To appear in *IEEE Transactions on Parallel and Distributed Systems*

Evaluation 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
Internet: {fredrik,per}@dit.lth.se, http://www.dit.lth.se/cachemire/

Abstract. We study the 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 as well as and in same cases even better than stride prefetching for five 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 these stride accesses, and (ii) sequential prefetching also exploits the locality of read misses with non-stride accesses. However, since stride prefetching in general results in fewer useless prefetches, it offers the extra advantage of consuming less memory-system bandwidth.

Corresponding author: Fredrik Dahlgren

Keywords: Hardware-Controlled Prefetching, Latency Tolerance, Performance Evaluation, Relaxed Memory Consistency, Shared-Memory Multiprocessors.

Evaluation 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 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 as well as and in same cases even better than stride prefetching for five 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 these stride accesses, and (ii) sequential prefetching also exploits the locality of read misses with non-stride accesses. However, since stride prefetching in general results in fewer useless prefetches, it offers the extra advantage of consuming less memory-system bandwidth.

1 Introduction

Large-scale shared-memory multiprocessors with a general interconnection network often suffer from a significant processor stall time, mostly because of the large latencies associated with read and write accesses. 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 [21] are quite effective at reducing the average read stall time. However, due to cache block coherence maintenance and replacements, additional 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 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 [16, 17] 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 non-binding hardware-based prefetching strategies in shared-memory multiprocessors are sequential [6, 20] and stride prefetching [5, 9, 13, 19]. 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, e.g. [1], require fairly complex detection mechanisms and also some modification to the processor.

This paper evaluates the relative performance as well as implementation implications of a variety 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. We continue in Section 3 to review previously proposed hardware-based prefetching schemes and develop in the process a classification scheme based on (i) how strides are detected and (ii) how prefetching is kept going, once it has been enabled. This classification is important in order to understand the implementation cost of each prefetching scheme as well as the performance benefit of a certain feature. In a previous study [8], we compared the effectiveness of two specific implementations of sequential and stride prefetching schemes. In this paper, we extend that work by a detailed analysis of a range of implementations as well as different strategies to keep prefetching going, once it has been enabled. In addition, while all results in [8] are for infinite caches only, we also cover in this paper the effect of finite caches on the performance of stride and sequential prefetching schemes.

In Sections 4 and 5, we evaluate the relative performance of the prefetching schemes using a detailed architectural simulator and a set of six scientific benchmarks, in which stride accesses dominate in four of them. Surprisingly, we find that one of the simplest variations of sequential prefetching does as well and in some cases even better than the most aggressive stride prefetching scheme for five out of the six applications we use. The intuitive reason for this is twofold: First, since strides are shorter than the block size (we have assumed a relatively small block size of 32 bytes), prefetching of consecutive blocks will cut the number of misses of stride accesses substantially. 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 in the applications we study. An observed shortcoming of sequential prefetching, however, is that its prefetch efficiency is in general lower which could speak in favor for stride prefetching schemes if the memory-system bandwidth is not sufficient. We end this paper by relating our work to that of others in Section 6 and conclude in Section 7.

2 The Baseline Architecture

The architectural framework in which we evaluate the implementation cost as well as the performance of a some stride as well as sequential prefetching schemes consists of a cache-coherent NUMA (non-uniform memory access-time) architecture. The shared-memory is distributed across the processing nodes, which are interconnected by a general interconnection network. The overall organization of each processing node is shown in Figure 1.

The key feature of this particular design of the processing node is its lockup-free [14, 22] second-level cache (*SLC*) which is a key component to the support of non-binding prefetching schemes as well as relaxed memory consistency models. Moreover, we assume a blocking-load processor which is interfaced to a simple, and thus fast, on-chip, first-level data cache *(FLC*).

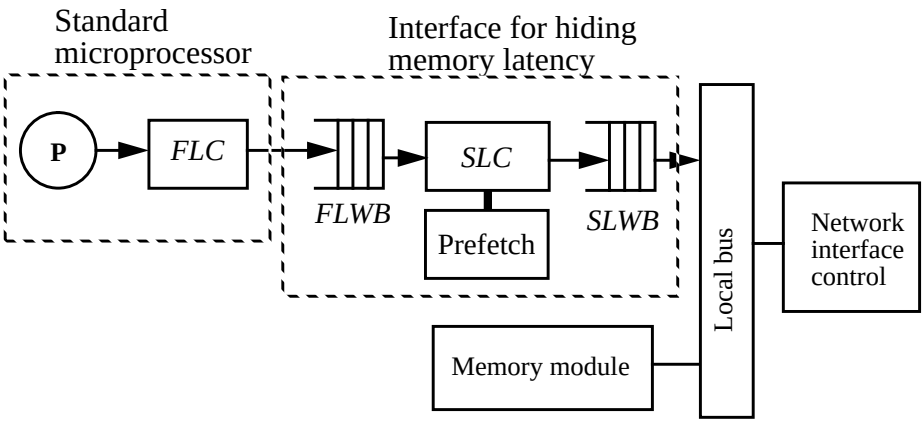

Figure 1. The processor environment and the simulated architecture.

The *FLC* is a write-through, direct-mapped data cache with no allocation of blocks on write misses that blocks on read misses, and has a block-invalidation pin connected to the outside of 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 lockup-free capability of the *SLC* is supported by the second-level write-buffer (*SLWB)* which buffers all pending requests such as prefetch, read miss, and invalidation requests.

Since the focus of this study is how to reduce the read-miss penalty for processors that block on loads, techniques for dealing with write-penalty reduction are orthogonal. One such technique is relaxation of the memory consistency model which has implications for the programmer as well as the design of the processing-node memory subsystem. To reduce the design space, however, our study assumes *release consistency* [11]. Under release consistency, write requests can be overlapped so that their penalties can be completely eliminated [11]. This overlapping of write requests is made possible through the lockup-free mechanism implemented in the *SLC* controller working in conjunction with the *SLWB*. Note that the *SLWB* is needed to buffer prefetch requests also under a sequentially consistent implementation.

We concentrate on off-chip mechanisms that interfere little or nothing with the processor chip. In order to focus on how to hide the large latencies involved with internode accesses rather than the substantially shorter latencies involved with local actions, this study only considers prefetching into the *SLC*. Thus, all prefetching mechanisms require modifications to the *SLC*. 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 Evaluated Prefetching Algorithms

We now describe the prefetching schemes, and how they are incorporated into the baseline system. We start in Section 3.1 by introducing a terminology for stride prefetching. We continue in Sections 3.2 and 3.3 with descriptions of the evaluated stride prefetching schemes, and in Section 3.4 with the evaluated

sequential prefetching schemes. Finally, in Section 3.5 we compare the implementation cost of the evaluated schemes and build intuition into expected performance differences.

3.1 Stride Prefetching Terminology

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

There are two issues that must be addressed in order for a stride-prefetching scheme to be effective. First, it must dynamically identify the stride in a stride sequence, which is done in the *detection phase*. Second, when a stride is detected it must issue prefetch requests 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

The stride detection phase aims at detecting a stride sequence. Since stride sequences are typically interleaved with read and write accesses that do not belong to a stride sequence, detecting a sequence is not trivial. However, on the premise that all such sequences are generated in loops, (e.g. the vector accesses in Figure 2), all accesses for a certain stride sequence originate from the same load instruction. Thus, by keeping track of the instruction address associated with each read or write access, it is possible to compare the data address of a memory access with previous accesses evolving from the same instruction and this way detect a stride. This 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* mechanism. The implication for our baseline architecture is that *FLC* read-miss requests must include the instruction address of the corresponding load instructions causing these misses. To avoid this, people have also considered detection mechanisms that do not require that the instruction address is available; rather they analyze all read addresses in order to isolate possible strides. Such detection mechanisms will be referred to as *D-detection* mechanisms.

Examples of I-detection algorithms can be found in [5, 9, 19]. In this study, we only consider prefetching into the *SLC;* i.e., only read requests that miss in the *FLC* can trigger prefetching. For all I-detection

schemes we consider, the instruction address of the load instruction that misses in the *SLC* is matched against previous 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 the 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, there will be 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 *2-state 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.

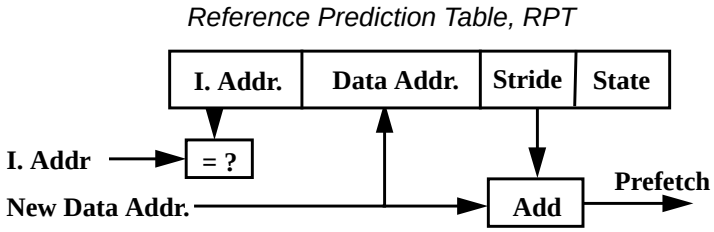

Figure 3: I-detection scheme

In order to study how the number of useless prefetches can be reduced by more selective mechanisms, we will also consider two other schemes. The most straightforward technique is a *3-state scheme* that requires three read-miss requests whose addresses are separated by the same stride and generated from the same instruction to start prefetching. The state transition graph of the 3-state scheme is similar to the stride prefetching scheme proposed by Sklenar in [19].

Another approach is the *4-state 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 4-state scheme is the state `no-pref`, which means that prefetches are stopped for a load instruction, if three incorrect predictions happen in a row. This reduces the number of useless prefetches.

We have slightly modified our implementation of the Baer and Chen scheme [1] as follows. Instead of using an additional program counter that looks ahead the same number of instructions as the miss latency we want to overlap, counted in instruction cycle-times, our implementation uses a mechanism incorporated in the *SLC* with the purpose of issuing prefetches to blocks belonging to the detected stride sequences which is explained in Section 3.3. 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].

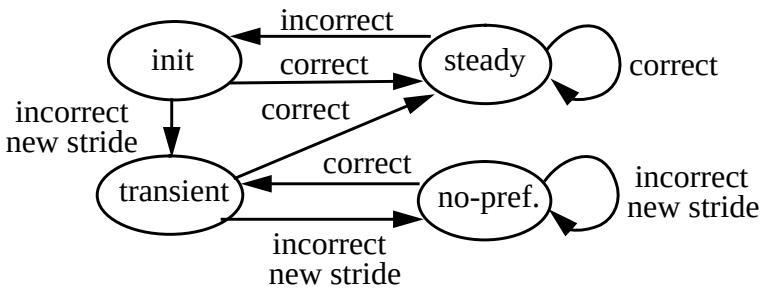

Figure 4: State-transition graph of the 4-state stride prefetching scheme.

In contrast to I-detection schemes, D-detection schemes must compare a data address against all previous data addresses in order to detect regularities, which makes the detection phase much more complicated. We consider the D-detection algorithm proposed by Hagersten in [13], which is the only one we are aware of. Conceptually, it works as follows. The address of each read miss is matched against recent misses buffered in a *miss list*, all possible strides are calculated, and a *frequency table* for strides is 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. The stride threshold is 3, which means that if the stride has not occurred previously four misses belonging to the same stride sequence are required before it is recorded in the list of common strides. When this happens, two additional misses are required to initiate prefetching.

3.3 Stride Prefetching Phase

Orthogonal to the selection of a stride detection mechanism is the choice of a stride prefetching mechanism 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. We have studied two simple but effective schemes for the prefetching phase, referred to as the *fixed* and *alive* prefetching schemes, which are both applicable for I-detect as well as D-detect.

The simplest prefetching scheme is the *fixed* scheme which works as follows. When a stride is detected starting at data address *B*, and the stride is calculated to *S*, the blocks to prefetch are *B+S*, *B+2*S*,..., *B+d*S*, where *d* is referred to as the *degree of prefetching*. If the processor continues to access blocks in the same stride sequence, there will be hits in the SLC for these blocks as long as they are not replaced or

invalidated. If the stride sequence contains more than *d* blocks, there will be a new miss. The data and instruction addresses of this miss are then matched against the corresponding entry in the RPT (I-detection) or in the stream list (D-detection). If the miss belongs to a recorded stride sequence, *d* new prefetches are issued. Thus, prefetch requests are only issued on read misses, and a fixed number (*d)* of prefetch requests are issued at a time.

The *alive* scheme we will evaluate is described in Figure 5. Like the fixed scheme, blocks *B+S, B+2*S, ..., B+d*S* are prefetched when a stride is initially detected at block address *B*. Unlike the fixed scheme, however, 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, the stride *S* is read from the RPT and the next block in the stride sequence at address *B+S+d*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. As a result, there will be no read misses beyond the ones in the detection phase[1]. The alive scheme is similar to the prefetching-phase mechanism proposed by Hagersten in [13].

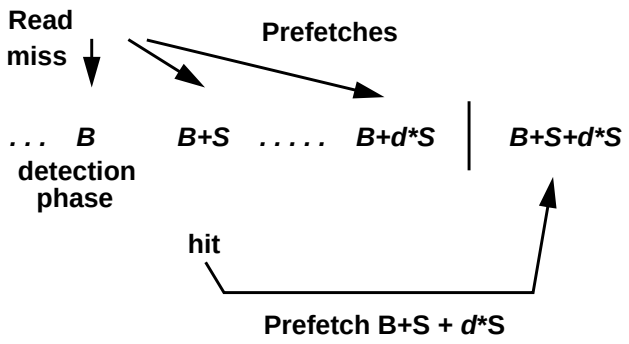

Figure 5: The alive scheme in the prefetching phase.

3.4 Sequential Prefetching

Sequential prefetching is based upon the assumption that cache misses exhibit high spatial locality; if the processor accesses block *B*, the probability is high that the processor will also reference block *B+1*. Therefore, if the cache experiences a miss on block *B*, it prefetches blocks *B+1, B+2,..., B+d*, if these blocks are not already in the cache, where *d* is referred to as the *degree of prefetching*. Sequential prefetching has been shown to perform well for parallel applications on a shared-memory multiprocessor by Dahlgren *et al.* in [6, 7], and in multiprocessor vector machines by Fu and Patel in [9]. In general, sequential prefetching requires much less hardware complexity. This is simply because no detection phase is needed and because sequential prefetching can adopt the same schemes for the prefetching phase as stride prefetching schemes.

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

This study covers two prefetching-phase mechanisms for sequential prefetching, in essence the same as fixed and alive for stride prefetching. *Fixed sequential prefetching* works as follows. On a read miss on block B, the corresponding read request is issued immediately and blocks *B+1, B+2,...,B+d* are prefetched. This scheme requires (i) no extra bit for prefetched blocks, (ii) no cache controller action on read hits; prefetching is only done when the processor is blocked because of the read miss. Thus, the extra complexity is negligible.

Alive sequential prefetching works as follows. On a read miss on block *B*, the corresponding read request is issued immediately and blocks *B+1*, *B+2*,..., *B+d* are prefetched. 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, etc. Like the alive stride prefetching scheme, only one miss will be encountered initially for a whole sequence. In the next section we will summarize the differences between the stride and sequential prefetching schemes in terms of complexity and expected performance.

3.5 Summary

In order to compare the prefetching schemes implementation-wise, Table 1 summarizes the extra hardware needed for the I-detection (4-state), the D-detection, and the sequential prefetching schemes. I-detection needs support from the processor chip in terms of the instruction address of the load instruction that missed in the *FLC*, while neither D-detection nor sequential prefetching need such support. Table 1 clearly shows that the hardware needed for the I- and D-detection stride prefetching schemes is substantially higher than for sequential prefetching, even though the size of the RPT or the associative buffers are fairly small. The complexity of handling an *SLC* miss is also substantial, especially for D-detection, whereas it is negligible for sequential prefetching. In particular, sequential prefetching with the alive prefetching phase and *d*=1 only needs to prefetch one block ahead (if any) on *SLC* misses or on hits on tagged blocks.

Table 1: Hardware extensions for three prefetching schemes.

	4-state I-detect	**D-detect**	**Sequential Prefetching**
Processor chip	Instruction address available for read misses in the *FLC*.	No	No
SLC	RPT (256 entries) + state machine + calculation of strides. Prefetched blocks are tagged.	4 associative buffers, 16 entries each. Calculation of strides At least one buffer is scanned and potentially updated on each miss Alive: Prefetched blocks are tagged	Fixed: No additional hardware except for issuance of prefetches Alive: Prefetched blocks are tagged.

The difference in hardware complexity between stride and sequential prefetching is entirely due to the mechanisms needed for the stride detection phase, which do not have a counterpart in sequential prefetching schemes. As a result, the stride prefetching schemes can predict future references for any constant stride, while the sequential prefetching schemes rely on a high spatial locality to perform well. Important questions concerning the effectiveness of these schemes become: (i) Are there enough references and misses that belong to stride sequences to justify the complexity of stride prefetching schemes? (ii) Are the stride sequences long enough that the misses needed to detect a stride sequence become negligible? (iii) Is the stride often shorter than the number of words contained in a cache block? If it is, sequential prefetching is expected to be as effective for stride sequences. Finally, (iv) how much spatial locality do accesses that do not belong to stride sequences exhibit? We focus on the above questions in the following sections.

4 Simulation Methodology and Benchmark Programs

We have developed simulation models of the baseline architecture and all the prefetching algorithms presented in the previous section. The simulation platform is the CacheMire Test Bench [3], a program-driven functional simulator of multiple SPARC processors. It consists of two parts: (i) a functional simulator and (ii) a memory-system simulator that models the functionality of the memory system so that the same interleaving of memory-system events is maintained as in the target system we model. 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 presence-flag vector associated with each memory block points to the processor nodes with a copy in their caches. An *SLC* read miss sends a read miss message to the *home* memory module (the node at which the physical memory page containing the block is allocated). If the home memory is the local node and if the block is clean (unmodified), the miss is serviced locally. Otherwise, the miss is serviced either in two or in four node-to-node transfers depending on whether the block is dirty (modified) in some other cache. A write access to a shared or invalid copy in the *SLC* sends an ownership request to the home node; in response, the home node sends invalidations to all nodes with a copy, waits for acknowledgments from these nodes, and, finally, sends an ownership acknowledgment to the requesting node plus the copy of the block if needed. While the home node is waiting for the completion of a coherence action—e.g., the invalidation of copies—the memory block is in a transient state. If a read miss or ownership request reaches the memory while the block is in a transient state, it is rejected and must be retried.

The architectural parameters that are kept constant in all simulations appear in Table 2. As for the size of the *FLC*, we have chosen 4 Kbytes to get a realistic replacement miss rate. Moreover, although a large block size would be advantageous for the sequential prefetching scheme to be effective for large strides, we pessimistically assume a fairly small block size of 32 bytes. In [6], Dahlgren *et al.* evaluate how sequential prefetching behaves for larger block sizes.

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 cycles. 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 the *SLWB* are 8 and 16 entries, respectively. The word size is 4 bytes.

We assume a single 4-by-4 Mesh network clocked at 100 MHz with wormhole routing and with a flit size of 32 bits. The node fall-through latency is two network cycles. Throughout the study, contention is accurately modelled in all parts of the system. The last two rows in Table 2 show the time it takes to service a read request provided that there is no contention and if the average distance between nodes are considered for the network latency. A read request is sent to the memory module where the block and its directory is located, denoted the *home* of the block. If the home has a clean copy of the block it responds directly, and if the home is not in the requesting node there will be a network latency of two node-to-node transfers, resulting in a total read stall of 75 pclocks. However, if the block is dirty, the home must first request a shared copy from the cache holding the dirty copy, which results in a total of four node-to-node transfers and a read stall time of 143 pclocks. 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.

Table 2. 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	30 pclocks
Read from Home (two node-to-node traversals)	75 pclocks
Read from Remote (four node-to-node traversals)	143 pclocks

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. These are summarized in Table 3. Four of them are taken from the SPLASH suite (MP3D, Water, Cholesky, and PTHOR) [18]. The other two applications (LU and Ocean) have been provided to us from 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`. For all measurements, we gather statistics during the parallel sections only according to the recommendations in the SPLASH report [18].

Table 3. Benchmark programs.

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

Table 4 shows the cold, coherence, and total cache read miss rates for each of the applications for the baseline architecture without prefetching and with infinite *SLC*s. Since we assume full inclusion between the *FLC* and the *SLC*, the miss rates is calculated as the total number of read misses in the *SLC*s divided by the total number of read accesses to shared data in the system. A miss is classified as being a cold miss of the block has not previously been fetched into the cache.

Table 4: Miss rates for the baseline architecture (no prefetching) and infinite *SLC*s.

Benchmark	Cold miss rates	Coherence miss rates	Total miss rates
MP3D	1.3%	8.9%	10.2%
Water	0.04%	0.72%	0.76%
Cholesky	1.0%	0.35%	1.35%
LU	0.86%	0.05%	0.91%
Ocean	0.02%	0.75%	0.77%
PTHOR	2.5%	3.8%	6.3%

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 for the results we show in the subsequent sections. We continue in Sections 5.2 - 5.3 by analyzing different stride detection and prefetching phases assuming an infinite *SLC*. We evaluate the relative effectiveness of stride and sequential prefetching in Section 5.4, before we broaden the results to cover finite second-level caches as well as larger data sets in Sections 5.5 and 5.6, 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 belong 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 for other accesses is low.

The analysis is based upon an execution on the baseline architecture, and we only consider the 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 Sections 5.2-5.4, we only consider requests from one processor in this section, which has been shown to be representative. In the measurements, we have characterized a stride sequence as being generated by the same load instruction, and it requires at least three read misses with equidistant addresses. We use I-detection in order to identify stride sequences, and at least three accesses with the same instruction address and with equidistant data addresses (the stride) are required to tag the accesses as belonging to a stride sequence. The results of this experiment are shown in Table 5.

Table 5: Application characteristics. Infinitely large second-level cache.

	MP3D	**Cholesky**	**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
Most common strides	1 (76%)	1 (95%)	21 (99%)	1 (93%)	65 (42%), 1 (31%)	1 (37%)

For MP3D, we see that only 9.2% of all misses belong to stride sequences. This means that stride prefetching based on I-detection is limited to these 9.2%, and since it requires some misses for the detection phase before prefetches are issued, the actual miss reduction will be even 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. The most common stride is 1 block (76% of all stride accesses belong to sequences with stride 1), which means that most of the stride access misses are also covered by sequential prefetching. In addition, the accesses to the `Particles` data structure show a reasonably high spatial locality, and we can expect sequential prefetching to perform reasonably well.

For Cholesky, Water, and LU, almost all misses belong to stride sequences. They are in general larger 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. For Cholesky and LU, almost all strides are 1 block, and sequential prefetching is expected to do equally well. For Water, on the other hand, most strides are substantially longer than one, and sequential prefetching is limited to the spatial locality of references belonging to different stride sequences and non-stride accesses.

For Ocean, most misses belong to stride sequences (66%), and the length of the sequences are 7.6 on average, which indicates that stride prefetching should do reasonably well. Even if 31% of all misses

belong to sequences with a stride of 1 block, this is not a sufficient reason for sequential prefetching to perform as well as stride prefetching, if there is no spatial locality in non-stride accesses. Thus, stride prefetching is expected to be more effective than sequential prefetching at reducing the number of read misses for Ocean.

For 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. In the following sections, we will evaluate different stride prefetching techniques and the relative effectiveness of stride and sequential prefetching, and we will use above metrics in order to explain the results. Subsequently, we will use the same key parameters and broaden the analysis in order to make an intuition about the relative effectiveness of stride and sequential prefetching for finite sized second-level caches in Section 5.5 and larger data sets in Section 5.6.

5.2 The Stride Detection Phase

In this section, we analyze the read miss reduction and prefetch efficiency of three different I-detection mechanisms denoted the 2, 3, and 4-state schemes as introduced in Section 3.2. To simplify the wording, we refer to them as 2-state, 3-state, and 4-state. The prefetch efficiency is the fraction of all prefetched blocks that are accessed during their lifetime in the cache. For all three mechanisms, we assume the *fixed* prefetching scheme with d = 1, 2, and 8, and an infinite *SLC*. Figure 6 shows the reduction of read misses relative to the baseline architecture (with no prefetching), while Figure 7 shows the prefetch efficiency. Read misses for blocks that are being prefetched but have not yet arrived in the cache are counted as hits in the diagram. This is justified by the observation that the read stall time for such accesses with the fixed prefetching scheme is on average significantly shorter than the total time of servicing a read miss. The reason for this is that prefetches are only issued at read misses. Since prefetching across page boundaries is not supported, and the memory is distributed across the memory modules page-wise, prefetch requests always go to the same memory module as the read-miss request that triggered prefetching. Since prefetch requests are issued immediately after the read-miss request is issued, they are often completed soon after the block for the read-miss request has returned.

As can be seen in Figure 6, the reduction of read misses for MP3D is very low, regardless of the degree of prefetching, d, and the detection mechanism. For 3-state and 4-state, the reduction lies between 2% and 5%, while it is slightly higher for 2-state. The reason for the modest reduction is explained in the previous section; the fraction of read misses that belongs to stride sequences is low. 3-state and 4-state are more selective than 2-state at detecting stride sequences and decide when to prefetch, which is why their reductions of read misses are lower. Whereas 3-state only prefetches when it is definitely a stride sequence and 4-state does not prefetch at instruction addresses that previously did not request stride sequences, 2-state continuously issue prefetches as long as it finds at least 2 read requests at the same instruction address. These expectations are shown in Figure 7; the prefetch efficiency of 2-state is extremely low for MP3D, which means that a large number of blocks which are never accessed are prefetched. On the other hand, the more selective detection mechanisms, 3-state and 4-state, have a much higher prefetch efficiency. As can

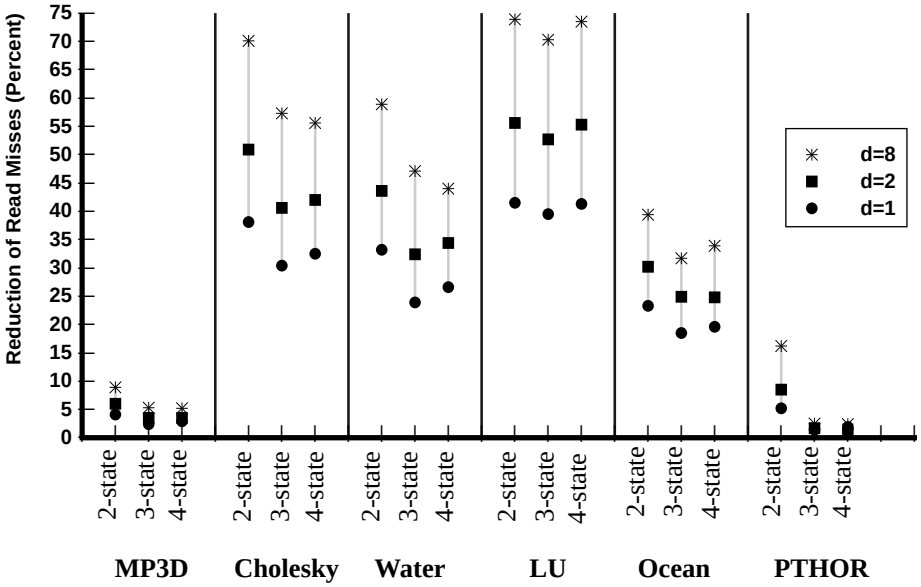

Figure 6. Reduction of read misses (in percent) relative to the baseline architecture.

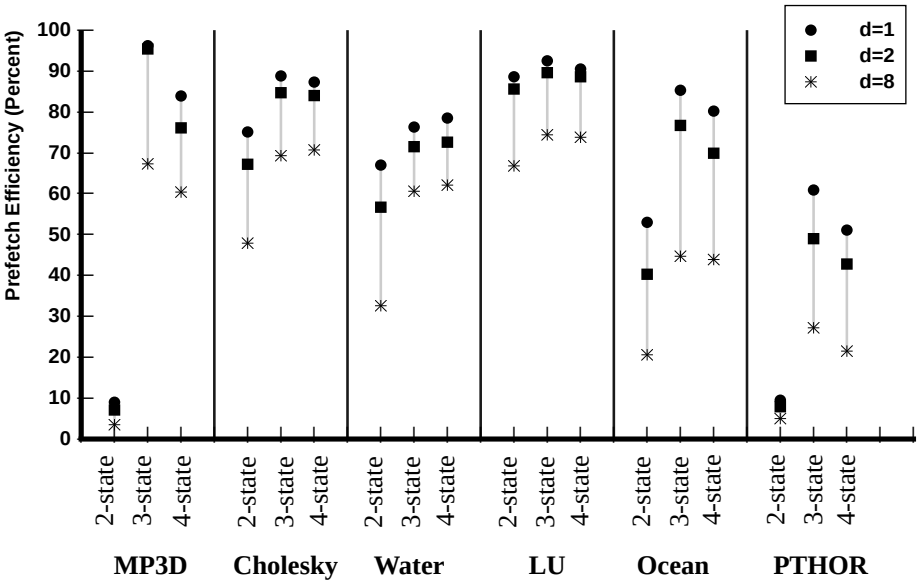

Figure 7. Prefetch efficiency in percent.

be seen for 4-state, the prefetch efficiency is lower for a higher *d* — it is decreased from 84% for *d*=1 to 60% for *d*=8. When the stride sequence comes to an end, all subsequent prefetches issued to that sequence will normally be useless. For the fixed prefetching scheme with *d*=1, the number of useless prefetches per stride sequence can be no more than 1, since no more than one prefetch is issued at a time to the same sequence, while for *d*=8 it can be up to 8.

For Cholesky, Water, and LU, the reduction of misses is much larger. This also follows from Table 5 in the previous section, where most read misses for these applications belong to stride sequences. In general, 2-state removes more misses because it prefetches more and needs fewer misses to detect a stride sequence. Unfortunately, it has a lower prefetch efficiency. A higher *d* means a larger reduction of read misses. Since the fixed scheme requires one miss for *d* blocks to be potentially prefetched, we note that the

number of misses will be decreased when *d* increases. On the other hand, a higher *d* can imply a lower prefetch efficiency which we see in MP3D.

For Ocean, the reduction of read misses as well as prefetch efficiency is between that of MP3D and LU. This is expected, since the fraction of read misses that does not belong to stride sequences is higher than in LU but lower than in MP3D (see Table 5.) In PTHOR, the reduction of misses shown in Figure 6 is extremely low, which is explained by the absence of stride sequences as seen in Table 5. 2-state shows a higher reduction, because it still issues a large number of prefetch requests, but just like in MP3D, most of these are useless, and the prefetch efficiency is thus extremely low.

Overall, even though 2-state is shown to reduce the number of misses slightly more than 3-state and 4-state, the cost is a higher traffic. For applications like Cholesky, Water, and LU, this is not a problem, since the prefetch efficiency is still reasonably high. On the other hand, for applications like MP3D and PTHOR, where the potential of stride prefetching is small, the low prefetch efficiency implies a large increase in network traffic at the same time as the number of read misses is only marginally reduced. This clearly shows the importance of a selective stride detection mechanism in order not to degrade the performance due to contention effects. For the other applications, especially when the stride sequences are long, the difference in the miss reduction capability between 2-state, 3-state, and 4-state is small. The difference between 3-state and 4-state turned out to be small. In the following, the only I-detection mechanism that will be considered is 4-state.

5.3 The Stride Prefetching Phase

In order to evaluate the relative merits of the fixed and the alive prefetching mechanisms, we show below the reduction of read misses in Figure 8, the read stall time in Figure 9, and the prefetch efficiency in Figure 10. All measurements assume 4-state as the stride detection mechanism and an infinite *SLC*.

As can be seen in Figure 8, the alive prefetching mechanism results in general in a larger reduction of read misses than fixed. In addition, the reduction for alive seems to be independent of *d*. The reason for this is that, for each stride sequence, almost all read misses encountered are those in the detection phase. After that, prefetching is continuously issued as long as prefetched blocks are being accessed and page boundaries are not crossed. (A page boundary crossing affects fixed and alive prefetching in the same way and leads to an extra miss whereafter prefetching is continued.)

As in the previous section, read misses for blocks that are being prefetched but have not yet arrived in the cache are counted as hits in the diagram. This is intuitively justified for the fixed scheme, since the processor is stalled during most of the latency of the prefetch requests. For the alive scheme, however, a prefetch is issued every time the processor reads a block tagged as prefetched, and the processor is potentially not stalled more than a few cycles before its next read request to the same stride sequence. Thus, a block that is being prefetched might soon be referenced, which leads to a processor stall while waiting for the prefetched block to arrive. Figure 9 shows the fraction of the read stall time that comes from read requests that miss in the *SLC* and encounter the full latency of the memory-system (labelled miss stall in Figure 9) and the fraction that comes from waiting for blocks that have already been requested by previously issued prefetches or write-miss requests (pending stall). The baseline architecture has less pending stall time since no prefetches are issued. For the alive scheme, labelled Alive-x (the alive scheme with

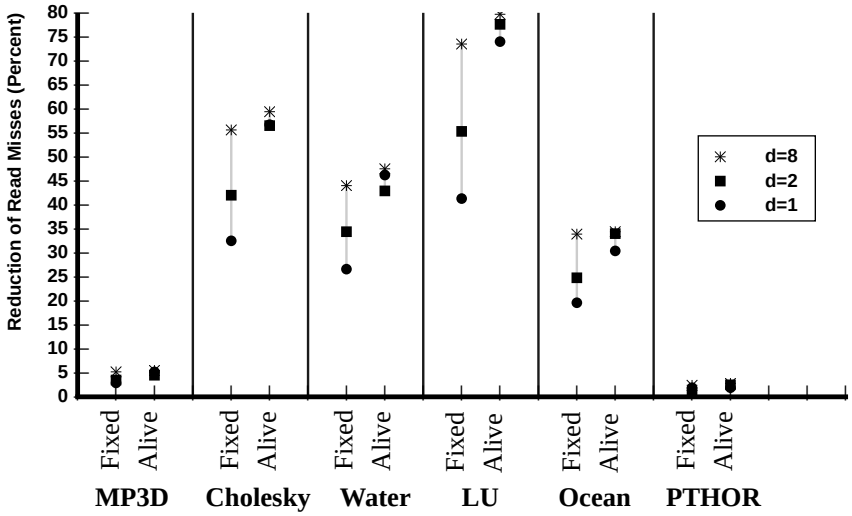

Figure 8: Reduction of read misses relative to the baseline architecture.

d=x), the pending stall component is decreased as *d* increases from 1 to 8. The reason is that for *d*=8, prefetching is carried out 8 blocks ahead of the processor's read requests to each stride sequence, while for *d*=1, prefetching is only 1 block ahead (when one block is requested the subsequent one is prefetched). On the other hand, for most of the applications, the miss stall time is marginally increased when *d* increases because of increased contention in the wormhole routed mesh and in the memory modules. Overall, the pending-stall component is small, and the difference between the read stall time for different *d* is small for the alive scheme.

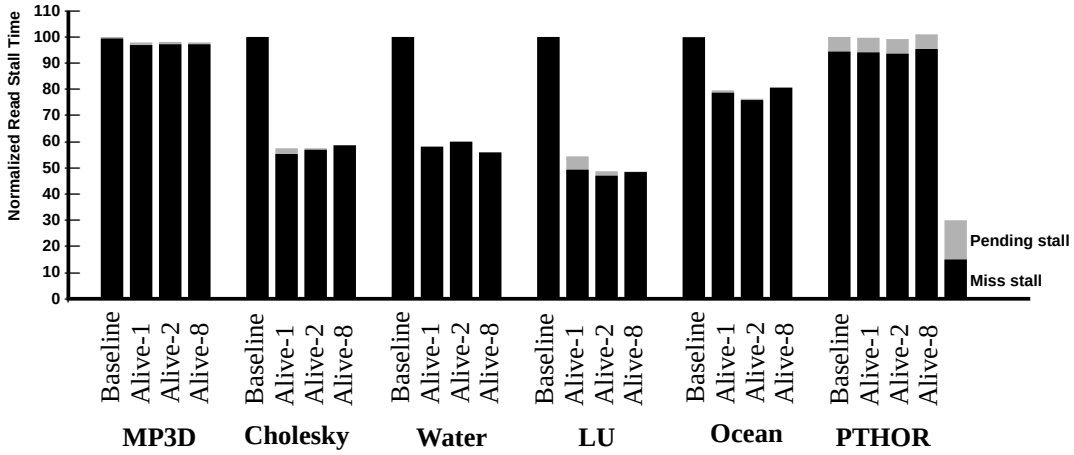

Figure 9: Read stall time relative to the baseline architecture divided into stall times because of read misses and wait times for pending request.

Continuing with the prefetch efficiency in Figure 10, the number of useless prefetches tends to be larger for alive at the end of each stride sequence, since prefetches are continuously issued as the stride sequence is being accessed. On the other hand, the number of useful prefetches is larger along the sequence, since no further read miss is encountered. As a result, the prefetch efficiency is approximately the same as long as *d* is the same, which is shown in Figure 10.

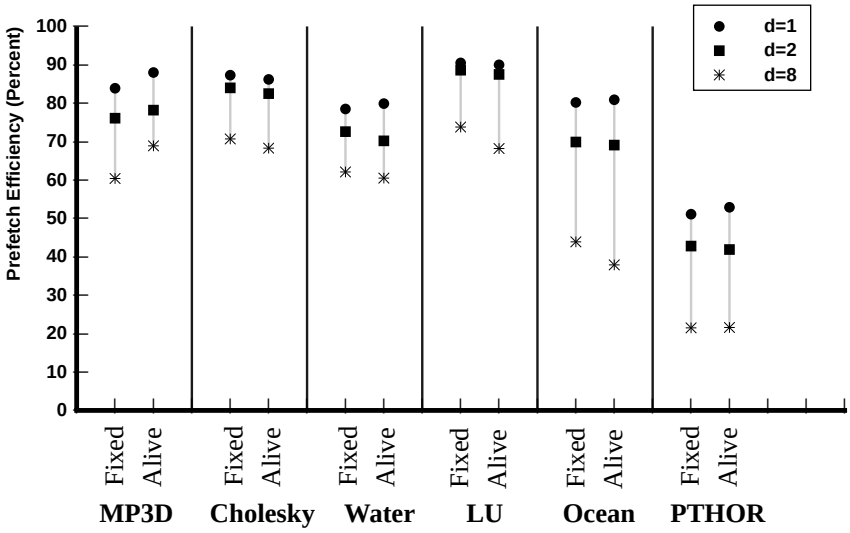

Figure 10: Prefetch efficiency.

Because of a lower prefetch efficiency, the amount of network traffic is much higher for *d*=8 than for *d*=1 for fixed as well as for alive. Since the miss-reduction capability of the alive scheme with *d*=1 is as large as for fixed with *d*=8, but at a lower network bandwidth consumption, we will in the following only consider the alive prefetching scheme with *d*=1 for stride prefetching.

5.4 Stride vs. Sequential Prefetching

In order to develop 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.

Figure 11 shows the number of read misses for each prefetching scheme relative to the baseline architecture. For each application, there are three bars: (from left) the I-detection scheme (the 4-state detection scheme), denoted I-det; the D-detection scheme, denoted D-det; and the sequential prefetching scheme, denoted Seq. Each bar consists of two sections: cold misses (on the bottom) and coherence misses. A miss is classified as being a cold miss if the block has never been fetched or prefetched into the corresponding cache previously. All three schemes use the alive prefetching phase mechanism with *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 behind this is that sequential prefetching covers most of the stride sequences (recall that 76% of all strides are 1 block) and that it exploits 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. Our expectation from Section 5.1 is that stride prefetching should perform reasonably well, while there is neither enough 1-block strides nor enough spatial locality of accesses belonging to

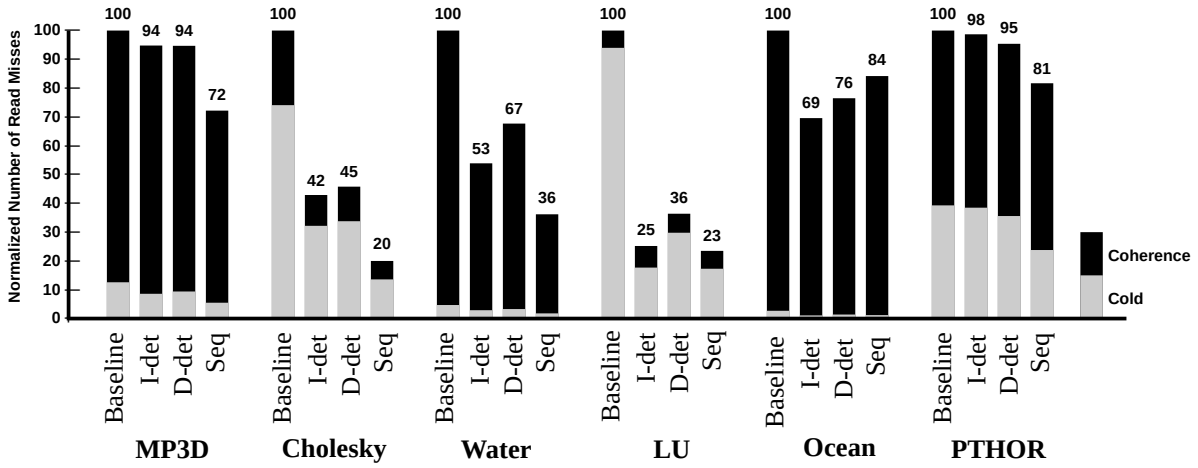

Figure 11: Number of read misses relative to the baseline architecture.

different strides for sequential prefetching to perform equally well. For PTHOR, all three techniques perform poorly, although sequential prefetching manages to reduce the number of misses to some extent. We also see that the prefetching schemes are more effective at reducing the cold miss rate than the coherence miss rate for all applications.

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.

Figure 12 shows the prefetch efficiency. For the applications for which all three schemes are effective at reducing the number of misses (Cholesky, Water, and LU), all three schemes also have a very high prefetch efficiency. For the other three applications, MP3D, Ocean, and PTHOR, I-detection has in general 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 through the instruction stream only, which means 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, e.g. in Ocean and PTHOR. These useless prefetches increase the traffic, and may introduce contention.

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 Figure 13. The reduction of the read stall times follows to some extent the reduction of read misses, but the reduction of the former is in general not as dramatic. The effect on the read stall times is due to two effects: (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. As a result, the load on the network is increased. Overall, sequential prefetching is more effective at reducing the read stall time for three out of six applications.

Figure 14 shows the total execution times. Each bar is decomposed into busy time, read stall time, acquire stall time (the time spent waiting for an acquire to complete), and buffer stall time (the processor is stalled due to a filled *FLWB*). The execution times follow the trends of read stall time reductions. The dia-

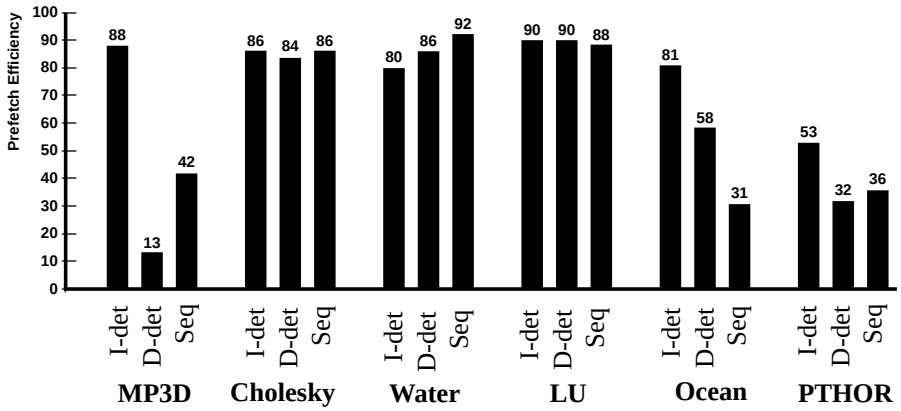

Figure 12: Prefetch efficiency.

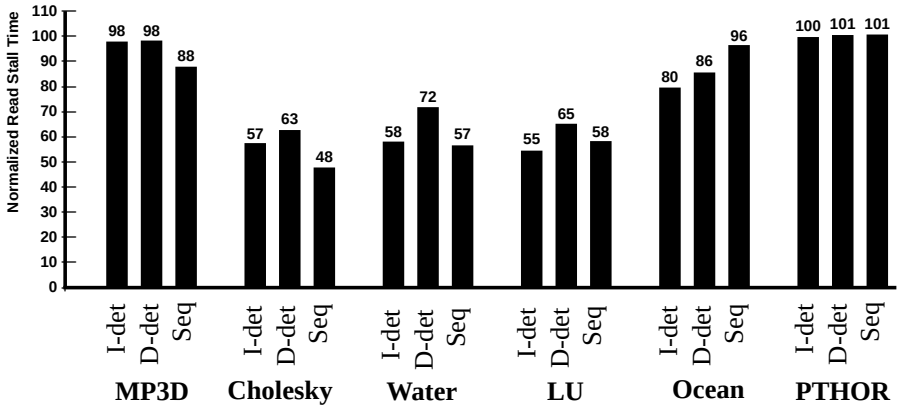

Figure 13: Read stall time relative to the baseline architecture.

gram shows that the execution time for sequential prefetching is lower than I-det as well as D-det for three out of six applications, and equally low as for I-det for two of the others. Ocean is the only application where sequential prefetching performs worse than stride prefetching.

The corresponding network traffic is shown in Figure 15. The network traffic clearly follows the prefetch efficiency trends shown in Figure 12, with the difference that the prefetch efficiency only shows the fraction of prefetches that are useful, but not how many blocks that were prefetched. This is why sequential prefetching, despite of a higher prefetch efficiency, generates more traffic than D-detection for MP3D and PTHOR — the number of issued prefetches is much larger under sequential prefetching since there is no selection mechanism that avoids prefetching when no stride is detected.

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

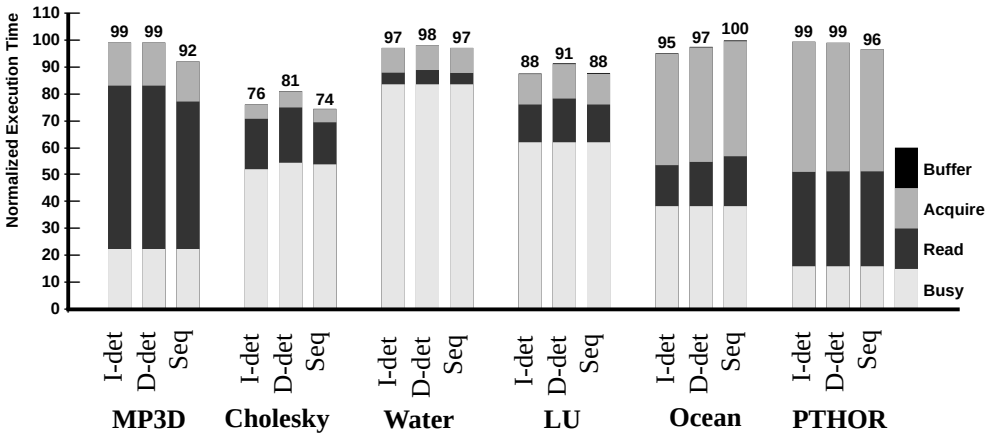

Figure 14: Execution time relative to the baseline architecture.

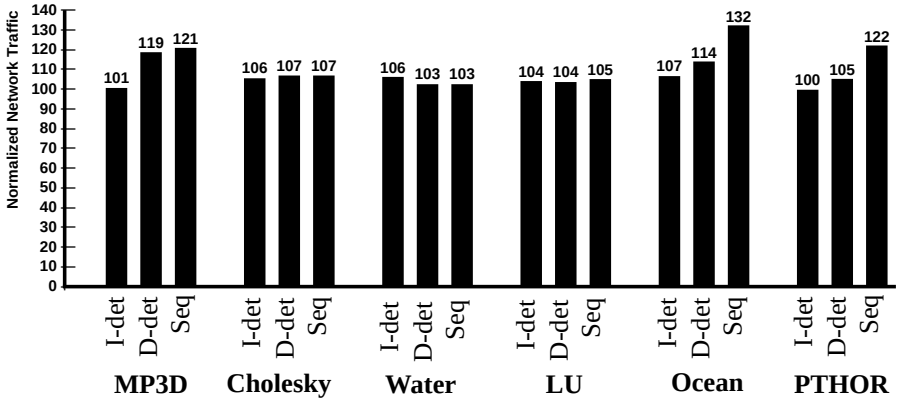

Figure 15: Network traffic relative to the baseline architecture.

explain the results. In the next two sections, we will use these parameters in order to broaden the results to cover finite caches as well as larger data sets.

5.5 Finite Sized Second-Level Caches

Table 6 shows the application characteristics for a 16kbyte direct-mapped *SLC*. The methodology used was the same as the one in Section 5.1. The first row shows the number of replacement misses relative to total number of read misses, while the rest of the table has the same layout as Table 5. A read miss to a block that was prefetched, but was replaced out from the cache before it was accessed, is classified as a replacement miss in Table 6. The most remarkable differences as compared to Table 5 (infinite *SLC*) are for MP3D and Ocean. For MP3D, more than 90% of all replacement misses belong to stride sequences with stride 1. For 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, 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].

Table 6: Application characteristics for finite 16 kbyte direct-mapped SLC.

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

Figures 16 - 18 show the number of read misses, the network traffic, and the read stall times for the 16kbyte *SLC*s. Since above sections have shown I-detect to consistently provide a higher performance than D-detect, we will in this section only show results for sequential prefetching and I-detect stride prefetching as compared to the baseline architecture with no prefetching. Since the fraction of all misses that belong to stride sequences is slightly lower for finite *SLC*s than for infinite *SLC*s (cf. Tables 5 and 6), I-detect stride prefetching does not perform as well for finite caches as for infinite caches, as can be seen by comparing Figure 16 with Figure 11. The only exceptions are for MP3D and Ocean, where the replacement misses almost always belong to stride sequences. Because of a larger number of useless prefetches for sequential prefetching, one could believe that cache pollution would lead to a larger number of read misses than for stride prefetching. However, as can be seen in Figure 16, sequential prefetching is more effective than I-detect at reducing the number of misses for all applications. Except for Ocean, these results are consistent with the results for infinite *SLC*s. For Ocean, the reason is the increased amount of unit strides and the fact that sequential prefetching utilizes spatial locality for non-stride accesses as well.

Looking at the network traffic in Figure 17, it is clear that sequential prefetching leads to a larger increase in network traffic as compared to I-detect stride prefetching. For sequential prefetching, it is interesting to notice that the amount of network traffic is closer to that of the baseline architecture for MP3D and Ocean, the two applications where the number of unit strides is increased. Figure 18 shows the corresponding read stall times. The major difference from the results shown in Section 5.4 is for MP3D and Ocean where both sequential prefetching and I-detect now perform well. For PTHOR, on the other hand, the large increase in network traffic for sequential prefetching results in a read stall time that is 10% longer than that of the baseline architecture.

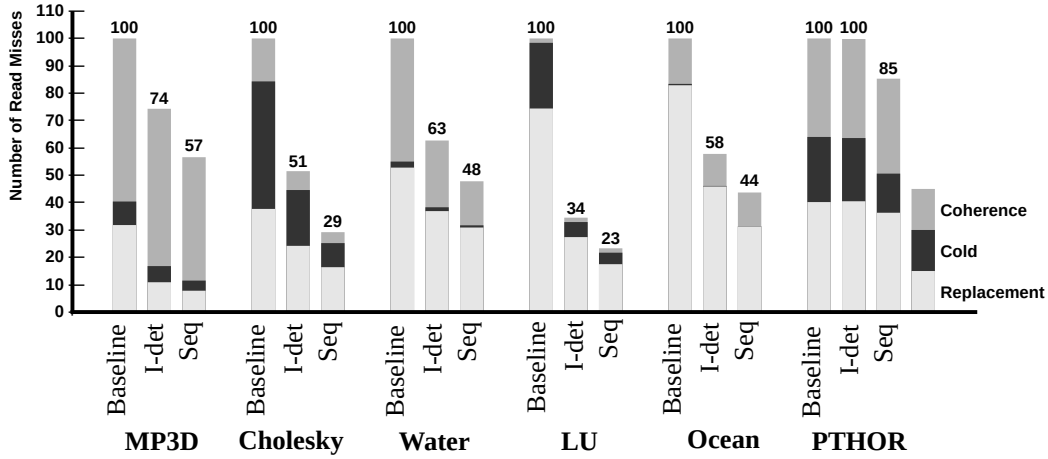

Figure 16: Number of read misses relative to the baseline architecture for 16Kbyte *SLC*s.

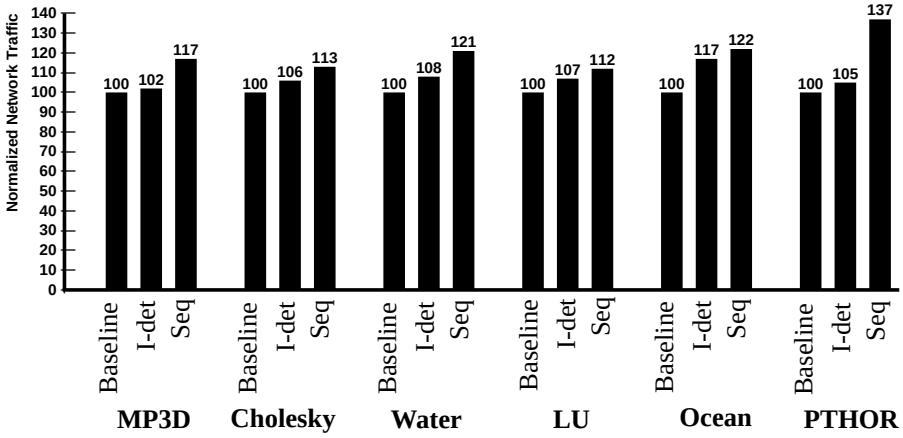

Figure 17: Network traffic relative to the baseline architecture for 16Kbyte *SLC*s.

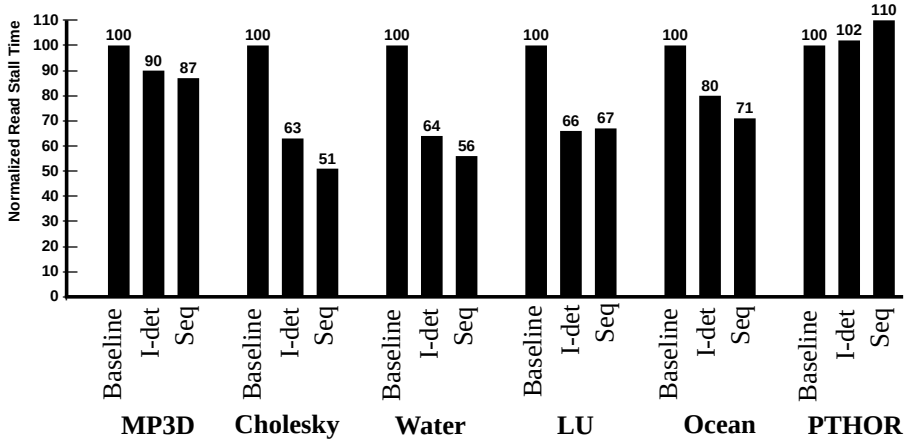

Figure 18: Read stall time relative to the baseline architecture for 16Kbyte *SLC*s.

Figure 19 shows the execution times. For the first four applications, the reduction of the execution time is similar to the results for infinite caches, as was shown in Figure 14. For Ocean, sequential prefetching reduces the execution time with 11%, which is more than stride prefetching does. The reason for this is the good spatial locality of replacement misses that results in a high prefetch efficiency. On the other hand,

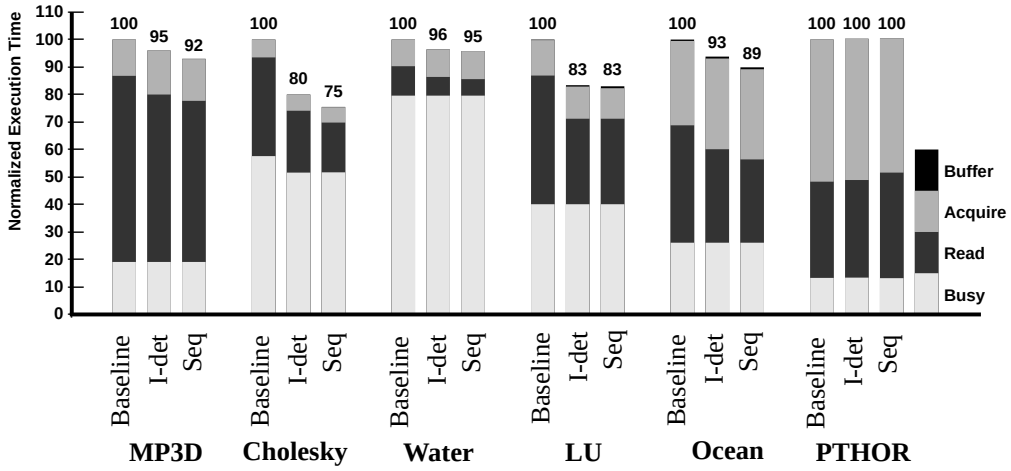

Figure 19: Execution time relative to the baseline architecture for 16Kbyte *SLC*s.

neither sequential nor stride prefetching lead to any reduction of the execution time, in contrast to the situation with infinite caches where sequential prefetching reduced the execution time with 4% (shown in Figure 14). Overall, sequential prefetching reduces the execution time more than I-detect stride prefetching does for four out of the six applications used.

5.6 Larger Data Sets

We have also studied how our results are affected by larger data sets for five of the applications, and studied how the three key application parameters varied with increased data sets. The results are shown in Table 6 in terms of expected tendencies. All trends correspond to infinite second-level caches. For MP3D, we simulated 40 K particles. For Cholesky, we used matrix bcsstk15. Water was run with 576 molecules for 8 time steps. LU was run with a 400x400 matrix. For Ocean, because of time limitations for simulations, we actually simulated a smaller data set, a 64x64 grid and a tolerance of 10^{-3}, and thereafter analyzed the code in order to predict the application and sharing behavior for larger data sets. 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.

Table 7: Expected application characteristics for infinite direct-mapped SLC and larger data sets.

	MP3D	Cholesky	Water	LU	Ocean
Read misses within stride sequence	about the same	increase	increase	increase	increase
Avg. length of sequence	limited	increase	increase	increase	increase
Most common strides	still 1 block	almost all 1 block	not changed	almost all 1 block	longer, > 65

For MP3D, using more particles 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 of Section 5.4, and we

still expect sequential 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 be negligible. On the other hand, sequential prefetching will continue to perform as well as stride prefetching, since almost all strides are one block in Cholesky and LU, and since the spatial locality of accesses belonging to different strides is still very high in Water. For 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 cache size due to the results in Section 5.5. Overall, for these five applications, the relative effectiveness of stride and sequential prefetching showed in above sections is not expected to be dramatically changed with larger and more realistic data sets, and we believe stride and sequential prefetching to perform equally well for Cholesky, Water, and LU, whereas sequential prefetching would perform better for MP3D. For Ocean, stride prefetching is expected to perform better as long as replacement misses are not dominating.

6 Discussion and Related Work

One of the advantages of I-detect over D-detect and sequential prefetching as implemented in this study, is the fact that I-detect does not issue 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 4-state transition graph; for instructions where prefetching has not turned out to be effective, no prefetches are issued. In [6] Dahlgren *et al.* propose an adaptive sequential prefetching mechanism that dynamically changes the degree of prefetching based on an approximation of the 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 on a low level. A similar mechanism is proposed for the prefetching phase of the D-detection scheme by Hagersten in [13]. The reason why we did not consider such mechanisms in the study is because this study should base the evaluation on equally simple prefetching phase mechanisms. 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 in order to detect the stride sequences, and uses the same 4-state transition graph. However, their mechanism differs in that it uses a lookahead-PC, which is a predicted program counter that ideally is as much ahead of the normal PC as the maximum latency of a read request. While our schemes prefetch blocks for a read request at a previous reference from the same load instruction, the scheme by Baer and Chen prefetches 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 4-state scheme and the scheme by Baer and Chen will be nearly identical.

The prefetching phase of Hagersten's prefetching scheme presented 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 alive 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 [9], Fu and Patel evaluate sequential prefetching and stride prefetching for multiprocessor vector caches. They do not assume any specific stride detection mechanism, rather they assume the information about the stride sequence and the stride to be provided by the vector instruction. For scalar and short-stride vector accesses, the prefetcher behaves as a sequential prefetcher, while for long-stride vector accesses it behaves as a stride prefetcher. They show that the stride prefetching scheme was more effective than sequential prefetching for their vector applications. In [10], Fu *et al.* evaluate a stride prefetching approach for scalar uniprocessors. The stride detection scheme is an I-detection scheme similar to the 2-state scheme that uses a *Stride Prediction Table* (SPT) which is in essence the same as the RPT as used in this paper as well as by Baer and Chen in [1]. They evaluate three different schemes for the prefetching phase for read accesses that belong to stride sequences: *pf_miss* which only issues prefetches on read misses, *pf_hit* which only issues prefetches on read hits, and *pf_all* that issues prefetches on all read accesses that belong to stride sequences. The pf_miss is similar to the fixed scheme with degree of prefetching d=1, whereas the pf_all scheme is similar to the alive scheme with d=1. They find that their stride prefetcher was very effective at reducing the number of read misses for applications dominated by vector accesses, while the overhead in terms of useless prefetches could be large for other applications. This is in accordance with our results, indicating a low prefetching efficiency for the 2-state scheme for applications where a minority of the misses belong to stride sequences. Further, they also show that, while pf_all is capable of reducing the number of read misses the most, pf_all and pf_hit have a substantially higher overhead in terms of useless prefetches.

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. Based on detailed simulations and a set of six scientific benchmarks executed under release consistency, we have studied the miss rates, the prefetch efficiency, the network traffic, and the effects on 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 results 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 1 block which means that sequential prefetching is as effective for most stride sequences, and (ii) sequential prefetching also exploits the locality of read misses for non-stride accesses. However, this study also shows that one of the most important design issues of a stride prefetching mechanism is the stride detection phase, since it affects the number of useless prefetches. Since the stride sequences of many applications are long enough for the stride detection overhead to be negligible, the detection phase can be optimized to keep the number of useless prefetches on a minimum. 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.

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 by the Swedish National Board for Industrial and Technical Development under contract number 9001797.

References

[1] J.-L. Baer and T.-F. Chen, “An Effective On-Chip Preloading Scheme To Reduce Data Access Penalty,” in *Proc. Supercomputing ‘91*, 1991, pp.176-186.

[2] R. Bianchini and T.J. LeBlanc, “A Preliminary Evaluation of Cache-Miss-Initiated Prefetching Techniques in Scalable Multiprocessors,” Tech. Rep. 515, Comput. Sci. Dept., University of Rochester, USA, May 1994.

[3] M. Brorsson, F. Dahlgren, H. Nilsson, and P. Stenström, “The CacheMire Test Bench — A Flexible and Effective Approach for Simulation of Multiprocessors,” in *Proc. 26th Ann. Sim. Symp.*, 1993,pp. 41-49.

[4] L.M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” in *IEEE Trans. Comput.*, vol. 27, pp. 1112-1118, Dec. 1978.

[5] T.-F. Chen and J.-L. Baer, “A Performance Study of Software and Hardware Data Prefetching Schemes,” in *Proc. 21st Int. Symp. Comput. Architecture*, 1994, pp.223-232.

[6] F. Dahlgren, M. Dubois, and P. Stenström, “Fixed and Adaptive Sequential Prefetching in Shared-Memory Multiprocessors,” in *Proc. Int. Conf. Parallel Processing*, Vol. I, 1993, pp. 56-63.

[7] F. Dahlgren, M. Dubois, and P. Stenström, “Combined Performance Gains of Simple Cache Protocol Extensions,” in *Proc. 21st Int. Symp. Comput. Architecture*, 1994, pp.187-197.

[8] F. Dahlgren and P. Stenström, “Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors,” in *Proc. First Int. Symp. High-Performance Comput. Architecture*, Jan. 1995.

[9] J. Fu and J.H. Patel, “Data Prefetching in Multiprocessor Vector Cache Memories,” in *Proc. 18th Int. Symp. Comput. Architecture*, 1991, pp.54-63.

[10] J. Fu, J.H. Patel, and B.L. Janssens, “Stride Directed Prefetching in Scalar Processors,” in *Proc. 25th Ann. Int. Symp. Microarchitecture*, 1992, pp.102-110.

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

[12] A. Gupta *et al.*, “Comparative Evaluation of Latency Reducing and Tolerating Techniques,” in *Proc. 18th Int. Symp. Comput. Architecture*, 1991, pp.254-263.

[13] E. Hagersten, “*Towards Scalable Cache Only Memory Architectures*,” PhD thesis, SICS Dissertation Series 08, Swedish Inst. of Computer Science, Oct. 1992.

[14] D. Kroft, “Lockup-free Instruction Fetch/Prefetch Cache Organization,” in *Proc. 8th Int. Symp. Comput. Architecture*, 1981, pp.81-87.

[15] R. Lee, P-C. Yew, and D. Lawrie, “Data Prefetching in Shared-Memory Multiprocessors,” in *Proc. Int. Conf. Parallel Processing*, 1987, pp. 28-31.

[16] T. Mowry and A. Gupta, “Tolerating Latency through Software-Controlled Prefetching in Scalable Shared-Memory Multiprocessors,” in *J. Parallel and Distrib. Computing*, Vol. 12, pp. 87-106, Jun. 1991.

[17] T. Mowry, “Tolerating Latency Through Software Controlled Data Prefetching,” PhD Thesis, Dept. of Comp. Sc., Stanford University, USA, March 1994.

[18] J.P. Singh, W.-D. Weber, and A. Gupta, “SPLASH: Stanford Parallel Applications for Shared-Memory,” in *Comput. Architecture News*, vol. 20, pp. 5-44, Mar. 1992.

[19] I. Sklenar, “Prefetch Unit for Vector Operations on Scalar Computers,” in *Comput. Architecture News*, vol. 20, pp. 31-37, Sep. 1992.

[20] A.J. Smith, “Sequential Program Prefetching in Memory Hierarchies,” in *IEEE Comput.*, Vol. 11, No. 12, pp.7-21, Dec. 1978.

[21] P. Stenström, “A Survey of Cache Coherence Scheme for Multiprocessors,” in *IEEE Comput.*, Vol. 23, No. 6, pp. 12-24, Jun. 1990.

[22] P. Stenström, F. Dahlgren, and L. Lundberg, “A Lockup-free Multiprocessor Cache Design,” in *Proc. Int. Conf. Parallel Processing*, Vol. I, 1991, pp. 246-250.