Sequential Hardware Prefetching in Shared-Memory Multiprocessors

Fredrik Dahlgren, *Member, IEEE Computer Society*,
Michel Dubois, *Senior Member, IEEE*, and Per Stenström, *Member, IEEE*

Abstract—**To offset the effect of read miss penalties on processor utilization in shared-memory multiprocessors, several software- and hardware-based data prefetching schemes have been proposed. A major advantage of hardware techniques is that they need no support from the programmer or compiler.**

***Sequential prefetching* is a simple hardware-controlled prefetching technique which relies on the automatic prefetch of consecutive blocks following the block that misses in the cache, thus exploiting spatial locality. In its simplest form, the number of prefetched blocks on each miss is fixed throughout the execution. However, since the prefetching efficiency varies during the execution of a program, we propose to adapt the number of prefetched blocks according to a dynamic measure of prefetching effectiveness. Simulations of this adaptive scheme show reductions of the number of read misses, the read penalty, and of the execution time by up to 78%, 58%, and 25% respectively.**

Index Terms—**Hardware-controlled prefetching, latency tolerance, memory consistency models, performance evaluation, sequential prefetching, shared-memory multiprocessors.**

I. Introduction

SHARED-MEMORY multiprocessors offer significant performance improvements over uniprocessors at an affordable cost while preserving the intuitively appealing programming model provided by a single, linear memory address space. With the advent of ultra-fast uniprocessors and of massively parallel systems, the shared-memory access penalty (i.e., the average time each processor is blocked on an access to shared data) becomes a significant and often a dominant component of the program execution time. This penalty mostly comes from the large latency associated with the traversal of the processor-memory interconnect. Private caches in conjunction with hardware-based cache coherence maintenance [27] are quite effective at hiding this latency. In a system with such caches, the memory access penalty is mainly dictated by the cache miss rate of read requests, i.e., the fraction of read requests that miss in the cache and possibly incur the latency of the processor-memory interconnect. Besides cold and replacement misses that also appear in uniprocessors, another type of misses called invalidation misses (or coherence misses) [13] show up in multiprocessors using write-invalidate cache coherence protocols and can be quite dominant. By contrast, the latency of write requests can easily be hidden by appropriate write buffers and weaker memory consistency models [1], [11], [15], [16].

Prefetching is a common technique to reduce the read miss penalty. Prefetching relies on predicting which blocks currently missing in the cache will be read in the future and on bringing these blocks into the cache prior to the reference triggering the miss. Prefetching approaches proposed in the literature are software- or hardware-based. Software-controlled prefetching schemes [5], [22], [23] rely on the programmer/compiler to insert prefetch instructions prior to the instructions that trigger a miss. In addition, both the processor and the memory system must be able to support prefetch instructions which can potentially increase the code size and the run-time overhead. By contrast, hardware-controlled prefetching schemes, such as the ones proposed by Hagersten [18] and Baer and Chen [2], relieve the programmer/compiler from the burden of deciding what and when to prefetch. Usually, these schemes take advantage of the regularity of data accesses in scientific computations by dynamically detecting access strides. Unfortunately, these solutions require complex hardware and do not work as well for applications with irregular strides. These observations have motivated us to seek for simpler and more general techniques.

The main theme of this paper is to evaluate two simple hardware-controlled prefetching techniques with a low implementation cost. These techniques are non-binding, meaning that the prefetched block is brought into the (second-level) cache and remains subject to invalidations by the coherence protocol.

The basic approach of both techniques is called *sequential prefetching*; the cache controller prefetches K consecutive blocks on a cache miss, where K is the *degree of prefetching*. This degree of prefetching is a critical parameter, which must be adjusted to reflect the spatial locality of shared-data accesses. To understand the usefulness of the schemes, we review how the miss rate, traffic, and write penalty are affected by the cache block size based on previous experimental results in Section II [12], [13], [17]. In general, the miss rate and the traffic as a function of the block size follow a U-shaped curve: whereas a larger block size reduces the cold miss rate, it often increases the overall miss rate as a result of false sharing.

We describe the two hardware schemes in Section III: fixed and adaptive sequential prefetching. Fixed sequential prefetching is the simplest form of prefetching in the sense that the degree of prefetching remains constant throughout the execution. An obvious limitation of this simple scheme is that the

Manuscript received Nov. 3, 1993.

F. Dahlgren and P. Stenström are with the Department of Computer Engineering, Lund University, P.O. Box 118, S-221 00 LUND, Sweden; e-mail: fredrik@dit.lth.se.

M. Dubois is with the Department of Electrical Engineering-Systems, University of Southern California, Los Angeles, CA 90089-2562.

IEEECS Log Number D95006.

degree of prefetching should vary with the workload and should also vary dynamically during the execution of the same application: e.g., in the start-up phase when the cold miss rate component is high, K should be large, whereas in steady-state, K should adjust to the spatial locality of the true-sharing miss component. Our adaptive sequential prefetch technique does just that and its hardware cost is limited to two extra bits per cache block and three 4-bit counters per cache.

The sequential prefetching schemes evaluated in this paper were originally proposed in [8]. In that paper, we presented some preliminary simulation results based on simplistic architectural assumptions. We extend the work in [8] by providing a more solid performance evaluation based on a detailed architectural simulator that also models network contention for a worm-hole routed mesh. Sections IV and V present the results from this evaluation using six benchmark applications from the scientific and engineering domain. We find that fixed sequential prefetching with $K = 1$ manages to reduce the read penalty by more than 30% for four of the six applications in spite of its simplicity. The adaptive technique, although slightly more expensive to implement, results in a considerably lower miss rate and acceptable traffic overhead because of its dynamic nature. We finally relate our results to others in Section VI and conclude the paper in Section VII.

II. Impact of Block Size on Access Penalties and Traffic

The effectiveness of prefetching on a cache miss depends on the amount of spatial locality and coherence activity. In this section, we review the effects of cache block size variations on the miss rate, traffic, and write penalty in multiprocessors. Increasing the block size is actually the simplest and most obvious prefetching technique: If we double the block size, each miss brings twice as much data into the cache.

A number of studies have reported on the effects of block size variations on the miss rate in the context of shared-memory multiprocessors [12], [13], [17]. To understand whether larger block sizes may help to reduce the overall miss rate, we need to distinguish between the different types of misses. In the case of infinite caches, the overall miss rate consists of two components: *cold misses* and *coherence misses* which we define in the table below. In the case of finite-size caches, there is also a third component, called *replacement misses*, which we ignore here for simplicity but which we will consider later in the paper when we experimentally compare the prefetching schemes.

Cold miss	The block has never been referenced by the processor.
Coherence miss	The block has previously been referenced to by the same processor, but another processor has subsequently invalidated the copy.

Coherence misses can further be classified as *true sharing misses* and *false sharing misses*. Loosely speaking, false sharing is the sharing of a block without actual sharing of data whereas true sharing is due to the sharing of data items. False sharing misses are thus useless in that they are not needed for the correct execution of the program, whereas true sharing misses are due to actual communication of new values and are essential.

Dubois et al. [12] analyzed how the number of cold misses, true sharing misses, and false sharing misses depends on the block size in the context of infinite caches. Their results show that the number of cold misses is significantly reduced for all applications as the block size increases. However, the number of false sharing misses increases with the block size B and even in the range of typical cache line sizes, i.e. between 32 and 256 bytes, false sharing can be quite significant. These effects are illustrated in Fig. 1. A somewhat unexpected observation was also made regarding the true sharing miss rate. It was observed that the number of true sharing misses decreases with increasing block sizes for most of the applications, indicating that a larger block size has a potential to exploit spatial locality in shared data references. Eggers and Katz [13] also evaluated the miss rate variations with the block size in four applications with block sizes from 4 bytes to 32 bytes for finite size caches. Their conclusions were similar, except for two applications, in which the increase in the number of coherence misses offsets the decrease in the cold and replacement miss rates for larger block sizes, resulting in an increasing overall miss rate.

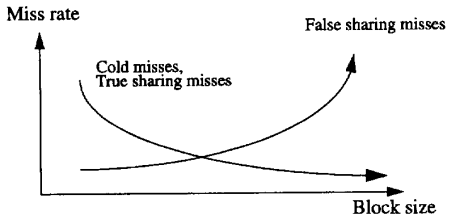


Fig. 1. How cold, true sharing, and false sharing misses depend on the block size.

Besides the miss rate, which is an important performance metric since it predicts the read penalty, the memory traffic is also affected by the block size and can increase memory-system contention if the bandwidth of the processor-memory interconnect is not sufficient. Gupta and Weber [17] have evaluated the data and invalidation traffics for various block sizes in the case of infinite caches and observed the same trends as for the miss rate: The overall traffic as a function of the block size follows a U-shaped curve.

Another important metric is the write penalty, i.e., the time the processors are stalled due to outstanding write requests. In the context of shared-memory multiprocessors with write-invalidate protocols, a write request may invalidate blocks residing in other caches. Gupta and Weber [17] further observed that the number of invalidation requests as a function of the block size also follows a U-shaped curve, which can be explained by the basic behavior of the cold, true, and false sharing components of the miss rate. Under sequential consistency [19], the write penalty is difficult to hide whereas under

relaxed memory consistency models, such as release consistency [15], the write penalty can be hidden almost completely [16]. Consequently, the write penalty as a function of the block size follows a U-shaped curve under sequential consistency.

To summarize, when we move to larger blocks the read penalty and the traffic follow a U-shaped curve. In addition, under sequential consistency, the write penalty also follows a U-shaped curve. The behavior of these metrics are explained by the decomposition of the miss rate in its three basic components. The minimum of these curve depends on the application. A good prefetching scheme should reduce the cold miss and true sharing miss rates without increasing the false sharing miss component. As we will see in the subsequent sections, sequential prefetching reduces the cold and true sharing miss components in a similar fashion as a larger block size does without increasing the false sharing miss component.

III. Two Simple Hardware-Controlled Sequential Prefetching Techniques

In this section, we describe in some detail the implementation of the fixed and adaptive sequential prefetching techniques. We begin by describing the processor environment on which we have based our performance evaluation, i.e., our simulations. The description is kept general, and implementation details of our simulation model will be described in Section IV.

A. Simulated Processing Node Architecture

Since the proposed prefetching schemes are non-binding, they are applicable to standard processors with blocking loads. Furthermore, in contrast to software-controlled prefetching, the processors do not need to support specific prefetch instructions. The environment to which these processors are interfaced is shown in Fig. 2.

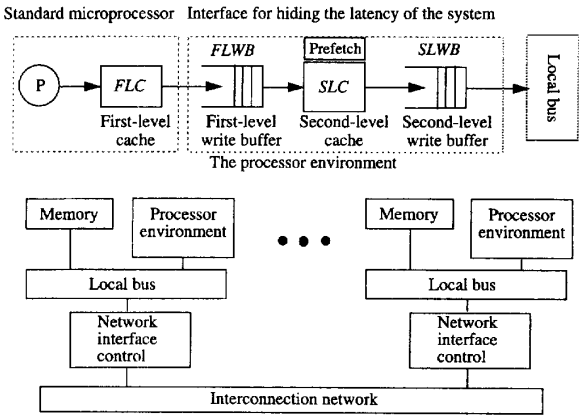


Fig. 2. The processor environment and the simulated architecture.

According to Fig. 2, the processing node consists of a processor, a first-level cache (*FLC*), a second-level cache (*SLC*), a first- and second-level write buffer (*FLWB* and *SLWB*), a local bus, a network interface controller, and a memory module. The *FLC* is a direct-mapped write-through cache with no allocation of blocks on write misses and is blocking on read misses. Writes and read miss requests are buffered in the *FLWB*. The second-level cache (*SLC*) is a direct-mapped write-back cache. For both prefetching techniques, we only prefetch into the *SLC*. In order to support prefetching, the *SLC* is lockup-free [28]. In addition, a second-level write buffer (*SLWB*) keeps track of outstanding requests (*SLC* read miss, prefetch, and write requests). No more than one request to the same block is allowed to be issued to the system, others are just kept in the *SLWB* while waiting for the pending request to that block to complete. Moreover, a read miss request may bypass write requests if they are for different blocks. How the *SLC* and the *SLWB* handle prefetch requests will be discussed later.

The *SLC* controller deals with all the complexities of the cache coherence protocol and the prefetching mechanisms. The techniques proposed do not rely on a specific cache coherence protocol, which is why it will be described later in Section IV. The *SLC* snoops on the local bus for consistency actions: If an invalidation to a block residing in the *SLC* is detected, the copy of the block is invalidated in both the *SLC* and the *FLC*. Requests from the local bus have priority over those from the first level write buffer. These interferences with the processor accesses are tolerable because most accesses hit in the *FLC*. In Section IV, we further discuss how this processing node interfaces to the rest of the multiprocessor memory system.

B. Fixed Sequential Prefetching—A Simple Hardware Prefetching Scheme

By fixed sequential prefetching we mean that K consecutive blocks are prefetched into the *SLC* on a reference to a block, i.e., blocks $n + 1 \ldots n + K$ are prefetched upon a reference to block n, if they are not present in the cache. Sequential prefetching has been extensively studied in the context of uniprocessors [25], [26] but, to our knowledge, have never been considered for general applications on multiprocessors. Although many sequential strategies have been proposed for uniprocessors, we have restricted ourselves to prefetching on a miss in the *SLC*. When a reference misses in the *SLC*, the miss request is sent to memory, and the cache is searched for the K consecutive blocks directly following the missing block in the address space. The blocks among the K consecutive blocks that are not present in the *SLC* and have no pending requests in the *SLWB* are prefetched. We refer to K as the *degree of prefetching*.

Fig. 3 shows the mechanism of the fixed sequential prefetching scheme. As a cache lookup is made for block address n, the next block address $(n + 1)$ is calculated. On a read miss, a read request is issued to the memory system and is kept in the *SLWB*. In the next cache cycle, the calculated address $(n + 1)$ is directed to the cache, and a cache lookup is made. If the block is not present in the cache, a prefetch request is issued and is kept in the *SLWB*. During that time, the subsequent block address is calculated $(n + 2)$. The number of iterations is determined by the degree of prefetching. The processor is blocked only during the time it takes to handle the first read

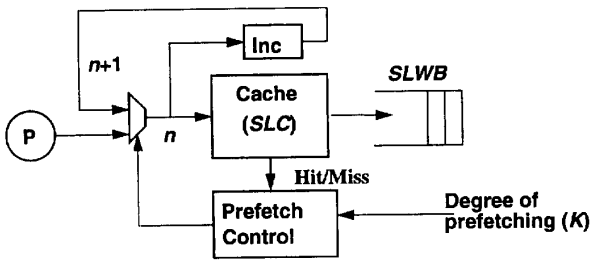


Fig. 3. The fixed sequential prefetching mechanism.

miss. Since the prefetch requests are issued one at a time and are pipelined in the memory system, they can be overlapped with the original read request.

Besides the simple extensions in the *SLC* to incorporate fixed sequential prefetching, the memory system must be able to handle three new network commands: a prefetch request and two reply messages denoted PreData and PreNeg. Whereas PreData carries the prefetched block, PreNeg tells the cache that the prefetch request can not be satisfied because the memory copy is in a transient state-some other cache is reading or writing to it.

Despite its simplicity, especially for the special case of $K = 1$ (also known as *one-block look-ahead*), this scheme is very effective for applications where the cold and true-sharing read misses exhibit a high spatial locality as we will see in Section V. To build intuition into this, let us assume a block size of B. For an implementation of fixed sequential prefetching with $K = 1$, the cold and true sharing miss rate components are essentially reduced to the level that corresponds to a block size of 2B. However, the false sharing miss rate component is expected to be essentially the same as that of no prefetching with block size B, because coherence is maintained on blocks of size B. To realize this, let us assume that two consecutive blocks of size B are fetched into the cache of processor i and that processor j writes to the second block. While processor i would experience a false sharing miss if it only accesses the first block and if the block size is 2B, the false sharing miss does not occur if sequential prefetching is applied to a block size of B. In fact, there are situations where the number of false sharing misses can be reduced by sequential prefetching, since a block that has been invalidated because of false sharing can be subsequently prefetched. Another issue is the amount of traffic as compared with a system with a block size of 2B. We differentiate between the traffic due to the transfer of block data and the traffic due to invalidations. We expect the data traffic to be the same as the one generated by the cold and true sharing miss rates for a block size of 2B plus the traffic generated by the false sharing miss rate for a block size of B. However, the invalidation traffic may actually increase, and consequently, the write penalty under sequential consistency may increase.

The optimum degree of prefetching, K, is dictated by the spatial locality of the application and by the block size. Depending on how far the block size B is from the minimum of the U-shaped curve for the miss rate, we should use a smaller or higher value for K. If the prefetching degree is too high, the cache is loaded with useless data, resulting in excessive memory traffic. However, as we will see, the best value of K varies dynamically during the execution of most benchmarks. For example, at the beginning of the execution, most misses are cold misses and their spatial locality is high, which suggests a large K. Conversely, when the caches reach steady-state, the true and false sharing miss components dominate the miss rate. Ideally, K should be adjusted to minimize the true sharing miss component while retaining the same false sharing miss rate. In the next section, we show how we can achieve this by an adaptive sequential prefetching scheme.

C. Adaptive Sequential Prefetching—Further Exploiting the Spatial Locality of Reads

The mechanism behind the adaptive scheme is basically the same as that of fixed sequential prefetching. For example, prefetching is activated by a read miss and blocks are prefetched into the *SLC*. In contrast to fixed sequential prefetching, however, the degree of prefetching is not fixed; rather it is controlled by a register, the LookaheadCounter.

The adaptive sequential prefetching scheme relies on adjusting the degree of prefetching (the value of the LookaheadCounter) dynamically by counting the useful prefetches, i.e., prefetched blocks that are actually referenced during their lifetime in the cache. To explain how this is achieved, we will first focus on how the algorithm measures the prefetch efficiency and then how the LookaheadCounter is adjusted to a certain prefetch efficiency. The mechanisms needed to achieve this task—two bits per cache line and three counters per cache—appear in Table I.

TABLE I
MECHANISMS NEEDED TO IMPLEMENT ADAPTIVE SEQUENTIAL PREFETCHING

PrefetchBit (per cache line)	Used to detect useful prefetches (needed when prefetching is turned on.)
ZeroBit (per cache line)	Used to detect when a prefetch would have been useful (needed when prefetching is turned off.)
LookaheadCounter (per cache)	The current degree of prefetching
PrefetchCounter (per cache)	Counts the number of prefetches that have been returned after each read miss
UsefulCounter (per cache)	Counts the number of useful prefetches

Conceptually, the algorithm measures the prefetch efficiency by counting the fraction of prefetched blocks that are referenced by the processors. If this fraction exceeds a preset threshold, the degree of prefetching is increased and, if it is below another preset threshold, the degree of prefetching is decreased.

The basic mechanisms used to measure the prefetch efficiency consist of two counters (the PrefetchCounter and the UsefulCounter) and a PrefetchBit per cache line which are all cleared from the very beginning. The fraction of useful prefetches is established as the ratio of the UsefulCounter and the PrefetchCounter as follows. The number of prefetched blocks is counted by incrementing the PrefetchCounter whenever a

prefetch acknowledgment is received from the memory system, independent of whether the prefetch was accepted (PreData) or not (PreNeg) (e.g., if the memory block was in a transient state and neither clean nor dirty) by the memory system. To count the number of prefetched blocks that are referenced, the PrefetchBit of a prefetched block is set; when a block is accessed with its PrefetchBit set, the UsefulCounter is incremented and the PrefetchBit is cleared.

Every time the PrefetchCounter reaches its maximum (i.e., it wraps around), the value of the UsefulCounter is matched against two preset thresholds to determine if the LookaheadCounter—initially set to one—should be changed. If the UsefulCounter exceeds the upper threshold, we are in a phase of execution where the program could benefit from a higher degree of prefetching and therefore the LookaheadCounter is incremented. If the UsefulCounter is lower than the lower preset threshold, the amount of prefetching is too high and the LookaheadCounter is decremented. Finally, if the UsefulCounter has a value between the two thresholds, the LookaheadCounter is not affected. In all cases, the UsefulCounter is cleared. In our evaluation, we have considered counters modulo 16 (4 bits).

When the LookaheadCounter reaches zero, prefetching is turned off. To turn it back on, we use the following mechanism. When a block is received on a read miss and prefetching is turned off, the ZeroBit in the corresponding *SLC* block frame, which is initially cleared, is set to indicate that the following block in the address space could have been prefetched and the PrefetchCounter is incremented. On a read miss, a cache lookup is made to the previous block (by address); if it hits and the ZeroBit is set, the UsefulCounter is incremented and the ZeroBit is cleared. The ZeroBit of a block is also cleared when the block is accessed and the LookaheadCounter is not zero to keep the number of ZeroBits that have been previously set to a minimum.

Except for the modifications at the *SLC*, the adaptive sequential prefetching scheme does not imply any new modifications besides those of the fixed sequential prefetching scheme.

IV. Simulation Environment, Architectural Model, and Benchmark Programs

The architectural simulation model is built on top of the CacheMire Test Bench [4]; a program-driven functional simulator of multiple SPARC processors. It consists of two parts:

1) a functional and
2) an architectural simulator.

The SPARC processors in the functional simulator issue memory references and the architectural simulator delays the processors according to the behavior and timing of the target memory system we model. Consequently, a correct interleaving of memory references is maintained by keeping track of the global time.

For all the experiments, we have assumed the same system-level model, i.e., the same interconnection network and cache coherence protocol. Inter-node cache coherence is maintained by a full-map, write-invalidate directory-based protocol similar to Censier and Feautrier's [6]: A bit vector is associated with each memory block to point to the processor nodes with copies in their caches. A second-level cache read miss causes a read request to be sent to the *home* memory module (the node at which the corresponding page is allocated). If the home is the *local* node and if the memory block is clean, the miss is serviced locally. Otherwise, reading the block from the home node requires two node-to-node traversals if the block is clean. If the block is dirty, the block must be read from the *remote* node, i.e., the node which has the dirty copy, and the home node must be updated which requires another two node-to-node traversals. A write request to a shared or invalid copy results in an ownership request to be sent to home. The home node transmits invalidation requests to all nodes with copies, waits for acknowledgments from these nodes, and finally, sends an invalidation acknowledgment to the requesting node notifying that no other copies exist. Acquire and release requests are supported by a queue-based lock mechanism similar to the one implemented in DASH [21]. To further reduce coherence overhead for synchronization accesses, we have allocated a whole memory block for each synchronization variable.

The simulated system consists of 16 processors, one per node. The first-level cache (*FLC*) is assumed to be a simple and fast on-chip cache (4 Kbytes direct-mapped and write-through). We model both infinite and finite (16 Kbytes) second-level caches (*SLC*) in order to isolate the effect of replacement misses. In the case of finite caches, we model all effects due to replacements including the handling the write-back requests. The *FLWB* is limited to eight entries and the *SLWB* is limited to 16 entries. Under release consistency (henceforth referred to as RC), the *SLC* allows multiple outstanding requests provided they are to different blocks. The timing model is based on a processor clock rate of 100 MHz. The *FLC* has the same cycle time as the processor, and a read hit in the *FLC* does not stall the processor. An *FLC* block fill takes 30 nsec. The access time of the *SLC* is 30 nsec (33 MHz). The *SLC* and its write buffer are connected to the network interface and the local memory module by a 256-bit wide split transaction bus clocked at 33 MHz. Thus, it takes 30 nsec to arbitrate for the bus and 30 nsec to transfer a request or a block. The network interface controller is clocked at 33 MHz. Furthermore, the memory is assumed to be fully interleaved with a cycle time of 90 nsec. The page size is 4 Kbytes and the pages are allocated in a round-robin fashion. To avoid triggering a page fault on a prefetch, prefetching across page boundaries does not occur.

The interconnection network is a worm-hole routed synchronous 2D-mesh, using 64-bit links and is clocked at 100 MHz to match the speed of the processors, resulting in an aggregate bandwidth of 800 Mbytes/sec out from and into each node. In contrast to the DASH-network that has one request and one reply mesh, the network simulated consists of only one mesh. Table II shows the processor stall times due to a read request in the case of a conflict-free system and an average distance of 2.67 links for each node-to-node traversal in

TABLE II
LATENCY NUMBERS FOR READ REQUESTS

Read from *FLC*	1 pclock
Read from *SLC*	6 pclocks
Read from Local Memory	30 pclocks
Read from Home (two node-to-node traversals)	75 pclocks
Read from Remote (four node-to-node traversals)	143 pclocks

TABLE III
BENCHMARK PROGRAMS

Benchmark	Description	Data Sets
MP3D	3-D particle-based wind-tunnel simulator	10 K parts, 10 time steps
Water	N-body water molecular dynamics simulation	288 molecules, 4 time steps
Cholesky	Cholesky factorization of a sparse matrix	The matrix bcsstk14
LU	LU-decomposition of a dense matrix	200x200 matrix
PTHOR	Distributed time digital circuit simulator	RISC circuit, 1000 time steps
Ocean	Ocean basin simulator	128x128 grid, tolerance 10^{-7}

the mesh. Since we model contention for all components in the system, all requests will normally take longer times.

We use six benchmark programs to drive our simulation models. Four of them are taken from the SPLASH suite (MP3D, Water, Cholesky, and PTHOR) [24]. These and the other two applications (LU and the multi-grid version of Ocean) have been provided to us from Stanford University. They are all written in C using the ANL macros to express parallelism [3] and compiled by `gcc` (version 2.1) using optimization level O2. We summarize them in Table III.

While the scheduling policy can affect the effectiveness of the prefetching algorithms, we have not particularly studied this aspect. Instead, our results are derived using the scheduling policies built into the applications and which are described in the SPLASH report [24]. (LU and Ocean are not described in [24] and use static scheduling.) Since the benchmarks use quite limited data sets, the initialization section is large compared to the parallel section. Because of this, all statistics are gathered in the parallel sections.

V. EXPERIMENTAL RESULTS

This section reports simulation results on fixed and adaptive sequential prefetching. In order to concentrate on the read penalty reduction provided by sequential prefetching, we will use release consistency for most of our experiments unless explicitly stated otherwise. Except for Section V.A, all simulations are carried out with a block size of 32 bytes. Sections V.A through V.D assume infinite second-level caches, whereas Section V.E concentrates on the effect of cache sizes. Finally, Section V.F deals with the impact of processor speed and network bandwidth on the results. To simplify the wording, we refer to fixed sequential prefetching as *fixed prefetch*, and to adaptive sequential prefetching as *adaptive prefetch* throughout this section.

A. Effects on Misses and Traffic of Fixed Sequential Prefetching

Some preliminary experiments with fixed prefetch showed that it was not useful to consider values of K greater than 1. Although a large K effectively cut the cold and true sharing misses in program phases where the spatial locality was high, the traffic was drastically increased for phases and applications where the spatial locality was low. Therefore, all results presented in this section are for a degree of prefetching equal to one ($K = 1$), although it is worth mentioning that a fixed K larger than 1 could be useful if the network bandwidth can sustain the extra traffic.

In order to explore the effects of sequential prefetching on the miss rate and traffic and compare it to a design with a larger block size, we simulate three designs:

1) no prefetching with the block size B, referred to as NoPrefetch(B),
2) fixed prefetch with block size B, FixedPrefetch(B), and
3) no prefetching with the block size 2*B, NoPrefetch(2B).

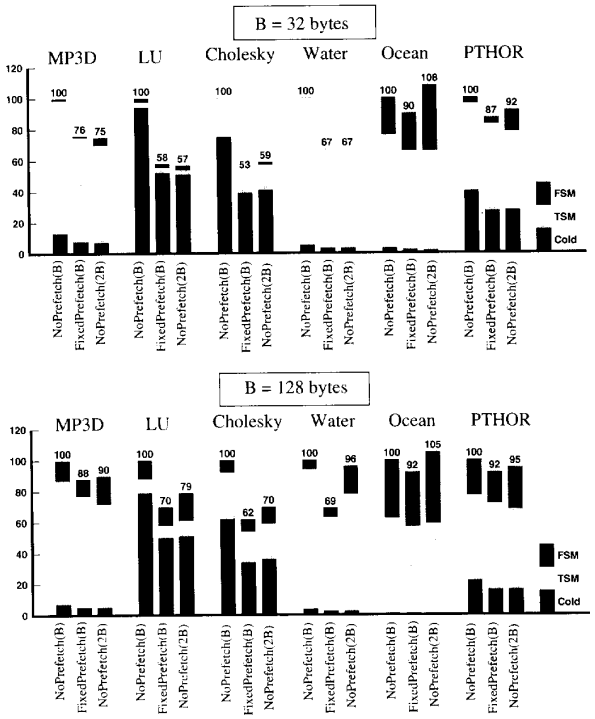


Fig. 4. Cold, true sharing, and false sharing misses relative to NoPrefetch(B). The upper diagram corresponds to B = 32 bytes and the bottom diagram corresponds to B = 128 bytes.

Concentrating first on the miss rate effects, we show in Fig. 4 the relative numbers of misses in the second level cache for the three designs. The upper diagram shows the results for B = 32 bytes and the lower diagram is for B = 128 bytes. For each application, the bars for designs 2) and 3) are normalized to design 1) (the leftmost bar). Each bar consists of three sections that, from the bottom to the top, correspond to cold misses, true sharing misses, and false sharing misses, respectively. The distinction between true sharing misses and false sharing misses are made according to the classification scheme presented by Dubois et al. in [12]: A miss is classified as a

cold miss if it has never been fetched into the cache, a true sharing miss if a new value is communicated during the rest of the lifetime of the block, and a false sharing miss otherwise.

Read misses for blocks that are being prefetched but have not yet arrived at the cache is counted as hits in the diagrams. This is justified by the observation that the stall time for such misses is on average significantly shorter than the total time of servicing a read miss. To realize this, assume the worst-case scenario that the processor accesses two consecutive blocks missing in the cache by two piggy-backed load instructions. While the processor has to wait for the first block to return, the cache has issued a prefetch request for the second block. Since the original miss request and the prefetch request go to the same memory module, the time difference between the two blocks' arrival into the cache is approximately given by the memory cycle time. We have observed that the read stall time due to pending prefetches constitutes less than a percentage of the total read stall time.

Regarding the reduction of the number of cold and true sharing misses for B = 32, we would expect FixedPrefetch(B) and NoPrefetch(2B) to be almost equally successful and Fig. 4 confirms this expectation. This verifies that fixed prefetch with $K = 1$ and no prefetching with a block of twice the size achieve similar cold and true sharing miss rates. The cold miss rate is cut by almost a factor two for most applications. Although the impact of cold misses on the overall miss rate is small for most applications, this is not the case for Cholesky and LU. In these applications, which are examples of direct methods, the cold miss component remains high throughout the whole execution.

Fig. 4 also shows that the true sharing miss rates in MP3D and Water are reduced, indicating that the spatial locality in true sharing misses is significant. In MP3D, true sharing misses are due to particle collisions and in Water they are due to molecule interactions. In both these applications the number of true sharing misses is significantly cut by doubling the block size.

Although the data in Fig. 4 demonstrate that FixedPrefetch(B) and NoPrefetch(2B) cut the numbers of cold and true sharing misses nearly identically, a small discrepancy exists, which can be explained as follows. Whereas sequential prefetching always prefetches the subsequent block of size B, doubling the block size means prefetching either the previous or subsequent half block of size B. We speculate that this explains the small differences between sequential prefetching at a block size B and no prefetching at a block size 2B in the upper diagram of Fig. 4.

Regarding the effects on false sharing misses, Fig. 4 shows a few differences between the three designs. Whereas NoPrefetch(B) and FixedPrefetch(B) exhibit nearly the same number of false sharing misses, NoPrefetch(2B) slightly increases the number of false sharing misses. This effect is especially significant for Ocean and PTHOR. All applications we have used in this evaluation are carefully written, and have very little false sharing at a block size of 32 bytes.

In order to take a closer look at the impact of false sharing effects, we have included results for B = 128 bytes in the bottom diagram of Fig. 4. Overall, it is unclear whether the spatial locality of misses is large enough to gain from sequential prefetching or an increased block size for large blocks. If we study the number of misses shown in the bottom diagram, and compare the results with the upper diagram (B = 32 bytes), we can see that the gain from FixedPrefetch(B) over NoPrefetch(B) is less but still significant. The reduced benefit is partly due to a smaller reduction of true sharing misses. Comparing FixedPrefetch(B) with NoPrefetch(2B) in the bottom diagram, we see that fixed sequential prefetching is more efficient at reducing the number of true sharing misses than if we increase the block size. Again, we speculate that the reason for this is that sequential prefetching always prefetch subsequent blocks in contrast to a block having twice the size.

The number of false sharing misses for 128 byte blocks is much larger than for 32 byte blocks.[1] Comparing the amount of false sharing for the three designs, we clearly see that NoPrefetch(B) and FixedPrefetch(B) exhibit a similar number of false sharing misses whereas NoPrefetch(2B) shows substantially more false sharing misses. This confirms our expectations that FixedPrefetch(B) does not significantly affect the false sharing read miss rate.

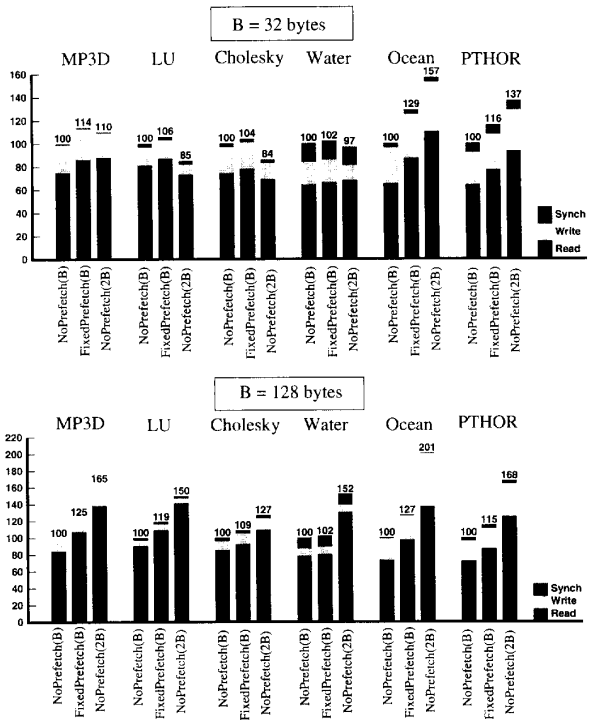


Fig. 5. Read, write,and synchronization traffic relative to NoPrefetch(B). The upper diagram corresponds to B = 32 bytes and the bottom diagram corresponds to B = 128 bytes.

Fig. 5 shows the network traffic for the above executions. The traffic is measured in number of bytes sent through the network during the execution, and is normalized to the traffic for NoPrefetch(B). Moreover, we have broken down the traffic into three components that, from bottom to the top, correspond

1. In [12] we detected more false sharing for MP3D and Ocean. This is mainly attributable to our use of smaller data sets in that study.

to the read traffic including all traffic associated with read and prefetch requests; the write traffic including all traffic associated with ownership acquisitions and invalidations; and finally synchronization traffic including all traffic associated with acquiring and releasing locks.

Starting with the four applications to the left that show a significant reduction of misses under FixedPrefetch(B) and NoPrefetch(2B), we first note that for LU and Cholesky with B = 32 bytes (the upper diagram) the read traffic is only slightly higher for FixedPrefetch(B) than for NoPrefetch(B). This indicates that almost all prefetches were useful. For NoPrefetch(2B), the read traffic is reduced as compared to NoPrefetch(B), which means that both subblocks of size B in each fetched block were almost always referenced. In addition, since there is less overhead to fetch one block of size 2B than two blocks of size B also helps reducing the number of messages and the overall traffic. MP3D and Water also show high prefetch efficiency (fraction of prefetches that are useful). While both fixed prefetch at a block size B and no prefetch at a block size 2B only marginally increase the amount of data sent over the network if the spatial locality is high, NoPrefetch(2B) results in less overhead in terms of headers than FixedPrefetch(B).

Continuing with the two applications to the right (Ocean and PTHOR) that exhibit low spatial locality, we can see from Fig. 5 that the traffic for NoPrefetch(2B) is much higher than for FixedPrefetch(B). The explanation is that since the spatial locality is low, both schemes result in higher read traffic. Moreover, NoPrefetch(2B) also suffers from increased false sharing which contributes to increase its read traffic.

Comparing the write traffic for FixedPrefetch(B) with NoPrefetch(B) we see in Fig. 5 that it is almost the same. Even if fixed sequential prefetching can never take advantage of the spatial locality of writes and thus reduce the write traffic, as NoPrefetch(2B) can, it keeps the write traffic at about the same level as NoPrefetch(B) regardless of the block size. This is in contrast to NoPrefetch(2B) that actually can increase the write traffic in the presence of false sharing as we clearly see in the Ocean and PTHOR applications in Fig. 5.

There are situations where prefetching might increase the write traffic. For example, one processor may prefetch a block with a dirty copy in another processor's cache, while the copy is still being modified. One could expect that preventing the prefetch of blocks with a dirty copy in another cache would make the increase in write traffic negligible. To test this intuition, we have compared fixed prefetch of one block with prefetching on dirty and without prefetching on dirty (i.e., only clean blocks in memory can be prefetched; prefetches of other blocks are acknowledged with a PreNeg message). The comparison was carried out for all six applications with block sizes of 16 and 32 bytes, and showed that when only clean blocks were prefetched, the number of read misses was reduced to a much smaller extent. In addition, the effect this had on the write traffic was small.

In summary, we have observed that fixed prefetch of the next consecutive block can exploit the spatial locality of reads as does doubling the block size, i.e., reduce the cold miss rate and true sharing miss rate to almost the same extent. However, by contrast to a larger block size where the false sharing miss rate is increased, it is essentially unaffected by fixed prefetch. Moreover, we have shown that the same reasoning applies to the read traffic; fixed prefetch can result in less read traffic than doubling the block size in the presence of false sharing and in about the same write traffic as no prefetch with the same block size.

B. Adaptive Prefetch vs. Fixed Prefetch and No Prefetch

Fig. 6 compares the read stall times with 32 byte blocks for adaptive and fixed prefetch, both normalized to the design with no prefetching. The read stall times for LU, Cholesky, and Water are drastically reduced by the adaptive scheme. For Cholesky, the adaptive scheme reduces the read stall time by 58% (from 100% to only 42%) of the original. For Ocean, the read stall time is slightly higher compared to fixed prefetch but still lower than for no prefetching. For PTHOR, a slightly higher read stall time for adaptive prefetch can be seen, mainly caused by contention, which we discuss next.

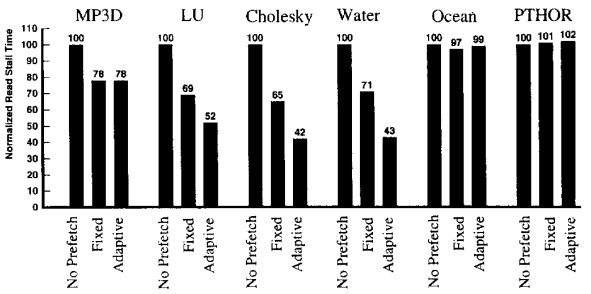


Fig. 6. Relative read stall times for fixed and adaptive prefetch normalized to no prefetching.

The simulated scheme for changing K (i.e., the value of LookaheadCounter) in adaptive prefetch is the following: After 16 prefetches (PrefetchCounter, modulo-16, wraps around) the value of UsefulCounter is checked. If UsefulCounter > 12, then K is incremented. If UsefulCounter < 8, then K is decremented (never below 0). If UsefulCounter < 3, then K is shifted right, i.e., divided by 2. If $K = 0$ (i.e., no prefetching) and UsefulCounter > 6, then K is set to 1. Since the local bus is clocked 3 times faster than the memories and since the adaptive scheme might send "bursts" of prefetch requests to the same memory module (because of the page-level interleaving) on read misses, the input buffers of the memory module might have to be large. To avoid this drawback, prefetch requests arriving to an almost full memory buffer immediately trigger a PreNeg message in the output buffer. The effect of this is twofold:

1) the number of messages in the input buffer is kept low, and
2) since this PreNeg counts as a useless prefetch, the adaptive protocol tends to reduce the degree of prefetching in the case of contention at memory modules.

Read misses to blocks that are being prefetched but have not yet arrived at the cache (the pending prefetch request is book-

kept in the *SLC*) are counted as hits by the UsefulCounter, since it indicates the spatial locality to be high. Except for MP3D, we have observed that the latency of serving such misses to blocks that are being prefetched constitutes a negligible fraction of the overall read stall times in Fig. 6; it did not exceed 0.65% of the read stall time. This indicates that the prefetches are issued sufficiently ahead of the point where the blocks are actually referenced. For MP3D, 16% of all prefetch requests arrived too late so the processor had to stall. However, the average stall time in these cases was on average only 31 pclocks as compared to the average read miss stall time of 132 pclocks. Although this indicates that a prefetch request, which was issued only a few pclocks after the read request, has been further delayed because of contention in the network and in the nodes, the effect this has on the read stall time is only 2.4%.

Fig. 7 shows the number of cold and coherence misses. For LU, Cholesky, and Water, the number of cold misses is drastically reduced. The spatial locality is high, which makes most prefetches useful, and the adaptive scheme uses a high degree of prefetching. In addition, the numbers of coherence misses for MP3D, Cholesky, and Water are also drastically reduced because of the high spatial locality of true sharing misses and of the low fraction of false sharing misses for B = 32 bytes. In the case of Ocean, the number of cold misses is reduced, whereas the number of coherence misses is slightly higher than for fixed prefetch. The reason for this is that the adaptive scheme uses a high K at the beginning when cold misses dominate, and a low K for the rest of the program, which is dominated by coherence misses. For PTHOR, the number of prefetches issued under the adaptive scheme is only about 50% of those issued under fixed prefetch.

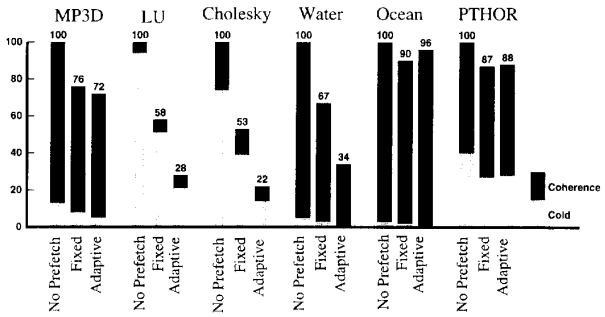


Fig. 7. Relative number of cold and coherence misses for fixed and adaptive prefetch normalized to the case with no prefetching.

Fig. 8 shows the prefetch efficiency and the number of issued prefetch requests under fixed and adaptive prefetch. *Prefetch efficiency* is defined as the fraction of all prefetch requests that are useful, i.e., prefetch requests that bring blocks that are later accessed. The prefetch efficiency of MP3D is between 40% and 50%. At that efficiency, the adaptive scheme issues about the same number of prefetches as does fixed prefetch, i.e., the LookaheadCounter is one most of the time. This can be seen in the right diagram; the number of issued prefetches for MP3D is about the same for fixed and adaptive prefetch, and consequently the prefetch efficiency is about the same. Applications with a high spatial locality of misses (e.g., LU, Cholesky, and Water) show a high prefetch efficiency. As a result, the number of issued prefetches is much higher for adaptive prefetch compared to fixed prefetch. Because of the larger number of issued prefetches, the prefetch efficiency is decreased for the adaptive scheme. Applications with low spatial locality of misses (e.g., Ocean and PTHOR) show low prefetch efficiency, and consequently the number of prefetches issued under adaptive prefetch is much lower compared to fixed prefetch.

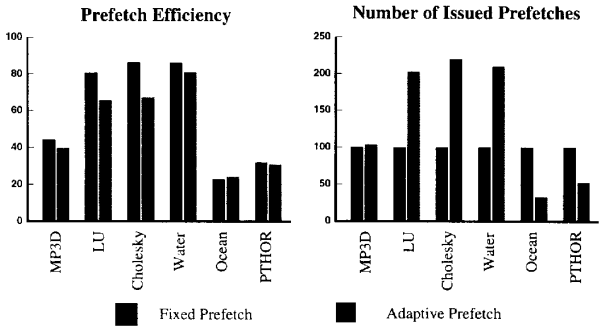


Fig. 8. Prefetch efficiency in percent (left) and number of issued prefetch requests in percent (right) for fixed and adaptive prefetch. The number of issued prefetch requests under adaptive prefetch is normalized to the corresponding number under fixed prefetch.

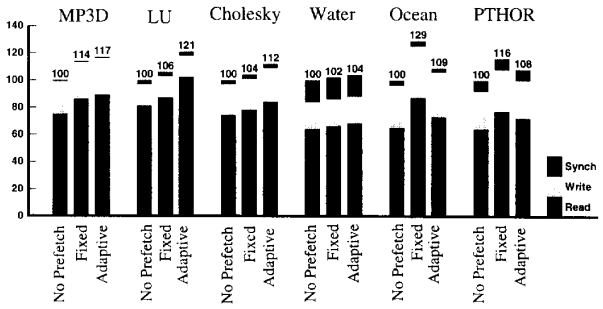


Fig. 9. Read, write, and synchronization traffic for fixed and adaptive prefetching relative to the case of no prefetching.

Fig. 9 shows the total amount of network traffic for fixed and adaptive prefetch normalized to the design with no prefetching with the same breakdown as in Fig. 5. It is interesting to study the correlation between the total traffic and the number of read misses from Fig. 7; a decrease (increase) in the number of read misses leads to increased (decreased) traffic. The large reduction of the number of read misses for LU, Cholesky, and Water comes from a large number of prefetches, and thus an increased read and write traffic. By contrast, the read and write traffic for Ocean and PTHOR is lower for adaptive prefetch as compared to fixed prefetch. This follows directly from a low prefetch efficiency for those programs (the spatial locality of misses is small); the reduction of read misses is small and the adaptive scheme issues fewer prefetches. We can also see that the adaptive prefetching scheme

does not significantly increase the write traffic, indicating that the effect of increased unnecessary sharing is small.

C. Dynamic Behavior of the Adaptive Prefetching Scheme

Fig. 10 shows the cold and coherence miss rates during the execution of LU with a block size of 32 bytes. The upper diagram shows a normal execution, and the lower diagram shows the behavior under the adaptive scheme. The left Y-axis shows the miss rate (%) as the total number of misses for the whole system divided by the total number of shared reads. The right Y-axis in the lower diagram shows the average value of LookaheadCounter for all 16 processors. One sample is taken every 10,000 shared reads, and the miss rates correspond to the average since the last sample, with the values of the LookaheadCounters taken at the time of the sample.

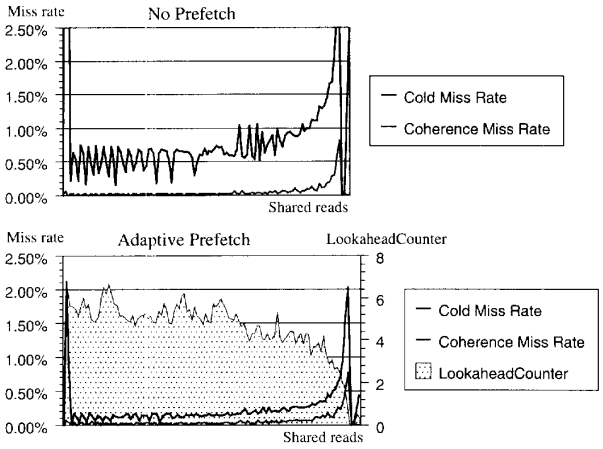


Fig. 10. The adaptive prefetch behavior for LU with a block size of 32 bytes.

The top diagram shows the normal execution of the program. LU has a large cold miss rate at the beginning (8% with no prefetching), but in contrast to usual assumptions about cold misses, the cold miss rate remains the dominant part of the miss rate throughout the whole execution, and goes up to 3% at the end. The number of coherence misses is negligible throughout the major part of the execution, but shows a peak in the end (although not as high as the peak in cold misses at the same time).

The bottom diagram shows the same program with the adaptive scheme. Since the values on the X-axis are the total number of shared reads for the system instead of time, and the number of reads is about the same for the two executions, it is easy to relate the diagrams to each other. Observe that the left Y-axis (the miss rate) is scaled equally in both diagrams. As can be seen, the value of the LookaheadCounter raises immediately to 7, and the prefetches manage to keep the cold miss rate down to below 2% for the first peak (originally 8%). Since the major part of the misses are cold misses, and the prefetch efficiency is high, the LookaheadCounter stays at a high level. At the end of the execution, the number of coherence misses increases. As a result, the LookaheadCounter goes down (because of a smaller fraction of useful prefetches), until prefetching completely stops at the point where the number of coherence misses peaks. The cold misses at the end do not have enough spatial locality to increase the degree of prefetching.

The choice of thresholds for the adaptive scheme, i.e., the values that determine when to increase or decrease the value of the LookaheadCounter, affects the number of prefetches sent to the network. Lower thresholds means that the LookaheadCounter tend to have a higher value, and the number of prefetches increases. The thresholds affect not only the number of prefetches, but also the prefetch efficiency (i.e., the fraction of the prefetches that are useful). Higher thresholds mean that a larger number of useful prefetches is needed to increase the degree of prefetching and favor higher prefetching efficiency at the possible expense of longer read stall times. With the thresholds we use (8 is the lower mark and 12 is the upper mark), the prefetch efficiency must be higher than 12/16 = 75% in order to increase the degree of prefetching, while the prefetch efficiency must be lower than 8/16 = 50% for the degree of prefetching to be reduced. The lower threshold thus determines the lowest spatial locality of read misses needed to start prefetching meaning that if it is lower, prefetching will stop.

We have also studied whether the ZeroBit heuristic discussed in Section III.C is effective in enabling prefetching, once it has been switched off. Specifically in MP3D, Ocean, and PTHOR, we found that prefetching was disabled many times and in these applications the ZeroBits were effective in switching on prefetching again.

D. Effects on Total Execution Time

Figs. 11 and 12 show the total execution times under sequential consistency (SC) and release consistency (RC), respectively, for fixed and adaptive prefetch normalized to an execution with no prefetching. To understand how well the prefetching schemes manage to reduce the read stall time, we have broken down the execution time into *busy time* (the time the processor is busy), *read stall*, *write stall*, *acquire stall* (the time during which the processor is waiting for acquiring a synchronization lock), and *release stall* (the time during which the processor is waiting for the release of a synchronization lock to complete). Under SC, we lump acquire and release stall into a group called synch stall. As can be seen from Fig. 11, the reduction of the total execution time under SC is as much as 27% for Cholesky, although parts of it (about 5%) stems from better load balancing and is not due to less read-miss penalty. Fig. 12 shows the total execution time under Release Consistency (RC). Under RC, the processor never has to stall on write or release requests, and consequently, the write stall and release stall times can be completely hidden [16] and are not shown. Although processors do not have to stall due to outstanding write requests, too many global writes might fill the *FLWB* and *SLWB*, and force a processor to stall. This is referred to as *buffer stall* and is a new stall component in the execution times of Fig. 12. Since the read stall time now constitutes a larger fraction of the execution time, we would expect to see a larger reduction of the execution time compared to SC. Although the read stall time is reduced under RC, the reduction is limited by an increased contention due to higher

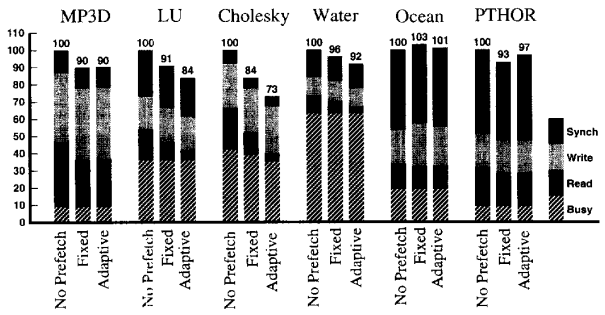


Fig. 11. Total execution time under SC for fixed and adaptive prefetch normalized to the case with no prefetching.

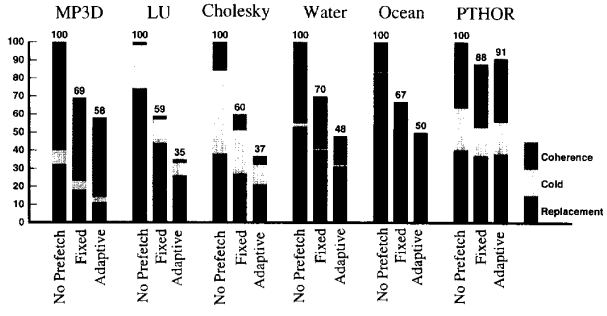


Fig. 13. Number of replacement, cold, and coherence misses for fixed and adaptive prefetch normalized to the case with no prefetching for 16 Kbyte second-level caches.

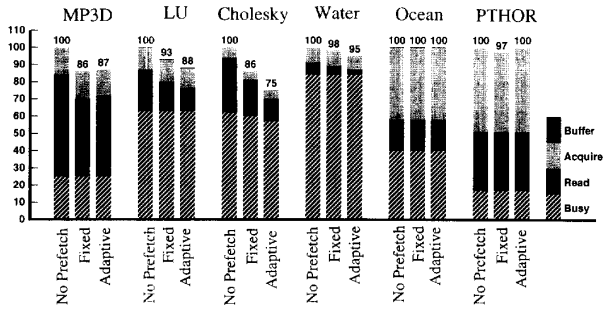


Fig. 12. Total execution time under RC for fixed and adaptive prefetch normalized to the case with no prefetching.

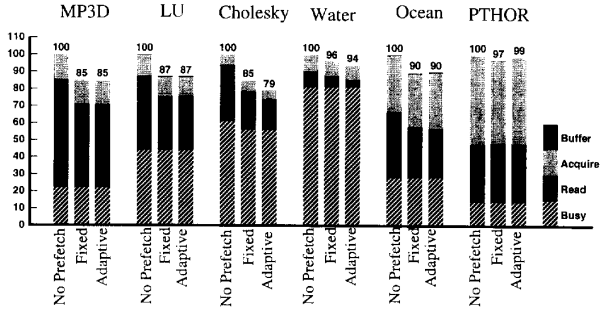


Fig. 14. Total execution time under RC for fixed and adaptive prefetch normalized to the case with no prefetching for 16 Kbyte second-level caches.

global access rate under RC. For example, the execution time reduction for Cholesky is only 25% under RC as compared to 27% under SC

E. Effects of Finite Size *SLC*

One damaging side-effect of prefetching into relatively small caches is cache pollution; a prefetched block that will not be accessed during its lifetime in the cache may replace a block that would be referenced. Since prefetching increases the number of blocks loaded into the cache, it also increases the number of blocks replaced, which might lead to an increased miss rate. On the other hand, if replacement misses have a high spatial locality, these will be covered by prefetching, i.e. the overall replacement miss rate will be lower. The question is whether the first or the latter effect will dominate. Figs. 13 and 14 below report results from simulations with finite first-level caches of 4 Kbyte, and finite second-level caches (*SLC*) of 16 Kbytes.

Fig. 13 shows the total number of read misses for each application decomposed into replacement, cold, and coherence misses from the bottom to the top. It is interesting to see that both fixed and adaptive prefetch show a significant reduction in the number of replacement misses for all applications but PTHOR. For all applications, replacement misses constitute a significant part of the total miss count for the 16 Kbyte *SLC* with no prefetching. As we can see, both prefetching schemes manage to reduce the replacement miss component significantly and cache pollution seems not to be a problem in terms of read misses. Note that the adaptive prefetching scheme would adjust to a reduced prefetching degree in the presence of cache pollution because it would count more useless prefetches.

Although the increased number of replacements does not show up in terms of read misses, since a large fraction of the replacement misses is covered by prefetching, the traffic is increased correspondingly. In order to include this effect, the total execution times for the corresponding executions assuming RC is shown in Fig. 14. By comparing this diagram with the diagram of Fig. 12 (the same execution but with infinite *SLC*), we see that the effectiveness of fixed prefetch seems to be independent of the *SLC* size, except for Ocean, where the prefetch efficiency of replacement misses shows a reduction of the read stall time at 16 Kbyte caches which did not show up for infinite caches. By contrast, the advantage of adaptive prefetch is reduced for smaller caches. One reason is the reduced prefetch efficiency, and the fact that replacement misses dominate, and therefore the effectiveness of prefetching is dictated by their spatial locality. Another effect we have noticed is memory contention; for LU and Cholesky, some parts of the program show high memory contention in the case of small caches due to bursts of messages to the same memory module, which is 3 times slower than the local bus. The same behavior has been observed with slower processors (33MHz instead of 100 MHz) and with other network speeds.

F. Processor Speed and Network Bandwidth Variations

In order to evaluate the sensitivity to the processor speed, we have also run simulations with a 33 MHz processor (recall that all the results above are for 100 MHz processors). The main observation is that the impact of both fixed and adaptive sequential prefetching on the read stall time, read miss count, and traffic is qualitatively unchanged. Therefore, since slower processors means that the processor busy time is increased, the reduction of the read stall time has a smaller impact on the total execution time for slower processors. The opposite effect can be seen if we study a slower memory system or interconnect. Since the latencies are increased, and thus constitute a larger fraction of the overall execution time, the relative gains of adding prefetching will be increased.

In order to evaluate the sensitivity of network contention on our results, we have also run simulations with a contention-free network with a fixed latency time (540 nsec). The main observation is that adaptive sequential prefetching managed to reduce the read stall time even further for applications with a high spatial locality of misses. Since the number of prefetches is dependent on the prefetching efficiency, and affects network traffic, applications with a high spatial locality tend to suffer more from increased contention with the adaptive prefetching scheme.

VI. Related Work

Baer and Chen [2] have proposed a prefetch mechanism that detects stride patterns in the references of the processor. If a stride is found, a hardware mechanism calculates the next potential miss address, and issues a prefetch request to that address. Whereas they originally evaluated the performance of the scheme in a uniprocessor environment, they have also studied its performance in a multiprocessor [7]. Since they use quite different architectural assumptions, a direct comparison of our experimental data with their results is difficult. However, in a follow-up study [10], we have found that sequential prefetching does similarly well in comparison to their stride prefetching scheme. The reason is that stride-one accesses often dominate. While Baer and Chen's scheme successfully detects these strides, sequential prefetching is also successful because the spatial locality is then high. Comparing the hardware cost of the two schemes speaks in favor of sequential prefetching because neither does it have to book keep strides and nor does it require the instruction address for its operation. The extra mechanisms needed are limited to two bits per cache line and three counters per cache.

Hagersten's proposal [18] is similar to Baer and Chen's and is called Right-On-Time (ROT) prefetching. His design includes a hardware stream and stride detector for deciding when to prefetch. A stride-list keeps track of common strides, and a miss-list contains the most recent misses among which new strides are found. In [10], we have compared the performance of this scheme with our adaptive scheme. Again, since stride-one accesses dominate in the applications we have studied and since sequential prefetching is also capable of exploiting spatial locality of nonstride accesses, sequential prefetching often does better with a significantly lower hardware complexity.

Fu and Patel [14] evaluate sequential prefetching and stride prefetching in the context of vector cache memories. They concentrate completely on vectorized numerical programs. For two applications they consider, stride prefetching performs somewhat better than sequential prefetching, but for the other two applications, sequential prefetching does better. They do not consider applications for shared memory multiprocessors. Other hardware-based proposals (such as [20] by Lee et al.) rely on data address lookahead in the processor, but these proposals are limited by the size of software basic blocks.

Mowry and Gupta have evaluated software-based prefetching for shared-memory multiprocessors [22]. Special prefetch instructions are inserted in the code by the programmer or by the compiler. Since they have evaluated their scheme using some benchmarks we have used, one can compare the merits of their scheme and ours by comparing the reduction of the read stall time (which does not include the additional overhead due to the execution of inserted prefetch instructions). With a block size of 16 bytes our adaptive scheme reduces the read stall time to 72% for MP3D. They reduced it to 65% with their pf2 implementation and to 44% with their pf4 implementation (heavily optimized by hand). For LU, our adaptive scheme reduced the read stall time to as low as 47%, while their most aggressively hand-inserted prefetches reduced it to 57% only. For PTHOR, which was the application that showed the worst performance, the adaptive scheme only managed to reduce the read stall time to 95%. By extensive rewriting of source code (reorganizing the element records and grouping entries together on the basis of whether they were likely to be modified) they reduced it to 86%.

Mowry and Gupta have also studied the benefits of read-exclusive prefetching [22] meaning that instead of prefetching a block in read-only mode, an exclusive copy is brought into cache in anticipation of a write access from the local processor. Under sequential consistency, read-exclusive prefetching has the additional advantage of reducing the write stall-time and indeed, Mowry and Gupta have shown that it is very effective. In another study [9], we have combined our adaptive prefetching mechanism with a previously proposed hardware-based mechanism to detect migratory sharing [29] resulting in a sequential prefetching scheme that brings exclusive copies of the block based on a dynamic prediction done by hardware. This hardware-based read-exclusive prefetching scheme results in drastic reductions of read as well as write stall-times yielding an execution-time reduction of up to 50% [9]. In summary, we feel that our hardware-based scheme is competitive with software-based schemes and does not add any burden on the programmer or the compiler.

VII. Conclusions

One of the contributions of this paper is the detailed performance analysis of a class of hardware-based prefetching schemes for shared-memory multiprocessors, called sequential prefetching and a simple adaptive version of this scheme. When a miss occurs in the cache, the subsequent K blocks are prefetched, where K is the degree of prefetching. In this paper, we have only considered sequential prefetching on read misses.

We have analyzed the behavior of fixed sequential prefetching with $K = 1$. When a read miss occurs in the cache the following block is always prefetched if it is not present in the cache. This simple prefetching technique is shown to cut the number of cold and true sharing misses by about the same amount as in a system with twice the block size, without increasing the number of false sharing read misses. Moreover, fixed prefetching normally results in less read traffic than in a system with twice the block size and in about the same traffic as in the system with no prefetch with the same block size. This indicates that fixed prefetch of one block is more effective than using a block of twice the size. Overall this simple prefetching scheme is shown to reduce the number of read misses by between 25% and 45% for four or five out of six applications (depending on the cache size), which resulted in a reduction of the read stall time by between 20% and 35% for a block size of 32 bytes.

Sequential prefetching takes advantage of spatial locality to reduce the read stall time. Spatial locality, however, varies among different programs and also during different parts of the execution of a given program. This calls for a technique where the degree of prefetching is high at those moments when it is useful, and is small or even zero at those moments when it does not pay off. Our proposed adaptive prefetching—another contribution of this paper—tunes the degree of prefetching based on a dynamic measure of prefetching effectiveness. Our simulations show a considerable reduction of the number of read misses and of the read stall time for those applications where the spatial locality of read misses is high, and show little added traffic for those applications where the spatial locality of misses is small. The number of read misses and thus the read stall time was significantly reduced for four or five applications out of six (depending on cache size). In one case the number of read misses was cut by 78%. Moreover, adaptive prefetching reduced the read stall time and the total execution time under release consistency by up to 58% and 25%, respectively.

We have also in detail evaluated the mechanisms needed to support adaptive sequential prefetching. The algorithm that we have experimentally evaluated needs three 4-bit counters per cache and two bits per cache line. Another critical component of the algorithm is to select appropriate thresholds for increasing/decreasing the degree of prefetching. If the available memory system bandwidth is high, one can choose lower thresholds as this will promote more prefetching, whereas the opposite holds for higher thresholds. Overall, considering the substantial performance improvement provided by adaptive sequential prefetching, the extra hardware complexity seems justified.

Acknowledgments

We are indebted to our colleagues Mats Brorsson, Håkan Grahn, and Jonas Skeppstedt of Lund University and to the anonymous reviewers for helpful comments on earlier drafts of this paper. This research has been supported in part by the Swedish National Board for Industrial and Technical Development under contract number 9001797 and by the National Science Foundation under Grant No. CCR-9115725.

References

[1] S.V. Adve and M.D. Hill, "Weak ordering—A new definition," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 2-14, 1990.

[2] J.-L. Baer and T.-F. Chen, "An effective on-chip preloading scheme to reduce data access penalty," *Proc. Supercomputing'91*, pp. 176-186, Nov. 1991.

[3] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, "Portable programs for parallel processors." Holt, Rinehart, and Winston Inc. 1987.

[4] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, "The CacheMire test bench—A flexible and effective approach for simulation of multiprocessors," *Proc. 26th Ann. Simulation Symp.*, pp. 41-49, 1993.

[5] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *Proc. ASPLOS-IV*, pp. 40-51, Apr. 1991.

[6] L.M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. Computers*, vol. 27, no. 12, pp. 1,112-1,118, Dec. 1978.

[7] T.-F. Chen and J.-L. Baer, "A performance study of software and hardware data prefetching schemes," *Proc. 21st Ann. Int'l Symp. Computer Architecture*, pp. 223-232, 1994.

[8] F. Dahlgren, M. Dubois, and P. Stenström, "Fixed and adaptive sequential prefetching in shared-memory multiprocessors," *Proc. 22nd Int'l Conf. Parallel Processing*, vol. I, pp. 56-63, Aug. 1993.

[9] F. Dahlgren, M. Dubois, and P. Stenström, "Combined performance gains of simple cache protocol rxtensions," *Proc. 22nd Ann. Int'l Symp. Computer Architecture*, pp. 187-197, 1994.

[10] F. Dahlgren and P. Stenström, "Effectiveness of hardware-based stride and sequential prefetching in shared-memory multiprocessors." *Proc. First Int'l Conf. High-Performance Computer Architecture*, pp. 68-77, Jan. 1995.

[11] M. Dubois and C. Scheurich, "Memory access dependencies in shared memory multiprocessors," *IEEE Trans. Software Engineering*, vol. 16, no. 6, pp. 660-674, June 1990.

[12] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström, "The detection and elimination of useless misses in multiprocessors," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp. 88-97, May 1993.

[13] S.J. Eggers and R.H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," *Proc. ASPLOS-III*, pp. 257-270, 1989.

[14] J. Fu and J.H. Patel, "Data prefetching in multiprocessor vector cache memories." *Proc. 18th Ann. Int'l Symp. Computer Architecture*, pp. 54-63, May 1991.

[15] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J.L. Hennessey, "Memory consistency and event ordering in scalable shared-memory multiprocessors," *Proc. 17th Ann. Int'l Symp. Computer Architecture*, pp. 15-26, May 1990.

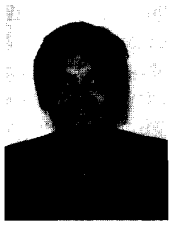
[16] K. Gharachorloo, A. Gupta, and J. Hennessy, "Performance evaluation of memory consistency models for shared-memory multiprocessors," *Proc. ASPLOS IV*, Apr. 1991.

[17] A. Gupta and W.-D. Weber, "Cache invalidation patterns in shared-memory multiprocessors," *IEEE Trans. Computers*, vol. 41, no. 7, pp. 794-810, July 1992.

[18] E. Hagersten, "Towards scalable cache only memory architectures." PhD thesis, Swedish Inst. of Computer Science, Oct. 1992 (SICS Dissertation Series 08).

[19] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Computers*, vol. 28, no. 9, pp. 690-691, Sept. 1979.
[20] R. Lee, P.-C. Yew, and D. Lawrie, "Data prefetching in shared-memory multiprocessors," *Proc. 1987 Int'l Conf. Parallel Processing*, pp. 28-31, Aug. 1987.
[21] D.E. Lenoski, J.P. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J.L. Hennessy, M. Horowitz, and M.S. Lam, "The Stanford DASH multiprocessor," *Computer*, pp.63-79, Mar. 1992.
[22] T. Mowry and A. Gupta, "Tolerating latency through software-controlled prefetching in scalable shared-memory multiprocessors," *J. Parallel and Distributed Computing*, vol. 2, no. 4, pp. 87-106, 1991.
[23] T. Mowry, M. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *Proc. ASPLOS V*, pp. 62-73, Oct. 1992.
[24] J.P. Singh, W.-D. Weber, and A. Gupta, "SPLASH: Stanford parallel applications for shared-memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5-44, Mar. 1992.
[25] A.J. Smith, "Sequential program prefetching in memory hierarchies," *Computer*, vol. 11, no. 12, pp. 7-21, Dec. 1978.
[26] A.J. Smith, "Cache memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, Sept. 1982.
[27] P. Stenström, "A survey of cache coherence scheme for multiprocessors," *Computer*, vol. 23, no. 6, pp. 12-24, June 1990.
[28] P. Stenström, F. Dahlgren, and L. Lundberg "A lockup-free multiprocessor cache design," *Proc. 1991 Int'l Conf. Parallel Processing*, vol. I, pp. 246-250, 1991.
[29] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing," *Proc. 20th Ann. Int'l Symp. Computer Architecture*, pp. 109-118, 1993.

Fredrik Dahlgren received his MS degree in computer science and engineering in 1990 and his PhD degree in computer engineering in 1994, both from Lund University. He is currently a research professor at the Department of Computer Engineering, Lund University. His areas of research include computer architecture, memory systems for shared-memory multiprocessors, and simulation techniques. He has published more than 15 papers on these subjects. He is a member of the IEEE Computer Society.



Michel Dubois holds a PhD from Purdue University, and MS from the University of Minnesota, and an engineering degree from the Faculté Poyltechnique de Mons in Belgium.

Dr. Dubois is an associate professor in the Department of Electrical Engineering at the University of Southern California. Before joining USC he was a research engineer at the Central Research Laboratory of Thomson-CSF in Orsay, France. His main research interests are computer architecture and parallel processing, with a focus on multiprocessor architecture, performance, and algorithms. He is a member of the ACM and a senior member of the IEEE.

Per Stenström is an associate professor of computer engineering at Lund University, where he has conducted research in parallel processing since 1984. He received an MS degree in electrical engineering in 1981 and a PhD degree in computer engineering in 1990, both from Lund University. His primary research interests are in parallel architectures, performance evaluation, and compiler optimization techniques for high-speed computer systems. He has authored or co-authored more than 30 papers in these areas. He is also an author of two textbooks on computer organization and architecture.

Dr. Stenstrom has been a visiting scientist at Carnegie Mellon University (1987), Stanford University (1991), and University of Southern California (1993), where he has studied various aspects of shared-memory multiprocessor architectures. He is on the editorial board of the *Journal of Parallel and Distributed Computing* and is a member of the IEEE.