 16 entries deep. Once the prefetch reaches the head of the prefetch buffer, the secondary cache is checked to see whether the line is already present. If so, the prefetch is discarded. Otherwise the prefetch is issued onto the bus, where it is treated like any normal memory request. When the prefetch response returns to the processor, it is placed in both the secondary and primary caches. If the processor is executing when this cache fill begins, it is stalled for four cycles (since the cache line size is four words) to model the effect that no loads or stores can be executed while the cache is busy. If a processor references a location it has prefetched before the result has returned, the reference request is combined with the prefetch request so that a duplicate set of messages is not sent out and so that the reference completes as soon as the prefetch result returns.

Since we did not want to be constrained by the limits of existing compiler technology to automatically add prefetching, and because such a compiler was not available to us, prefetches were introduced manually at the source level of each application through macro statements. These macros covered both *read* and *read-exclusive* prefetches, as well as single cache line and block prefetches. A read prefetch brings data into the cache in a read-shared mode, while a read-exclusive prefetch also acquires exclusive ownership of the line, enabling a write to that location to complete quickly.

5.2 Prefetching Results

We begin with a description of how prefetching was inserted into each application, and then discuss the results for both sequential and release consistency.

MP3D: Most of the time is spent in a loop where each processor takes a particle and moves it through one time step. The overwhelming majority of cache misses are caused by references to two structures within this loop: (i) the particle which is being moved (34% of misses), and (ii) the space cell where the particle resides (50%). Particles are statically assigned to processors and are allocated from the shared memory local to each processor, while the memory for the space cells is distributed uniformly among the processors.

Since a particle must be referenced to determine the space cell it occupies, we prefetch a particle record two iterations before its turn to be moved. In the iteration following the prefetch, the particle is read, and the associated space cell is determined and prefetched. As a result, when it is time for the particle to be moved, both the particle and space cell records are available in the cache. We also prefetch several other references that occur at time step boundaries. The end result is that prefetches are issued for 87% of the misses that occur in the version without prefetching (we will refer to this as the *coverage factor*). Read-exclusive prefetches are used since the objects are modified during each iteration. Introducing these prefetches required adding 16 lines to the source code.

LU: The matrix columns are statically assigned to the processors in an interleaved manner, and are allocated from the corresponding local memories. The main computation done by each processor consists of reading a pivot column once it is produced,

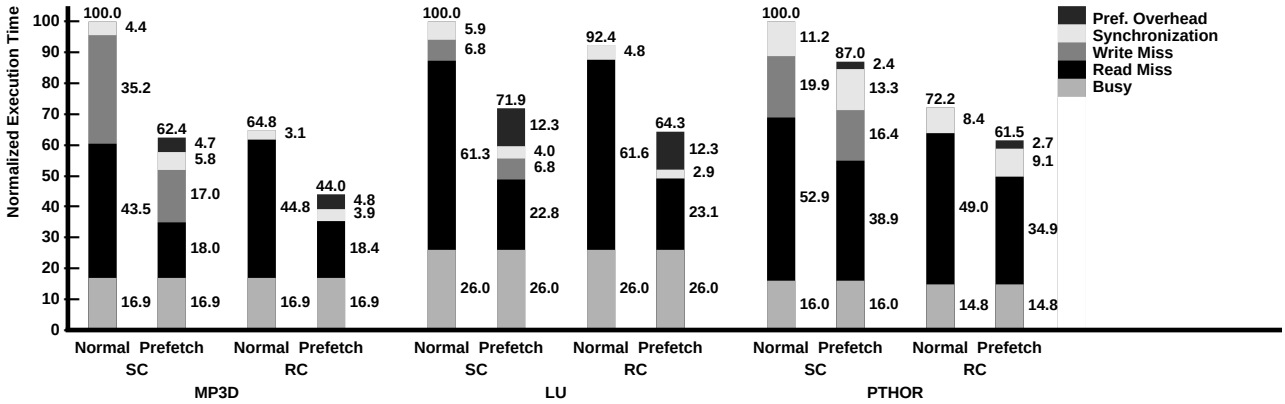

Figure 4: Effect of prefetching.

and applying the pivot column to each column to its right that the processor owns. There are three primary sources of misses in LU: (i) the pivot column when it is read for the first time (8%); (ii) the pivot column when it is replaced by a column it is applied to and needs to be refetched (17%); and (iii) the owned columns that the pivot column is applied to (64%). This last set of misses occurs because the combined size of the owned columns is larger than the size of the cache.

Each time the pivot column is applied to an owned column, we prefetch the pivot column in read-shared mode and the owned column in read-exclusive mode. Although prefetching the pivot column each time causes redundant prefetches, it reduces the misses when the pivot column is replaced from the processor's cache, resulting in a total coverage factor of 89%. We found that it is better to evenly distribute the issue of prefetches throughout the computation rather than prefetching an entire column in a single burst, in order to avoid hot-spotting problems. A total of 8 lines were added to the source code.

PTHOR: In the main computational loop, each processor picks up an activated logic element, computes any changes to the element's outputs, and schedules new input events for elements that are affected by the changes. One of the main data structures in the program is the *element record*, which stores all information about the type and state of the element. Several fields in the record are pointers to linked lists, or are pointers to arrays that in turn point to linked lists. Prefetching is complicated by the presence of linked lists, since to prefetch a list it is necessary to dereference each pointer along the way.

We first reorganized the element record and grouped entries based on whether they were likely to be modified, likely to be read but not modified, or likely not to be referenced. Whenever a processor picks an element from a task queue, we prefetch the element record entries accordingly. In addition, we prefetch the first several levels of the more important linked lists. Due to the complex control structure of the application, it is difficult to determine where the misses occur. Despite the aid of profiling markers that helped determine which sections of code were generating misses, we were only able to increase the coverage factor to 56%. A total of 29 lines were added to the source code.

The results of the prefetching experiments are shown in Figure 4. Notice that a new section has been added to the execution time bar to account for prefetching overhead. This includes any extra instructions executed to do prefetching (e.g., evaluation of conditional statements that help decide whether to prefetch or not, instructions to do address computation, and the prefetch instruction itself), any time for which issuing a prefetch stalls the processor due to a full prefetch buffer, and any stall time due to the primary cache being filled with a prefetched line.

For sequential consistency we see that most of the benefit comes from reduced read latencies, and that this more than offsets the added prefetch overhead. While read-exclusive prefetching effectively reduces write latencies for MP3D, it offers little or no improvement for LU (where write latencies are already small because owned columns are allocated from local shared memory) and PTHOR (where prefetches are issued for only a small fraction of writes). Prefetch overhead is substantial in the case of LU since there is very little computation between references, causing the prefetch generation instructions to be a large fraction of total instructions. The overhead due to primary cache fills is much less of a problem. The main difference we see when prefetching is combined with release consistency is that the write latency has already been eliminated, so the benefits come strictly through reduced read latency.

The benefits of prefetching are limited by several factors. First, inserting the prefetches can be difficult. This was especially true for PTHOR. The difficulty is both in identifying the references that need to be prefetched and in scheduling the prefetches far enough in advance to effectively hide latency. We are currently working on compiler technology to automate this process. Secondly, even if a reference is prefetched far enough in advance, cache interference may cause it to be knocked out of the cache before it is referenced. This interference can be either self-interference in the form of replacements or external interference caused by invalidations. The effects of self-interference were noticeable for LU, while MP3D was affected by external interference. Finally, the overhead of adding prefetches can potentially offset much of the gain that is realized through reduced latencies, as we see in the case of LU.

The advantage of prefetching is that significant gains can be achieved by inserting only a handful of prefetches when the access patterns are regular and predictable. For MP3D, adding only 16 lines to the source code resulted in speedups of 1.60 and 1.47 under SC and RC, respectively. Another important advantage of prefetching, in contrast to other latency hiding techniques such as multiple contexts, is that it can be implemented using existing commercial processors, as has been done in DASH [18].

6 Multiple-Context Processors

Although prefetching is useful for many applications, it requires explicit programmer or compiler intervention. Processors with multiple hardware contexts [3, 12, 13, 26, 29] do not have this disadvantage. They make use of increased concurrency to hide latency. Each processor has several processes assigned to it, which are kept

as hardware contexts. When the context that is currently running encounters a long-latency operation, it is switched out and another context starts executing. In this manner the memory latency of one context can be hidden with computation of another context. Given processor caches, the interval between long-latency operations (i.e., cache misses) becomes fairly large, allowing just a handful of hardware contexts to hide most of the latency [29]. This is in contrast to the early multiple-context processors such as the HEP [26], where context switches occurred on every cycle.

The performance gain to be expected from multiple context processors depends on several factors. First, there is the number of contexts. With more contexts available, we are less likely to be out of ready-to-run contexts. On the other hand, the number of contexts is constrained by hardware costs and available parallelism in the application. Secondly, there is the context switch overhead. If the overhead is a sizeable fraction of the typical run lengths (time between misses), a significant fraction of time may be wasted switching contexts. Shorter context switch times, however, require a more complex and possibly slower processor. Thirdly, the performance depends on the application behavior. Applications with clustered misses and irregular miss latencies will make it difficult to completely overlap computation of one context with memory accesses of other contexts. Multiple context processors will thus achieve a lower processor utilization on these programs than on applications with more regular miss behavior. Lastly, multiple contexts themselves affect the performance of the memory subsystem. The different contexts share a single processor cache and can interfere with each other, both constructively and destructively. Also, as is the case with release consistency and prefetching, the memory system is more heavily loaded by multiple contexts, and thus latencies may increase.

We presented a preliminary investigation of the benefits of multiple-context processors in cache-coherent multiprocessors in a previous study [29]. More recently, there have also been two analytical evaluations of multiple contexts [2, 24]. In this study we present a more detailed evaluation of the performance of multiple-context processors, and we also consider the combined effect with other latency hiding techniques. We use processors with two and four contexts. We do not consider more contexts per processor because 16 4-context processors require 64 parallel threads and some of our applications do not achieve very good speedup with that many threads. We use two different context switch overheads: 4 and 16 cycles. A four-cycle context switch overhead corresponds to flushing/loading a short RISC pipeline when switching to the new instruction stream. An overhead of sixteen cycles corresponds to a less aggressive implementation. In our study, we include additional buffers to avoid thrashing and deadlock when two contexts try to read distinct memory lines that map to the same cache line.

6.1 Results with Multiple-Context Processors

We start our investigation of multiple contexts with an evaluation of their benefit under sequential consistency. Later we will examine the combined benefit when the consistency model is relaxed and prefetching is added.

Figure 5 shows performance results for 1, 2, and 4-context processors with context switching penalties of 4 and 16 cycles. Each bar in the graphs is broken down into the following components: *busy* time which represents actual work done by the processor, *switching* time incurred when switching from one context to the next, *all idle* time which is the total time when all contexts are idle waiting for a reference to complete, and *no switch* time which represents time when the current context is idle but is not switched out. Most of the latter idle time is due to the fact that the processor is locked out of the primary cache while fill operations of other contexts complete. Under sequential consistency, some of the *no switch* idle time is due to write hits in the secondary cache, which stall the processor for two cycles.

MP3D benefits greatly from the use of multiple contexts (top graph of Figure 5). The median run lengths are about 11 cycles long, and the average read miss latencies are about 50 cycles long. With a context switch overhead of four cycles, we expect 5 contexts to hide most of the miss latency. With two contexts we see some reduction in *all idle* time and with four contexts an additional portion of this idle time is eliminated. However, with multiple contexts we now have additional idle time in the form of context switch overhead. This time is especially significant when the context switch overhead is 16 cycles. It is interesting to note that with two contexts there is very little performance improvement in going from a switch penalty of 16 cycles to one of 4 cycles. The context switch time saved simply shows up as additional *all idle* time. The latencies have not changed much, but were partially masked by switch overhead when the switch latency was 16 cycles.

The behavior of LU (middle of Figure 5) is completely dominated by cache interference. With a single context, the read and write hit rates are 66% and 97% respectively. With two contexts they deteriorate to 56% and 38%, and with 4 contexts they are down to 50% and 16%. These additional misses lead to more context switches and more time wasted on context switching. With 16 cycle context switch overhead, performance gets worse as more contexts are added. Even though some of the latency is hidden, the time wasted on context switches dominates. With the 4 cycle context switch overhead, some gains are possible. The median run lengths are 6 cycles long, and the average miss latencies are 20–27 cycles long. The miss latencies are low because a high proportion of them are due to the owned columns of the matrix, which are allocated from the local portion of shared memory.

PTHOR (bottom of Figure 5) shows another interesting effect. There is not enough parallelism available in the application to achieve good speedup with a large number of processors or contexts. So even though the run lengths and latencies (7 and 60–80 cycles respectively) suggest good multiple-context behavior, the gains achieved with two contexts are small. Four contexts actually do worse than two, no matter what the context switch overhead is. Since there is not enough parallelism available, the additional contexts spend most of their time busy-waiting on an empty task queue. The additional instruction cycles used for spinning on the task queue are reflected in the graphs as extra busy time. During this time they hold up the useful work being done by the other contexts that did manage to find a task. We note that when PTHOR is run with only four processors instead of sixteen, multiple contexts achieve much greater gains: four context-processors run about twice as fast as single-context processors.

The conclusion from the results of our experiments with multiple contexts under sequential consistency is that multiple contexts can increase performance significantly when the run length to latency ratio is favorable. However, enough parallelism must be available in the application to keep the additional contexts busy. We further observe that destructive interference of the contexts in the processor cache can undo any gains achieved. Interference is more of a problem with multiple contexts than with prefetching because multiple working sets interfere with each other in the same cache. The smaller the number of cycles required for context switching, the lower the total overhead due to multiple contexts. A context switch cost of 16 cycles introduces significant overhead, whereas the overhead is much more reasonable with a 4-cycle switch penalty.

6.2 Effect of Combining other Schemes with Multiple Contexts

We have seen that multiple contexts with sequential consistency can increase performance substantially under favorable circumstances. An interesting question is whether multiple contexts can gain any extra performance when combined with relaxed consistency mod-

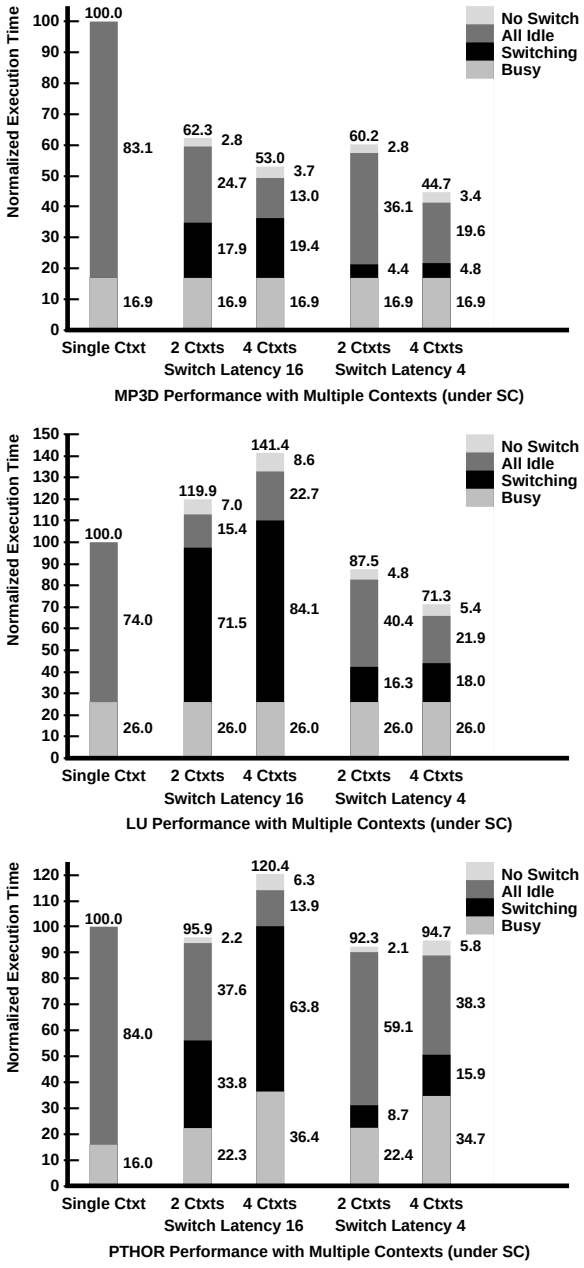

Figure 5: Effect of multiple contexts.

els. The left and middle sections of the graphs in Figure 6 show the performance of multiple contexts with SC and RC, respectively. We only show results for a context-switch overhead of 4 cycles. The major difference between release consistency and sequential consistency is that write misses are no longer considered long latency operations from the processor's perspective, since writes are simply put into the write buffer. (Note: A decomposition showing write latencies with single contexts can be found in Figure 3.) We thus find that median run lengths between switches have increased (from 11 to 22 cycles for MP3D and from 6 to 14 cycles for LU, PTHOR is unchanged) and that fewer contexts are required to eliminate most of the remaining read miss latency. As a result, the gains achieved with four contexts over two contexts are also diminished. As is apparent from the results, there is some benefit from relaxing the consistency model with multiple contexts. For the 4-context case, performance improved by a factor of 1.32 for MP3D, 1.23 for LU, and 1.21 for PTHOR when going from SC to RC.

Finally, let us consider the combined effect of multiple contexts and prefetching (see the right portion of the graphs in Figure 6). The main benefit of combining the two, of course, is that each scheme can compensate for the other scheme's weaknesses. For example, prefetching can increase the hit rate, thus increasing the run lengths and ensuring that a small number of contexts suffice. Similarly, multiple contexts can ensure that the processor does not remain idle for misses where prefetching was not effective. However, the two schemes also have negative interactions. First, both prefetching and multiple contexts add overhead. So if the latency of a reference could be totally hidden by one scheme alone, the second one only contributes overhead. Secondly, the two techniques may interfere with each other. For example, when multiple contexts are used, the time between issue and use of a prefetch may increase substantially, thus increasing the chance of the prefetched data being invalidated or replaced from the cache before being referenced. Depending on the relative magnitudes of the above effects, the performance of an application may increase or decrease when both schemes are used. Our data show that when four contexts are used, the negative effects overwhelm the positive effects (see Figure 6) for all applications. However, when only two contexts are used, additional use of prefetching helps achieve higher performance than when only a single context with prefetching is used or when two contexts without prefetching are used.

Although prefetching and multiple contexts both attempt to hide the same latency, there are several distinguishing features. The major advantage of prefetching is that it does not require a special processor. Also, it allows many more accesses to be outstanding at any given time, thus permitting their latencies to be overlapped. With prefetching, each processor can issue an essentially unlimited number of prefetch requests. Multiple contexts, on the other hand, are limited by the total number of contexts, which is expected to be a small number. The advantage of multiple contexts is that they can handle very irregular access patterns which cannot be prefetched efficiently. Another significant advantage is that multiple contexts do not require software support.

In our study, when we added prefetching to the applications, we did not have multiple contexts in mind. We believe this has had some negative impact on the results for combined prefetching and multiple contexts. For example, for a single-context processor, it is reasonable to be quite aggressive and add prefetches in situations where we expect only a small portion of the latency to be hidden. However, for a multiple-context processor, this may be a bad decision. If the multiple context processor would have hidden the latency anyway, prefetch overhead has been added without any benefit.

In summary, release consistency helps multiple contexts because it eliminates writes as long latency operations, thus increasing run lengths and allowing the remaining latency to be hidden with fewer contexts. The benefit of adding prefetching to multiple contexts is small, and may even be negative, especially when little latency is left to hide. Inserting prefetches with more awareness of their effect on the performance of multiple contexts may achieve better results.

7 Concluding Remarks

While several latency hiding techniques have been proposed in the past, a study evaluating the relative performance benefits of these techniques and their combinations had been lacking. In this paper, we have presented such an evaluation for four techniques—coherent caches, relaxed memory consistency, prefetching, and multiple contexts—using a common set of architectural assumptions and benchmarks. As expected, the largest single improvement in runtime, a factor of 2.2 to 2.7, came from coherent caches. Relaxing the consistency model provided additional gains by a factor of 1.1 to

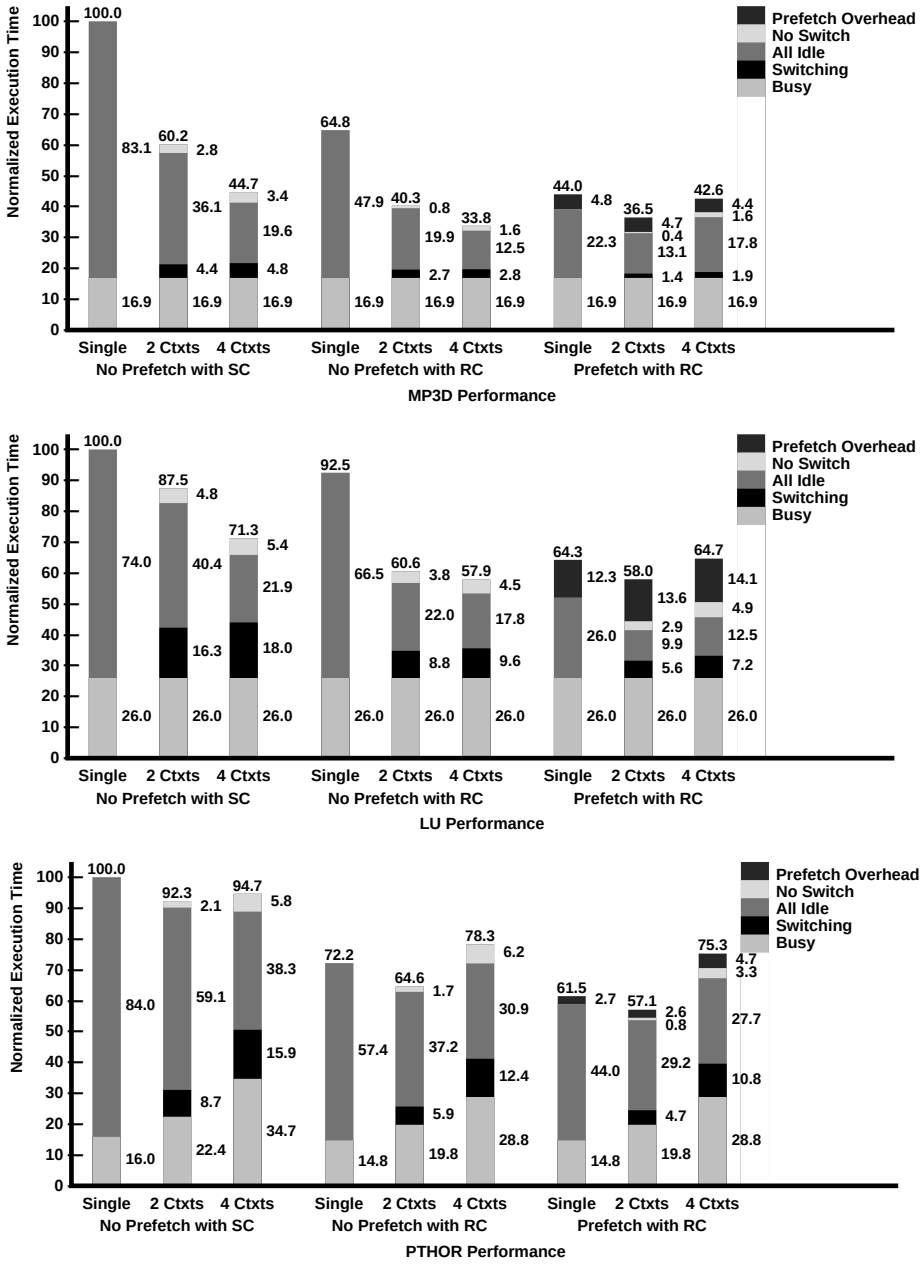

Figure 6: Effect of combining the schemes (multiple-context schemes have a 4-cycle switch latency).

1.5, arising mainly from the hiding of write latencies. Similar to the gain from caches, this gain is automatic as long as programs use explicit synchronization. The main hardware requirement (in addition to coherent caches) is lock-up free caches, since the relaxed models allow multiple outstanding references. Lock-up free caches, however, are also necessary for prefetching and for multiple-context processors, and thus form a universal requirement for latency hiding techniques.

As intended, prefetching was very successful in reducing the stalls due to read latencies (26%–63% less). Prefetching was less effective in reducing write latency under the strict consistency model, but combined well with the relaxed consistency model to eliminate both read and write latency. The speedups of relaxed consistency and prefetching over sequential consistency were 2.3 for MP3D, 1.6 for LU, and 1.6 for PTHOR. While prefetching has the drawback that it requires compiler or programmer intervention, a significant advantage is that it requires no major hardware support beyond that needed by RC, and it can easily be incorporated into systems built using existing commercial microprocessors.

The multiple context approach, while requiring significant hardware support, provided mixed results when the context-switch overhead was 16 cycles. In cases where the concurrency was low (e.g., PTHOR) or where there was substantial cache interference (e.g., LU), the use of multiple contexts made the performance worse. The use of relaxed consistency helped multiple context performance by hiding write latencies and increasing the run lengths. Under an aggressive implementation, with use of 4 contexts and a context-switch overhead of 4 cycles, the performance benefits were a factor of 3.0 for MP3D, 1.7 for LU, and 1.3 for PTHOR. The interaction of multiple contexts with prefetching was shown to be complex. Oftentimes the performance became worse when the two were combined together. To achieve better results, it appears that the prefetching

strategy must become more sensitive to the presence of multiple contexts.

8 Acknowledgments

We thank the reviewers for their comments. This research was supported by DARPA contract N00014-87-K-0828. Anoop Gupta is partially supported by a NSF Presidential Young Investigator Award with matching funds from Sumitomo, Tandem, and TRW. Wolf-Dietrich Weber is partially supported by IBM and Kourosh Gharachorloo is partially supported by Texas Instruments.

References

[1] S. Adve and M. Hill. Weak ordering - A new definition. In *Proc. Int. Symp. Comput. Arch.*, pages 2–14, May 1990.

[2] A. Agarwal. Performance tradeoffs in multithreaded processors. MIT VLSI Memo 89-566, Lab. for Comput. Sci., Submitted for publication, September 1989.

[3] A. Agarwal, B.-H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proc. Int. Symp. Comput. Arch.*, pages 104–114, May 1990.

[4] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. Comput. Syst.*, 4(4):273–298, 1986.

[5] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 434–442, June 1986.

[6] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proc. Int. Symp. Comput. Arch.*, pages 2–15, May 1989.

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Int. Conf. Arch. Support Prog. Lang. Oper. Syst.*, April 1991.

[8] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 15–26, May 1990.

[9] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[10] J. R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[11] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Int. Conf. Supercomputing*, pages 354–368, 1990.

[12] R. H. Halstead, Jr. and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proc. Int. Symp. Comput. Arch.*, pages 443–451, June 1988.

[13] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proc. Int. Symp. Comput. Arch.*, pages 131–140, June 1988.

[14] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proc. Int. Symp. Comput. Arch.*, pages 81–85, 1981.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):241–248, September 1979.

[16] R. L. Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors*. PhD thesis, University of Illinois at Urbana-Champaign, May 1987.

[17] R. L. Lee, P.-C. Yew, and D. H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proc. Int. Conf. Paral. Proc.*, pages 28–31, August 1987.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. Int. Symp. Comput. Arch.*, pages 148–159, May 1990.

[19] E. Lusk, R. Overbeek, et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[20] J. D. McDonald and D. Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[21] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Paral. Distr. Computing*, to appear in June 1991.

[22] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proc. Int. Conf. Paral. Proc.*, pages 764–771, 1985.

[23] A. K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[24] R. H. Saavedra-Barrera, D. E. Culler, and T. von Eicken. Analysis of multithreaded architectures for parallel computing. In *ACM Symp. Paral. Alg. Arch.*, July 1990.

[25] G. E. Schmidt. The Butterfly parallel processor. In *Proc. Int. Conf. Supercomputing*, pages 362–365, 1987.

[26] B. J. Smith. Architecture and applications of the HEP multiprocessor computer system. *SPIE*, 298:241–248, 1981.

[27] L. Soule and A. Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[28] J. Torrellas, M. S. Lam, and J. L. Hennessy. Measurement, analysis, and improvement of the cache behavior of shared data in cache coherent multiprocessors. Technical Report CSL-TR-90-412, Stanford University, Feb. 1990.

[29] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proc. Int. Symp. Comput. Arch.*, pages 273–280, June 1989.

[30] A. W. Wilson, Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proc. Int. Symp. Comput. Arch.*, pages 244–252, June 1987.