Effectiveness of Dynamic Prefetching in Multiple-WriterDistributed Virtual Shared Memory Systems

Magnus Karlsson and Per Stenström

Department of Computer Engineering, Chalmers University of Technology
SE-412 96 GÖTEBORG, Sweden
email: {karlsson,pers}@ce.chalmers.se

Abstract

We consider a network of workstations (NOW) organization consisting of bus-based multiprocessors interconnected by an ATM interconnect on which a shared-memory programming model is imposed by using a multiple-writer distributed virtual shared memory system. The latencies associated with bringing data into the local memory are a severe performance limitation of such systems.

To tolerate the access latencies, we propose a novel prefetch approach and show how it can be integrated into the software-based coherence layer of a multiple-writer protocol. This approach uses the access history of each page to guide which pages to prefetch. Based on detailed architectural simulations and seven scientific applications we find that our prefetch algorithm can remove a vast majority of the remote operations which improves the performance of all applications. We also find that the bandwidth provided by ATM switches available today is sufficient to accommodate prefetching. However, the protocol processing overhead of available ATM interfaces limits the gain of the prefetching algorithms.

Keywords: Shared-memory multiprocessors, Distributed virtual shared memory, Latency tolerance, Software-controlled prefetching, Performance evaluation.

Corresponding Author: Magnus Karlsson

1 Introduction

Shared memory multiprocessors like the Stanford DASH [18] and the MIT Alewife [1] offer high performance, but at the price of a high hardware complexity and production cost. One way to drastically cut the cost of large configurations is to use commodity computing platforms, such as bus-based multiprocessors, that are interconnected to form a *network of workstations* (NOW). Several research groups are currently studying performance and design issues for this style of building large-scale parallel computing platforms [2,7,8]. In this paper we consider a NOW organization built from clusters that are interconnected by ATM technology and where each cluster is a bus-based multiprocessor server.

In order to support a shared-memory programming model on top of the message-passing substrate that a NOW provides, a *distributed virtual shared memory* (DVSM) *system* is a particularly attractive solution in terms of cost-effectiveness. In a DVSM system, a shared-memory programming model is supported by replicating data at the granularity of page-sized chunks and consistency can then be maintained by invalidating remote page copies when a processor modifies its local copy. Unfortunately, owing to the fact that accesses by processors in different servers are usually co-located on the same page, efficiency is severely hampered by false sharing. However, if a relaxed memory consistency model is supported, this false sharing effect can be reduced by allowing several processors to modify different data on the same page as long as modifications are propagated at the point of the next synchronization or at some point later when data is requested. Such *multiple-writer DVSM systems*, exemplified by TreadMarks [16], are the focus of this paper.

While a preliminary performance evaluation of the NOW system above [15] indicated a decent performance level, the latency associated with the ATM interconnect was found to be one of the main limiting factors to high performance because it directly affects the penalty associated with access faults in a DVSM system. One popular way of coping with devastating latency components in any NOW system is to adopt a latency tolerating technique. In this paper we propose a novel prefetching scheme extension to a multiple-writer DVSM system.

To understand how our prefetch approach is different from others it is useful to classify existing prefetching techniques into *static* and *dynamic* techniques. Static techniques analyze the code at compile-time and insert prefetch instructions that bring data closer in the memory hierarchy in anticipation of a future access miss (fault). Mowry et al. [19] studied the effectiveness of such a static prefetching technique in the context of uniprocessors to cut replacement and cold cache misses and demonstrated that it can be very effective to attack misses caused by vector accesses in loops. Mowry also extended this static technique to multiprocessor computations [20] to also attack coherence misses by conservatively assuming that all shared data are invalidated after a synchronization. This can result in a high instruction overhead for shared data. By contrast, the approach taken by dynamic prefetch approaches is to predict which data to prefetch based on previous access history for the data at run-time. In stride prefetching [6], for example, when three accesses separated by the same stride have been detected, prefetching for consecutive accesses with the same stride is enabled. While the strength of this scheme is the absence of instruction overhead, it requires some non-trivial modifications to the processor and the caches. Also, the prefetch heuristic ought to be simple to be tractable.

Our proposed prefetch scheme, called *history prefetching*, is also dynamic in that it utilizes the access history of data. Instead of requiring hardware support, however, we integrate our prefetch algorithm in the software layer in a multiple-writer distributed virtual shared memory system; thus, the prefetch heuristic can be made more sophisticated than in hardware-based schemes. The algorithm assumes the existence of information regarding which pages that have been accessed locally between two synchronizations and which of those pages that were invalidated by remote clusters in the past. The history prefetching algorithm scans the access history for different access patterns. Noting that this is a general approach, we focus in this paper on its effectiveness for producer-consumer sharing only. When the algorithm finds a page with such an access pattern, it prefetches that page to avoid future faults on load accesses for consumers or store accesses for producers.

Based on detailed architectural simulations we study how history prefetching can increase the performance of seven application programs picked from the scientific domain. History prefetching is found to remove *all* load and store faults, after some initial misses for some applications. By contrast, for irregular algorithms it is less efficient owing to the fact that previous access patterns cannot guide which pages to prefetch. This motivates us to also adopt sequential prefetching [10,23] of consecutive pages that are not present in the local memory. While sequential prefetching alone is only effective when spatial locality is high, the combination of history and sequential prefetching improves performance for all applications.

Our simulations assume ATM technology that is available commercially. We find that while the bandwidth of the switch fabric is sufficient to accommodate the extra traffic caused by prefetching, the protocol processing taking place in commercially available ATM interfaces limits the performance gain of prefetching.

As for the rest of the paper, we start in the next section to conceptually describe what features in a multiple-writer protocol are utilized in order to implement history and sequential prefetching. Then in Section 3, to exemplify in detail how our prefetching schemes are successfully implemented in a real DVSM system, we focus on their integrations in the TreadMarks system; a state-of-the-art DVSM system. Section 4 deals with the experimental methodology, while Section 5 presents the experimental results. Related work is discussed in Section 6 and finally the paper is concluded in the last section.

2 Dynamic Prefetching Schemes in Multiple-Writer DVSM Systems

In order to understand what features are needed to support our proposed prefetching schemes, we provide in Section 2.1 the main characteristics of multiple-writer distributed virtual shared memory systems and the basic approach behind dynamic prefetching in this architectural framework. Then in Sections 2.2, 2.3 and 2.4, we describe the prefetch heuristics of history and sequential prefetching and their combination, respectively. These heuristics are later simulated in Section 5.

2.1 Multiple-Writer DVSM Systems and Prefetching Approach

We consider systems consisting of a number of clusters interconnected by a network. A DVSM system implements a shared-memory programming model on top of the message passing sub-

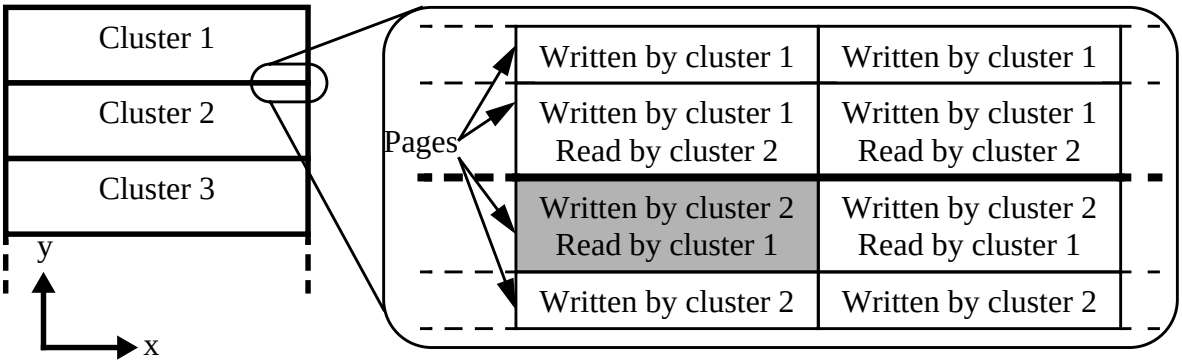

Figure 1: The partitioning of the data set in the S.O.R. application.

strate by replicating pages. To maintain consistency when a processor modifies data in a page, an exclusive copy is typically obtained by invalidating other copies of the page.

The basic mechanism that triggers coherence actions for a certain page is its two sets of access attributes (or states) maintained by the MMU: *Accessible* and *Invalid* (Inaccessible). For example, an invalid page that is accessed triggers an access fault that invokes the DVSM system software which brings a copy of the page from a remote cluster. Conversely, a page changes from accessible to invalid when an invalidation for the page is received. Clearly, access faults can be long-latency operations in that they can involve message exchanges across clusters which encounter the latency of the software system as well as the underlying interconnection network.

Traditional DVSM systems suffer from a high access fault rate because the fairly large pages are frequently accessed by multiple clusters and cause devastating false sharing effects. To reduce false sharing effects, multiple-writer protocols have been proposed [5]. Assuming that the program executes correctly under a relaxed memory consistency model [14], the main idea is to keep the page in the accessible state between two synchronization points by not propagating any invalidations until a synchronization point is reached. If a page has been modified by several clusters, invalidations from these clusters are received by the local cluster. If the local cluster subsequently accesses the page, it has to acquire the modifications from each and every one of the remote clusters that wrote to the page in order for the page to be accessible again. A system that uses this approach is TreadMarks which will be presented in more detail in Section 3.

Since accesses to invalid pages and invalidations incoming from other clusters are handled by the software layer implementing the multiple-writer protocol, one could log such events to predict which pages to prefetch. This log could then be scanned for certain access patterns; in our case we search for different producer-consumer access patterns that happened in the past and speculate that the same access patterns will be repeated in the future. An example of this is shown next.

S.O.R, which is one of the applications we evaluate, is an iterative algorithm that in every iteration performs a nearest-neighbor calculation on each element in a grid. The algorithm sweeps first along the x-axis and when it reaches the end of it, it starts to sweep along the x-axis of the next y-value. As shown in Figure 1, each cluster uses its own data set plus one row of each of its neighbor's data sets. The grid is stored in row-wise order. Only at the borders between the cluster's sets of elements will an element be read or written by more than one cluster. Every iteration starts with a barrier synchronization. We define an *interval* to be the time between two synchroni-

zations when at least one cluster wrote to the page. Consequently, in this example every iteration is one interval long.

At the barrier synchronization in the beginning of the second iteration cluster 1 will receive invalidations for all the pages it read from and that cluster 2 wrote to during iteration number one. When cluster 1 wants to read one of the invalidated pages, in the end of the second iteration, it has to send a message to cluster 2. By prefetching into the local memory of cluster 1 all pages modified by cluster 2 one could avoid the long latency of the subsequent access faults for these pages.

To guide which pages to prefetch, each cluster maintains the access history for all pages, called *page history.* This history is a list of entries indicating (1) if a page has been accessed locally and is logged when an access fault for the page occurs and (2) which clusters that wrote to a page during a certain interval, deduced from incoming invalidations. The page history is sorted in descending time order, i.e., the most recent entry appears first in the list. So at the barrier, in the S.O.R. application, during the start of the second iteration, cluster 1 would put two entries into the page history of the shaded page in Figure 1. One entry that cluster 2 wrote to the page during the last interval and one entry stating that cluster 1 read the page during the last interval. At the next barrier synchronization the same sequence of events is entered into the page history of the shaded page, because the same accesses occurred during the second iteration too. By looking at the page history, cluster 1 can detect that two consecutive intervals with the same access pattern have occurred. Now, cluster 1 can predict that this will be the case during this interval too, thus it sends out a prefetch for the page. So, when cluster 1 later accesses the page, it can potentially be in the accessible state already, thus having a potential to improve performance. This is the main principle on which history prefetching operates.

2.2 History Prefetching Heuristics

The history prefetching algorithm is invoked as a handler on one of the idle processors in a cluster after each synchronization point. The algorithm looks for different flavors of producer-consumer sharing and to specify these, we distinguish between the *Local* cluster being the cluster that initiates a prefetch and the *Remote* cluster being one or more of the other clusters in the system.

There are two types of prefetches, *read-prefetches* and *write-prefetches,* that are issued as a result of that the prefetching algorithm predicts that the page will be read (consumed) or written to (produced) by Local in the future, respectively. A read-prefetch is issued if Local accessed (that is read or wrote to) a page during the last two intervals, and Remote wrote to the page during those two intervals and the page is invalid right after a synchronization. This means that Local acts as a consumer and Remote as a producer. The write from Remote indicates that the page was invalidated at the start of each interval. On the top of the left column of Figure 2 this access pattern is displayed. Here, R.W means that at least one Remote cluster wrote to the page and R.A that at least one Remote cluster either read, wrote, or did nothing to the page. L.W stands for Local write, L.R for Local read and L.A for a Local access, either a read or a write. The dashed lines are the boundaries between the intervals. The prefetch decision is made in the right-most interval (interval *i*) of each column.

The algorithm will also, somewhat more speculatively, read-prefetch when Local read the page in the first preceding interval and accessed it in the second preceding interval, but Remote

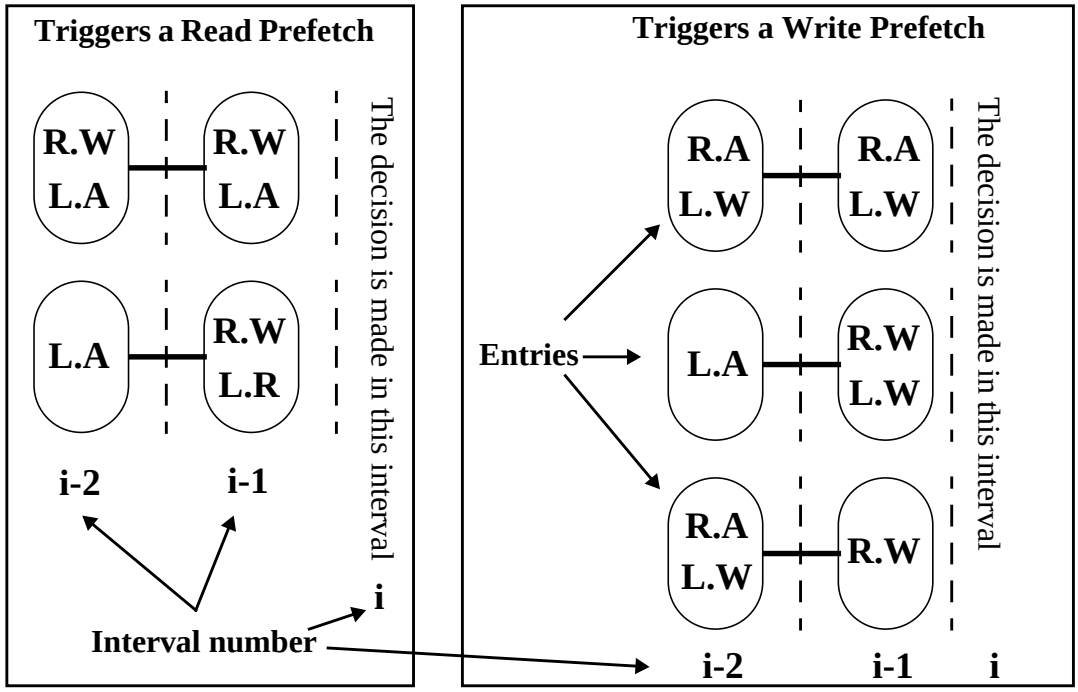

Figure 2: Access patterns that will trigger a prefetch. The dashed lines are interval boundaries. R.W means that at least one remote cluster wrote to the page and R.A that at least one remote cluster either read or wrote to the page. L.W stands for local write, L.R for local read and L.A for a local access, either a read or a write. Write-prefetches have precedence over read-prefetches.

wrote to the page only during the first preceding interval. The algorithm only needs to look at the last two intervals to decide if a page history will trigger a read-prefetch. More technically this is stated: *A read-prefetch will be triggered if the two most recent consecutive intervals in the page history contain a Local access and a Remote write each, or if the most recent interval contains a Remote write and a Local read followed by only a Local access in the second interval.*

Write-prefetches are issued by Local when the algorithm predicts that it is a producer of a page. Note that a write-prefetch has got precedence over a read-prefetch, because some of the patterns that trigger a write-prefetch are more specialized versions of read-prefetch patterns. As seen at the top of the right column of Figure 2, the first pattern that will trigger a write-prefetch is as follows: Remote either read or wrote to the page and Local wrote during the first preceding interval followed by the same behavior during the interval before that, i.e the access pattern during the first interval repeated in the next. The second pattern is if Remote and Local wrote to the page during the first preceding interval followed by a read or write access by Local in the second preceding interval. The third and final one is if only Remote wrote to the page during the first preceding interval and during the second preceding interval Local wrote but is necessarily not the only writer.

Regarding the performance improvements provided by history prefetching, we would expect that it is effective as long as the access patterns are repeated more than one interval. As will be shown in Section 5, iterative algorithms do extremely well. However, history prefetching can fail for irregular access patterns that are not repeated over a few intervals. In the next section we will study a previously proposed hardware-based scheme that we apply to our software-based multiple-writer protocol that turns out to be effective in this case.

2.3 Sequential Prefetching Heuristics

Large matrices are typically swept through once in direct methods. For example, consider multiplication of two matrices that each consists of several pages. Clearly, one could in this case benefit from prefetching consecutive pages on an access fault. This simple technique has been proposed in the context of hardware-based cache prefetch algorithms by Smith [23] and is known as *sequential prefetching*. We will simulate a variation of this simple scheme that on the first access to a page checks if any of the next N pages are invalid, where N is the degree of prefetching. If so, read-prefetches are sent for those pages.

2.4 Combining History and Sequential Prefetching

While history prefetching (HP) is expected to be effective in removing access faults for regular applications with producer-consumer sharing, sequential prefetching (SP) is expected to wipe out access faults in direct methods. Therefore, we have also considered a combination which works as follows. To successfully combine history prefetching with sequential prefetching (hereafter called the *combined scheme*), we need to introduce another status bit for each page. This bit, called the *prefetch-type-bit*, indicates which type of prefetching was used to bring the page to an accessible state. It is initialized to SP the first time a page is brought into memory. The prefetch-type-bit is set to HP when a producer-consumer access pattern is found in the page history for that page and thus the page is prefetched with the history prefetching scheme.

When a processor accesses a page with the prefetch-type-bit set to SP, sequential prefetching will be applied to the N following pages. This implies that sequential prefetching will be used on all pages successfully prefetched with sequential prefetching and for all the access faults, i.e. pages that were not predicted with either algorithm. A processor accessing a page with the prefetch-type-bit set to HP means that the page was successfully predicted with history prefetching, so it will not trigger any sequential prefetches as long as it is successfully predicted. This means that if the history algorithm is good at predicting the accesses, sequential prefetching will only be used for the first couple of accesses to the page. After that, history prefetching will take over. When history prefetching fails, i.e. when an access miss is encountered, sequential prefetching will be used for that page.

3 Implementation and Performance Case Study

We have integrated the prefetching schemes in Section 2 into a simulation model of a network of workstations that uses TreadMarks to support a shared-memory model. In Section 3.1 we give an overview of this system; our detailed technological assumptions are later described in Section 4. Then in Section 3.2, we explain how dynamic prefetching is integrated into TreadMarks.

3.1 Simulated System Environment

In order to understand the implications of integrating the prefetching schemes described in the previous section in a real system, we have considered the system depicted in Figure 3. It consists of a number of bus-based multiprocessor servers interconnected by an ATM switch. Each cluster contains a number of processors and memory modules and an I/O bus that is connected to an ATM interface. The intra-cluster coherence mechanism is a write-invalidate snooping cache

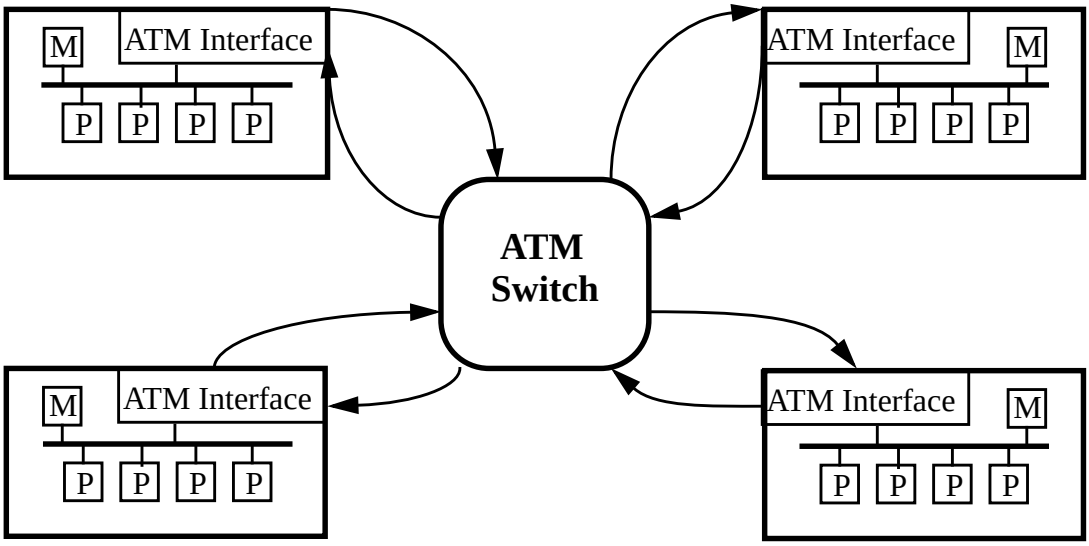

Figure 3: The base system of the study. An ATM switch connecting 4 multiprocessors/servers with 4 compute processors each. All communication from one cluster to another travels through the ATM interfaces of the communicating clusters.

coherence protocol, whereas inter-cluster coherence is maintained by TreadMarks. While the implementation details of TreadMarks can be found in [16], we will here focus on the general approach taken by TreadMarks to maintain consistency and in particular the data structures that TreadMarks maintains that we have used to implement dynamic prefetching.

TreadMarks is a multiple-writer protocol based on lazy release consistency (LRC). Under LRC, consistency-related information is only communicated by the releaser of a lock to an acquirer. The key data structure to carry out this task is a *write notice* which tells the acquirer which cluster has written to a certain page during a certain interval. To find out which modifications that the acquirer has not seen, a vector timestamp (containing one timestamp entry for each cluster) is associated with each interval. Each cluster maintains its local vector timestamp which is increased each time it enters a new interval e.g. enters or exits a critical section. By having the acquiring cluster send its timestamp to the current holder of the lock, the releaser can based on the difference between the two vector timestamps propagate a list consisting solely of the write-notices tagged with a timestamp greater than the one that the acquirer provided, i.e. only the write notices that the acquirer does not already have. A write-notice for a certain page will invalidate this page locally. It will then be recorded in a *write-notice list* associated with each page that is maintained in each cluster. The write-notice list can theoretically be infinite. Therefore, TreadMarks does garbage collection naturally at barrier synchronizations. Lacking a barrier it occasionally does a barrier synchronization to do garbage collection. Let us now consider the actions taken on access faults.

When an invalidated page is accessed, the local write-notice list for this page is used to make the page consistent by sending so called *diff requests* to all clusters that have modified the page. These clusters then return the diffs, i.e., encodings of the modifications to the page, that then are applied locally to make the page consistent before the state of the page is switched to read-only.

When a store access occurs to a page that is invalid, the page is first made read-only. Then in order to allow the local cluster to modify the page a copy is made, called a *twin*. A modification of

the local page copy creates a write notice as explained above and when another cluster requests a diff from this cluster, the diff is created by comparing the original page copy with the twin.

The coherence actions taken by TreadMarks of course impose execution overhead on the processor that invokes the handlers. These handlers are scheduled on the processors according to a round-robin algorithm.

3.2 Integrating Dynamic Prefetching into TreadMarks

All prefetching schemes in Section 2 are quite easy to integrate into the TreadMarks system based on the data structures that already exist. Recalling that a page history for each page is needed, where an entry is either an invalidation from outside or a local access (load or store) in a certain interval, we note that the write-notice list managed by TreadMarks maintains this information for invalidations and local stores. To support history prefetching we simply augment each write notice with a bit, called *used-bit*, that indicates whether the page has been accessed locally in this interval. This way, Local's reads are recorded in the page history. To be able to record an access to a prefetched page, it is read and write protected. This way a page fault will occur when it is accessed so that the used-bit can be set at that time. If the page is valid due to a prefetch but the used-bit is not set, i.e the page has not been used by Local, when the history prefetching algorithm scans this page's history; this means that the prefetch was not successful and another attempt will not be made until an access fault is encountered. At this time the used-bit is set to one. This way, the algorithm can adapt to changes in the sharing pattern of the pages. The history prefetching algorithm can scan the write-notice list and will detect an interval boundary by comparing the timestamps associated with two consecutive write notices. However, to avoid scanning the same set of entries twice, a bit called *already-scanned* associated with each page is set whenever it has been scanned. When a write-notice arrives for that page the already-scanned bit is reset to zero, indicating that these entries need to be scanned again. Finally, a write prefetch will trigger a twinning of that page and the garbage collector has to be slightly modified to avoid purging the write-notice list beyond the second interval down the line.

Sequential prefetching is trivially implemented by just examining the access attributes of the N pages that follow the page that triggered a load or store access fault. For any of the prefetching schemes that we try, a read-prefetch triggers requests for diffs just like a load access fault would do and a write-prefetch, in case of history prefetching, will also create a twin.

In some cases a write-notice will arrive to a page while a prefetch for the same page is pending. When the prefetch reply message arrives at the cluster, the already-scanned bit is checked for the page. If the bit is zero it means that another write-notice arrived for that page while the message was out in the network, so another message is sent to fetch the newly missing diff. When this one arrives, the page is set to accessible state, unless yet another write-notice arrived in the mean time.

4 Experimental Methodology

Herein, the simulation framework used to study the relative performance of the systems is presented. First in section 4.1 all the architectural parameters of the simulated systems are presented and in the last section the benchmark programs used are introduced.

4.1 Timing Model Parameters

The detailed simulation models are built on top of the CacheMire Test Bench [4]; a program driven functional simulator that potentially issues a memory reference from each simulated SPARC processor in each processor cycle. These references are then fed into the architectural model where they are delayed according to its timing model. This way the same interleaving is obtained in the simulator as in the real system. Next, we will discuss the assumptions for the timing model.

We simulate a system according to Figure 3 consisting of 4 clusters containing 4 processors each for a total of 16 processors. Each processor is a SPARC assumed to be clocked at 300 MHz with a peak performance of 1 instruction per cycle. In order to isolate the latency-tolerating capability of prefetching without taking interconnect bandwidth limitations into account, contention is not modeled in any part of the ATM interconnect by default. In Section 5.4, however, we will concentrate on how the bandwidth limitations of ATM affects the effectiveness of the prefetching algorithms. The minimum time through the ATM switch for a cell is 2.5 μs and it can hold a maximum of 256 cells. Through the ATM interface a message of one single cell (ATM fixed-size package) travels in 20 μs. It uses the AAL5 protocol for transmitting the ATM cells. A larger message takes the minimum time plus 1.25 μs for each extra cell generated. These values correspond to a commercially available 622 Mbit/s ATM interconnect.

Internally, the bus contention in each cluster is not modeled. Contention for the memories of each cluster is simulated. The caches are 16 Kbytes and direct mapped. The internal cache coherence protocol is a bus-based write invalidate protocol. Throughout the study the cache block size is set to 32 bytes and the page size is set to 1024 bytes. The default page size chosen is intentionally picked small to match the scaled down data sets we use for our applications. For example, S.O.R. would only generate 2 to 4 coherence actions per scan with a 4Kbyte page size, instead of 8 to 16 with 1Kbyte pages. In Section 5.4 we will look at how sensitive history prefetching is to variations in these parameters. The memory access time is 100 ns, the bus cycle time is 50 ns, and finally the cache access time is a single processor clock (pclock).

While we model the execution of the application code in detail, we do not simulate the actual execution of software handlers. Instead, our approach is to model the time they take by charging time for the handlers in TreadMarks in our timing model. Contention in the processors due to software handlers and local computations are simulated in detail. This is essential to factor in the overhead of the DVSM system that incorporates the prefetching algorithms. We have compiled the timing assumptions for the software handlers in Table 1. The latency charged to set up the ATM interface for sending a message is assumed to take 1000 pclocks. Most software coherence operations have a fixed cost that only depends on the page size. For example, to create a twin takes 1856 pclocks and the creation of a diff takes 5264 pclocks in the worst case that the whole page was modified. Creating a twin essentially results in that a consecutive region of memory corresponding to the page size is copied from one location to another. In each iteration, the algorithm loads a word from the source region and stores it into the destination region before a compare and a conditional branch instruction are executed. Thus, each iteration corresponds to four instructions. In the worst case scenario, the cache does not contain any date from the source region which means that every eight loads will miss in cache with our assumption that the block size is

eight words. Since the hit rate then is 87.5% and the miss penalty is 26 pclocks, the average execution time for a load in each iteration is 4.25 pclocks. Each iteration then takes 7.25 pclocks. Consequently, to create a twin then takes 7.25 x 256 = 1856 pclocks. The other time components our model assumes are calculated in a similar fashion.

On top of these costs context switch latencies are added and we assume that a page fault/context switch takes 2500 pclocks. So the creation of a twin will actually take 2500 + 1856 + 2500 = 6856 pclocks (a context switch + the twin creation time + another context switch). With our 300 MHz processors this takes 23 μs. Most of the other operations in Table 1 have an initial cost in the interval of 100 to 300 pclocks. On top of these latencies we charge a fixed cost of 40 pclocks for each incorporation of one write notice into the data structure. For each write notice sent at a barrier or a lock, an extra 20 pclocks are added to account for search and processing time. The latency parameters are consistent with those given for the TreadMarks system [16,8,11], when available.

Table 1: Software handler operations and their costs in pclocks. #WN stands for number of write notices. The diff sizes are all in 32-bit words.

Software handler Operation	Latency (pclocks)	Software Handler Operation	Latency (pclocks)
Context switch / Interrupt	2500	Communication Overhead	1000
Twin Creation	1856	Diff Creation	5264
Receive a Page	1056	Send a Page	1056
To apply a Diff (1)	10 * size of the diff	Request a Page	100
Incorporation of a Write Notice (2)	40	Search for a Write Notice (3)	20
Send Lock Acquire	300	Forward Lock Acquire	150
Receive Lock	100 + #WN*(2)	Release Lock	300 + #WN*(3)
Request Diffs	200	Send Diffs	150 + size of the diffs
Receive Diffs	100 + (1)	Barrier Entrance	200 + #WN*(3)
Barrier Release at Home	200 + #WN*(3)	Barrier Exit	250 + #WN*(2)

The history prefetching algorithm starts to search for patterns that will trigger a prefetch right after a synchronization. It starts at the top of the page array and traverses downwards. To go to the next write notice list and check if this list needs to be examined, i.e., check the already-scanned bit, takes 30 pclocks. If the write-notice list needs to be examined it takes an extra 50 pclocks. To send a prefetch takes the same time as the corresponding normal non-prefetch operation, plus the above extra. Our search algorithm is rather conservative in the way that it looks at the already-scanned bit of all the pages after every synchronization. A faster implementation, that we do not

consider, would be just to directly look at the entries associated with the pages that we just received write notices for. This would reduce the history prefetching overhead considerably. As will be seen, the overhead is small so this optimization does not seem justified.

4.2 Benchmark Programs

We have run seven scientific applications on the simulated system. They are all written in C using the ANL macros and compiled by gcc (version 2.1) with optimization level -O2. S.O.R., S.O.R.high and Shuffling FFT (sFFT) were taken from the CacheMire benchmark suite [12]. Quicksort (Qsort), TSP and a modified version of Water, has been provided to us by Rice University. Finally, LU was obtained from the SPLASH suite [22]. The applications can be characterized by being iterative algorithms, direct methods or others.

One of the iterative algorithms, S.O.R., makes computations on a grid with 1024 by 1024 elements. It is the same as the one discussed in Section 2.1. Statistics are gathered after the first iteration to reduce the start-up effects. Water is another iterative algorithm that simulates molecular dynamics and is originally obtained from the SPLASH suite. A modification was made to the program to reduce the number of lock accesses. It simulates 512 molecules for 2 time steps. Among the direct methods, sFFT Fourier transforms 65536 elements with the shuffling FFT method. Qsort sorts an array of 128K integers, using bubblesort to sort subarrays of less than 1/2K of elements. Finally, TSP uses a branch-and-bound algorithm to solve the travelling salesman problem for a 17-city tour. These applications were written with TreadMarks in mind and do not stress the system a lot, so we added two more applications that are not that nice. S.O.R.high is another parallelization of S.O.R which leads to a significantly higher communication-to-computation ratio. and, finally, LU decomposes a 500x500 sized dense matrix.

5 Experimental Results

In this section we study the effectiveness of the dynamic prefetching schemes. Section 5.1 considers history prefetching and Section 5.2 sequential prefetching while Section 5.3 deals with the performance of their combination. In Section 5.4 we will examine the effects of varying some of the key parameters in the system.

5.1 History Prefetching

In Figure 4 the execution time for the system with history prefetching (HIS) is shown relative to the execution time for the system with no prefetching (NO) for each application. To understand what components of the execution times are affected by prefetching, we have broken down each execution time bar into six components. The bottommost component is the busy time whereas all other components are overhead caused by (from bottom to top) locks, barriers, read and write stalls, and finally ‘remote’ is the fraction of the execution time spent executing the coherence handlers invoked by messages from other clusters that cannot be overlapped by other stall time components.

For S.O.R., TSP, Qsort and sFFT the speed-ups for the base system without any kind of prefetching are adequate. This can be seen from the busy time components. For example, TSP has a busy time component of almost 55% which means a speed-up of 16 x 0.55 = 8.8 compared to an

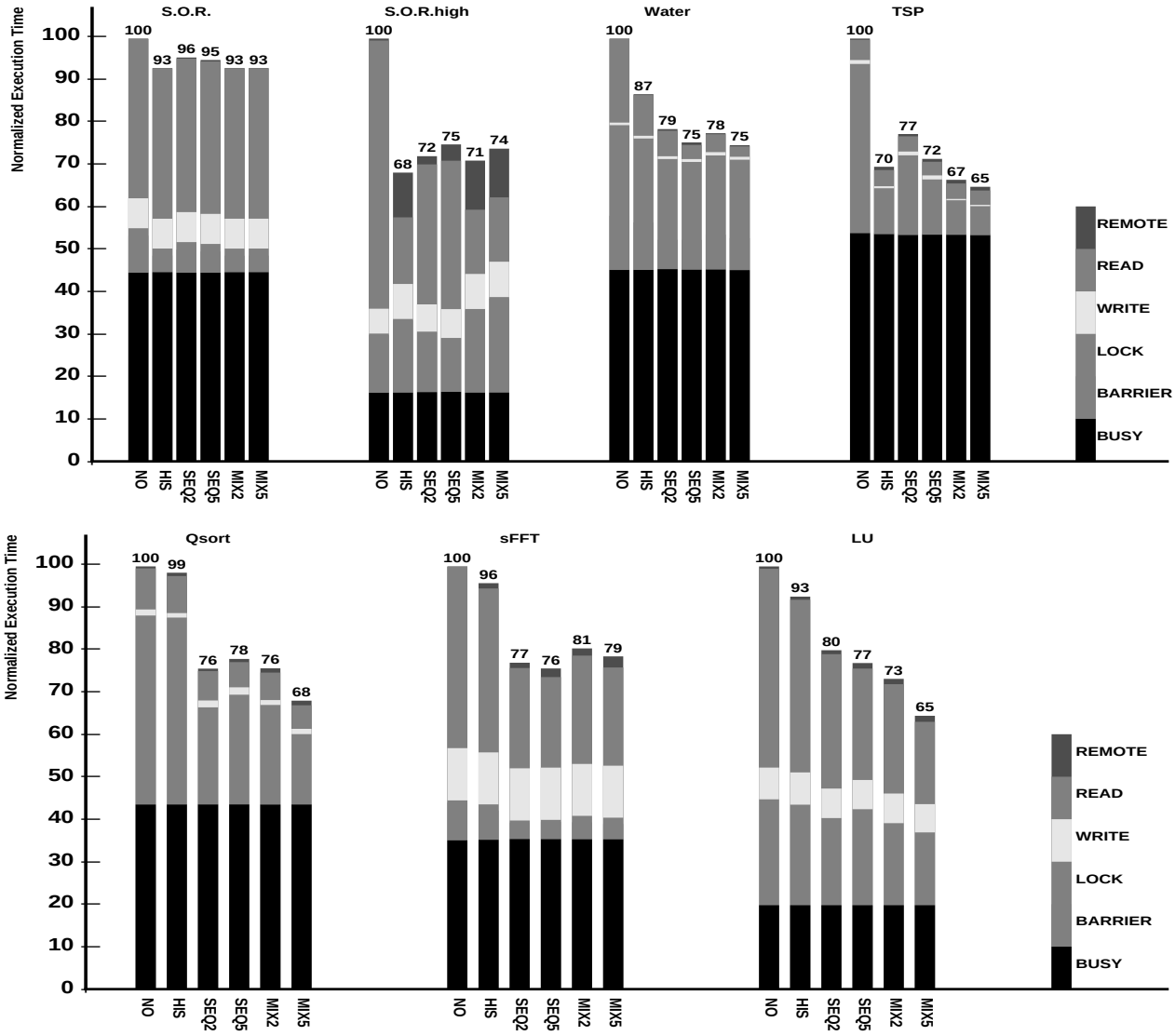

Figure 4: Execution times for the system with history prefetching (HIS), sequential prefetching (SEQN where N is the degree of prefetching) and for the combined prefetching scheme (MIXN) relative to the execution time for the system with no prefetching.

ideal sequential execution. By contrast for S.O.R.high and LU the speed-ups are low to very low. By introducing history prefetching into a DVSM system even these challenging applications have a potential to perform well, thus broadening the suitable application spectrum for DVSM systems.

The three leftmost applications are iterative in nature so history prefetching should be effective. As expected, the execution time is reduced by between 7% and 32%. By looking at how the stall time components are affected, we note that the most striking effect is that a major part of the read stall-time components are wiped out. In order to understand in more detail why history prefetching is so effective, we show in Figure 5 the *coverage* and in Table 2 the *efficiency* for history prefetching for each application. Coverage is defined as the percentage of all read and write requests that are eliminated by prefetching pages in advance. Since an issued prefetch may not always return the data on time, we also show the fraction of access faults that are eliminated by prefetching (Access Fault Coverage) meaning those prefetches that returned data before the page was accessed. Finally, efficiency is the fraction of prefetches that are useful meaning that they eliminate read and write requests.

Starting with S.O.R., history prefetching manages to eliminate all read and write requests although some prefetches do not return the data on time (the access fault coverage is 85%). Interestingly, history prefetching also lowers the barrier stall time. This is an effect of better load balancing because many of the long-latency operations, that are not uniformly distributed across clusters, have now been removed. The reason that the busy time is so low for S.O.R. is that two of the three cache blocks that are used in the computation are mapped to the same block in the cache. Therefore there will be two replacement misses every time one pixel in the array is calculated.

S.O.R.high does the same computation as S.O.R but with a different parallelization which results in a higher communication-to-computation ratio. Like S.O.R. history prefetching also helps this application to cut the read stall time but with a slightly lower coverage of 96% owing to the fact that some prefetches are predicted after the page is accessed (thus they are not issued at all). A remarkable observation is that the barrier, write, and remote stall times are increased which is explained by queueing delays to the software handlers as a result of the larger number of prefetch messages triggered by this application. We will get back to this effect when we later look at the traffic statistics.

In Water the global data structures are protected by locks and accesses to these are fairly regular. Therefore, Water also benefits from history prefetching; the execution time is shortened by 13%. However, the coverage is 38% and the efficiency is 52%. These are lower than S.O.R.'s due to some irregularities in the access patterns that cause useless prefetches and read and write requests that are not eliminated.

The last four applications are not iterative in nature and so history prefetching is not expected to be very effective. Interestingly, as Figure 4 reveals, history prefetching does fairly well for some of the applications. In TSP the accesses to most of the small global data structures are fairly regular. So history prefetching fares pretty well; a 30% decrease in execution time. The coverage and efficiency are fairly high; 67% and 72% respectively. By contrast, Qsort performs poorly. In Qsort, processors grab a task consisting of sorting an array of numbers of varying length and performs it and when it has completed it grabs another one from a global task queue. Since the assignments of tasks to processors are unpredictable, history prefetching is not effective. Therefore, the execution time of the history prefetching system is only decreased by 1% and the coverage is low.

Shuffling Fast Fourier Transform (sFFT) is a direct method. It contains only a few synchronizations. The calculation is divided into two stages. After the first one there is very little information in the page history. Only some of the pages can be prefetched for the next phase of the calculation. As can be seen in Figure 4, the execution time is only decreased by 4% due to this fact. The poor performance can also be seen from the coverage that is a puny 27%.

LU is also a direct method, like sFFT. The main difference is that LU contains significantly more synchronization operations which provides a longer page history and, as a result, a potential to more opportunities for history prefetching to make predictions. Unfortunately, the accesses are fairly irregular so the execution time is only reduced by 7%. The fact that the history prefetching algorithm cannot predict irregular access patterns can be deduced from the low coverage (31%) and the high efficiency (84%).

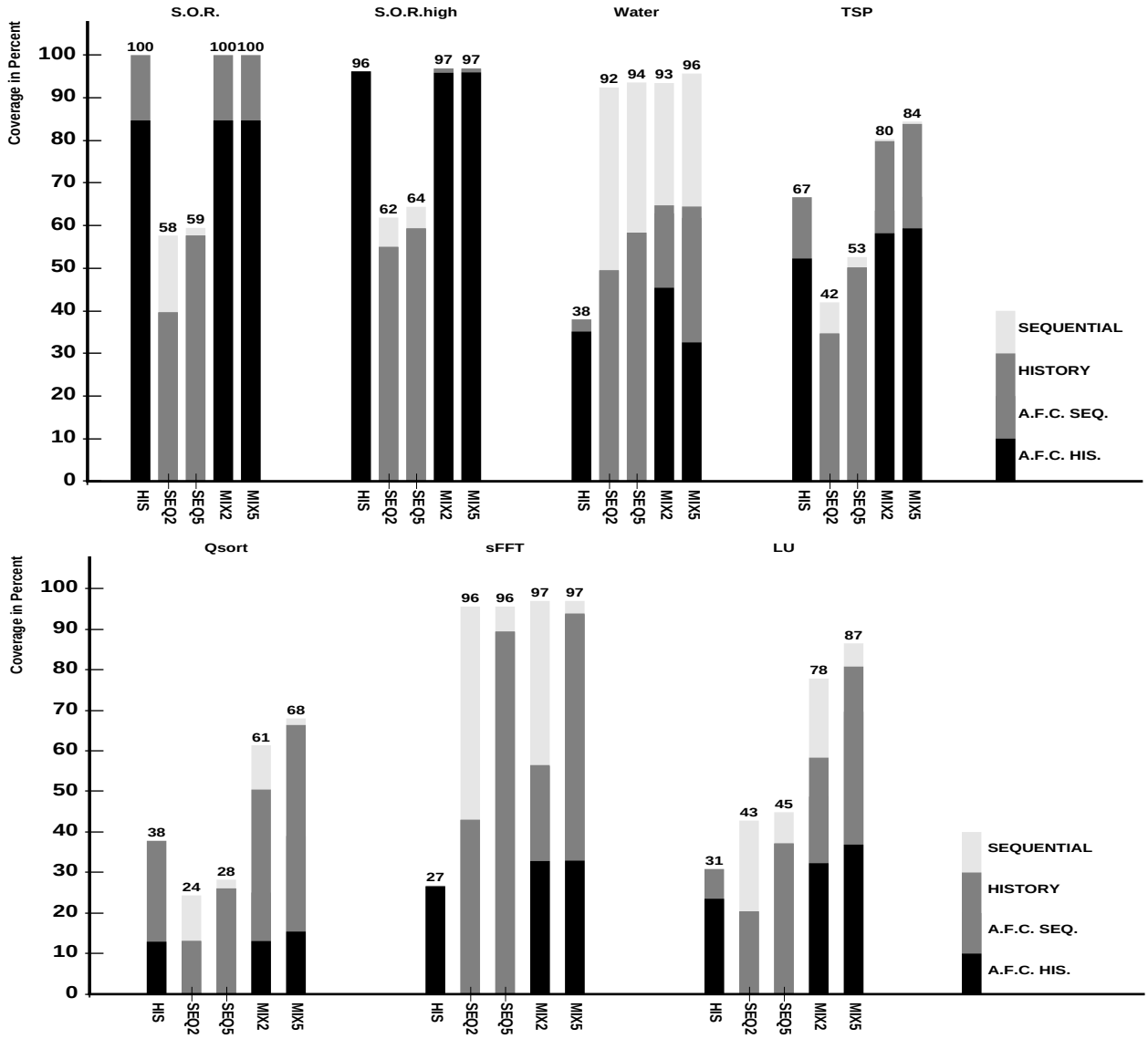

Figure 5: Coverages and access fault coverages for all prefetching schemes. The bottommost sections (the two bottommost sections for MIX2 and MIX5) is the fraction of access faults eliminated.

While our system uses ATM switches that provide a fairly high bandwidth (622 Mbits/s), it is nevertheless important to understand how much the traffic increases by prefetching. For the baseline system (NO) the bandwidth consumed by the applications was highest for S.O.R.high and never exceeded 45% of the ATM switch bandwidth. Let us now look at how the traffic is affected by history prefetching.

In Table 2 the network traffic caused by HIS relative to NO is shown for each application. For S.O.R. HIS generates 30% more network traffic than NO even though the coverage and efficiency are 100%. This is caused by an increase in write-notice traffic because a page was prefetched before the cluster that modified it was finished with its modifications. As a result, when a prefetch has switched a page into read-only mode, the same page will be written to which results in a write-notice to be sent for this interval as well. This same effect also explains why S.O.R.high yields 89% higher traffic and an increased write stall time as seen in Figure 4. Fortunately, this traffic increase seems to be consistent with the bandwidth provided by the ATM switch. Less than 45% of its bandwidth is occupied by S.O.R.high which is the most communication intensive

application. For Water, TSP, Qsort and LU history prefetching also causes higher traffic because of lower efficiency. However, these traffic increases are fairly moderate. For sFFT, on the other hand, the efficiency is fairly low (58%). This stems from the fact that after the second and last phase of the calculation there is enough information to predict all the future accesses, so prefetches are sent out for these pages but these prefetches will not be used.

Table 2: The efficiency for each prefetching scheme and application (in percent).

Application	HIS	SEQ2	SEQ5	MIX2	MIX5
S.O.R.	100	91	82	100	100
S.O.R.high	99	52	32	96	89
Water	52	79	73	48	48
TSP	72	57	56	65	64
Qsort	58	39	27	48	43
sFFT	58	78	76	66	65
LU	84	44	32	61	54

Table 3: Network traffic for each prefetching scheme relative to the system with no prefetching (in percent).

Application	HIS	SEQ2	SEQ5	MIX2	MIX5
S.O.R.	130	107	117	130	130
S.O.R.high	189	172	275	196	210
Water	142	118	153	133	153
TSP	104	115	121	118	120
Qsort	113	131	159	154	170
sFFT	119	102	106	120	123
LU	109	129	153	132	136

Overall, history prefetching is very effective for regular algorithms with producer-consumer sharing but also helps three out of the other four applications by eliminating some of the long-latency operations. Moreover, the extra bandwidth consumed by the prefetches is not very large compared to the bandwidth provided by ATM.

5.2 Sequential Prefetching

The third and fourth bars in Figure 4 show the execution times for sequential prefetching relative to the base system. SEQ2 has a degree of prefetching of two and SEQ5 has one of five. Starting with the two versions of S.O.R., which are iterative algorithms, sequential prefetching does not perform as well as history prefetching although it manages to cut the execution time in comparison with NO. The reason that sequential prefetching fares pretty well in this application is due to the fact that the effectively shared pages are aligned in a row on the grid boundaries. Figure 5 shows the coverages for the different prefetch schemes and demonstrates that the coverages for SEQ2 and SEQ5 are fairly high (between 58% and 59%). Also the efficiencies for SEQ2 and

SEQ5 under S.O.R are pretty high (between 91% and 82%), but for S.O.R.high the efficiencies for SEQ2 and SEQ5 are very low compared to HIS (52% and 32% compared to 99%). If we look at the traffic in Table 2 generated by the prefetch schemes for the two S.O.R. applications, SEQ2 and SEQ5 generate less traffic than HIS for S.O.R but more than NO because of less useless write-notice messages than under HIS. By contrast, for S.O.R.high the efficiencies for SEQ2 and SEQ5 are much lower than for HIS and results in a very high traffic level for SEQ5 (175% higher than under NO) which is expected to cause severe contention problems for ATM switches.

Continuing with Water which is also an iterative application, the pages of the main global data structure are accessed in a row-by-row manner. This accounts for the success of sequential prefetching for Water. The execution time drops by between 21% and 25% for sequential prefetching compared to 13% for history prefetching. In addition, the coverages and the efficiencies are much higher as can be seen in Figure 5 and Table 2 because history prefetching produces a lot of useless prefetches due to some irregularities in the application. The network traffic for SEQ2 is slightly lower while it is slightly higher for SEQ5. Despite Water, sequential prefetching is not as robust as history prefetching for regular algorithms.

For TSP sequential prefetching works fine, but it is not as good as history prefetching. The relative execution time is lowered by 23% and 28%. The global data set in TSP is small, so if a cluster misses once in the beginning of the data set, sequential prefetching pretty much prefetches a large part of the used global data set (at least for SEQ5). If a page later on is accessed in this interval, a performance improvement is gained. The coverage for SEQ2 and SEQ5 is moderate and between 42% and 53% while the efficiency is between 57% and 56%. Only 15% and 21% higher network traffic is produced, but it is higher than for HIS.

In Qsort, we recall history prefetching has a problem because it is difficult to predict what will be accessed next in Qsort since processors get tasks (sorts or merges) from a global task queue. However, looking within each individual task, a vector is typically accessed sequentially giving SEQ2 and SEQ5 a potential to perform well. As expected, the execution time is shortened by between 24% and 22% for SEQ2 and SEQ5, compared to a 1% decrease for HIS. The cost is an increase in network traffic; 31% and 59% compared to 13% for HIS.

sFFT, which is a direct method, is expected to run better under sequential prefetching than under history prefetching. This can be also seen in Figure 4. The performance is improved by 23% and 24% for SEQ2 and SEQ5; a lot compared to 4% for HIS. The coverage is at an impressive level of 96% percent for both of them and an efficiency of 78% and 76%, respectively. The network traffic is only increased by 2% and 6% as compared to 19% for HIS, although this traffic should be lower if history prefetching was switched off by the end of the execution.

Finally, LU is also a direct method but with less regularities. Sequential prefetching manages to squeeze the execution time down under history prefetching. It is decreased by 20% and 23% compared to 7% for HIS. The bad thing is that the network traffic soars to new heights and increases by between 29% and 53%. This is mainly due to the poor efficiency of sequential prefetching which is between 32% and 44%.

Overall, while history prefetching is advantageous for applications with access regularities that are repeated, sequential prefetching does better for direct methods with less regular behavior. Next, we look at whether a combination of them can yield additive gains.

5.3 Combining History Prefetching with Sequential Prefetching

Figure 4 shows the execution times for all prefetching schemes including the combined history and sequential prefetching schemes of Section 2.4. MIXN stands for the combined scheme with a degree of prefetching equal to N. Starting with S.O.R. and S.O.R.high the combination of both protocols perform as well as HIS. This is because history prefetching takes over after two iterations and switches off sequential prefetching. In S.O.R.high, some of the pages are predicted too late. These pages will trigger some useless, or mostly useless, sequential prefetches that just increases the network traffic some, as can be seen from the traffic statistics in Table 2.

For Water, the combined scheme performs somewhat better than both history prefetching and sequential prefetching on their own. Looking at Figure 4, execution time is shortened by between 22% and 25% compared to NO. Figure 5 shows the coverages of the applications. Each bar is decomposed into two sections: The ratio of prefetches that were covered fully by history and sequential prefetching, denoted access fault coverage (A.F.C.), and the ratio of prefetches that were covered fully or partly by each algorithm, plainly denoted coverage in the figures. Comparing the coverage of HIS with the coverage of MIX2, we see that the coverage for MIX2 is higher than for HIS. This is due to the fact that the sequential prefetching done in MIX2 produces useful prefetches that does not take place in HIS.

The successes/failures of these prefetches are also recorded in the page history. This fact gives the history prefetching in MIX more information to make a prediction. So for some pages where it was impossible for HIS to make a prediction because there was not enough information can now be predicted with history prefetching. The higher the degree of prefetching, the higher is the sequential part of the coverage. For MIX2 the network traffic is 33% higher than for NO, and for MIX5 it is 53% higher.

For TSP and Qsort we have made the same observation. MIX2 and MIX5 improve performance more than history and sequential prefetching do on their own and the network traffic is somewhere in between HIS and SEQ; sequential prefetching often produces the most traffic. The coverage increases due to the effects discussed in the previous paragraph and the efficiency lands somewhere between HIS and SEQ, where history prefetching is always more effective.

sFFT did not have enough accesses and synchronizations for history prefetching to be effective. Here most of the prefetches are sequential prefetches and the performance is decreased slightly going from SEQ2 and SEQ5 to MIX2 and MIX5 due to mispredictions from the history prefetching algorithm.

With LU the combined prefetching scheme dramatically outperforms the other schemes on their own. The execution time is 35% lower under MIX5 than NO; SEQ5 does not cut the execution time that much. The coverage is also much higher for the combined protocol (between 78% and 87%); a good example of fruitful cooperation between sequential and history prefetching.

The network traffic is around 34% over that for NO. This is well below the value for SEQ5 but above the values for history prefetching.

In summary, the combination of history and sequential prefetching always outperforms or equals the performance of the prefetching schemes on their own. The coverage of the combined scheme is always higher or equal to the coverage for history prefetching on its own. The traffic of sequential prefetching is higher than the traffic for the combined scheme for five of the seven applications. Another desirable property of all prefetching schemes is that they impose very little overhead as can be seen from Figure 4. The only case where the remote overhead is significant is for history and the combined scheme under S.O.R.high where it accounts for around 15% of the execution time; for the other applications, the remote overhead is negligible.

5.4 Variation Analysis

In this section we will briefly examine the effects of varying some of the key parameters of this architecture, e.g. ATM interconnect contention, page size and bus contention. The one parameter with the most pronounced effect on prefetching is the interconnect technology used. So we will start by analyzing the effects of this.

The assumptions for the ATM interconnect in this study is based on off-the-shelf switches and interfaces. We have shown in the previous sections that the prefetching algorithms are capable of tolerating a substantial part of the long latencies incurred by the interface and the switch. In addition, we have also shown that the bandwidth provided by the switch fabric of ATM technology available today (622 Mbit/s) seems sufficient for the extra bandwidth consumed by the prefetching algorithms. However, the substantial protocol processing taking place in the ATM interface itself might severely hamper the effectiveness of the latency-hiding capability of prefetching. To study such contention effects in detail, we have also modeled the queuing delays in the ATM interface and the switch fabrics. The simulated switch fabric is a shared-buffer implementation containing 256 entries.

When we encountered the contention effects of the interface, we found that the gains of the prefetching algorithms were smaller or reversed by the longer latencies incurred by the queing delays in the interface. This can be seen by looking at the leftmost two bars (NO and MIX5) for each application in Figure 6. Interestingly, we found that the switch fabric itself did not experience any cell losses.

One way to reduce the devastating contention effects in the interface would be to aggregate a number of coherence messages (spanning a size of between 1 to 22 ATM cells) into one single message. This way the protocol processing overhead of sending a single coherence message will be amortized over each aggregated message. In Figure 6 this scheme is denoted AGG and it aggregates 10 messages at a time. Overall, we see that the execution times of all applications, except S.O.R.high, are reduced by the combined prefetching algorithm with aggregation (AGG) although the reduction is not as high as when contention in the ATM interconnect was not encountered. The AGG scheme performed better than both sequential prefetching for all the applications, so they are not shown in the figure.

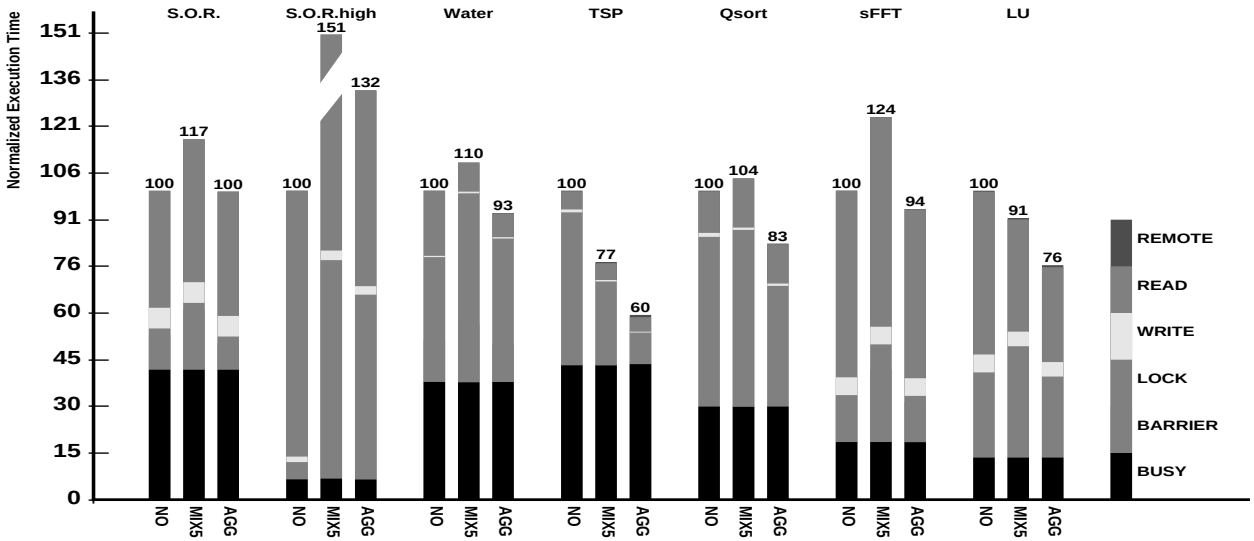

Figure 6: Execution times for the system with ATM contention switched on. The aggregation number is 10 for AGG.

For S.O.R.high the execution time is actually increased when prefetching over an ATM interconnect is used. As we saw in Section 5.3, S.O.R.high has a very high bandwidth demand and when this can be satisfied history prefetching is also effective for this application. But the ATM interfaces cannot accommodate this even though the ATM switches can, and therefore the execution time is actually increased. Another reason for the poor improvement of S.O.R.high is due to the fact that all the processors in each cluster have to wait for one of the coherence messages that is aggregated into the first ATM message in each iteration. There are two ways of alleviating this problem. First, one could introduce a priority scheme that serves vital messages first, and second, one could allow only a predefined number of pending prefetches in the system.

Now, let us concentrate on the observations regarding the switch fabric itself. As we would expect, virtually no cell losses were recorded under any prefetching algorithm except for history prefetching (HIS) alone. Under HIS, three out of the seven applications experienced cell losses: S.O.R.high, Water, and sFFT. The fraction of cells that were lost are 6.5%, 0.7%, and 1%, respectively. We tracked these cell losses to the bursts of prefetches after each barrier synchronization. While the retransmission of messages caused by the cell losses for the latter two applications results in acceptable execution time overhead, the high cell loss rate for S.O.R.high is not acceptable. However, this application has been used to stress the system; a more natural parallelization of this kernel is the S.O.R. which did not result in any cell losses. Overall, except for S.O.R.high, the ATM switch fabric can accommodate the traffic caused by the prefetching algorithms.

Another important analysis is how history prefetching reacts to different page sizes. We used a small page size because our data sets were small. To see if that decision affects our conclusions we scale the data set by 4 and increase the page size to 4 Kbytes. When history prefetching is turned on it turned out that it decreased the read and write stalls as much as earlier (in percent, see Table 4), but the overall execution time reduction will be less due to the higher busy time. S.O.R., S.O.R.high and Qsort were not included because they yielded exactly the same coherence traffic as before. LU was not simulated due to simulation constraints and the data set of TSP was not increased due to the same reason.

The last factor we are going to analyze is the bus traffic to see if our not to simulate bus contention. The results of these simulations are presented in Table 5. For sFFT, S.O.R. and LU the increase in bus traffic was less than 7% and therefore negligible. For TSP, Qsort and Water the increase in bus traffic is 75%, 47% and 13%, respectively. But as the bus traffic is low for all these applications this extra traffic can easily be accommodated by a modern bus, e.g. the S.G.I Challenge POWERpath-2 (< 1.2 GBps) [13]. This is not the case for S.O.R.high, where the bus traffic is increased by 79% to around 745 Mbytes/second which would degrade performance even further for this application.

Table 4: The results of the simulations with a page size of 4 Kbytes.

Application	Read Stall Reduction	Write Stall Reduction	Execution Time Reduction
Water	32%	0%	8%
TSP	0%	57%	22%
sFFT	10%	0%	1%

Overall, the interconnect used is a crucial factor if history prefetching should be effective even for high bandwidth applications. The extra bus traffic generated by history prefetching can easily be accommodated by a contemporary bus for six out of the seven applications.

Table 5: Bus traffic measurements for the seven applications in Mbytes/second.

Application	Bus Traffic (NO)	Bus Traffic (HIS)	Increase
S.O.R.	560	560	0%
S.O.R.high	416	745	79%
Water	24	27	13%
TSP	5	9	75%
Qsort	12	17	47%
sFFT	220	234	6%
LU	189	202	7%

6 Discussion and Related Work

Looking back on previous events in the system, as history prefetching does, is not that new. There are other more sophisticated versions of sequential prefetching that do this, for example hardware-based adaptive sequential prefetching [10] and stride prefetching [6] for tightly coupled systems. The former adapts the degree of prefetching for each page/block so that a near optimum number is attained. It still needs a high spatial locality in the application to be effective and it does not eliminate the initial miss, like history prefetching is able to do. Stride prefetching, on the other hand, does not rely on a high spatial locality. It detects strides in the accesses and can thus prefetch every Nth page/block. This works fine as long as the accesses are evenly spaced. Though, it has been shown that strides are often short and sequential prefetching can exploit this fact efficiently [9]. What we think is new, however, is that the prefetching schemes in this paper are sup-

ported in software and can thus be much more sophisticated than the above discussed hardware schemes.

There are also static software-controlled prefetching schemes with hand-inserted prefetch instructions [21]. The advantage of these schemes is that they can handle very irregular applications with success. The main disadvantage is that the programmer has to understand the code to a higher degree than one would need to without static software prefetching. With history prefetching no knowledge of the application is necessary. If the programmer would like to put this effort into the program, static software-controlled prefetching would be a good complement to history prefetching, taking care of the irregular access patterns that can be found in the source code.

A different approach than ours was tried out by Bianchini et al in [3]. Their approach is also based on the previous history of TreadMarks, but it is more optimistic than our scheme because as soon as a page is invalidated (that the node was using) it will fetch that page again. This will probably create more useless prefetches than our scheme thus degrading performance. Only two out of six applications benefited from their scheme. A comparison is hard to make because they are using uniprocessors as nodes while we used SMPs.

7 Conclusions

The high latencies of today's networks of workstations have motivated us to study the efficiency of one latency tolerating technique, namely prefetching. The previous dynamic prefetching schemes have been designed with mainly hardware shared memory machines in mind. In a distributed virtual shared memory system more advanced schemes can be implemented, due to the fact that it uses a software layer to support a shared-memory model. We have proposed a new prefetching algorithm called history prefetching that is specifically designed for DVSM systems. It records the accesses and synchronizations of the system and uses these to predict future access. History prefetching is best suited for regular applications.

Through detailed simulations of a system encompassing sixteen processors distributed across four clusters and driven by seven applications, we have found that history prefetching works really well for iterative and regular applications. Coverages and efficiencies up to one hundred percent are reported. For the direct methods and the irregular algorithms sequential prefetching works better. But by combining the two prefetching approaches a new prefetching scheme is proposed that works better than any of the two schemes on their own. Basically, if an access fault is encountered sequential prefetching is used and if a page was successfully predicted with history prefetching it will not trigger any further sequential prefetches. Execution time decreases up to 35 percent are reported. The network traffic was reduced or at a comparable level compared to sequential prefetching for five of the seven applications. The coverages range from 61 to 100 percent and the efficiencies range from 43 to 100 percent for the combined scheme.

While the prefetching algorithms proposed in this paper can be applied to a wide spectrum of DVSM systems and hardware platforms, we specifically considered a network of workstation system using ATM switches as an interconnect. Based on our simulations we have found that the bandwidth provided by off-the-shelf ATM switches is sufficient to accommodate the extra bandwidth needed by prefetching. However, the protocol processing in the ATM interfaces available

today causes severe contention effects that limit the gains of our prefetching algorithms. Therefore, to make use of similar latency-tolerating techniques calls for more efficient ATM interfaces.

Overall, history prefetching combined with sequential prefetching seems to be a promising way to hide some of the latencies on a DVSM system at the cost of a slight increase in network traffic.

Acknowledgments

The authors are indebted to Mats Brorsson of Lund University and Jan Jonsson of Chalmers University of Technology for their suggestions and comments and to Robert Fowler of Rice University that provided us with many of the applications. This research has been funded in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under project numbers P2363-1, P855 and by the Swedish Research Council on Engineering Science (TFR) under contract number 94-315.

References

[1] Agarwal, A., Chaiken, D., Johnson, K., et al., "The MIT Alewife machine: A large-scale distributed-memory multiprocessor," in: Dubois, M. and Thakkar, S.S., eds., Scalable Shared Memory Multiprocessors (Kluwer Academic Publishers, Boston, MA 1990), pp. 240-261.

[2] Anderson, T., Culler, D., Patterson, D., "A Case for NOW (Networks of Workstations)," *IEEE Micro.* Vol. 15, No. 1, Feb. 1995, pp. 54-64.

[3] Bianchini, R., Kontothanasis, L. I., Pinto, R., De Maria, M., Abud, M., Amorim, C. L., "Hiding Communication Latency and Coherence Overhead in Software DSMs," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 1996, pp. 198-209.

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

[5] Carter, J. B., Bennett, J. K. and Zwaenepoel, W., "Implementation and Performance of Munin," *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991, pp. 152-164.

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

[7] Chiou, D., Ang, B. S., Arvind, Beckerle, M. J., Boughton, A., Greiner, R., Hicks, J. E. and Hoe, J. C., "START-NG: Delivering Seamless Parallel Computing", *Proceedings of EURO-PAR'95*, Lecture Notes in Computer Science, No. 966, Springer-Verlag, Berlin, August 1995, pp. 101-116.

[8] Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R. and Zwaenepoel, W., "Software Versus Hardware Shared-Memory Implementation: A Case Study," *Proceedings of the 21st International Symposium on Computer Architecture*, April 1994, pp. 106-117.

[9] Dahlgren, F. and Stenström, P., "Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared Memory Multiprocessors," *Proceedings of the First International Symposium on High Performance Computer Architecture (HPCA)*, January 1995, pp. 68-77.

[10] Dahlgren, F., Dubois, M. and Stenström, P., "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 7, July 1995, pp. 733-746.

[11] Dwarkadas, S., Keleher, P., Cox, A. L. and Zwaenepoel, W., "Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology," *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 144-155.

[12] Ekstrand, M., "CaBS — The CacheMire Benchmark Suite," *M.Sc. Thesis*, Department of Computer Engineering, Lund University, March 1993.

[13] Galles, M., Williams, E., "Performance optimizations, Implementaion, and Verification of the SGI Challenge Multiprocessor," *Proceedings of the 27th Hawaii International Conference on System Sciences*, Vol. 1, 1994, pp. 134-143.

[14] Gharachorloo, D., Lenoski, J., Laudon, P., et al, "Memory Consistency and Event Ordering in Scalable Shared Memory Multiprocessors," *Proceedings of the 17th International Symposium on Computer Architecture*, May 1990, pp. 15-26.

[15] Karlsson, M. and Stenström, P., "Performance Evaluation of a Cluster-Based Multiprocessor Built from ATM Switches and Bus-Based Multiprocessor Servers," *Proceedings of the Second International Symposium on High Performance Computer Architecture (HPCA2)*, February 1996, pp. 4-13.

[16] Keleher, P., Cox, A.L., Dwarkadas, S. and Zwaenepoel, W., "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems," *Proceedings of the 1994 Winter USENIX Conference*, January 1994, pp. 115-132.

[17] Keleher, P., Cox, A.L. and Zwaenepoel, W., "Lazy Release Consistency for Software Distributed Shared Memory," *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 13-21.

[18] Lenoski, D., Laudon, J., et al., "The Stanford DASH Multiprocessor," *IEEE Computer*, Vol. 25, No.3, March 1992, pp. 63-79.

[19] Mowry, T., Lam, M. and Gupta, A., "Design and Evaluation of a Compiler Algorithm for Prefetching," *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, October 1992, pp. 62-73.

[20] Mowry, T., "Tolerating Latency through Software-Controlled Data Prefetching," Ph.D. Thesis, Stanford University, March 1994.

[21] Mowry, T. and Gupta, A., "Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, Vol. 12, No. 2, June 1991, pp. 43-93.

[22] Singh, J. P., Weber, W-D. and Gupta, A., "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, Vol. 20, No. 1, March 1992, pp. 5-44.

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