Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors

Todd Mowry and Anoop Gupta
Computer Systems Laboratory
Stanford University, CA 94305

To appear in the Journal of Parallel and Distributed Computing, June 1991.

Abstract

The large latency of memory accesses is a major obstacle in obtaining high processor utilization in large scale shared-memory multiprocessors. Although the provision of coherent caches in many recent machines has alleviated the problem somewhat, cache misses still occur frequently enough that they significantly lower performance. In this paper we evaluate the effectiveness of *non-binding software-controlled prefetching*, as proposed in the Stanford DASH Multiprocessor, to address this problem. The prefetches are non-binding in the sense that the prefetched data is brought to a cache close to the processor, but is still available to the cache coherence protocol to keep it consistent. Prefetching is software-controlled since the program must explicitly issue prefetch instructions.

The paper presents results from detailed simulation studies done in the context of the Stanford DASH multiprocessor. Our results show that for applications with regular data access patterns—we evaluate a particle-based simulator used in aeronautics and an LU-decomposition application—prefetching can be very effective. It was easy to augment the applications to do prefetching and it increased their performance by 100-150% when we prefetched directly into the processor's cache. However, for applications with complex data usage patterns, prefetching was less successful. After much effort, the performance of a distributed-time logic simulation application that made extensive use of pointers and linked lists could be increased only by 30%. The paper also evaluates the effects of various hardware optimizations such as separate prefetch issue buffers, prefetching with exclusive ownership, lockup-free caches, and weaker memory consistency models on the performance of prefetching.

1 Introduction

Techniques that can cope with the large latency of memory accesses are essential for achieving high processor utilization in large scale shared-memory multiprocessors. A number of different solutions have been proposed. For example, many recent multiprocessors [2, 13, 17] provide caches to help reduce the latency seen by the processor. More recently, weaker memory consistency models [1, 4, 6, 7] have been proposed that allow buffering and pipelining of memory references to hide latency. Still another technique is the use of processors with multiple hardware contexts [2, 10, 11, 26]. These processors tolerate latency by switching from one context to another when they encounter a high latency memory access. The various techniques that have been proposed are not mutually exclusive, but are complementary and offset the limitations of one another.

In this paper, we evaluate the effectiveness of another powerful latency hiding technique, namely *non-binding software-controlled prefetching*. Our study is done in the context of the Stanford DASH multiprocessor [17], a large scale shared-memory machine that provides for coherent caches, a weaker memory consistency model, and support for software directed prefetching. A prefetch operation in DASH is an explicit non-blocking request to the memory system that brings the prefetched location into a cache close to the processor.

Prefetching in DASH is non-binding in the sense that prefetched data remains visible to the cache coherence protocol [18] to keep it consistent until the processor actually reads the value through a binding access (e.g. a register load operation). In contrast, with binding prefetching [9, 14] the value of a later reference is bound (e.g. a processor register is loaded) at the time the prefetch completes. As a result, there are restrictions placed on when a binding prefetch can be issued, since the prefetched value may become stale if another processor modifies the same location. For example, a binding prefetch cannot be issued if there is a synchronization reference

between the prefetch and the subsequent binding reference. Binding prefetching studies done in [14] reported significant performance loss due to such interactions. Non-binding prefetching imposes no such restrictions on when a prefetch can be issued; the coherence protocol ensures that the value fetched by the final binding-read will be correct. This flexibility considerably simplifies the task of generating prefetches and increases their effectiveness by allowing them to be issued earlier. The choice between binding and non-binding prefetching is largely dictated by whether cache coherency is maintained by hardware or software, since hardware-based coherency is a requirement for non-binding prefetching.

Prefetching in DASH is software controlled in that to initiate it the processor must explicitly execute a prefetch instruction. Prefetches can be introduced either explicitly by the programmer, or automatically by the compiler, or perhaps dynamically by the runtime system of a programming language. In contrast to earlier studies where data prefetching was controlled by the hardware, for example through instruction look-ahead in [16], software control allows the prefetching to be done selectively (thus reducing overhead) and extends the possible interval between the issue of prefetch and the actual use of that data (thus increasing effectiveness) [9, 21]. The disadvantage, of course, is that programmer or software intervention is required.

The benefits due to prefetching come from several sources. The most obvious benefit occurs when a prefetch is issued early enough 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. When multiple prefetches are issued back-to-back, 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 [3]. By selectively prefetching data that are to be modified directly with ownership, it is possible to significantly reduce the write latencies and the ensuing network traffic for obtaining ownership.

The results presented in this paper are based on our experience with three parallel applications to which we explicitly added prefetching at the source level. The applications we studied are a particle-based simulator used in aeronautics [20], an LU-decomposition program, and a digital logic simulation program [24]. Our results show that for applications with regular data access patterns, as found in the first two applications, it is easy to add prefetching. By adding as few as 10-20 new lines of code, we were able to issue prefetches corresponding to about 90% of the original cache misses and we could increase the performance by 100-150% when prefetching directly into the processor's primary cache. The overhead due to issue of prefetch instructions was less than 15% of execution time. For the logic simulation application, however, prefetching was much more difficult to add. This was caused primarily by the extensive use of pointers and linked-lists by the program. Linked lists make prefetching difficult because unless a pointer has already been fetched, a miss will occur when it is dereferenced to prefetch the next element in the list. After substantial effort, we could manage to increase the performance only by 30%. The paper also evaluates the impact of various hardware optimizations such as separate prefetch issue buffers, prefetching with exclusive ownership, lockup-free caches, prefetching directly into primary cache, and weaker memory consistency models on the performance of prefetching.

The paper is organized as follows. The next section describes the multiprocessor architecture used in our experiments, the benchmark applications, and the simulator used to collect performance results. Section 3 predicts the maximum benefits we expect to see through prefetching. Section 4 presents the results of our studies. Sections 5 and 6 contain a discussion of related work and conclusions.

2 Multiprocessor Architecture, Benchmark Applications, and Simulator

In order to evaluate the benefits of prefetching it is necessary to focus on a specific class of multiprocessor architectures. The reason for this is that the benefits due to prefetching may vary greatly depending on the architecture chosen. For example, the benefits for a bus-based multiprocessor with cache miss latencies of twenty cycles will be very different from the benefits for a large scale multiprocessor with miss latencies of a hundred or more cycles. This section presents the architectural assumptions that we make, the benchmark applications, and the simulation environment used to get 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 architecture. The architecture consists of several processing nodes (or clusters) connected through a low-

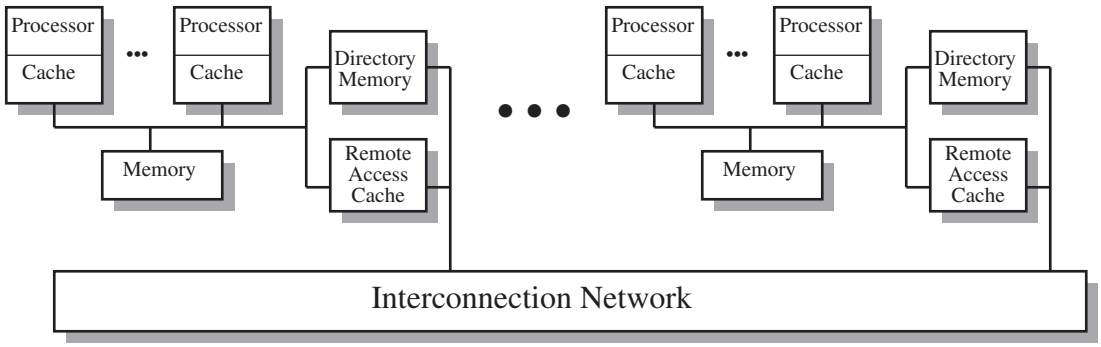

Figure 1: The DASH architecture.

latency scalable interconnection network. Physical memory is distributed among the nodes. Cache coherence is maintained using a distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching it, and 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.

Figure 2 shows the organization of the processor environment in the DASH prototype. Each CPU consists of a MIPS R3000/R3010 processor, a 64 Kbyte instruction cache, and a 64 Kbyte write-through primary data cache. The write-through cache enables processors to do single cycle write operations. The primary data cache interfaces to a 256 Kbyte secondary write-back cache. The interface consists of a read buffer and a write buffer. Both the primary and secondary caches are direct-mapped and support 16 byte lines. The DASH architecture uses the release consistency model [7] for memory consistency.

The latency of a memory access in DASH depends on where in the memory hierarchy the access is serviced. Figure 3 shows the latency for servicing an access at different levels in the hierarchy, assuming no contention (the simulations done in this paper do model contention, however). The following naming convention is used for describing the memory hierarchy. The *local cluster* is the cluster that contains the processor originating a given request, while the *home cluster* is the cluster that contains the main memory and directory for a given physical memory address. A *remote cluster* is any other cluster. The latency 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, since the release consistency model is used.

Since the DASH prototype currently being built uses regular commercial processors (MIPS R3000/R3010), there are no special prefetch instructions available. Consequently, to implement prefetching, the application's cacheable address space is double mapped into a portion of the I/O address space. To prefetch a location, a write is done to the corresponding location in this special I/O address space. The advantage of using I/O writes is that they get put into the write buffer and do not block the processor. Once the prefetch reaches the head of the write buffer, it is issued onto the bus. If the prefetch is for a remote memory location, it is converted into a regular memory-request message by the directory controller and sent to the home cluster.[1] The prefetch response is stored in the remote access cache (RAC), a special 256 Kbyte cache associated with each cluster (see Figure 1), when it returns. When the processor subsequently reads the prefetched location, the data is supplied by the RAC. If the data has not arrived back at the RAC when the regular request reaches it, the RAC is intelligent enough not to issue a duplicate request to the home cluster. The processor request is satisfied as soon as the reply to the original prefetch request arrives.

DASH provides three different types of prefetch operations [18]. The two main types that we use in this study are *read* prefetches and *read-exclusive* prefetches. A read prefetch brings data into the RAC in a read-shared

[1] In the DASH prototype currently being built, the processor's own caches are not checked for the presence of the prefetched location. This is because an I/O write is a non-cacheable reference and the caches simply ignore it. For similar reasons, no checking is done to see if any processor within the local cluster already has a copy of the prefetched line.

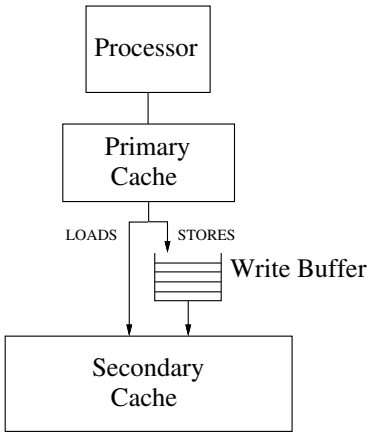

Figure 2: The processor environment.

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	12 pclock
Fill from Local Cluster	22 pclock
Fill from Home Cluster (Home $\neq$ Local)	61 pclock
Fill from Remote Cluster (Remote $\neq$ Home $\neq$ Local)	80 pclock
Write Operations	
Owned by Secondary Cache	3 pclock
Owned by Local Cluster	17 pclock
Owned in Home Cluster (Home $\neq$ Local)	57 pclock
Owned in Remote Cluster (Remote $\neq$ Home $\neq$ Local)	76 pclock

Figure 3: Latency for various memory system operations in processor clocks (1 pclock = 40ns).

mode. The read-exclusive prefetch, on the other hand, brings the data into the RAC in an exclusive-ownership mode, which enables a write to that location to complete quickly since ownership has already been obtained. DASH also provides a *deliver* operation, whereby a processor can directly send data to a remote cluster's RAC. Such producer-initiated prefetches are not evaluated in this study.

The DASH prototype, as is currently being built, imposes a number of constraints that limit the benefits obtained from prefetching. Most of the constraints arose from our desire to minimize the changes necessary to the commercial processor boards used in the prototype. These constraints include the following: (1) the write buffers are only four entries deep; (2) reads cannot bypass writes in the write buffer (although this is allowed by the release consistency model used); (3) the processor's own cache is not checked before a prefetch is issued onto the bus; (4) the caches of other local processors are not checked before a prefetch request is sent to the home cluster; (5) prefetches cannot bypass writes in the write buffer and cannot be bypassed by reads; (6) only a single miss request can be outstanding from a secondary cache at a time; and (7) prefetched lines are brought into the RAC rather than the processor's own cache.

In this paper, to separate out the prefetching issues from the implementation decisions in the prototype, we will examine the performance of prefetching under implementations of varying aggressiveness. The BASE implementation that we use for the bulk of the paper assumes the DASH prototype, but uses 16 entry deep write buffers and removes constraints 2-4 mentioned above. Later, in Section 4.2, we consider the performance impact of more aggressive implementations which remove constraints 5-7. We consider the effects of separate prefetch buffers, lockup-free caches, and of directly prefetching into the primary cache.

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. The three applications we studied are MP3D, LU, and PTHOR. Table 1 shows some general statistics for the benchmarks when using 16 processors (as is the case throughout this study).

Table 1: General statistics for the benchmarks.

Application	Instructions Executed *(x 1000)*	Total Data Refs *(x 1000)*	Shared Data Refs *(x 1000)*	Shared Reads *(x 1000)*	Shared Writes *(x 1000)*	Locks	Barriers	Shared Data Size *(KBytes)*
MP3D	5240	1737	1592	1084	507	0	448	524
LU	27,773	8369	8274	5543	2727	3184	32	640
PTHOR	18,997	8588	4577	3946	468	79,114	2208	2856

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. The main synchronization consists of barriers between each time step. For our experiments we ran MP3D with 10,000 particles, a 64x8x8 space array, and simulated 5 time steps.

LU performs LU-decomposition for 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 all columns to the left of a column have modified that column, 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 that column is used to modify all columns that the processor owns. 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 [24] is a parallel logic simulator based on the Chandy-Misra simulation algorithm. Unlike centralized-time algorithms, this algorithm does not rely on a single global time during simulation. The primary data 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 looks up the net data structure to determine which elements are affected by the output change and schedules the newly activated elements on to task queues. For our experiments we simulated several clock cycles for a small RISC processor consisting of 11,000 two-input gates.

2.3 Simulation Environment

An event driven simulator is used to simulate the major components of the DASH architecture at the behavioral level. For example, 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, with one processor per cluster. We chose only one processor per cluster to avoid the interference (both positive and negative) between processors in the same cluster which would significantly complicate the analysis of prefetching behavior. The latency parameters for the simulated architecture are derived from the DASH prototype (see Figure 3). The architecture simulator is tightly coupled to the Tango reference generator [8] 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. Operating system references are not modeled. Shared memory used by a program is evenly distributed across all clusters using a round-robin page allocation scheme.

We now arrive at a difficult methodological problem that occurs 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/applications than those that would be run on the real machine. The question arises of how to scale the machine parameters so as to get realistic performance estimates. For example, consider the MP3D application. In real life, the application is designed to run with enough particles so as to fill the complete main memory of a machine. Since at each time step in the application all particles are moved (i.e. the complete memory is swept through) the caches are expected to miss on each particle. If we would have retained 64 Kbyte first-level and 256 Kbyte second-level caches in the simulator, as in DASH, then we would have to run MP3D with at least 125,000 particles to achieve realistic cache behavior. This would have taken extremely long to run.

We see no easy answer to the above question. For this study, however, we have chosen to use scaled-down cache sizes to get a more realistic problem size to cache size ratio. For MP3D, we scaled down the processor caches to 2 Kbyte first-level and 4 Kbyte second-level caches.[2] In this configuration, we get similar miss ratios to the full-size caches with only 10,000 particles and thus reduce the simulation time substantially. The data sets for the other two applications were also adjusted to get realistic cache hit ratios and reasonable runtimes. For LU, the data set size is chosen such that the data starts fitting into the combined caches when only the bottom 45% of the matrix remains to be factored. As a result, the processors get poor cache hit ratios in the beginning, and high hit ratios towards the end of the computation. This kind of behavior is not atypical in many numerical applications. For PTHOR, our experiments use a circuit with 11,000 gates. However, on the real machine, we expect to be using circuits with hundreds of thousands of gates. We thus reduce the cache size and the circuit size proportionately. To evaluate the impact of the scaled caches used throughout this paper, we present results for larger cache sizes in section 4.2.1.

3 Predicting the Performance Benefits of Prefetching

In this section we develop a simple model that helps predict the performance benefits of prefetching in an architecture that has write buffers and two levels of cache, similar to DASH. We use this model to predict the maximum expected speedup for each benchmark for both the BASE architecture, where we prefetch only into the RAC, and also for the case of prefetching into the primary cache. The model can be used to determine whether it is worthwhile to add prefetching to an application.

3.1 Runtime Without Prefetching

The execution time of an application can be broken down into the following components: (1) busy time when the processor is doing useful work, (2) stall time on account of the write buffer being full when the processor tries to issue a write, (3) stall time for reads that miss in the primary cache, and (4) stall time for synchronization operations which block the processor. In an ideal RISC processor architecture with no interlocks, the busy time is equal to the number of instructions executed. The stall time for read misses can be further broken down as follows: (1) time spent waiting for writes to complete before the read miss can issued to the secondary cache,[3] (2) time spent fetching the line into the secondary cache if the read misses there as well, and (3) time spent filling the primary cache from the secondary cache before restarting the processor. We incorporate all of the components mentioned above into the following equation for total execution time of an application without prefetching:

$$\begin{aligned} \text{Execution Time} &= \text{Busy Time} + \text{Write Stall Time} + \text{Read Stall Time} + \text{Sync Stall Time} \\ &= N + Wl_{wb\ write} + Rm_p(l_{wb\ read} + m_s l_{miss} + l_{fill}) + Sl_{sync} \end{aligned} \quad (1)$$

where

[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]This is assuming either a strong consistency model or a secondary cache which is not lockup-free, as in the case of the BASE architecture.

N, W, R and S represent the total number of instructions, writes, reads and synchronization operations, respectively.

$l_{wb\ write}$ is the average time a processor stalls on a write due to a full write buffer.

m_p and m_s represent the fraction of reads which miss in the primary cache, and the fraction of these misses which subsequently miss in the secondary cache, respectively.

$l_{wb\ read}$ is the average time that a processor is stalled on a read miss waiting for writes to complete.

l_{miss} is the average latency for satisfying a secondary cache read miss.

l_{fill} is the time that the processor stalls on a primary read miss if the line is found in the secondary cache.

l_{sync} is the average latency for completing a synchronization operation.

If we normalize equation (1) by dividing out by the number of instructions, N, we get the following expression for CPI:

$$\text{CPI}_{\text{nopf}} = 1 + wl_{wb\ write} + rm_p(l_{wb\ read} + m_s l_{miss} + l_{fill}) + sl_{sync} \tag{2}$$

where w, r and s are the number of writes, reads and synchronization operations per instruction, respectively.

For non-uniform memory access (NUMA) machines, we can express the secondary cache read miss time as follows:

$$l_{miss} = p_{local} l_{local} + (1 - p_{local}) l_{remote} \tag{3}$$

where p_{local} is the probability that the reference is to local memory, and l_{local} and l_{remote} are the latencies for accessing a line from local and remote memory, respectively. Assuming that references are evenly distributed across c clusters, as we do in our simulations, $p_{local} = 1/c$. Therefore,

$$l_{miss} = \Big(\frac{1}{c}\Big) l_{local} + \Big(\frac{c-1}{c}\Big) l_{remote} \tag{4}$$

3.2 Prefetching into the Remote Access Cache

Prefetching into the remote access cache (RAC) can reduce miss latencies to the access time of local memory. Since prefetching is controlled by software, it adds a new overhead component to execution time, as shown below.

Execution Time = Busy Time + Write Stall Time + Read Stall Time + Sync Stall Time + Prefetch Overhead

Considering each component individually, the busy time does not change as compared to the case without prefetching since the same amount of useful work is being done. The stall time for writes depends on the likelihood of the write buffer filling up, which can either increase or decrease with prefetching. On one hand, if prefetches are issued as writes (as in DASH), the write buffer will have more requests to process. On the other hand, read-exclusive prefetches allow the write buffer to retire writes more quickly. Read-exclusive prefetches also reduce the amount of time a read miss has to wait for the write at the head of the write buffer to complete. Prefetched secondary read misses should experience a latency close to l_{local}, if they have been prefetched early enough, while non-prefetched misses should experience the normal miss latency. The stall time due to blocking synchronization operations may change due to second-order effects, but we ignore this effect in our model. The overhead of prefetching includes both the extra instructions which must be executed in order to generate and issue the prefetch as well as any time that the processor stalls during prefetch issue on account of the write buffer being full. Taking all of the above effects into account, the CPI when prefetching into the RAC is:

$$\begin{aligned} \text{CPI}_{\text{RAC}} = {} & 1 + wl_{wb\ write} + \\ & rm_p\Big\{l_{wb\ read} + m_s[f_{prefetched} l'_{miss} + (1 - f_{prefetched}) l_{miss}] + l_{fill}\Big\} + \\ & sl_{sync} + f_{prefetched} rm_p m_s o_{pf} \end{aligned} \tag{5}$$

where $f_{prefetched}$ is the fraction of secondary read misses that are prefetched (which we will refer to as the *coverage factor*), l'_{miss} is the latency of a prefetched secondary read miss, and o_{pf} is the overhead of generating a single prefetch. In the best case, l'_{miss} would be equal to l_{local} and o_{pf} would be a single cycle.

As we mentioned above, the values of $l_{wb\,write}$ and $l_{wb\,read}$ may change significantly due to prefetching. In equation (5), the term $f_{prefetched} rm_p m_s o_{pf}$ represents the overhead of introducing prefetches. We neglect the overhead of prefetches for lines which are already in the secondary cache in this model. Equation (5) can also be used to model prefetching directly into the secondary cache by setting l'_{miss} equal to zero.

3.3 Prefetching into Primary Cache

If we prefetch into the primary cache, the latency for a prefetched read miss can be reduced to a single cycle. The following equation models the case of prefetching into the primary cache:

$$\begin{aligned} \text{CPI}_{\text{PC}} &= 1 + wl_{wb\,write} + \\ &\quad rm_p f'_{prefetched} l_{pf} + rm_p(1 - f'_{prefetched})(l_{wb\,read} + m_s l_{miss} + l_{fill}) + \\ &\quad sl_{sync} + f'_{prefetched} rm_p o_{pf} \end{aligned} \tag{6}$$

where $f'_{prefetched}$ is the fraction of primary (rather than secondary) read misses that are prefetched, and l_{pf} is the average stall time of a prefetched read reference. l_{pf} will be zero if the prefetches are always issued early enough.

Once again, read-exclusive prefetching may further reduce the values of $l_{wb\,write}$ and $l_{wb\,read}$ by increasing the hit rate of writes in the secondary cache. Notice that the total number of prefetches ($f'_{prefetched} rm_p$) does not include the m_s factor which appears in equation (5), since we now distinguish prefetched read misses with respect to the primary rather than the secondary cache. It is possible that prefetching directly into the caches may cause some interference and therefore alter m_p and m_s, but we ignore this effect in our model. With larger caches this is less likely to be a problem.

3.4 Maximum Expected Speedups

In this subsection, we use data about the reference and miss rates of the benchmark applications to estimate the best speedup they may get from prefetching. The data (see Table 2) were collected from simulations of an ideal architecture which has the same cache sizes as the BASE architecture used in this study and where all memory references complete in a single cycle. (The data in Table 2 do not include references to private data, which are assumed to hit in the cache.) The resulting synchronization delays reflect the intrinsic load imbalance in each application. In the most optimistic case, read-exclusive prefetching will eliminate write buffer stall times, all read misses will be prefetched, and the overhead of a prefetch will be a single cycle. In the case of prefetching into the RAC, the latency of a prefetched secondary read miss will be l_{local}, and in the case of prefetching into the primary cache, l_{pf} will be zero, meaning that the read hit rate will be perfect. As we can see from Table 2, the applications show the potential for a two to three-fold performance gain when prefetching into the RAC, and a four to six-fold improvement when prefetching directly into primary cache.

Table 2: Maximum expected speedups for benchmarks.

Application	Writes per Inst (w)	Reads per Inst (r)	Sync Ops per Inst (s)	Primary Miss Rate (m_p)	Secondary Miss Rate (m_s)	Avg Sync Latency (l_{sync})	$\text{Speedup}_{\text{RAC}}$	$\text{Speedup}_{\text{PC}}$
MP3D	0.0968	0.2069	0.000085	0.1997	0.9858	141	2.3	4.3
LU	0.0982	0.1996	0.000116	0.4115	0.8217	509	2.7	6.4
PTHOR	0.0246	0.2078	0.004281	0.2564	0.8456	47	2.3	4.3

4 Simulation Results

This section is divided into two parts. In the first subsection, we consider each application individually and evaluate the benefits of prefetching on the BASE architecture (see Section 2.1). We discuss not only how much faster each application runs with prefetching, but also where the benefits come from and how much

effort and sophistication is required in introducing the prefetches. In the second subsection, we evaluate the impact of several architectural variations on the benefits of prefetching. We study the impact of varying cache sizes, sequential versus release memory consistency models, separate prefetch issue buffers, lockup-free caches, prefetching directly into the processor's cache, and the benefits of read-exclusive versus strictly read prefetching.

4.1 Application Case Studies

One of the main aims of our current study was to gain a quantitative idea of the performance improvements possible through non-binding software-controlled prefetching. We did not want to be constrained by the limits of existing compiler technology to automatically add prefetching. For example, existing compiler technology [9, 21] would not have handled the linked-lists and pointers in PTHOR. Also, since we did not have a prefetching compiler available to us, the prefetch statements were added manually to the programs. Prefetches were introduced at the source level through macro statements. These macros covered both read and read-exclusive prefetching, as well as single cache line and block prefetches. In all cases, the address to be prefetched was the first argument to the macro and the block size was the second argument. Block prefetches were expanded into multiple single line prefetches, which were issued sequentially by the processor.

4.1.1 MP3D

The MP3D application spends most of its time executing 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 high-level data structures within this loop: (1) the particle which is being moved, and (2) the space cell where the particle resides. Each particle structure is two cache lines long, and each space cell structure is three lines long. References to the particles and space cells within this loop account for 37% and 54% of the secondary cache read misses respectively, for a total of 91%.

We experimented with several different prefetching strategies for MP3D. The results of these experiments are shown in Table 3 and Figure 4. The results in the figure are all normalized to the case when no prefetching is used. The bars are to be interpreted as follows. The bottom section (*busy*) represents the busy time or useful cycles executed by the processor. The *reads* section on top of it indicates the fraction of runtime for which the processor was stalled due to read miss latencies. Note this does not include the time that a read miss stalls while it is waiting for the write at the head of the write buffer, if present, to retire.[4] In fact, this time for which the reads are stalled for the write buffer plus the time for which writes themselves stall the processor is represented as the *writes* section. The *sync ops* section on top of that represents the time spent waiting for lock accesses and barriers. Finally, the section on the very top (*prefetches*) represents the overhead of prefetching. This includes both the extra instructions executed due to prefetching (e.g. evaluation of conditional statements that help decide whether to prefetch or not, instructions to do address computations for prefetching, and the cost of issuing the prefetch itself) and the time for which prefetches stall the processor if the write buffer is full. The latter overhead can become noticeable when block prefetches are issued.

Table 3: Statistics on MP3D prefetching strategies.

Strategy	Source Lines Added	Coverage	Prefetched Read Miss Latency	Overall Read Miss Latency	Speedup
nopf	0	0%	N/A	69.1	1.00
pf1	1	37%	40.0	64.1	1.08
pf2	2	91%	37.0	41.8	1.50
pf3	6	91%	20.3	26.9	1.78
pf4	16	95%	22.0	25.3	1.86

The contents of Table 3 are fairly straightforward to interpret. The two columns that need some explaining are the prefetch read latency and the overall read miss latency. The former refers to the latency seen by a read

[4]Recall that the BASE architecture allows reads to bypass pending writes. However, since the secondary cache is not lockup-free [12, 22], it cannot allow both a read and a write operation to access cache at the same time. Consequently, the read miss must stall for the write at the head of the write buffer, if present, to retire.

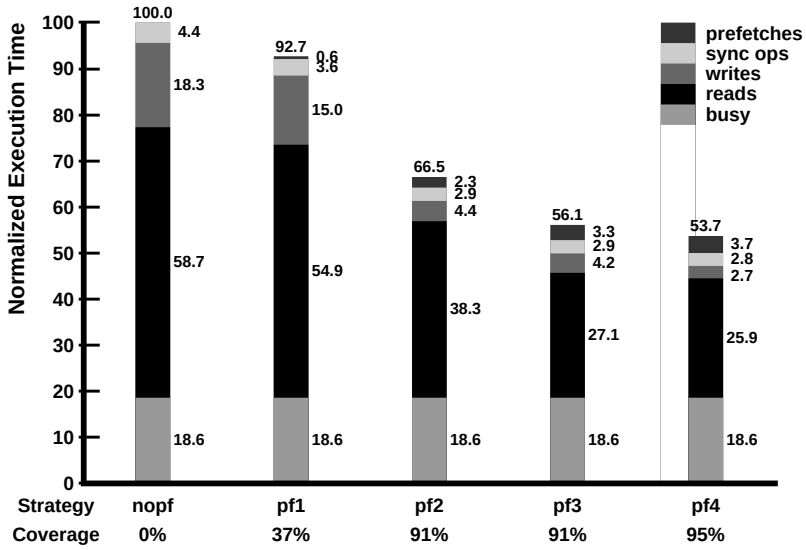

Figure 4: MP3D case study.

miss (for which a prefetch had previously been issued) from the time it gets past the write buffer to the time that the processor is restarted. The minimum such time for the BASE architecture is 14 processor cycles. In general, this would be higher since the prefetch may not have been issued early enough. The latter refers to the overall miss latency for both prefetched and non-prefetched read misses.

The various prefetching strategies that we explored for MP3D are described below.

nopf: This represents the base case when no prefetching is done. As we can see from Figure 4, about 59% of the time is lost due to read latencies and another 18% due to write buffer waiting. Of the latter 18%, almost 99% of it is due to the reads stalling for the write at the head of the write buffer to retire.

pf1: In this strategy particle records are prefetched in a read-exclusive state immediately before they are first referenced. The coverage (percentage of misses prefetched) is 37% and the performance improves by 8%. Since we are prefetching right before use, where does the improvement come from? The improvement comes from two sources: pipelining and reduced write buffer waiting.

Considering pipelining, recall that the particle consists of 2 cache lines. Even though the full latency of a remote access is seen for the first cache line, only the local latency is seen for the second line. This pipelining reduces the stall time due to read latencies from 58.7% in nopf to 54.9% in pf1, as can be seen in Figure 4.

The write buffer waiting is reduced due to the read-exclusive prefetching. Since the prefetched line is brought directly into the RAC with ownership, the first write access to the particle can be quickly retired from the write buffer. (In DASH, a write cannot be retired from the write buffer until ownership is obtained for the associated cache line.) This results in a smaller stall for the read following the write. From Figure 4 we see that the write buffer time goes down from 18.3% to 15.0%.

pf2: In this strategy, both particle and space cell records are prefetched in a read-exclusive state immediately before they are first referenced. Note that the prefetches for the space cell record cannot be issued until the particle prefetches have already completed, since we need to know the location of the particle to determine the space cell that it occupies. The coverage goes up to 91% and the performance improves by 50% over the no prefetching case.

The reasons for the increase in performance are the same as for pf1, pipelining and reduced waiting time for write buffer. However, as can be seen from Figure 4, the improvements are much larger than those for pf1. Pipelining benefits are larger because: (i) the space cell consists of three cache lines versus two lines for the particle record, and (ii) the average read miss latency for a particle is smaller than that for the space cell. The reason for the smaller latency for particles is that they are statically partitioned among the processors, so they can never be dirty in any remote cluster; this is not the case for space array cells. The benefits due to reduced waiting time for the write buffer also increase for the same reasons. Since the space array cell can be dirty in a remote cluster, getting the ownership for its cache lines takes a much longer time.

pf3: In the pf3 strategy, a particle record is prefetched two iterations ahead of its turn to be moved. In the iteration following the prefetch, the particle record is read, the associated space cell is determined, and prefetches for the space array cell are issued. As a result, when the time comes for the particle to be moved, both the particle and space array records are available locally. While requiring more sophisticated prefetching, this approach removes the processor stalls in pf2 for the first cache line of both space cell and particle records. As shown in Figure 4, this increases the performance noticeably over strategy pf2. For the same reason, as seen in Table 3, there is a big drop in the average latency experienced by prefetched reads as compared to pf1 and pf2 strategies.

pf4: In this final case, we went through several other routines in MP3D and also tried to prefetch references that occur at time step boundaries. This increased the coverage to 95% of all misses. Overall, performance improves by 86% over the version of MP3D with no prefetching.

We now consider the overheads and complexity of prefetching. As can be seen from Figure 4 the overheads are quite small. They vary from about 0.6% of the original normalized cycles for pf1 to 3.7% for pf4. The sophistication required to issue the prefetches, however, is more difficult to characterize. On the surface, especially for pf1 and pf2 strategies, prefetching seems fairly straightforward. We did not have to perform any complex transformations to the program to do prefetching, and a compiler should be able to do it. Even for pf3, where we use a software pipeline for prefetching, a sophisticated compiler should be able to handle it. At a deeper level, however, the compiler's job may be much harder. For example, it was quite easy for us, using semantic information about the application, to realize that the particle and space-cell arrays were the critical data structures, they were going to have high miss rates in the cache, and they were thus good candidates for prefetching. Such focusing in on critical data structures will be much harder for compilers. Similarly, we evenly distribute the prefetching of particles and space cells with computation. We do not prefetch a cache full of particles and corresponding space cells because we know that the space cells will be invalidated from the processor's cache by other processors modifying them. Again, such inferences will be much harder to make automatically.[5]

4.1.2 LU

In LU, the columns of the matrix are statically assigned in an interleaved manner to the processors. The main computation done by each processor consists of reading a pivot column once it is produced, 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) when the pivot column is read for the first time; (ii) the pivot column may be replaced from the cache by one of the columns that it is applied to, and hence may have to be refetched; and (iii) the misses on the owned columns that the pivot column is applied to. This last set of misses occur because the combined size of the owned columns may be larger than the size of the cache. According to the assumptions we make in this study, the caches are large enough to hold several columns at a time (so that the pivot column is not likely to be replaced frequently), but they are not large enough to hold all of the columns that a processor owns.

Table 4: Statistics on LU prefetching strategies.

Strategy	Source Lines Added	Coverage	Prefetched Read Miss Latency	Overall Read Miss Latency	Speedup
nopf	0	0%	N/A	63.5	1.00
pf1	1	8%	61.3	69.2	0.94
pf2	20	8%	21.1	60.0	1.02
pf3	5	29%	18.2	51.2	1.10
pf4	6	93%	26.0	31.0	1.74
pf5	8	93%	28.3	31.9	1.83

Figure 5 and Table 4 show the results of our case study of LU. The prefetching strategies we studied are the following:

[5]We note that it was possible to prefetch particles and space cells because we use non-binding prefetches. It would not be legal to prefetch either of these data structures if binding prefetching [9, 16] is used, since there is always a small chance that they will be modified between the time they are prefetched and the time they are used.

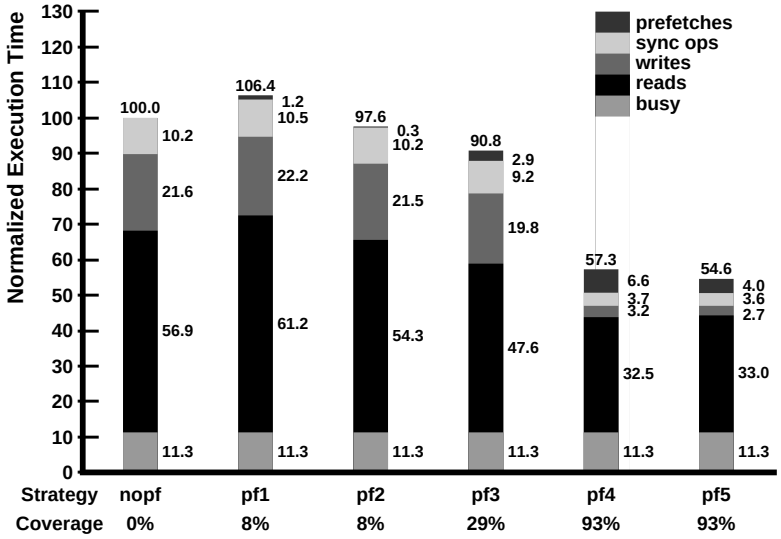

Figure 5: LU case study.

nopf: Case when no prefetching was used. As discussed above, a large miss rate is caused by the fact that the owned columns do not fit into the processor's cache, causing almost 78% of the time to be spent in read miss latency and in reads trying to get around the write buffer. Another 10% of the time is spent in synchronization, while processors wait for the pivot column to be produced.

pf1: In this strategy, the pivot column is prefetched as a single block as soon as it is produced. It is a simple strategy in that the compiler may easily notice that the pivot is being produced remotely and that it is being used repeatedly, so it should be prefetched. Unfortunately, this results in only 8% coverage, but even worse, the performance decreases by 6%. The cause for the lower performance was traced to hot-spotting at the home cluster where the pivot column is stored. Since all processors prefetch the *entire* pivot column at about the same time, there is an enormous build-up of queues at the cluster where the pivot column is stored. Simulation statistics show that the average prefetch request was stalled for nearly two hundred cycles due to the queueing delays. The problem is further aggravated by the fact that our architecture does not have a block-transfer mode, and as a result, the pivot column is supplied cache-line by cache-line to each requesting processor.

pf2: This is the same as strategy pf1, except that the prefetches are interleaved with applying the pivot column to the first owned column. The strategy required much more complex modification to the code than the simple block prefetch—for example, prefetches of pivot elements are introduced 4 iterations before they are needed, and since each prefetch fetches two pivot elements (because of the cache line size) they are issued only on every alternate iteration. However, the hot-spotting was significantly reduced and the performance improves slightly, by 2%, over the no-prefetch case.

pf3: The pivot column prefetches are evenly distributed, as in the pf2 strategy, except that the pivot is prefetched each time it is used to modify another column. This obviously results in redundant prefetches, and Figure 5 shows that the overhead for prefetches goes up from 0.3% for pf2 to 2.9% for pf3. However, it cuts down the misses when the pivot column is replaced from the processor's cache, thus increasing the coverage to 29%. The performance increase remains small though, only about 9%.

pf4: The pivot column is prefetched evenly in read-shared mode each time it is used, as in strategy pf3, and the column to be modified is prefetched in read-exclusive mode as a block, increasing the coverage to 93%. Although prefetching the pivot column as a block led to disastrous hot-spotting in pf1, we consider block prefetches in this case because one might intuitively expect less hot-spotting for the column to be modified since it is accessed by only one processor. This is indeed the case, and this strategy results in a substantial performance increase—74% over the case with no prefetching. Compared to pf3, the stalls due to read latencies go down significantly, but there is also a large decrease in the stalls due to reads waiting for the write buffer head to retire (from 19.8% to 3.2%). This decrease is made possible by the read-exclusive prefetching for modified columns. The synchronization time also decreases from 9.2% to 3.7% since read-exclusive prefetches enable the pivot column to be generated more quickly. Finally, we

note that the prefetch overhead jumps from 3% to almost 7%. This is partly because more prefetches are being issued, but also because the block prefetch causes the write buffer to fill up which stalls the processor.

pf5: This is the same as strategy pf4 except that the modified columns are also prefetched evenly, rather than as a block. This provides only a small performance improvement over pf4. As Figure 5 shows, the performance increase comes primarily from reduced prefetch overhead, since evenly distributed prefetching decreases the likelihood of the write buffer filling up.

In summary, we see that for LU prefetching was relatively easy to add and it increased the performance by about 83%. This case study also points out that, in general, evenly distributed prefetching performs better than block prefetching due to either reduced hot spotting or reduced prefetch issue overhead. We also observed that in some cases it helps to prefetch redundantly, as we do for the pivot column, since the data once prefetched may be replaced in the cache. Of course, this requires that the overhead of prefetching be small. Finally, as in the case of MP3D, LU shows that read-exclusive prefetching significantly lowers the stalls experienced by read accesses waiting for writes to get ownership.

4.1.3 PTHOR

The main computational loop in PTHOR consists of the following steps. The processor picks up an activated element, it checks pending events on the element's inputs, and determines the changes, if any, on the element's outputs. It then looks up the net data structure to determine which elements are affected by the output changes and it finally schedules these newly activated elements onto task queues. 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. Each element record is 12 cache lines long. Several fields in the element record are pointers to linked lists, or are pointers to arrays that in turn point to linked lists. For example, the pending events for an element are kept in an array (one entry per input), where each entry in the array is a linked list of events. 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 considered several different strategies for prefetching, and the results are presented in Figure 6 and Table 5.

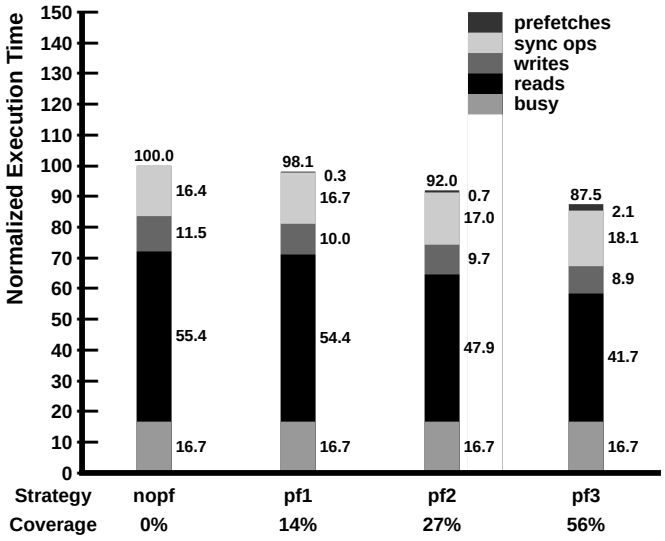

Figure 6: PTHOR case study.

nopf: This is the no prefetching case. Similar to the other two applications, most of the stall time is due to read latencies. However, stall time due to synchronization is also quite large. This is because of the fine-grain tasking used by PTHOR.

pf1: In this strategy, as soon as an activated element is picked from the task queue, the complete element record structure is block prefetched in read-exclusive mode. The strategy is quite simple and intuitive and can easily be added automatically by a compiler. Unfortunately, the performance improvements were poor. As shown in Figure 6, a coverage of 14% is achieved and the performance improvement is only 2%.

Table 5: Statistics on PTHOR prefetching strategies.

Strategy	Source Lines Added	Coverage	Prefetched Read Miss Latency	Overall Read Miss Latency	Speedup
nopf	0	0%	N/A	62.5	1.00
pf1	2	14%	33.1	60.9	1.02
pf2	9	27%	23.5	53.0	1.09
pf3	29	56%	27.5	45.1	1.14

pf2: In this case, we first reorganized the element record and grouped entries together 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, it first issues a block read-exclusive prefetch for the section that is likely to be modified, and a block read-shared prefetch for the likely-read section. In addition, we prefetch the input array that contains pointers to the lists of pending events, and the event at the head of each list. Similarly, the output array is also fetched. As shown by Figure 6, we get a modest coverage of 27% and the performance increases by about 9%. As usual, benefits come from reduced read latencies, and slightly decreased waits for the write buffer. The overheads of prefetching were negligible.

pf3: In this strategy, in addition to the prefetches in pf2, we prefetch several more items related to the output structure. For example, we prefetch the array that contains the delays associated with each of the outputs, we prefetch the list that keeps track of other elements that are connected to the current one, and so on. We put profiling markers in the code to determine which sections of the code were generating unprefetched read misses, and added prefetches there to increase our coverage. Although it was often clear that the gains from the prefetches we were adding were going to be small (since the data was used soon after the prefetch was issued), we added them anyway to increase coverage. With all this effort, we succeeded in increasing the coverage to 56% of read misses with a performance increase of 14%.

Adding prefetching to PTHOR was a qualitatively different experience from adding prefetching to MP3D and LU. While it took a significantly longer time to do so, the resulting coverage and speedup were smaller. Regarding how a compiler may fare in automatically adding prefetching, it is difficult to predict. It is unlikely that the compiler will be able to reorganize the record structure for the logic elements according to usage patterns, as we did. On the other hand, it is likely that the compiler will be more thorough in picking all corner cases where prefetching is possible. For example, we considered only the key functions in the code; the compiler will automatically examine all functions in the code and add prefetching to them. Of course, it may add excessive prefetching causing large overheads, but for the moment, this issue remains open.

4.2 Effect of Architectural Variations

In this subsection, we evaluate the impact of several architectural variations on the performance of prefetching. The variations we study include changes in the size of the processor caches, sequential versus release consistency models, separate prefetch issue buffers, lockup-free caches, prefetching directly into the processor's cache, and read-exclusive versus strictly read prefetching. All performance evaluations are done using the best prefetching strategy developed for each application in the previous subsection.

4.2.1 Cache Size Variations

The performance results reported in the previous subsection were based on the BASE architecture with 2 Kbyte first-level and 4 Kbyte second-level caches. The reasons for choosing such small cache sizes were discussed in Section 2.3, the main point being that we did not want to run applications with artificially small data sets (that can be simulated in reasonable time) with large realistic caches. In the following paragraphs we evaluate the effects of larger caches. The results of our experiments are presented in Figures 7, 8 and 9 and Table 6.

Figure 7 presents results for MP3D for three cache sizes. The combinations explored for the first-level and second-level caches are 2Kbyte/4Kbyte (from the BASE architecture), 16Kbyte/32Kbyte, and 64Kbyte/256Kbyte

(from the DASH prototype). All performance results are normalized to the BASE architecture. The data set size of the particles accessed by each processor is approximately 20Kbytes, and that of the space cell array is 192Kbytes (only a subset of the space cell array is accessed on each iteration). Given the large size of the space cell array it is not surprising that the performance with the 2K/4K and 16K/32K caches is similar. With the 64K/256K caches all data should fit into the second-level cache. However, the performance still doesn't increase enormously because the space cells are continuously being invalidated from the cache and have to be refetched. (A space cell is invalidated whenever a particle owned by another processor moves into the same space cell.) The relative performance gains due to prefetching also remain similar with the different cache sizes.

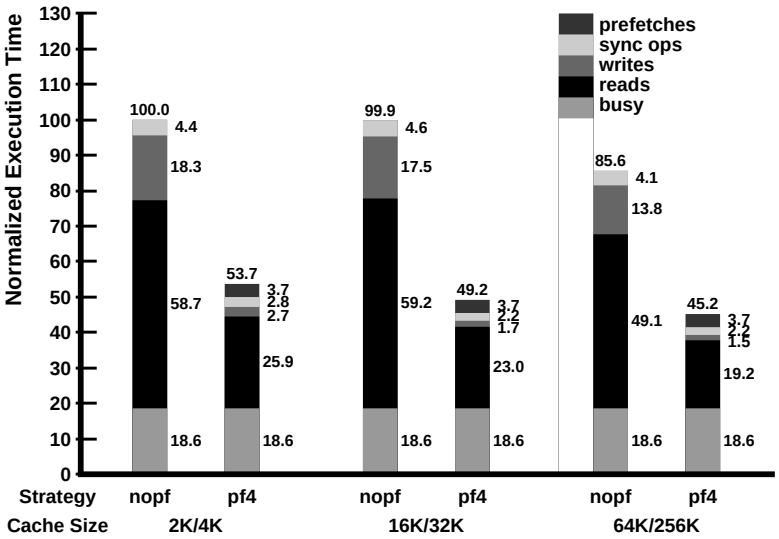

Figure 7: Effect of cache size variations on MP3D.

Figure 8 presents results for PTHOR. We use the same cache sizes as for MP3D. Again, we do not see dramatic changes in application performance even as the cache sizes are increased by a factor of 32. This is partly attributable to the fine-grain sharing that occurs in PTHOR. The result is that a significant portion of the cache misses are due to true-sharing (data produced by one processor is being consumed by another processor), and such misses do not decrease with larger caches. The benefits from prefetching do not qualitatively change with larger caches.

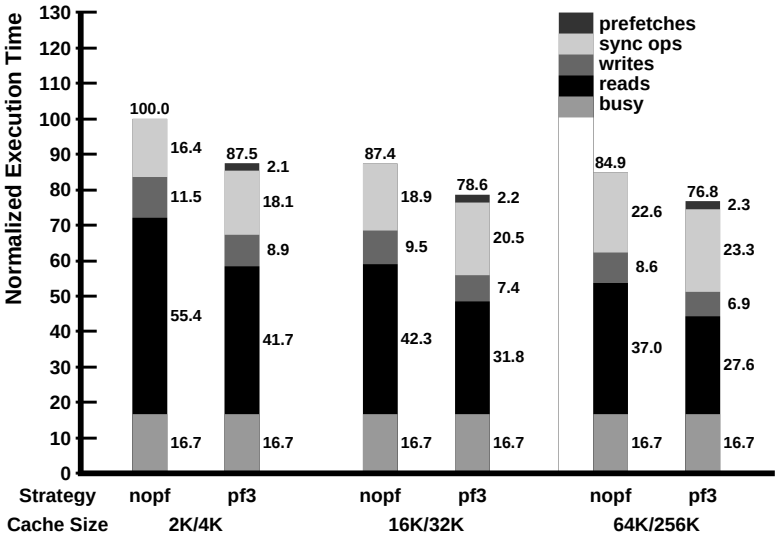

Figure 8: Effect of cache size variations on PTHOR.

Figure 9 presents results for LU. Here we use 4K/8K cache sizes for the middle run. We do not use the 16K/32K caches, as for MP3D and PTHOR, since most of the data would have fit into the cache and the results would have looked the same as for the 64K/256K configuration. For LU, we do see a big difference

in the performance as the cache size is increased. Consider the 4K/8K caches first. Although the absolute performance improves significantly for the no-prefetch case, the benefits due to prefetching remain large. The main reason is that the 8 Kbyte secondary cache is still too small to accommodate all of the owned columns. The owned columns start fitting into the cache only during the last 25% of the computation (recall that as computation proceeds certain columns are never touched again), so there is still plenty of computation during which prefetching is helpful. However, once we move to 64K/256K caches, all owned columns fit into the cache, and as a result the benefits of prefetching are drastically reduced. Still, we can see minor reductions in latencies due to prefetching because: (i) it helps to prefetch the pivot column when it is accessed for the first time, and (ii) it helps to prefetch the owned column references in case they get replaced in the cache due to interference with the pivot column. Unfortunately, the overhead of prefetching nullifies most of this gain. To reduce prefetch overhead, we tried several other less aggressive prefetching strategies, but in each case the decreased prefetch overhead was offset by higher memory reference latencies.

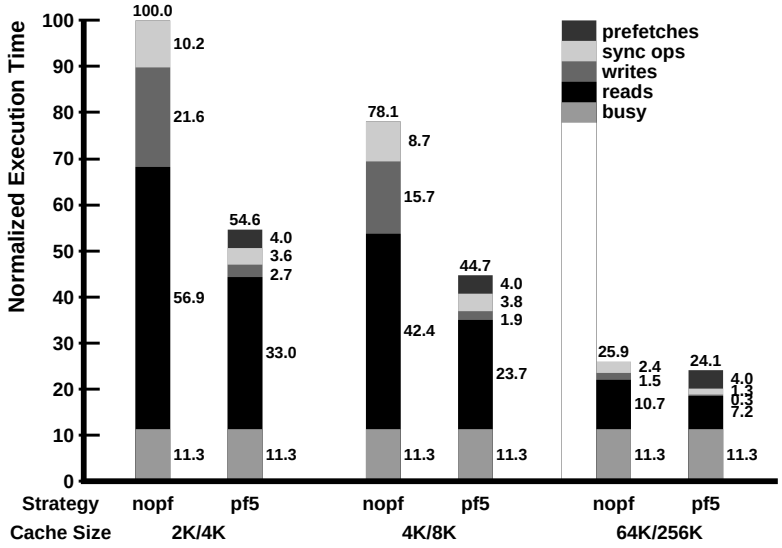

Figure 9: Effect of cache size variations on LU.

Table 6: Hit rates of applications for various cache sizes.

	Cache Size					
	BASE Architecture		Intermediate		DASH Prototype	
Application	Read Hit Rate	Write Hit Rate	Read Hit Rate	Write Hit Rate	Read Hit Rate	Write Hit Rate
MP3D	80.3%	68.4%	81.6%	69.2%	85.8%	73.3%
PTHOR	78.3%	46.9%	83.4%	52.0%	85.9%	52.7%
LU	66.2%	55.1%	75.9%	66.7%	95.4%	97.2%

In summary, although we use fairly small cache sizes in the BASE architecture to provide a more realistic ratio between application data set sizes and cache sizes, the benefits due to prefetching do not qualitatively change for MP3D and PTHOR when larger caches are used. This is mainly due to the fact that the memory system traffic due to true data sharing in the application does not change with larger caches. For LU, as long as the actively referenced data set is larger than the cache, the gains remain large. When the data does begin to fit, the gains of prefetching are expectedly small due to the very high hit ratios.

4.2.2 Sequential versus Release Consistency

The memory consistency model supported by a multiprocessor architecture determines the amount of buffering and pipelining that may be used to hide or reduce memory access latencies. The strictest model is that of *sequential consistency* (SC). It requires the execution of a parallel program to appear as some interleaving

of the execution of the parallel processes on a sequential machine. While conceptually intuitive, the model imposes several restrictions on the buffering and pipelining of memory accesses. One of the least strict models is the *release consistency* model (RC) [7]. It requires synchronization accesses in the program to be identified, but allows significant overlap of memory accesses. In this section we explore the impact that the memory consistency model has on the benefits of prefetching.

The BASE architecture, which we have been using so far, uses RC. The writes are buffered and the reads are allowed to bypass pending writes. However, since the second-level caches in the BASE architecture allow only a single access to be outstanding (they are not lockup-free), a read miss must stall for the write access at the head of the write buffer to complete before it can be issued to the secondary cache and the bus.[6] The SC implementation that we evaluate allows write accesses to be buffered. However, any subsequent read (whether a hit or a miss in the first-level cache) cannot be issued until all previous writes have completed. Figure 10 shows the performance of the benchmarks under the SC and RC models. Overall, the speedup due to prefetching is similar for both SC and RC, although RC does better in terms of absolute performance.

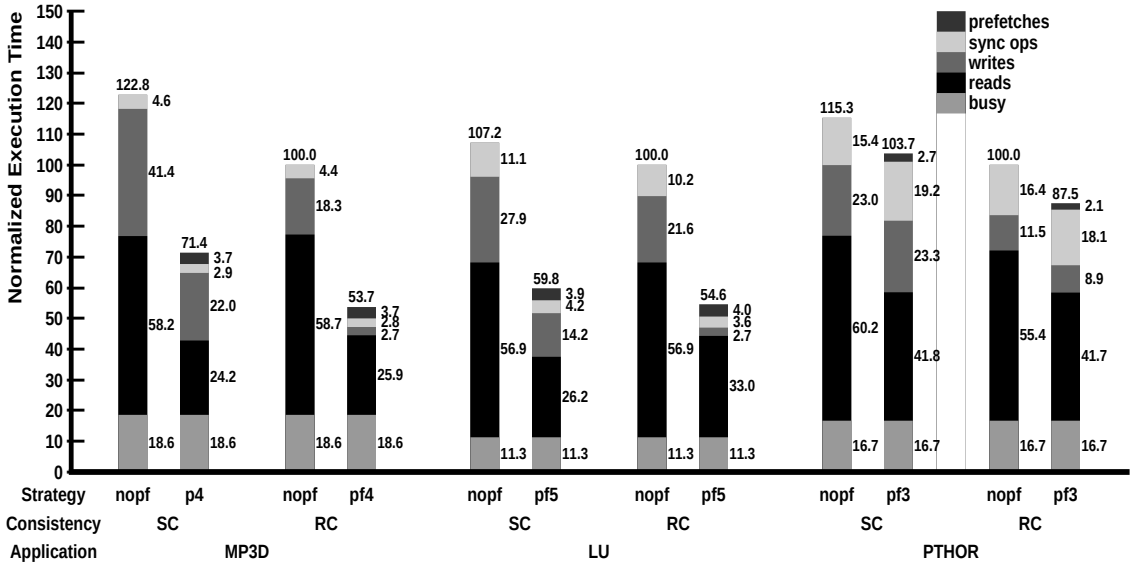

Figure 10: Sequential versus release consistency.

Comparing the no-prefetching case under the two models, the main performance increase for RC comes from the reduced stall time waiting for the write buffer, since the reads can bypass the pending writes [6]. The latency experienced once the read is past the write buffer (shown by the *reads* section of the bars in the figure) is similar for both models, as is to be expected.[7] When prefetching is added, the latencies decrease considerably. However, the latencies for prefetched reads are smaller for SC than for RC. The reason is that the interval between issue of prefetch and issue of binding read is larger for SC than for RC, since reads get blocked for writes in SC. As a result, there is a greater likelihood that the prefetched data is already present in the RAC for SC.

The read-exclusive prefetching helps decrease the time reads spent waiting for the write buffer significantly for both SC and RC, though the stall time is still sizeable for SC (22% for MP3D, 14% for LU, and 23% for PTHOR). The reason is that the writes still have to go over the cluster bus to the RAC to get ownership, and this takes a significant amount of time. If we were prefetching directly into the primary/secondary cache, this time could be decreased considerably more for SC. This would also help RC, but not as much, since the current stall times are not as large. Therefore, if the prefetching coverage is high and read-exclusive prefetches bring data directly into the processor cache, we will see the absolute performance under SC and RC models get quite close to each other.

[6]The BASE architecture therefore does not achieve the full potential of RC. We will fix this problem later in Section 4.2.4.

[7]For PTHOR we see a difference in total read stall time due to the fact that fewer reads are issued as the application runs faster, since a process spends less time polling an empty task queue waiting for work from other processors.

4.2.3 Separate Prefetch Issue Buffer

In the BASE architecture, prefetches go directly into the write buffer along with regular write accesses. This has two disadvantages. First, the issue of prefetch requests may be delayed if they get queued behind writes, which in turn may increase the latency for subsequent reads. Second, there is a greater chance of the write buffer getting full, thus increasing the stall time for writes. To explore the performance impact of these factors we introduced a separate prefetch issue buffer to the BASE architecture, which is similar to the write buffer except that it only handles prefetch requests. The prefetch buffer is 16 entries deep and arbitrates for the bus independently of the secondary cache.

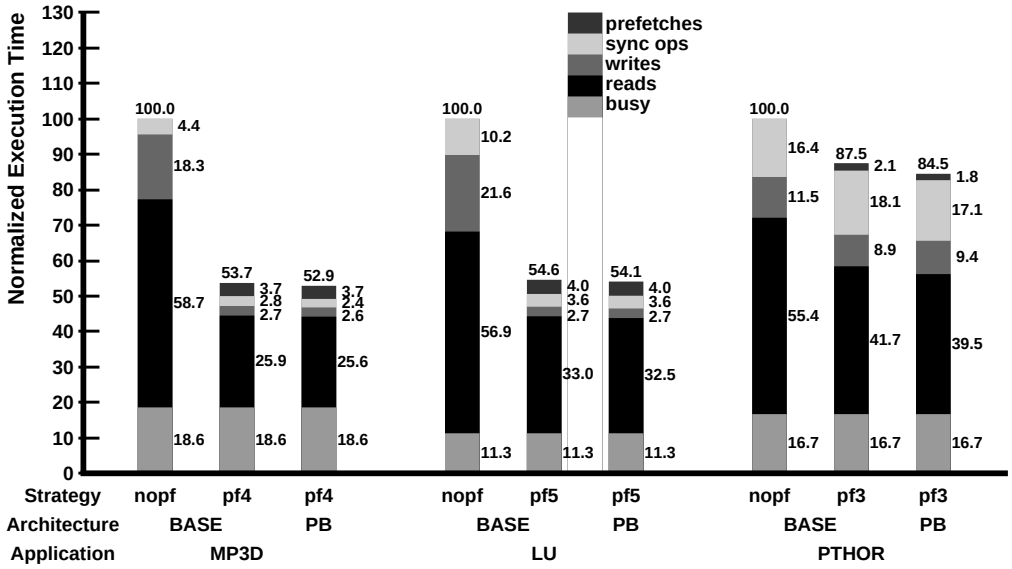

Figure 11: Benefits of prefetch buffers.

Prefetch buffers improve performance slightly for each of the benchmarks, as shown in Figure 11, where the PB architecture is the BASE architecture plus prefetch buffers. The improvement is caused primarily by a reduction in read miss time, due to earlier prefetch issue. The effects on write stall time varied: MP3D benefited from emptier write buffers, while PTHOR (which uses very little read-exclusive prefetching) suffered from increased bus contention. Synchronization time was decreased for both MP3D and PTHOR, due to the fact that releasing synchronization operations (e.g. unlock operations) were not queued behind prefetches in the write buffer.

Overall, given the small performance benefits of separate prefetch issue buffers, the added cost of providing them in an architecture may not be worthwhile. The benefits could be higher if we made extensive use of large block prefetches, but as we showed in Section 4.1, large block prefetches can lead to hot-spotting and reduced performance.

4.2.4 Lockup-free Caches

The BASE architecture that we have used so far permits only a single cache access to be outstanding at a time. Most existing architectures have this constraint due to the added hardware complexity required to allow multiple outstanding accesses. Consequently, primary read misses must stall for the write at the head of the write buffer to complete, although they are not required to do so by the release consistency model. This accounts for a substantial portion of execution time (18% for MP3D, 22% for LU, and 11% for PTHOR in the no-prefetch strategies as shown in Figure 12) which is reduced but not eliminated by prefetching. To examine the effects of removing write buffer waiting time, we evaluate a version of the BASE architecture which has lockup-free caches [12, 22]. A lockup-free cache permits multiple accesses to be outstanding, thus allowing a read miss to proceed without waiting for writes to complete. The results of our experiments are presented in Figure 12, where the LFC architecture is the BASE architecture with lockup-free caches.

Lockup-free caches dramatically reduce write stall times for all applications, both with and without prefetching. The reason why write stall times persist at all is the unusual case where a read miss occurs for a cache entry which is also the target of a pending write miss. Since only a single access can be outstanding at a time

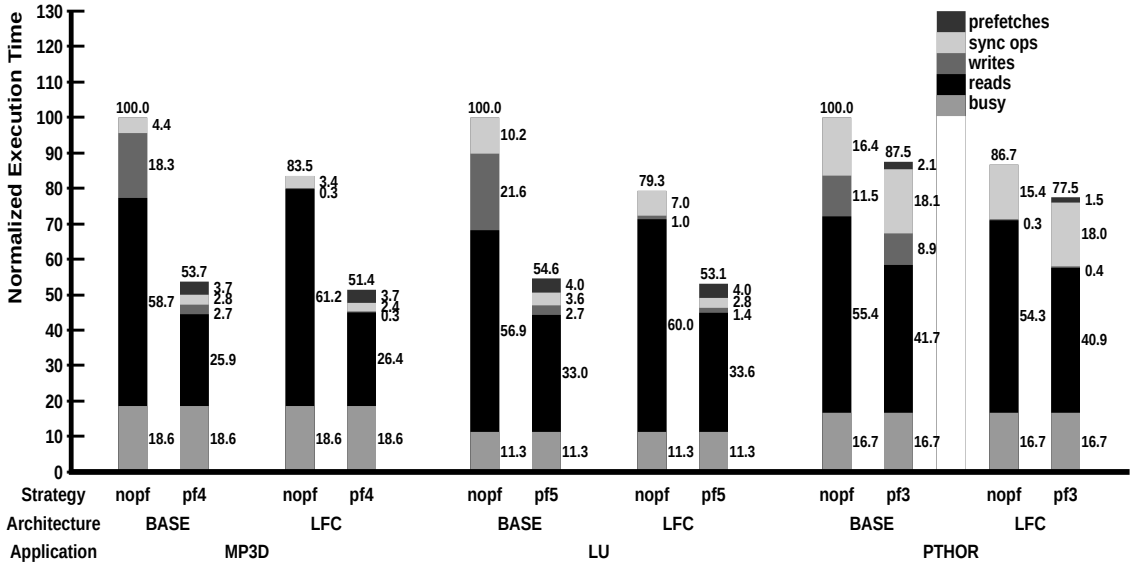

Figure 12: Lockup-free caches.

for a given cache entry, the read must wait until the write completes. Both MP3D and LU showed a slight increase in read stall time due to increased contention as the processors were able to issue references at a faster rate.[8] Figure 12 also shows a slight decrease in synchronization stall time for each application, due to the fact that the writes before the releasing synchronization operation can be pipelined and can thus complete earlier. Similar observations have been made in [6].

In summary, lockup-free caches along with release consistency effectively eliminate almost all stall time due to writes, even in the case where no prefetches are issued. It is therefore sufficient to focus on decreasing the latency of read misses rather than write misses in architectures that provide aggressive implementations of weaker consistency models with lockup-free caches.

4.2.5 Prefetching into Primary Cache

While prefetching into the RAC reduces miss latencies to that of local memory access time, the latency of a miss can be eliminated altogether if we prefetch directly into the processor's primary cache. The analysis presented in Section 3 indicated that significant performance gains may be achieved by prefetching into primary cache (see Table 2). In order to do this, the processor caches must be lockup-free since multiple prefetch requests can be outstanding at a time. In this subsection, we evaluate this potential for increased performance by expanding upon the lockup-free cache architecture introduced in the previous subsection to also include prefetching into the primary cache.[9] This architecture is labeled LFC-PC in Figure 13 and Table 7.

There is an additional cost associated with prefetching directly into the primary cache, however. During the time when the cache is being filled with a prefetched line, the cache tags will be busy and the processor will not be able to execute any loads or stores. We model this effect by stalling the processor for four cycles, since the cache line size is four words, whenever a prefetched line is inserted into the cache. This additional stall time is included as prefetch overhead (the uppermost section of the bar) for the LFC-PC architecture in Figure 13.

Table 7 shows that prefetching directly into the primary cache improves the performance of each benchmark by increasing the hit rates. However, the gains are much smaller than the upper bounds predicted in Section 3 for two reasons: (i) the hit rates are much lower than those assumed in Section 3, and (ii) the latency of read misses increases significantly beyond that of the no-prefetch case. Ideally, each prefetched reference will hit in the cache, thus reducing the number of misses by the prefetching coverage factor. There are two reasons why

[8] Although the average read miss time increased for PTHOR, the total read stall time decreased due to fewer reads being issued. This is caused by the fact that each process spends less time examining its task queue waiting for work when the application runs faster.

[9] We do not evaluate the case of prefetching only into secondary cache. The disadvantages of prefetching directly into primary cache might be stalling the processor while the primary cache tags are busy, and replacing more important data in the primary cache. For the DASH prototype, writing a prefetched line into the secondary cache would stall the processor for the same amount of time regardless of whether the line is written into primary cache as well. Also, we do not expect primary cache interference to be a problem since the caches are not tiny and because the data that is prefetched is likely to be referenced soon.

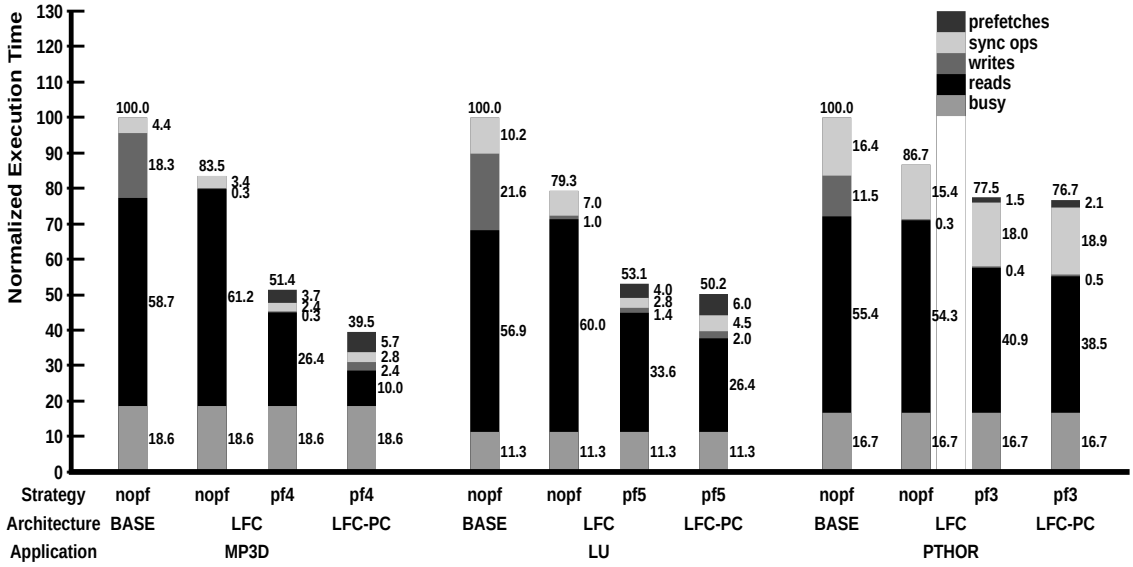

Figure 13: Prefetching into primary cache with lockup-free caches.

Table 7: Statistics on prefetching into primary cache with lockup-free caches.

Application	Architecture	Strategy	Primary Read Hit Rate	Secondary Write Hit Rate	Average Read Miss Latency	Speedup Relative to LFC	Speedup Relative to BASE
MP3D	LFC	nopf	81.9%	68.5%	72.0	1.00	1.20
		pf4	81.9%	68.5%	25.8	1.62	1.95
	LFC-PC	pf4	96.7%	96.1%	103.4	2.11	2.53
LU	LFC	nopf	58.9%	55.0%	67.3	1.00	1.26
		pf5	58.9%	55.0%	32.6	1.49	1.88
	LFC-PC	pf5	84.7%	95.0%	130.9	1.58	1.99
PTHOR	LFC	nopf	75.1%	48.4%	63.8	1.00	1.16
		pf3	75.1%	48.4%	45.9	1.11	1.29
	LFC-PC	pf3	81.4%	61.8%	67.7	1.12	1.30

this does not occur: firstly, cache interference, caused either by the same processor in the form of replacements or by other processors in the form of invalidations; and secondly, an insufficient interval between prefetch issue and reference. Although the latter case results in a miss, there can still be a performance benefit due to reduced miss latency. Each benchmark behaved differently with regard to these effects. For MP3D, which shows the greatest improvement of all the benchmarks, the problem is interference between processors due to the sharing of space cells. In the interval between prefetching and referencing a space cell, another processor occasionally writes to the same space cell, invalidating it from the processor's cache before it can be referenced. In the case of LU, interference between the pivot column and the column being modified often causes a prefetched line to be replaced in the cache before it can be referenced. PTHOR is qualitatively different from the other benchmarks since many of the prefetches were added to improve coverage, regardless of whether they could be issued early enough to completely hide the latency. Consequently, a prefetched line is frequently referenced before it has had time to return to the cache, resulting in a miss. The latency of such a miss is smaller than that of a non-prefetched miss.

Intuitively, one might expect the latency of a non-prefetched read miss to be similar regardless of whether prefetching is used. Instead, as we see in Table 7, the latency is much larger when prefetching into primary cache than in the no-prefetch case (almost twice as large, in the case of LU). The reason for this is increased contention in the memory system as the processors issue references at a much faster rate, resulting in much longer queueing delays.

Overall, the performance of an architecture with lockup-free caches can be significantly improved by prefetch-

ing directly into primary cache. Performance is improved by 153% and 99% over the BASE architecture for MP3D and LU, respectively. However, the gains for PTHOR are limited to a 30% improvement because prefetches are not issued early enough to arrive in the cache before being referenced. Although we modeled stalling the processor whenever prefetched data was written into the cache, we found the resulting performance loss to be very small in comparison to the benefits of reduced read latencies.

4.2.6 Read versus Read-Exclusive Prefetching

In this subsection, we evaluate the benefits of read-exclusive prefetching, which reduces write miss latencies. Unlike read misses, which directly stall the processor for their entire duration, the rate at which writes are retired affects performance more indirectly, since writes can be buffered. A processor stalls while waiting for writes to complete in two situations: (i) when executing a write instruction if the write buffer is full, and (ii) during a read miss if previous writes must complete before the read miss can proceed. The impact of the former effect can be reduced through larger write buffers. Throughout this paper, we use 16 entry write buffers, which we have found to be large enough to make this effect negligible. The impact of the latter effect depends on whether reads are permitted to bypass writes (as allowed by the release consistency model), and whether the cache permits multiple outstanding accesses (as allowed by a lockup-free cache). Since the benefits of read-exclusive prefetching are sensitive to whether the caches are lockup-free, we evaluate both the BASE and LFC architectures.

The prefetching studies presented so far have all used read-exclusive prefetches whenever appropriate. The percentage of prefetches which were read-exclusive is 100% for MP3D, 50% for LU (the columns to be modified are prefetched read-exclusively, the pivot columns are not), and 28% for PTHOR. The percentage is low for PTHOR since many of the prefetched objects are linked lists and other structures that are not modified after initialization. To evaluate the case where read-exclusive prefetches are not available, we replace each read-exclusive prefetch with a read prefetch of the same address. The results of our experiments are presented in Figure 14.

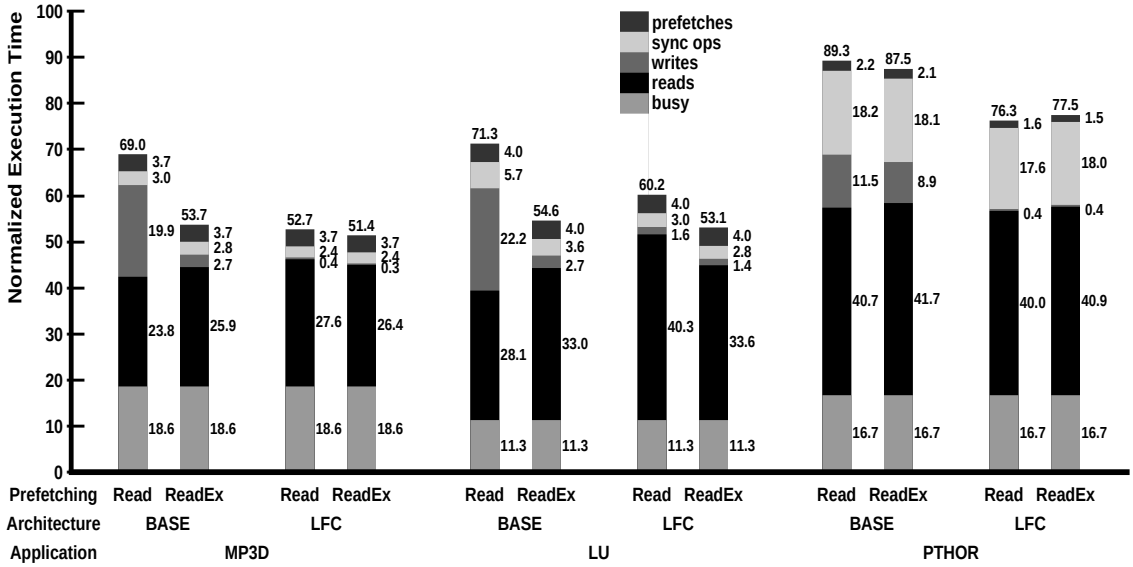

Figure 14: Read versus read-exclusive prefetching.

For the BASE architecture, read-exclusive prefetching substantially reduces write buffer stall time and hence execution time for MP3D and LU. The gains of read-exclusive prefetching are very small for PTHOR, since only a small fraction of write misses are being prefetched due to the difficulty we had in determining what to prefetch. The overall benefits are also reduced by the fact that the ratio of reads to writes is greater than eight for PTHOR, while it is closer to two for MP3D and LU, as shown in Table 1.

For the LFC architecture, Figure 14 shows that read-exclusive prefetching offers very little benefit in reducing write buffer stall times. This is to be expected, since this time has already been eliminated by the lockup-free caches [6]. However, read-exclusive prefetching can still improve performance by reducing the traffic associated with cache coherency. This benefit occurs in situations where a line is read before it is written. By prefetching the line with exclusive ownership to begin with, it is not necessary to fetch the line twice—first in a shared state during the read, and then with ownership to complete the write. Figure 14 shows a small improvement in read

stall times for MP3D and LU due to lower contention in the memory system as a result of reduced coherency traffic.

In summary, read-exclusive prefetching can provide significant performance benefits in architectures that have not already eliminated write stall times through aggressive implementations of weaker consistency models with lockup-free caches. Even if write stall times cannot be further reduced, read-exclusive prefetching can improve performance somewhat by reducing the traffic associated with cache coherency.

5 Related Work

First, let us consider the simplest kind of prefetching that occurs in cache-based systems where the line size is greater than one word—whenever the processor misses on a location, the hardware ensures that other words that fall into the same cache line are prefetched. Such prefetching relies on the spatial locality in program accesses and has the advantage that it requires no software intervention. Large cache lines also successfully utilize the block transfer capabilities of modern memory systems. Although the utility of multiword cache lines is almost universally accepted nowadays [23, 21], recent data [5, 15, 25] show that cache lines should be kept small for multiprocessors. The primary reason is that the spatial locality is considerably reduced in the process of parallelizing applications for multiprocessors, and large cache lines can result in a significant increase in memory system traffic. Also, unless prefetching or non-blocking loads are used, no overlap between cache miss service and computation is possible. Consequently, we do not expect prefetching due to large cache line sizes to play a significant role in solving the latency problem in parallel computers.

Lee [14, 16] evaluates a binding, hardware-controlled prefetching scheme on an architecture with software-based cache coherence. Instruction lookahead is used at runtime to prefetch data that has been explicitly marked as cacheable at compile time. The target of the prefetch is a FIFO queue, called the prefetch buffer, associated with each processor. The hardware is assumed to predict branch outcomes randomly and continue prefetching along the predicted path. If the prediction turns out to be incorrect, the instruction lookahead buffer and the prefetch buffer are flushed. Furthermore, since the prefetching is binding, synchronization operations inhibit further prefetching. One major disadvantage of Lee's scheme is that the prefetching window is limited by the size of the instruction lookahead buffer and the corresponding prefetch buffer. In a non-binding software-controlled scheme, such as ours, the prefetch instructions can be moved up by an arbitrary amount. This ability to move up prefetches by large amounts is especially important in large scale multiprocessors where latencies can be quite large. Lee also found the effectiveness of his prefetching scheme to be seriously limited by branches and synchronization instructions.

Gornish, Granston, and Veidenbaum [9] evaluate a binding, software-controlled prefetching scheme. They present prefetching algorithms that focus on array references and attempt to find the earliest point where prefetches can be issued without violating memory coherence. Since prefetching is binding, all control and data dependencies have to be carefully considered. The primary prefetch mechanism they use is an asynchronous block transfer from the global memory to the local memory of a processor (the local memory could be a software controlled cache). The proposed schemes are evaluated using a large number of numerical subroutines. Although the speedups predicted from static analysis are quite high, over 2-fold, the speedups obtained using detailed simulations are limited to 10-20%. This work differs from ours in that only numerical subroutines and array references are considered. Since our current focus was not automated prefetching, we evaluate the benefits for complete and complex applications, such as MP3D and PTHOR. Secondly, their prefetching constraints are quite different from ours because of binding versus non-binding prefetching. Finally, their techniques attempt to prefetch all references from the innermost loop as a single large block transfer. Our results (see data for LU) show that large block transfers can result in hot-spotting and can reduce performance. Prefetching via use of software pipelining appears to be a better technique. We also note that large block transfers in a hardware cache coherent multiprocessor, such as DASH, must be implemented as a series of accesses to the constituent cache lines—it is not possible for the memory controller to simply pick the data from the main memory as a block and send it to the requesting processor. The reason is that different cache lines within the block may be present in a modified state in caches of different processors.

Porterfield [21] evaluates both hardware-controlled and software-controlled non-binding prefetching schemes for a uniprocessor. The evaluation is done using several scientific applications from RiCEPS, the Rice Computer Evaluation Program Suite. Hardware prefetching is simulated by using long cache lines (1 word, 16 word, and 32 word cache lines). The results are mixed with significant improvement in hit rates for some applications and little improvement for others. The improvements are expected to be even less for multiprocessors, as was

discussed earlier in this section. For software prefetching, a compiler algorithm is presented which prefetches array references in vector loops one iteration ahead. This is similar to the pipelining approach that we use for MP3D and LU, though we often issue the prefetches several iterations ahead to compensate for the large latency in multiprocessors. Porterfield also discusses prefetching overhead, increased traffic, and percentage of useful prefetches. The prefetching overhead was found to be quite high due to prefetch of unnecessary data. While the software mechanism presented is essentially the same as our non-binding approach, the study is done in the context of a uniprocessor, where many of the multiprocessor issues do not arise. Also no detailed speedup results are presented.

6 Conclusions

In this paper we have evaluated the performance benefits of non-binding software controlled prefetching. Our study is done in the context of the Stanford DASH multiprocessor, a large scale cache-coherent shared memory machine being built at Stanford. In adding prefetching to applications, we found both the non-binding and software-controlled aspects to be essential to obtaining good performance benefits. Non-binding prefetching allowed us to fetch data far in advance, even though there was a possibility that data may not be used or that it may be modified before use. Software control allowed us to be selective in only prefetching data that was likely to miss in the cache, thus reducing overhead. It also allowed us to move prefetches earlier than would have been possible in hardware schemes that use a limited lookahead window.

Our results on the BASE architecture (where the data is only prefetched into a cluster cache) show that prefetching increased the performance of MP3D and LU applications by 86% and 83% respectively. To achieve this performance gain, only 16 lines of code had to be added to MP3D and 8 lines to LU. Our experience with these two applications indicates that use of software pipelines to do prefetching works better than prefetching large blocks of data, as indicated in some earlier studies. The latter has the potential of causing hot-spots, thus degrading performance. In contrast to LU and MP3D, the PTHOR application showed an improvement of only 14% and required an additional 29 lines of code. The major hurdle in getting additional speedup was issuing the prefetches early enough, so that the data arrives by the time it is needed. This was very difficult to achieve for PTHOR because it makes extensive use of linked lists.

We also evaluated the impact of architectural variations on the benefits of prefetching. Our results for different cache sizes showed that the benefits of prefetching can be substantial even when the caches are very large. The reason is that in parallel machines processors are frequently stalled while fetching data that another processor has just produced (e.g. the pivot column in the LU application). Such stalls cannot be reduced by going to larger caches, but prefetching can help. Our studies comparing sequential and release consistency models showed that gains from prefetching remain large under both models, but the absolute performance of the applications is noticeably better under the release consistency model. The most effective hardware optimizations were the combination of lockup-free caches and prefetching directly into the primary cache. Lockup-free caches used in an aggressive implementation of the release consistency model eliminated stalls due to writes, while prefetching into the primary cache reduced stalls due to reads by improving the cache hit rate. Although we modeled the processor stalls when prefetched data was being written into the cache, we found the performance loss due to such stalls to be minimal. When prefetching into the primary cache, we improved performance by 150%, 100%, and 30% over the non-prefetching case for MP3D, LU, and PTHOR respectively.

Finally, in this paper we have focused on the performance potential of software controlled prefetching. Our technique was to manually add prefetch instructions to the programs. We are currently exploring how a compiler can add the prefetching automatically, as well as studying other applications.

7 Acknowledgments

We would like to thank Jeff McDonald, Ed Rothberg and Larry Soule for their help with the MP3D, LU and PTHOR applications, respectively. We would also like to thank Earl Killian of Mips Computer Systems, Inc. for help with the basic architecture simulator and Kourosh Gharachorloo for modifying the simulator to model lockup-free caches. We thank Helen Davis and Steve Goldschmidt for creating and supporting Tango. Dan Lenoski and Jim Laudon provided insightful discussions. Todd Mowry and Anoop Gupta are supported by DARPA contract N00014-87-K-0828. Anoop Gupta is also supported by an NSF Presidential Young Investigator Award.

References

[1] Sarita Adve and Mark Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[3] James Archibald and Jean-Loup Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[4] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, 21(2):9–21, February 1988.

[5] Susan J. Eggers and Randy H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 2–15, May 1989.

[6] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[7] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[8] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[9] E. Gornish, E. Granston, and A. Veidenbaum. Compiler-Directed Data Prefetching in Multiprocessors with Memory Hierarchies. In *International Conference on Supercomputing*, 1990.

[10] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, June 1988.

[11] J. S. Kowalik, editor. *Parallel MIMD Computation : The HEP Supercomputer and Its Applications*. MIT Press, 1985.

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[13] David J. Kuck, Edward S. Davidson, Duncan H. Lawrie, and Ahmed H. Sameh. *Experimental Parallel Computing Architectures: Volume 1 – Special Topics in Supercomputing*, chapter Parallel Supercomputing Today and the Cedar Approach, pages 1–23. North-Holland, New York, 1987.

[14] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Data prefetching in shared memory multiprocessors. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 28–31, August 1987.

[15] Roland L. Lee, Pen-Chung Yew, and Duncan H. Lawrie. Multiprocessor cache design considerations. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 253–262, June 1987.

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

[17] Dan Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of Scalable Shared-Memory Multiprocessors: The DASH Approach. In *Proceedings of COMPCON'90*, pages 62–67, 1990.

[18] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

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

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

[21] Allan K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Department of Computer Science, Rice University, May 1989.

[22] Christoph Scheurich and Michel Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 118–125, 1988.

[23] Alan Jay Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[24] Larry Soule and Anoop Gupta. Parallel Distributed-Time Logic Simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[25] 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, February 1990.

[26] Wolf-Dietrich Weber and Anoop Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 273–280, June 1989.