Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors*

Fredrik Dahlgren and Per Stenström

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 LUND, Sweden

Abstract

We study the relative efficiency of previously proposed stride and sequential prefetching—two promising hardware-based prefetching schemes to reduce read-miss penalties in shared-memory multiprocessors. Although stride accesses dominate in four out of six of the applications we study, we find that sequential prefetching does better than stride prefetching for three applications. This is because (i) most strides are shorter than the block size (we assume 32 byte blocks), which means that sequential prefetching is as effective for stride accesses, and (ii) sequential prefetching also exploits the locality of read misses for non-stride accesses. However, we find that since stride prefetching causes fewer useless prefetches, it consumes less memory-system bandwidth.

1. Introduction

Shared-memory multiprocessors with a general interconnection network often suffer from a significant processor stall time, mostly because of the latencies associated with read and write accesses in large-scale shared-memory multiprocessors. While the latency of write accesses can easily be hidden by appropriate write buffers and relaxed memory consistency models [11], most processors have to stall on read accesses until the requested data has been provided by the memory system. Private caches in conjunction with hardware-based cache coherence maintenance [18] are quite effective at reducing the average read stall time. However, due to cache block coherence maintenance and replacements, other techniques are needed to reduce the number of read misses.

Prefetching is a promising approach to reduce the number of read misses, and thus to reduce the read penalty. Non-binding prefetching [12] does this by bringing into the cache the blocks which will be referenced in the future and are not already present in cache. The value returned by the prefetch is not bound; the prefetched block is still subject to invalidations and updates by the cache coherence mechanism. Non-binding prefetching approaches proposed in the literature can be either software- or hardware-based. Software-controlled prefetching schemes [14] rely on the user/compiler to insert prefetch instructions prior to a reference triggering a miss. By contrast, hardware-controlled prefetching schemes utilize the regularity of data accesses in applications, and need no software support to decide what and when to prefetch.

Two promising hardware-based prefetching strategies in shared-memory multiprocessors are sequential [6, 17] and stride prefetching [5, 10, 13, 16]. Sequential prefetching tries to exploit spatial locality across block boundaries by prefetching consecutive blocks in anticipation of future misses. By contrast, stride prefetching detects and prefetches blocks associated with strides only but does not require spatial locality to be effective. Clearly, the relative performance between the two approaches depends on the amount of spatial locality and stride accesses in an application.

Another notable difference between the two approaches is the amount of hardware support needed. Whereas sequential prefetching in its simplest form only requires a counter associated with each cache [6], previously published stride prefetching schemes [1] require fairly complex detection mechanisms and also modifications to the processor.

This paper evaluates the relative performance as well as implementation implications of stride and sequential prefetching schemes in a unified framework consisting of a cache-coherent NUMA architecture which is discussed in detail in Section 2. In Section 3 we describe the prefetching schemes we evaluate. We then evaluate the relative performance of the prefetching schemes in Sections 4 and 5 using detailed architectural simulators and a set of six scientific applications, in which stride accesses dominate in four of them. Surprisingly, we find that one of the simplest variations of sequential prefetching does better or same as the most aggressive stride prefetching scheme in five out of the six applications. There are two intuitive reasons for this. First, when strides are shorter than the block size, sequential prefetching that brings consecutive blocks will be effective. Second, unlike stride prefetching schemes, sequential prefetching can also attack misses caused by non-stride accesses that exhibit a high spatial locality. This type of misses is fairly dominant

*This work was supported by the Swedish National Board for Technical Development (NUTEK) under the contract P855.

0-8186-6445-2/95 $04.00 © 1995 IEEE

in the applications we study. An observed shortcoming of sequential prefetching, however, is that its prefetch efficiency in general is lower which speaks in favor for stride prefetching schemes if the memory-system bandwidth is limited. We end this paper by relating our work to that of others in Section 6 before we conclude in Section 7.

2. Architectural Framework

The architectural framework we assume is a cache-coherent NUMA architecture in which memory is distributed across the processing nodes and where each node is organized according to Figure 1.

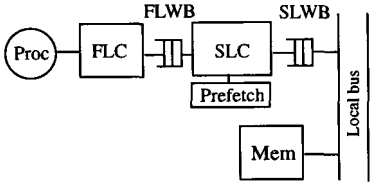

Figure 1: Processing node organization.

The key feature of this particular design of the processing node is its lockup-free [19] second-level cache *(SLC)* which supports non-binding prefetching schemes as well as relaxed memory consistency models [11]. Moreover, we assume a blocking-load processor which is interfaced to a simple, and thus fast, on-chip, first-level data cache *(FLC)*.

The *FLC* is a write-through, direct-mapped data cache with no allocation of blocks on write misses, is blocking on read misses, and has a block-invalidation pin from outside the chip. Write requests, synchronization requests and read-miss requests issued by the *FLC* are buffered in FIFO order in the first-level write-buffer (*FLWB*). The *SLC*, which is larger than the *FLC*, is a write-back cache. Because the *FLC* is direct-mapped and write-through, there is full inclusion between the *FLC* and *SLC*, which makes it possible to migrate all mechanisms for maintaining cache coherence to the *SLC*. Besides implementing a write-invalidate protocol, which is presented in Section 4, the *SLC* is made lockup-free by the second-level write-buffer *(SLWB)* which buffers all pending requests such as prefetch, read miss, and invalidation requests.

Throughout this study, we evaluate prefetching into the *SLC* only. Thus, all prefetching mechanisms require modifications to the *SLC*. As a result, the prefetch mechanisms only observe block references, and subsequent read requests to the same block will hit in the *FLC* and will thus not be visible to the prefetching mechanism. In order not to cause any page fault in the virtual memory system because of a useless prefetch, prefetching across page-boundaries is not allowed. The next section explains how this baseline architecture is extended with various stride and sequential prefetching mechanisms.

3. Simulated Prefetching Algorithms

We now describe the prefetching schemes, and how they are incorporated in the baseline system. We introduce in Section 3.1 a terminology for stride prefetching and continue in Sections 3.2 and 3.3 with descriptions of the evaluated stride prefetching schemes. Then in Section 3.4 we describe the sequential prefetching scheme we evaluate.

3.1 Stride Prefetching Terminology

Stride prefetching aims at detecting sequences of data accesses whose addresses are equidistant with a certain stride. To illustrate some key concepts, let us consider the matrix multiplication program of Figure 2 for which matrices A and B are allocated row-wise in memory. In the inner loop, the read accesses from A form a stride of one vector element. By contrast, the read accesses from B has a stride equal to N; the size of a row. We will refer to a sequence of read accesses with a constant stride as a *stride sequence*.

```
for (i=0; i<L; i++)
  for (j=0; j<M; j++)
    for (k=0; k<N; k++)
      C[i,j] = C[i,j] + A[i,k] * B[k,j];
```

Figure 2. Example matrix multiplication algorithm.

Two issues must be addressed in order for a stride-prefetching scheme to be effective. First, the stride in a stride sequence must be dynamically identified which is done in the *detection phase*. Second, when a stride sequence is detected, prefetch requests must be issued early enough so that the block will be available in the cache when the processor eventually accesses it. This is done in the *prefetching phase*. Note that the second issue is applicable also to other hardware-based prefetching schemes and to other access patterns than stride sequences.

The performance improvement provided by stride-prefetching schemes is dictated by the fraction of read misses caused by stride sequences. Moreover, since prefetching can not start until a stride sequence is detected, the length of a stride sequence becomes critical as to how many misses can be removed. In the next section, we specifically study various approaches to detect strides.

3.2 The Stride Detection Phase

Stride detection is complicated because stride sequences are typically interleaved with read accesses that do not belong to a stride sequence. However, if a stride sequence is generated in a loop, (e.g. the vector accesses in Figure 2), all accesses belonging to this sequence typically origi-

nate from the same load instruction. By keeping track of the instruction address associated with each stride access, it is possible to filter out accesses evolving from the same stride sequence and this way detect a stride. This detection approach requires that the instruction address (i.e. the program counter) is available to the detection mechanism; hence, we refer to it as an *I-detection scheme*. To implement this scheme in the baseline architecture requires that *FLC* read-miss requests are accompanied by the instruction address of the corresponding load instructions that caused these misses. Examples of I-detection schemes can be found in [5, 10, 16]. In this study, we only consider prefetching into the *SLC;* only read requests that miss in the *FLC* can trigger prefetching. Moreover, we assume that the instruction address of the load instruction that misses in the *SLC* is matched against the entries in a *Reference Prediction Table* (RPT) which is organized as a cache.

The simplest stride prefetching scheme works as follows. The first time a certain load instruction misses in the *SLC*, the corresponding instruction address, I (used as a tag), and data address, *D1*, are inserted in the RPT, and the state is set to the initial state *no-prefetch*. Subsequently, when a new read miss is encountered with the same instruction address and with a data address *D2*, we will have a hit in the RPT. Potentially, this is the beginning of a stride sequence, and the following actions take place: (i) the stride is calculated as $S = D2 - D1$, (ii) *D2* as well as *S* are inserted in the RPT, and (iii) the state is set to *prefetch*. At this point, we can prefetch *D2+S*, *D2+2*S*, etc. The structure of the RPT is shown in Figure 3. This simple stride scheme succeeds in detecting most strides, but has the drawback of producing useless prefetches in situations where the same load instruction is executed twice and the addresses do not form a stride sequence.

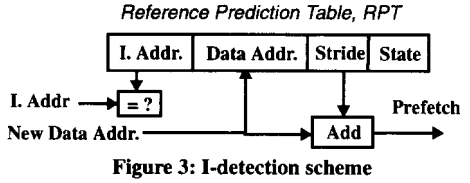

Figure 3: I-detection scheme

The I-detection scheme analyzed in this paper is more sophisticated, and is influenced by the scheme proposed by Baer and Chen in [1], where the sequence of events that leads to the detection of a stride is shown by the state-transition graph in Figure 4. The second time the same instruction address appears, a stride is calculated, the state is set to `init`, and prefetching begins. All read requests presented to the *SLC* are matched against the RPT in order to see whether the processor continues to access a detected stride sequence (`correct`) or whether a new possible stride sequence is initiated by the same load instruction (`incorrect`). When the same load instruction has generated accesses that belong to the same stride sequence three times in a row, the state becomes `steady`. A single `incorrect` (possible change of stride sequence) does not imply a recalculation of the stride; instead, a transition to state `init` occurs. However, a second incorrect prediction in a row leads to the state `transient`, and a new stride is calculated as the difference between the preceding two data addresses from that instruction. One of the most important features of this scheme is the state `no-pref`, which means that prefetching for a load instruction is stopped, would three incorrect predictions happen in a row. This reduces the number of useless prefetches.

We have slightly modified our implementation of the scheme according to Baer and Chen [1] as follows. Instead of using an additional program counter that looks ahead a number of instructions corresponding to the miss latency we want to overlap, our implementation uses a mechanism incorporated in the *SLC* which is explained in Section 2.3. The RPT in our evaluations is organized as a 256-entry, direct-mapped cache; having the same size and organization as the one used by Chen and Baer in [5].

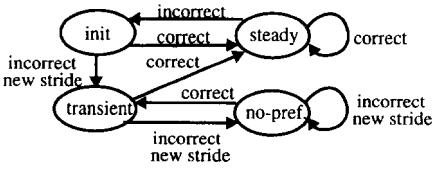

Figure 4: State-transition graph of the I-detection stride prefetching scheme.

A shortcoming of I-detection schemes is that they rely on the availability of the program counter of the load instruction that causes the miss. *D-detection schemes* do not need this. Instead, they compare the data address of the read-miss request with previous previously recorded data addresses in order to detect stride sequences which makes the detection phase much more complicated. We consider the D-detection scheme proposed by Hagersten in [13]. Conceptually, it works as follows. The address of each read miss is matched against recent misses buffered in a *miss list*, then all possible strides are calculated, and a *frequency table* for strides is finally updated. If a stride has appeared a certain number of times, called the *stride-threshold*, it is moved to a *list of common strides*. If a stride that is calculated already belongs to the list of common strides, a stride is potentially detected and inserted in the *stream list*. In our implementation, the miss list, the frequency table, the list of common strides, and the stream list each has 16 entries and uses a LRU replacement policy. A stride threshold of 3 means that four misses belonging to the same stride sequence are required before it is

recorded in the list of common strides, if the stride has not occurred previously. When a stride is recorded in the list of common strides, two additional misses are required to initiate prefetching.

3.3 Stride Prefetching Phase

Orthogonal to the selection of a stride detection scheme is the choice of a stride prefetching scheme which aims at determining how many blocks to prefetch. As soon as a stride sequence has been detected, prefetch requests are issued for predicted addresses. Since the length of a stride sequence is not known, a heuristic is needed to decide how many blocks to prefetch. Of course, if a predicted block already resides in the cache, or if the block is already requested but has not yet arrived, a prefetch request is never issued. To simplify comparisons, we have used the same prefetching algorithm for the I-detection and the D-detection schemes as follows.

The prefetching algorithm is shown in Figure 5. When a stride is detected starting at address *B*, and the stride is calculated to *S*, blocks *B+S*, *B+2*S*,..., *B+d*S* are prefetched, where *d* is referred to as the *degree of prefetching*. These *d* blocks are tagged as prefetched, which requires 1 bit per block in the *SLC*. Moreover, if the *SLC* subsequently encounters a read request by the same instruction to the tagged block *B+S*, and if there is a hit in the RPT, *S* is read from the RPT and the next block in the stride sequence at address *B+d*S+S* is prefetched. Subsequently, the 1-bit tag of the accessed block is reset. Consequently, as long as the processor accesses the same stride sequence, the prefetching mechanism continues to prefetch, and prefetched data will be available when the processor needs it. Ideally, with this algorithm there will be no read misses beyond the ones in the detection phase.[1] The prefetching-phase mechanism proposed by Hagersten in [13] implements a similar prefetching scheme.

3.4 Sequential Prefetching

Sequential prefetching brings into cache blocks *B+1*, *B+2*,...*B+d*, when a miss is caused to block *B*, where *d* is the degree of prefetching. Thus, sequential prefetching is effective if the spatial locality is high and has been shown to perform well for parallel applications on shared-memory multiprocessors by Dahlgren *et al.* in [6, 7, 9], and in multiprocessor vector machines by Fu and Patel in [10]. In general, sequential prefetching requires much less hardware support because no detection mechanism is needed and sequential prefetching can use the same schemes for the prefetching phase as stride prefetching schemes do.

1. This assumes that the stride sequence is within the same page and that a prefetched block is neither invalidated nor replaced.

The prefetching-phase mechanism for the sequential prefetching scheme in this study is in essence the same as for stride prefetching, and works as follows. On a read miss to block *B*, the corresponding read request is issued immediately and blocks *B+1*, *B+2*,..., *B+d* are prefetched if they are not in the cache already. These blocks are tagged as prefetched. Each time a processor hits on a block that is tagged as prefetched, the 1-bit tag is reset, and the block that appears *d* blocks ahead is prefetched. For example, if the processor continues to access consecutive blocks, it will hit on *B+1*, which is tagged, and block *B+1+d* will be prefetched. If *B+2* is accessed block *B+2+d* is prefetched. We note that only one miss will be encountered initially for a whole sequence.

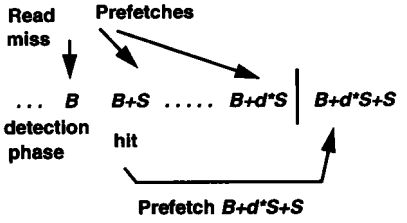

Figure 5: The stride prefetching phase.

In summary, when comparing stride and sequential prefetching schemes we note that stride prefetching schemes are effective for consecutive accesses separated by a constant stride, whereas sequential prefetching is expected to only capture stride sequences for strides smaller than or equal to the block size. Unlike stride prefetching schemes, however, sequential prefetching is effective for non-stride accesses if the spatial locality is high. As for the difference in hardware complexity, sequential prefetching schemes need no detection mechanisms and are therefore less complex. We next study the relative performance of the schemes in this section.

4. Simulation Methodology

We have developed simulation models of the baseline architecture and all the prefetching algorithms presented in the previous section. The simulation platform used is the CacheMire Test Bench [3], a program-driven functional simulator of multiple SPARC processors. It consists of a functional and a memory-system simulator. The SPARC processors in the functional simulator issue memory references and the architectural simulator delays the simulated processors according to its timing model. Consequently, the same interleaving of memory references is maintained as in the target system we model. To reduce the simulation time, we simulate all instructions and private data references as if they always hit in the first-level cache.

We model a system containing 16 processing nodes with an organization according to Figure 1. At the system level, the baseline architecture implements a write-invalidate protocol with a full-map directory similar to Censier and Feautrier's [4]. A *SLC* read miss is sent to the node where the page resides. If the memory copy is clean memory responds directly; otherwise, memory will request a fresh copy from the node keeping the modified copy before it can supply a block copy to the requesting node. Therefore, a read miss may encounter zero, two, or four node-to-node traversals.

The architectural parameters that remain the same in all simulations appear in Table 1 (due to contention, however, the reads might be further delayed). As for the size of the *FLC*, we have chosen 4 Kbytes to get a realistic replacement miss rate for the applications we use. Moreover, although a large block size would be advantageous for the sequential prefetching scheme to be effective for large strides, we pessimistically consider a block size of 32 bytes. The effectiveness of sequential prefetching for larger block sizes (128 bytes) has previously been reported in [6, 7].

The timing assumptions are based on a processor clock (or pclock) rate of 100 MHz. The *FLC* has the same cycle time as the processor with an *FLC* fill time of 3 pclocks. The *SLC* is built from static RAMs with a cycle time of 30 ns. Whereas we assume an infinitely large *SLC* by default, we will also study the effect of a finite *SLC*. The memory in each node is fully interleaved with an access time of 90 ns and a 256-bit wide local split-transaction bus clocked at 33 MHz. The sizes of the *FLWB* and of the *SLWB* are 8 and 16 entries, respectively.

Table 1. Fixed architectural parameters.

Parameter	Value (1 pclock = 10 ns)
Number of processors	16
First-level cache (*FLC*) size	4 Kbytes
Block size (*FLC* and *SLC*)	32 bytes
Read from *FLC*	1 pclock
Read from *SLC*	6 pclocks
Read from local memory	28 pclocks

We assume a single 4-by-4 mesh network synchronously clocked at 100 MHz with wormhole routing and with a flit size of 32 bits. The node fall-through latency is three network cycles. In all experiments, contention is accurately modelled in all parts of the system. Synchronization is based on a queue-based lock mechanism at memory similar to the one implemented in DASH, with a single lock variable per memory block. In addition, pages (4 Kbytes) are allocated across nodes in a round-robin fashion based on the least significant bits of the virtual page number. Finally, we assume release consistency [11].

The selection of benchmarks is critical in any evaluation study, and this one is not an exception. Since the success of stride prefetching is dictated by the occurrence of strides in the applications, we have chosen four applications that have a substantial amount of stride accesses and two where strides are rare. Four of them are taken from the SPLASH suite (MP3D, Water, Cholesky, and PTHOR) [15]. The other two applications (LU and Ocean) are developed at Stanford University. They are all written in C using the ANL macros to express parallelism and are compiled by `gcc` (version 2.1) with optimization `O2`. In all measurements, we gather statistics only in the parallel sections as recommended in the SPLASH report [15].

MP3D was run with 10K particles for 10 time steps. Cholesky used the bcsstk14 matrix. Water was run with 288 molecules for 4 time steps. A 200x200 matrix was the input to LU. Ocean used a 128x128 grid with the tolerance factor set to 10^{-7}. Finally, PTHOR used the RISC-circuit for 1000 time steps.

5. Experimental Results

We start in Section 5.1 with an analysis of the application characteristics in terms of some key metrics in order to provide an intuition of the results. We evaluate the relative effectiveness of stride and sequential prefetching in Section 5.2 and consider key variations such as finite caches and larger data sets in Sections 5.3 and 5.4, respectively.

5.1 Application Characteristics

In order to quantify the potentials of stride and sequential prefetching, we have used the following metrics: (i) the fraction of the original read misses that belongs to stride sequences, (ii) the average length of the stride sequences, and (iii) the strides. In particular, if only a small fraction of all read misses belongs to stride sequences, stride prefetching is not expected to be effective. In addition, if the length of the stride sequence is short, the detection phase will be a significant part of the sequence, which limits the effectiveness of stride prefetching. If the stride is close to 1 block, sequential prefetching will perform well, while for strides larger than 1 block, stride prefetching has a possibility to outperform sequential prefetching provided that the spatial locality of read misses is low.

The analysis is based on an execution on the baseline architecture, and we only consider read requests that miss in the second-level cache. In order to concentrate on cold and coherence misses, we assume an infinite *SLC*. In contrast to Section 5.2, we only consider requests from one processor in this section, which has been shown to be representative. To detect the start of a stride sequence, we use I-detection and also require that at least three equidistant

accesses are caused by the same load instruction. The results of this experiment are shown in Table 2.

In MP3D, we see that only 9.2% of all misses belong to stride sequences. This means that stride prefetching based on I-detection can remove at most 9.2% of the misses. Moreover, since some misses are required for the detection phase before prefetches are issued, the actual miss reduction will be lower. Since the average length of a stride sequence is only 5.2 block references, and at least two references are required to detect a stride, we cannot expect to gain more than a 5-6% reduction of read misses from I-detection stride prefetching. We also see that the most common stride is 1 block (76% of all stride accesses belong stride 1 sequences), which means that most of the stride-access misses are also covered by sequential prefetching.

Table 2: Application characteristics. Infinitely large second-level cache.

	MP3D	Chol.	Water	LU	Ocean	PTHOR
Read misses within stride sequences	9.2%	80%	79%	93%	66%	4.1%
Avg. length of sequence	5.2	7.2	8.0	16.9	7.6	3.4
Dominant stride (blocks)	1 (76%)	1 (95%)	21 (99%)	1 (93%)	65(42%), 1(31%)	1 (37%)

In Cholesky, Water, and LU, almost all misses belong to stride sequences. They are in general longer than in MP3D, on average between 7.2 and 16.9 references, which means that the detection overhead is expected to be smaller. Stride prefetching thus has a potential to perform well for these applications. In Cholesky and LU, almost all strides are 1 block, and sequential prefetching is expected to do equally well. In Water, on the other hand, most strides are substantially larger than one, and sequential prefetching is limited to the spatial locality of references belonging to different stride sequences and non-stride accesses.

In Ocean, most misses belong to stride sequences (66%), and the length of the sequences are 7.6 on average, indicating that stride prefetching is expected to do reasonably well. The most common stride is as long as 65. Therefore, if there is low spatial locality in non-stride accesses, stride prefetching is expected to be more effective than sequential prefetching at reducing the number of read misses for Ocean. Finally in PTHOR, there are almost no stride sequences, and those that exist are very short. In addition, other experimental observations have shown that the spatial locality of misses is low in PTHOR [6]. Therefore, neither stride nor sequential prefetching are expected to work well for PTHOR.

In this section, we have used some key parameters in order to build an intuition of the relative potentials of stride and sequential prefetching. We will now evaluate the relative effectiveness of stride and sequential prefetching using this data to explain the results.

5.2 Stride vs. Sequential Prefetching

In order to build intuition in how the read stall time as well as the amount of network traffic are affected by each type of prefetching scheme, we start by showing the effects they have on the number of read misses and the prefetch efficiency.

The top diagram of Figure 6 shows the number of read misses for each prefetching scheme relative to the baseline architecture. For each application, there are three bars: (from left to right) the I-detection scheme, denoted I-det; the D-detection scheme, denoted D-det; and the sequential prefetching scheme, denoted Seq. For all schemes we assume that the degree of prefetching is d=1.

Concentrating on MP3D, we can see that I-detection and D-detection reduce the number of read misses by only 5%, which is consistent with the results from Section 5.1. Sequential prefetching, on the other hand, reduces the number of misses by 28%. The reason why sequential prefetching is more effective is that it covers most of the stride sequences (recall that 76% of all strides are 1 block) and that it exploits spatial locality (accesses to the `Particles` data structure in MP3D show a fairly high spatial locality). This shows that the spatial locality is high enough for sequential prefetching to outperform stride prefetching. For Cholesky, Water, and LU, all three prefetching techniques perform well. Still, sequential prefetching shows fewer read misses than stride prefetching, while I-detection is more effective than is D-detection. While these results can be expected for Cholesky and LU from the observations in Section 5.1, the reason why sequential prefetching works well for Water is the high spatial locality of accesses belonging to different stride sequences. For Ocean, on the other hand, stride prefetching is more effective than sequential prefetching which is to be expected from the observations in Section 5.1 that large strides dominate. Apparently, the spatial locality of accesses belonging to different strides is not sufficiently high either. For PTHOR, all three techniques perform poorly, although sequential prefetching manages to reduce the number of misses to some extent.

Overall, for all applications but Ocean, sequential prefetching is more effective than stride prefetching at reducing the number of misses, while I-detection is in general more effective than D-detection.

The middle diagram of Figure 6 shows the prefetch efficiency. For Cholesky, Water, and LU, i.e. the applications for which all three schemes are effective at reducing the number of misses, all schemes also have a very high prefetch efficiency. For the other three applications, MP3D, Ocean, and PTHOR, I-detection in general has a higher prefetch efficiency than D-detection and sequential prefetching. The reason why I-detection has a reasonably high prefetch efficiency for all applications is because it is more selective in the detection phase; a stride is detected if it is caused by the same load instruction meaning that the probability is high that a detected stride sequence *is* a stride sequence. The sequential prefetching scheme, on the other hand, always prefetches on a cache miss, regardless of whether the spatial locality is high or low. As a result, the sequential prefetching scheme issues a larger number of useless prefetches for the applications where the reduction of read misses is not as high, i.e., in Ocean and PTHOR. These useless prefetches increase the traffic, and may cause contention. Another potential drawback of useless prefetches is that the replication can increase invalidation traffic. However, this issue was studied for sequential prefetching in [6, 7], where such effects were found to be negligible.

We now interpret how the read stall time is reduced based on the effects on read misses and prefetch efficiency. The read stall times are shown in the bottom diagram of Figure 6. The reduction of the read stall times follows to some extent the reduction of read misses, but the reduction of the read stall times is in general not as dramatic. The effect on the read stall times is twofold: (i) the number of misses is reduced, and (ii) the amount of traffic sent to the network during an execution is increased because of useless prefetches causing contention. Overall, sequential prefetching is more effective at reducing the read stall time for three out of six applications despite the lower prefetch efficiency in some cases.

Overall, the above results show that sequential prefetching is more effective at reducing the number of read misses, while the I-detection scheme in general has a higher prefetch efficiency because it is more selective in the detection phase. However, sequential prefetching reduces the read stall time more than the other schemes do for three out of six applications, despite its much simpler and less sophisticated hardware mechanism. We have also shown that the key application parameters presented in Section 5.1 are useful to explain the results. In the next two sections, we will use these parameters in order to study the effects of finite caches as well as larger data sets.

5.3 Effects of Finite Sized Second-Level Caches

One aspect that is not covered in above sections that might have a significant impact on the results are finite second-level caches. Table 3 shows the application characteristics for a 16k byte, direct-mapped *SLC*. The methodology used to capture the numbers is the same as the one we used in Section 5.1. The first row shows the number of replacement misses relative to the total number of read misses, while the rest of the table has the same layout as Table 2. The most remarkable differences as compared to Table 2 (infinite *SLC*) are in MP3D and Ocean. In MP3D, more than 90% of all replacement misses belong to stride sequences with stride 1. In Ocean, 84% of all replacement misses belong to stride sequences, and 92% of these belong to sequences with stride 1. Now, stride prefetching is expected to work reasonably well for MP3D and Ocean too. Because most strides are 1 block, however, sequential prefetching will cover these misses as well. This explains the results on stride prefetching for MP3D presented by Chen and Baer in [5] as well as the results on sequential prefetching with finite *SLC*s presented by Dahlgren *et al.* in [6, 7].

Table 3: Application characteristics for a finite 16 kbyte direct-mapped SLC.

	MP3D	Chol.	Water	LU	Ocean	PTHOR
Percentage repl. misses	32%	45%	45%	76%	82%	39%
Read misses within stride sequences	34%	73%	67%	91%	81%	4.8%
Avg. length of sequence	7.0	8.7	8.8	13.2	6.2	3.6
Dominant stride (blocks)	1 (96%)	1 (97%)	21 (98%)	1 (91%)	1(87%), 65(9%)	1 (25%)

5.4 Effects of Larger Data Sets

We have also studied how our results are affected by larger data sets for five of the six applications, and studied how the three key application parameters varied with increased data sets for infinitely *SLC*s. The results are shown in Table 4 in terms of expected tendencies. All trends correspond to infinite second-level caches. The reason why PTHOR is not analyzed with a larger data set is because of time limitations for simulations. However, we do not expect that the results for PTHOR will change dramatically.

Using more particles in MP3D does not increase the fraction of read misses within stride sequences, and the average length of the stride sequences will be slightly increased but is limited. Thus, the effectiveness of I-detect is not expected to increase significantly as compared to the results in Section 5.2, and we still expect sequential

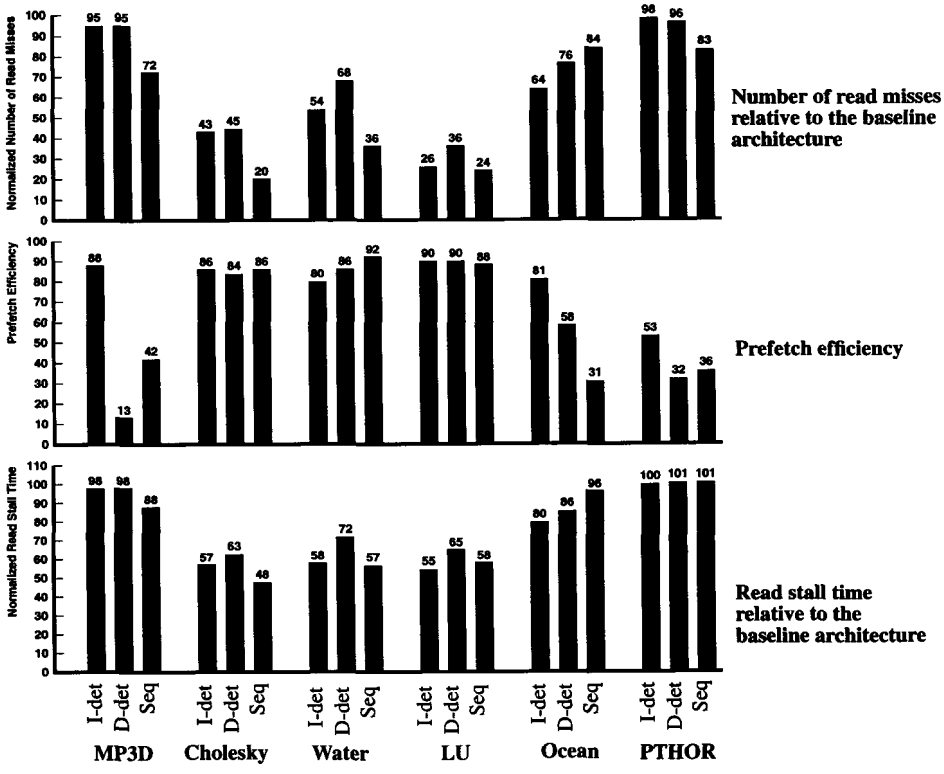

Figure 6: (Top) number of read misses relative to the baseline architecture, (middle) prefetch efficiency, and (bottom) read stall time relative to the baseline architecture.

prefetching to be more effective. For Cholesky, Water, and LU, the fraction of read misses within stride sequences will be closer to 100% and the sequences will be longer as we increase the data sets. Therefore, we expect stride prefetching to be more effective as we increase the data sets. In addition, since the sequence length will be increased, the detection overhead will diminish. On the other hand, sequential prefetching will continue to perform as well, since almost all strides are 1 block in Cholesky and LU, and since the spatial locality of accesses belonging to different strides is still very high in Water. In Ocean, stride prefetching is expected to perform better for larger data sets. For infinitely large *SLCs*, the spatial locality will not be increased, and sequential prefetching is expected to perform poorly. For finite caches, the effectiveness of sequential prefetching depends on the number of replacement misses according to the results in Section 5.3. Overall, for these five applications, the relative effectiveness of stride and sequential prefetching shown in above sections is not expected to be dramatically changed with larger and more realistic data sets.

Table 4: Expected application characteristics for an infinite direct-mapped *SLC* and larger data sets.

	MP3D	Chol.	Water	LU	Ocean
Read misses within stride sequence	about the same	higher	higher	higher	higher
Avg. length of sequence	limited	longer	longer	longer	longer
Dominant stride (blocks)	1	1	21	1	longer

6. Discussion and Related Work

In [9], we comparatively evaluated various I-detection stride prefetching schemes. We found that the relative difference in terms of their miss-reducing capabilities is small, while there can be a substantial difference in the prefetch efficiency. Since the stride-detection phase of many applications can be long enough for the stride detection overhead to be negligible, the detection phase can be optimized to keep the number of useless prefetches at a low level. The I-detection scheme described and evaluated in this paper was the one that performed the best. In [9], we also evaluated various degrees of prefetching, d. For the prefetching scheme evaluated in this paper, however, there was little difference between different values of d, which is why we in this paper only present results for d=1.

One of the advantages of I-detection schemes over D-detection and sequential prefetching schemes as implemented in this study, is the fact that I-detection schemes do not issue as many useless prefetches. This is partly because using instruction addresses seems to be an efficient way to detect stride sequences, and partly because of the `no-pref` state of the state-transition graph; for instructions where prefetching has turned out to be useless, no prefetches are issued. In [6], Dahlgren *et al.* propose an adaptive sequential prefetching mechanism that dynamically changes the degree of prefetching based on a heuristic measure of spatial locality. In particular, the degree of prefetching can reach zero, which means that no prefetch requests are issued. This means that for phases of execution where the spatial locality is low, no or few prefetches are issued, and thus the traffic can be kept at a low level. A similar mechanism is proposed for the prefetching phase of the D-detection scheme proposed by Hagersten in [13]. The reason why we did not consider such mechanisms in this study is because our focus is to compare the effectiveness of stride detection mechanisms with sequential prefetching. Hence, we have used the same prefetching-phase mechanism for all schemes. The need for a more sophisticated prefetching-phase mechanism is higher for sequential prefetching and D-detection, since they are less selective in the detection phase, and such mechanisms will be considered in future work.

The stride prefetching scheme of Baer and Chen, proposed in [1] and used in [5], also uses an RPT for detection of stride sequences along with the same control algorithm. However, their mechanism differs in that it uses a lookahead-PC, which is a predicted program counter that ideally is as far ahead of the regular PC so that the maximum latency of a read request can be hidden. While we prefetch blocks for a read request at a previous reference from the same load instruction, they prefetch a block when the load instruction that accesses that block appears at the address of the lookahead-PC. Thus, their scheme might result in fewer useless misses, but requires more sophisticated hardware mechanisms on the processor chip. The potential of their scheme is still limited by the same application parameters as the ones we have identified in this paper, and we feel that the performance difference between the two is small. In particular, if the stride sequences are long, and the number of misses to detect a stride becomes insignificant, the effectiveness of the I-detection scheme evaluated in this paper and the scheme by Baer and Chen will be nearly identical.

The prefetching-phase mechanism of Hagersten's prefetching scheme in [13] is different from the one used in this study. When his scheme detects a stride, it prefetches the next block, and when that block is requested, the next block is prefetched. So far it is similar to the prefetching scheme in this study. However, if the prefetched block is accessed before it has arrived to the cache, the number of blocks that are prefetched is increased. In this way, the intention is to adjust the lookahead distance to the latency of a prefetch request. In this study, we have seen no need for such a mechanism, since the time the processor has to stall due to outstanding prefetches is on average very short. In addition, by using the same prefetching phases for I-detect, D-detect, and sequential prefetching, we have been able to compare the fundamental characteristics of each scheme using the same assumptions.

In [2], Bianchini and LeBlanc propose a stride prefetching technique with software support called *hybrid prefetching*. The compiler or programmer provides the hardware with stride information, which means that the detection phase does not have to be implemented in hardware. Thus, their proposed scheme is less expensive in terms of hardware support than the stride prefetching schemes considered in this study, but imposes constraints for the programmer or for the compiler.

7. Conclusions

We have evaluated the relative effectiveness of hardware-based stride and sequential prefetching for shared-memory multiprocessors. The major assumptions of the work are a write-invalidate protocol and prefetching into the second-level cache only, and we assume in principal no additions to the processor chip. By detailed simulations and a set of six scientific parallel applications, we have studied the effects on the miss rate, the prefetch efficiency, and the read stall time. In order to understand the results, we have identified some key application parameters that are useful to predict the relative performance of stride and sequential prefetching schemes. These parameters include the fraction of read misses within stride sequences and the length of stride sequences and the strides themselves.

Our detailed architectural simulations show that sequential prefetching does better or at least equally well as stride prefetching, despite its much simpler hardware mechanisms. This is because (i) most strides are shorter than a single memory block which means that sequential prefetching is as effective for most stride sequences, and (ii) sequential prefetching also exploits the spatial locality of read misses for non-stride accesses. Because of the lower fraction of useless prefetches, stride prefetching can perform better than sequential prefetching if the memory-system bandwidth is not sufficient. However, in situations where the memory bandwidth is not a limitation, it appears that sequential prefetching is more cost-effective because of the much simpler hardware mechanisms needed.

References

[1] Baer, J.-L. and Chen, T.-F. "An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty," in *Proceedings of Supercomputing '91*, pp.176-186, November 1991.

[2] Bianchini, R. and LeBlanc, T.J. "*A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors*," Technical Report 515, Computer Science Department, University of Rochester, May 1994.

[3] Brorsson, M., Dahlgren, F., Nilsson, H., and Stenström, P. "The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors," in *Proceedings of the 26th Annual Simulation Symposium*, pp. 41-49, 1993.

[4] Censier, L.M., and Feautrier, P. "A New Solution to Coherence Problems in Multicache Systems," in *IEEE Transactions on Computers*, 27(12), pp. 1112-1118, December 1978.

[5] Chen, T.-F. and Baer, J.-L. "A Performance Study of Software and Hardware Data Prefetching Schemes," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp.223-232, April 1994.

[6] Dahlgren, F., Dubois, M., and Stenström, P. "Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors," in *Proceedings of the 1993 International Conference on Parallel Processing*, Vol. I, pp. 56-63, 1993.

[7] Dahlgren, F., Dubois, M., and Stenström, P. "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," in *IEEE Trans. on Parallel and Distributed Systems*, to appear.

[8] Dahlgren, F., Dubois, M., and Stenström, P. "Combined Performance Gains of Simple Cache Protocol Extensions," in *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pp.187-197, April 1994.

[9] Dahlgren, F. and Stenström, P. "*Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors—A Comparative Evaluation*," Technical Report No. DT-191, Dept of Computer Engineering, Lund University, November 1994.

[10] Fu, J. and Patel, J.H."Data Prefetching in Multiprocessor Vector Cache Memories," in *Proc. of the 18th Annual Int. Symposium on Computer Architecture*, pp.54-63, May 1991.

[11] Gharachorloo K., Gupta A., Hennessy J. "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," in *Proc. of ASPLOS IV*, pp. 245-257, 1991.

[12] Gupta, A. et al. "Comparative Evaluation of Latency Reducing and Tolerating Techniques," in *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp.254-263, May 1991.

[13] Hagersten, E. "*Towards Scalable Cache Only Memory Architectures*," PhD thesis, Swedish Institute of Computer Science, Oct. 1992 (SICS Dissertation Series 08)

[14] Mowry, T. and Gupta, A. "Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors," in *Journal of Parallel and Distributed Computing*, Vol. 12, No 2., pp. 87-106, 1991.

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

[16] Sklenar, I. "Prefetch Unit for Vector Operations on Scalar Computers," in *Computer Architecture News*, 20(4):31-37, September 1992.

[17] Smith, A.J. "Sequential Program Prefetching in Memory Hierarchies," in *IEEE Computer*, Vol. 11, No. 12, pp.7-21, Dec. 1978.

[18] Stenström, P. "A Survey of Cache Coherence Scheme for Multiprocessors," in *IEEE Computer*, Vol. 23, No. 6, pp. 12-24, June 1990.

[19] Stenström, P., Dahlgren F., and Lundberg L. "A Lockup-free Multiprocessor Cache Design," in *Proc. of the 1991 Int. Conference on Parallel Processing*, Vol. I, pp. 246-250, 1991.