Relative Performance of Hardware and Software-Only Directory Protocols Under Latency Tolerating and Reducing Techniques

Håkan Grahn and Per Stenström

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

August, 1995

Abstract

In both hardware-only and software-only directory protocols, i.e., protocols where the directory is managed by software handlers invoked on the compute processor, the performance is often limited by memory access stall times. To increase the performance, several latency tolerating and reducing techniques have been proposed and shown effective in the context of hardware-only directory protocols. However, in the context of software-only directory protocols, the efficiency of a technique depends not only on how effective it is as seen by the local processor, but also on how it impacts the software handler execution overhead in the node where a memory block is allocated.

Based on architectural simulations and case studies of three latency tolerating and reducing techniques, we find that prefetching can degrade the performance of software-only directory protocols due to too many useless prefetches, i.e., the software handler overhead increases more than the read stall time is reduced. Further, even though a relaxed memory consistency model hides all write latency also for software-only directory protocols, the software handler overhead is virtually unaffected and now constitutes a larger portion of the total execution time. Overall, latency tolerating techniques for software-only directory protocols must be chosen with more care than for hardware-only directory protocols.

1 Introduction

Private caches and a directory-based cache coherence protocol implemented in hardware, also called a *hardware-only directory protocol*, constitute an important approach to achieve high performance in shared-memory multiprocessors. However, during the last few years several other approaches have been proposed in order to reduce the hardware complexity and/or increase the protocol flexibility by migrating parts of the coherence protocol to software [1, 5, 7, 13, 17, 18, 24]. In so called *software-only directory protocols* [5], the management of the directory is migrated from a hard-wired memory controller to software handlers executed on the compute processor. As a result, the hardware complexity can be reduced at the expense of slightly lower performance; between 60% and 86% of the hardware-only protocol performance has been reported in [13].

In both hardware-only and software-only directory protocols, performance is often limited by processor stall times resulting from *memory access latencies*. To reduce the processor stall times, and thus increase the performance, several *latency tolerating and reducing techniques* have been

proposed and evaluated in the context of hardware-only directory protocols [8, 10, 12, 16, 22, 26]. In addition to the processor stall times, the invocation of software handlers on the compute processor in software-only directory protocols can prolong the execution time in two ways. First, the handler latency might end up on the memory access path and thus delay the memory access latency seen by the requesting processor. In [13, 14], we propose strategies and techniques to remove or hide the handler latency from the memory access path for read misses. Second, the handler latency also burdens the compute processor in the home node, i.e., the node where a memory block is allocated. Unfortunately, the latter protocol execution overhead, referred to as *p-time*, was found to have very limited possibilities to be overlapped by other processor stall times [13]. Therefore, p-time has a fundamental, and possibly large, impact on the effectiveness of latency tolerating and reducing techniques in software-only directory protocols.

To illustrate the problem let us consider prefetching, a latency tolerating technique shown to be effective in hardware-only directory protocols [9, 22]. By using special prefetch requests, data is brought into the cache prior to its use, thus reducing the read stall time as well as the number of global read miss requests. However, some prefetch requests are useless because the prediction of which block to prefetch is not perfect or a prefetched block might be evicted from the cache before the processor accesses it. In software-only directory protocols this can be a severe problem since each useless prefetch *increases* p-time without reducing the read stall time. As a result, a high number of useless prefetches might even increase p-time so much that the read stall time reduction is outweighed. By contrast, consider increasing the block size instead. A larger block size can potentially reduce both the number of cache misses and the read stall time. In contrast to prefetching, the number of coherence interrupts is also reduced which results in a *reduced* p-time.

As we have discussed, the effectiveness of latency tolerating and reducing techniques depends on how such techniques impact p-time. They can be divided into three classes depending on whether p-time *increases*, *decreases*, or is *invariant* when the technique is applied. First, techniques that increase p-time are expected to be relatively more efficient in hardware-only than in software-only directory protocols. Second, we expect the performance difference between hardware-only and software-only directory protocols to decrease when techniques that reduce p-time are used. Finally, for techniques where p-time is invariant, p-time will contribute relatively more to the total execution time given that the stall times are equally reduced in both hardware-only and software-only directory protocols. Therefore, we expect the performance difference between hardware-only and software-only directory protocols to increase for such techniques.

In this paper we study these effects by considering three latency tolerating and reducing techniques and compare the performance of hardware-only and software-only directory protocols under each of these techniques. We have chosen hardware-based *adaptive sequential prefetching* [10], *migratory optimization* [8, 26], a technique that dynamically detects migratory data blocks

and optimizes their coherence protocol actions, and *release consistency* [12] as example techniques that increase, reduce, and do not affect p-time, respectively.

To evaluate the relative performance between hardware-only and software-only directory protocols for each of the three techniques we use architectural simulations of a detailed multiprocessor model and four applications from the SPLASH benchmark suite [25]. We find that adaptive sequential prefetching, a technique that increases p-time, often actually *degrades* the performance of software-only directory protocols due to a large p-time overhead. By contrast, adaptive sequential prefetching increases the performance for hardware-only directory protocols. Further, we find that the migratory optimization technique, which reduces p-time, has a potential to boost the relative performance of software-only directory protocols as compared to hardware-only directory protocols. Finally, release consistency, which does virtually not affect p-time, usually increases the relative performance difference between software-only and hardware-only directory protocols.

We begin in the next section to describe our baseline architecture. Based on this architecture, we discuss the expected performance effects of latency tolerating and reducing techniques in Section 3. Then, in Section 4 we present the experimental environment that our results in Section 5 are derived from. Finally, in Section 6 we discuss and generalize our findings before we conclude the study in Section 7.

2 The Baseline Architecture and the Coherence Protocol

In this section we describe our baseline architecture by only providing enough information to understand the performance effects of the latency tolerating and reducing techniques we discuss in the next section. Later, in Section 4, we present our architectural assumptions when we describe the simulation methodology.

Our baseline architecture is a sequentially consistent cache-coherent NUMA architecture where a number of processor nodes are connected by a network. Each node consists of a processor with its cache hierarchy, a memory module, a local bus, and a network interface connecting the processor node to the network as shown in Figure 1.

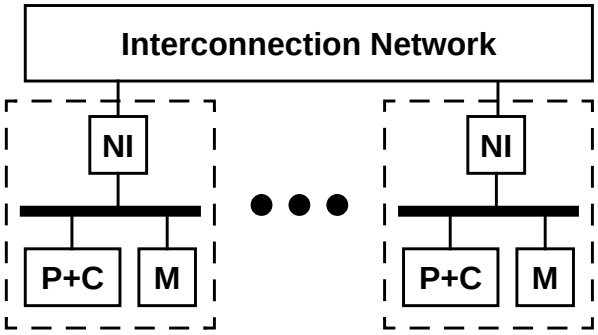

Figure 1: An overview of the assumed CC-NUMA architecture.

To understand the effects latency tolerating and reducing techniques have on the execution time, a basic understanding of how the cache coherence protocol works is essential. The cache coherence protocol is a write-invalidate protocol with a full-map directory [4], i.e., for each memory block a presence flag vector is used to indicate which nodes having a copy of the block. When describing the coherence actions of the protocol we will refer to the nodes involved as follows; *home* is the node where the page containing the block is allocated, *local* is the node where a request originates from, and finally, *remote* is any other node involved in a coherence action. The coherence protocol actions are as follows.

A processor read that misses in the cache, initiates a read miss request which is sent to home. If the block is *clean* in home, i.e., an updated copy exists, home responds with a block copy to local and updates the directory. Otherwise, if the block is *dirty* in home, i.e., a remote cache has an exclusive and possibly modified copy, home issues a write-back request to remote; remote updates home; and finally, home forwards a block copy to local, updates the directory, and the block ends up clean in home.

A processor write to a non-exclusive block in the cache, results in an ownership request sent to home. Home inspects the directory and sends explicit invalidations to the other caches with a block copy. Each cache receiving an invalidation, responds with an acknowledgment to home. When home has collected all acknowledgments, it grants ownership to local and the block ends up as dirty in home. During the time write-back requests and invalidations are pending, the block is in a transient state ‘busy’ and read and write requests to the block have to be retried.

In hardware-only directory protocols, the memory protocol engine consists of three parts implemented in hardware: a memory controller, a directory, and a state memory. The controller is responsible for processing incoming coherence requests, take correct actions depending on the state of the memory block, and also manage the directory. By contrast, in a software-only directory protocol, the management of the directory is migrated from the complex hard-wired controller to software handlers executed on the compute processor. These handlers are responsible for processing coherence requests and managing the directory which now is stored in main memory and not in a special hardware directory. Unfortunately, this migration has a cost. Upon each coherence request, the processor is interrupted in order to execute a software handler, thus interfering with the application execution which possibly prolongs the total execution time of the application. In Section 4 we describe in detail how a processor node for software-only directory protocols is organized.

In [13], we proposed several strategies to reduce the number of processor interrupts for read requests and we assume the most aggressive one in this study. This aggressive strategy has hardware support for request forwarding from home to remote for read miss requests to dirty blocks and also supports data block transfers to local in parallel with the processor interrupt for directory

updates. Moreover, read misses to blocks allocated in the local node do not interrupt the processor as proposed in [14].

The memory protocol-engine in home has a central role in our coherence protocol. The implementation of this protocol-engine differs between hardware-only and software-only directory protocols; in the latter case the processor is interrupted on each coherence request to home. As we will see in the next section, this fact has a fundamental effect on the relative performance between hardware-only and software-only directory protocols under latency tolerating and reducing techniques.

3 Execution Time Effects of Latency Tolerating and Reducing Techniques

To understand the effects latency tolerating and reducing techniques have on program execution time, we start in this section to develop a simple execution time model. Based on this model, we reason about the expected performance effects of three latency tolerating and reducing techniques which differ in the way they impact the protocol execution overhead and the total execution time. Then, in Section 5 we evaluate how well our expectations match the simulation results.

3.1 A Simple Execution Time Model

As a basis for the rest of the paper, we use a simple model of the execution time of a parallel application. This model allows us to reason about various effects that latency tolerating and reducing techniques have on the execution time components. In Figure 2, we show how the execution time of a parallel application can be decomposed for hardware-only and software-only directory protocols, i.e., the left and the right bars of Figure 2, respectively. We first discuss each of the bars and then, at the end of this section, we comment on the relative size of the bars.

The execution time components under a hardware-only directory protocol are shown in the left bar of Figure 2. The bottom part, denoted B_{hw}, corresponds to the busy time or the processor utilization, i.e., the time the processors execute application code. The next part corresponds to the read stall time, denoted R_{hw}, which is the time the processors are stalled due to read misses in the caches. Further, the part denoted W_{hw} corresponds to the write stall time, i.e., the time the processors are stalled waiting for ownership requests to complete. Finally, the top part of the bar corresponds to the synchronization stall time, denoted S_{hw}, which is the time the processors wait for, e.g., locks and barriers. The total execution time under a hardware-only directory protocol, denoted E_{hw}, is the sum of the different components we have described, i.e., $E_{hw} = B_{hw} + R_{hw} + W_{hw} + S_{hw}$.

The right bar in Figure 2 shows the execution time breakdown of a parallel application under a software-only directory protocol. We find that also under a software-only protocol the execution time consists of busy time, B_{sw}, read stall time, R_{sw}, write stall time, W_{sw}, and synchronization stall time, S_{sw}. However, in addition to the other stall times, overhead due to protocol execution, denoted P_{sw} in Figure 2, shows up in a software-only directory protocol. This protocol execution

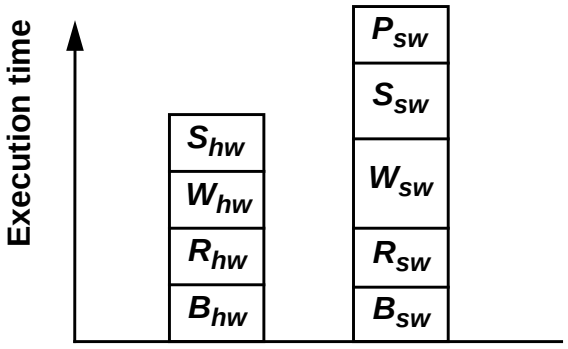

Figure 2: Execution time breakdown for hardware-only (left) and software-only (right) directory protocols.

overhead arises from the fact that each coherence request to home interrupts the compute processor in the home node. When a software handler is invoked, the execution of it may be overlapped with other stall times and does not disturb the application execution. Otherwise, the handler execution is only partly, or in the worst case not at all, overlapped with other stall times. In this latter case, the handler execution prolongs the total execution time of the application. The sum of handler execution times that are not overlapped with other stall times is referred to as *p-time* and denoted P_{sw} in Figure 2. In [13], we found very limited possibilities to overlap protocol execution with other stall times, and to simplify the reasoning in this section, we assume that the protocol execution cannot be overlapped at all with other stall times. The total execution time under a software-only directory protocol, denoted E_{sw}, is the sum of the busy time, all stall times, and p-time, i.e., $E_{sw} = B_{sw} + R_{sw} + W_{sw} + S_{sw} + P_{sw}$.

In this paper we will focus on the relative performance between hardware-only and software-only directory protocols. More specifically, we are interested in how the relative performance changes as we apply different latency tolerating and reducing techniques. Therefore, we use the *execution time ratio, ETR*, between hardware-only and software-only directory protocols as our primary measure of the relative performance. We define ETR as follows:

$$ETR = \frac{E_{sw}}{E_{hw}} = \frac{B_{sw} + R_{sw} + W_{sw} + S_{sw} + P_{sw}}{B_{hw} + R_{hw} + W_{hw} + S_{hw}}$$

We close this section by commenting on the relative sizes of the stall times under hardware-only and software-only directory protocols. In this study, we do not use any applications with dynamic load balancing that might affect the busy time, and therefore $B_{hw} = B_{sw}$ for all applications. In [13], we showed that by using efficient strategies for data forwarding, the read stall time under a software-only directory protocol is essentially the same as in a hardware-only directory protocol. Therefore, we assume in this section that $R_{hw} = R_{sw}$. By contrast, we found in [13] that the write stall time and the synchronization overhead were significantly higher in a software-only directory than in a hardware-only directory protocol, i.e., $W_{hw} < W_{sw}$ and $S_{hw} < S_{sw}$. As a result, *ETR* will be larger than one for the baseline systems; *ETR* values between 1.16 and 1.68 were reported in [13].

3.2 Prefetching — A Technique that Increases the Protocol Execution Overhead

Prefetching is a latency tolerating technique that have been proposed and evaluated, both as a software controlled technique [3, 19, 21] and as a hardware-based technique [6, 10]. We start with the general characteristics of prefetching and then specifically discuss the scheme used in this study. Further, in this study we only consider non-binding, read-shared prefetching, i.e., the block is fetched in a shared mode, but we will discuss other forms of prefetching as well in Section 6.

The goal of read-shared, non-binding prefetching is to bring data into the cache in advance so the processor encounters a cache hit instead of a miss. The data is fetched in a shared state and is still visible to the coherence protocol, as opposed to binding prefetching where data is loaded directly into, e.g., a register. As a result, the data might be evicted from the cache, due to an invalidation or a replacement, before the processor accesses it. In this situation the processor still encounters a miss and the prefetch was *useless*. Useless prefetches also originate from the fact that the prefetch scheme might fetch blocks that the processor never accesses. All these useless prefetches cause both unnecessary network traffic, and more important in this study, increase the occupancy in the memory protocol engine of the home node. This occupancy is usually no problem in a hardware-only directory protocol since each prefetch occupies the controller only a short amount of time. By contrast, in a software-only directory protocol this occupancy is directly translated to protocol execution overhead in the home node, i.e., p-time increases.

When prefetching is applied to both hardware-only and software-only directory protocols, we expect *ETR* to increase for the following reason. Prefetching attacks and reduces the read stall time, in the ideal case the read stall time is equally reduced in both hardware-only and software-only directory protocols, i.e., $R'_{hw} = R'_{sw} < R_{hw} = R_{sw}$. The write and synchronization stall times may increase a little as a result of contention. However, the big difference between hardware-only and software-only protocols is p-time, which *increases* as a result of useless prefetches, i.e., P_{sw} increases. This increase in combination with the decreased read stall time makes P_{sw} a relatively larger part of the total execution time under software-only directory protocols. As a result, we expect *ETR* to increase when prefetching is used. In Section 5.1, we present simulation results for prefetching and compare them with the expectations in this section.

Since useless prefetches are present to various degree in all prefetching schemes, we experimentally consider only one scheme. In Section 6, we discuss how other prefetch schemes relate to the one we have simulated. As an example of a prefetching scheme we have chosen *adaptive sequential prefetching* [10]. While the implementation details are found in [10], we here concentrate on the behavioral aspects. In sequential prefetching a fixed number of consecutive blocks, *K*, are prefetched for each cache miss the processor encounters. The *K* prefetched blocks are those directly following the missing block in the address space. However, the optimal number of blocks to prefetch on each miss varies during the execution. In adaptive sequential prefetching this shortcoming is addressed. By measuring the number of useful prefetches, i.e., the number of

prefetched block that the processor actually accesses before they are evicted from the cache, the scheme tunes the value of *K* according to the dynamic behavior of the application.

3.3 Migratory Optimization — A Technique that Reduces the Protocol Execution Overhead

In the previous section we discussed prefetching and found that the main obstacle of prefetching in the context of software-only directory protocols was the increased p-time. Therefore, in this section we will discuss a technique that *decreases* p-time when it is applied. We start by describing the technique and then the expected impact on the *ETR*.

Migratory sharing [15] is a program behavior not uncommon in parallel applications, e.g., data accessed in critical sections and by short read-modify-write sequences such as "`i=i+1`" exhibit migratory sharing. For example, consider the following scenario: First, one processor reads a data block and then it modifies the block, i.e., it obtains exclusive ownership of the block. Then, another processor reads and modifies the block in the same way, and thus, the block *migrates* around among the processors. Note that migratory data blocks are referenced by only one processor at the same time but by many processors in the long run.

In a system with a write-invalidate protocol each 'migration' of the block between two processors incurs two global actions; first a read miss request and then an ownership request. In a sequential consistent system, both these actions stall the processor resulting in both read and write stall times. However, two independent studies [8, 26] came up with the same solution to detect migratory blocks and optimize the coherence protocol for them. This *migratory optimization technique* dynamically detects migratory blocks at the home node, which sees all read miss and ownership requests, by recognizing two subsequent read-write sequences by two different processors. The coherence protocol then handles migratory blocks with a single read-exclusive request instead of one read miss and one ownership request, thus avoiding the write stall.

The performance gain achieved by the migratory optimization technique is the removal of global ownership requests for migratory blocks. As a result, both the write stall time and the occupancy in the home node is reduced, i.e., W_{hw}, W_{sw}, and P_{sw} in Figure 2 are reduced. The number of ownership requests is equally reduced in the hardware-only and the software-only directory protocol with this technique. However, each ownership request is more expensive, both in terms of stall time and occupancy, in a software-only than in a hardware-only directory protocol and therefore we expect W_{sw} to decrease more than W_{hw} does. In addition, the migratory optimization also reduces P_{sw} which leads us to expect *ETR* to decrease.

3.4 Release Consistency — A Technique that not Affects the Protocol Execution Overhead

Under sequential consistency, which is our default memory consistency model, the processor stalls on each access to shared data in order to enforce a global order of accesses. This is a severe restriction and can be relaxed. Under relaxed memory consistency models, ordering is only enforced on special, hardware-recognizable synchronization primitives. One of the most relaxed

consistency models is *Release Consistency* [12], *RC*. In RC, one distinguishes between *acquires* (acquiring a lock or a flag) and *releases* (releasing a lock or a flag). RC specifies that the processor is not allowed to proceed after an acquire before the acquire has completed and that a release is not allowed to be issued until all preceding requests have completed. The most important implication of RC in our framework is that the processor does not stall on write and release requests, i.e., all write stall time can be hidden and the synchronization stall time might decrease.

Relating the expected performance effects of RC to the bars in Figure 2, we begin by concluding that $W_{hw} < W_{sw}$ under sequential consistency. Since $W'_{hw} = W'_{sw} = 0$ under RC, we expect the software-only directory protocol to gain relatively more from RC than the hardware-only directory protocol does. Further, we expect both S_{hw} and S_{sw} to decrease, but a slight increase in R_{hw} and R_{sw} can occur as a result of higher contention within the processor nodes. Finally, since RC is not expected to change the number of coherence requests in the system, we expect the protocol execution overhead, i.e., P_{sw}, to be unaffected. To summarize, given that the stall times are equally reduced in both hardware-only and software-only directory protocols, the protocol execution overhead, P_{sw}, will be a relatively larger component of E_{sw} when RC is used, and thus *ETR* is increased. However, since $W_{hw} < W_{sw}$ under sequential consistency, the write stall time reduction under RC for a software-only directory protocol can outweigh the relatively higher p-time under RC which might even lead to a decreased *ETR*. In Section 5.3 we show how our intuitions relate to simulation results.

4 Architectural Parameters and Benchmark Programs

In this section we go through our detailed architectural assumptions, including timing parameters, and our set of parallel benchmark applications. The simulation models are built on top of the CacheMire Test Bench [2], a simulation framework and programming environment. The framework consists of multiple SPARC processors simulated at the instruction level and an architectural simulator of the multiprocessor model. The processors issue memory references to the architectural simulator which delays the processors according to its timing model. Thus, the same interleaving as in the target system is obtained. Instruction and private data references are not fed to the architectural simulator since we assume they expire a single-cycle cache hit.

4.1 Detailed Architectural and Timing Assumptions

We simulate a multiprocessor with 16 nodes and the organization of a processor node is shown in Figure 3. Each node contains a processor with its cache hierarchy, a memory module, a network interface, and a local bus. We comment on the organizational differences between hardware-only and software-only directory protocols where necessary.

The two-level cache hierarchy consists of a 2 Kbytes on-chip write-through first-level cache (*FLC*) and a 64 Kbytes off-chip copy-back second-level cache (*SLC*). Both caches are direct-mapped with a default block size of 64 bytes and full inclusion is maintained. Similar to many

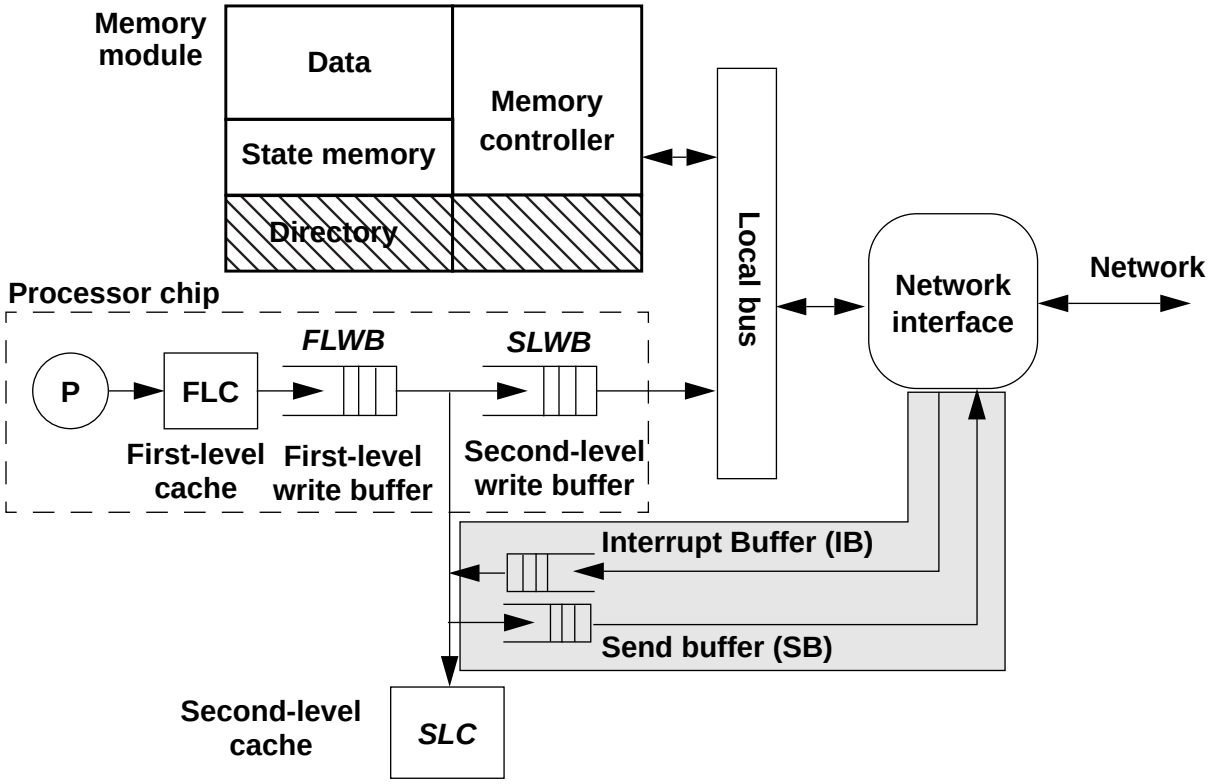

Figure 3: The organization of a processor node. The striped area only applies to hardware-only directory protocols and the shaded area only applies to software-only directory protocols.

contemporary microprocessors, we assume that the processor manages its own caches, i.e., the *SLC* controller is on-chip. A first-level write buffer (*FLWB*) with 16 entries connects the *FLC* and the *SLC* and allows processor writes to complete in a single cycle. Further, the *SLC* is lockup-free and supports multiple pending requests which is essential under, e.g., release consistency and prefetching. The buffering of pending requests are done in a 16 entries large second-level write buffer (*SLWB*).

The memory module consists of a data portion, a state memory containing the global state for each memory block, and a memory controller. In a addition, in a hardware-only directory protocol a separate directory (striped in Figure 3) is needed to record which caches having copies of the different memory blocks. Further, the memory controller also implements the memory actions in the coherence protocol. The page size in the system is 4 Kbytes and the pages are allocated to the memory modules in a round-robin fashion, i.e., pages with consecutive page numbers are allocated in consecutive memory modules. Moreover, synchronization operations, i.e., acquires and releases, are supported by a queue-based mechanism similar to the one implemented in Dash [20].

The network interface is responsible for routing messages between the node and the network, and is connected to the processor and the memory module through a 128 bits wide split transaction bus. The network interface also collects invalidation acknowledgments and notifies the memory-protocol engine when the last acknowledgment has arrived. In a software-only directory protocol, the network interface also routes messages to the interrupt buffer and from the send

buffer (shaded in Figure 3). When a coherence request arrives at the home node, the message is put into the interrupt buffer, the processor is interrupted, and a software handler is executed. This handler might generate new coherence messages which are put in the send buffer. The interrupt and send buffers are interfaced to the *SLC* bus, have the same access time as the *SLC*, and are accessible through memory mapped addresses. For more details, see [14], where we go through the design considerations for a processor node supporting software-only directory protocols. As suggested in [14], we assume that directory entries are cached when the software handlers access them.

As for the timing assumptions, we assume that the SPARC processors and their *FLC*s are clocked at 100 MHz, i.e., 1 pclock = 10 ns. The *SLC* interface is 128 bits wide and the *SLC* is implemented in SRAM with an access time of 30 ns to the first word in a block and 10 ns between the other words in the same block, resulting in a block transfer time of 60 ns. The memory module has an 128 bits wide interface and is implemented in DRAM with an access time of 90 ns to the first word and 10 ns between the rest of the words in a block, resulting in a 120 ns access time for a whole block. Both the local bus and the network interface are assumed to run at 100 MHz. While we also vary the network latency later in this study, the default interconnection network has an infinite bandwidth and a constant latency of 300 ns, i.e., 30 pclocks, which approximately corresponds to the latency in a 50 MHz mesh network with 32-bit flits. However, contention is correctly modelled in all parts within a processor node. Finally, the default software handler execution time is 50 pclocks excluding memory and buffer access times. To the default handler execution time, we add 6 pclocks for each message the handler sends, 13 pclocks for each read or write of the state of a memory block, and finally, 1 pclock or 16 pclocks for each directory access depending on whether a directory entry is cached or not.

4.2 Benchmark Programs

In our experimental evaluation we use three parallel programs taken from the SPLASH suite [25] (Water, Ocean, and MP3D) and one that has been provided to us by Stanford University (LU). The programs are written in C with parallelism and process coordination expressed using the *parmacs* macros from Argonne National Laboratory. The applications are compiled with `gcc` version 2.1 and optimization level `-O2`. All applications are from the scientific and engineering domain, and a short description of them together with the data set sizes we use are summarized in Table 1. Statistics are gathered in the parallel section of the applications to avoid initialization effects, which we assume to be negligible in an execution with more realistic data sets. Water and MP3D are two applications with a high degree of migratory sharing, while producer-consumer sharing dominates in LU and Ocean. As for the miss rates, we have found that coherence misses dominates in Water and MP3D, cold and replacement misses dominate in LU, and finally, coherence and replacement misses dominate in Ocean.

Table 1: The parallel programs together with the data set sizes we use in our simulations.

Application	Description	Data set size/Input data
Water	Water molecular dynamics simulation	288 molecules, 4 time steps
LU	LU-decomposition of a dense matrix	200x200 matrix
Ocean	Simulate eddy currents in an ocean basin	128-by-128 grid, tolerance 10^{-7}
MP3D	Particle-based wind-tunnel simulator	10,000 particles, 10 time steps

5 Performance Effects of Latency Tolerating and Reducing Techniques

In this section we will go through the relative performance between hardware-only and software-only directory protocols under adaptive sequential prefetching (Section 5.1), under migratory optimization (Section 5.2), and under release consistency (Section 5.3). Finally, in Section 5.4 we evaluate the performance effects of different block sizes.

5.1 Adaptive Sequential Prefetching

In this section we evaluate how the relative performance between a hardware-only and a software-only directory protocol changes when adaptive sequential prefetching is applied. The execution times of the four applications are shown in Figure 4. For each application, four bars are shown. The two left bars correspond to the execution time for a hardware-only directory protocol without (HW) and with (HW-P) adaptive sequential prefetching, and the two right bars correspond to a software-only directory protocol without (SW) and with (SW-P) adaptive sequential prefetching. For each bar, the execution time is decomposed, from bottom to top and with the notations from Figure 2, into the following components: the busy (B_x), the read stall (R_x), the write stall (W_x), and the synchronization stall times (S_x), where x is either *hw* or *sw*. Finally, for software-only directory protocols, the protocol execution overhead (P_{sw}) is shown at the top. To simplify the discussion, we will use these notations in the rest of this section.

We first focus on the relative execution times of HW and SW in Figure 4. By comparing the execution times, we find that the execution time ratios between SW and HW are between 1.16 (Water) and 1.76 (MP3D). These *ETR* values are in accordance with the results earlier presented in [13, 14].

Prefetching aims at reducing the read stall times, thus obtaining a shorter execution time. As we see in Figure 4, the execution times under HW-P is lower than under HW for all applications. In addition, simulation results show that prefetching reduces R_{hw} with 36%, 27%, 13%, and 11% for Water, LU, Ocean, and MP3D, respectively.

Continuing with the performance effects under the software-only directory protocol, we find by comparing the read stall times under SW and SW-P that prefetching reduces R_{sw} with 34%, 4%, 2%, and 12% for Water, LU, Ocean, and MP3D, respectively. So in that respect, prefetching helps performance. Unfortunately, both W_{sw} and S_{sw} increase by up to 25% (Ocean) and 24%

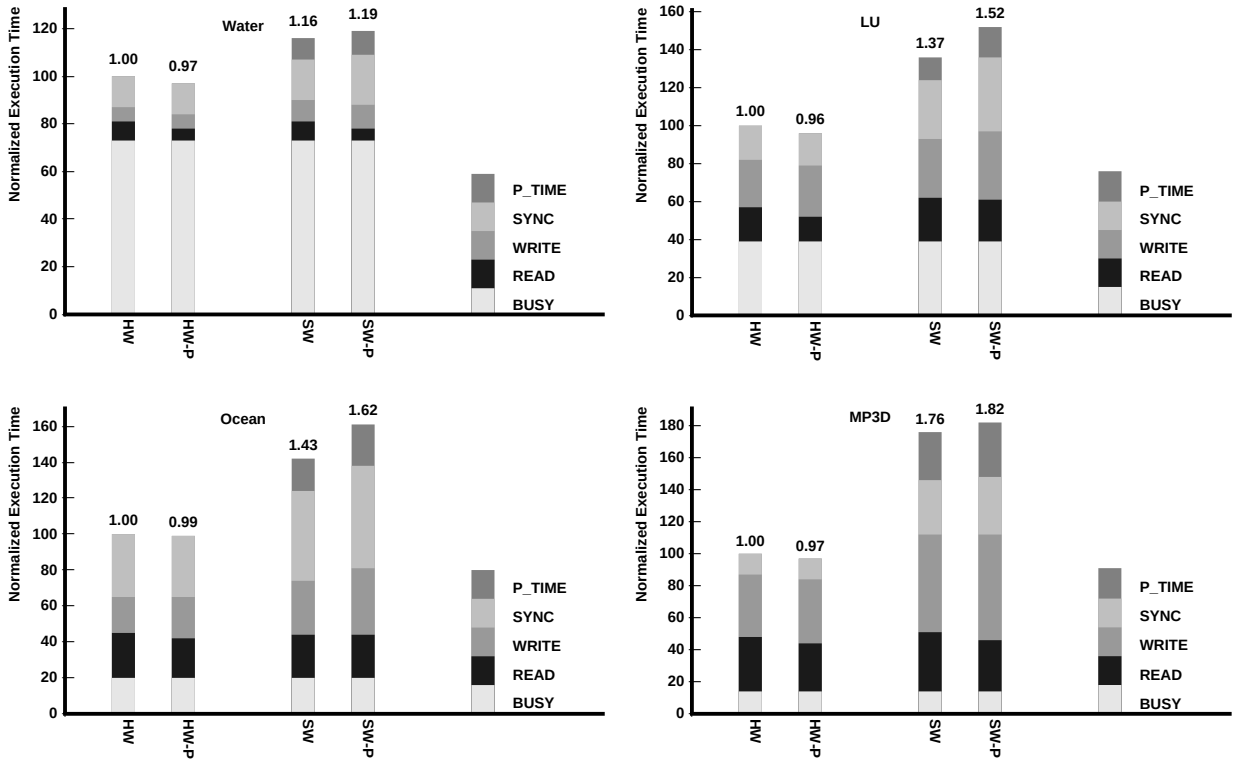

Figure 4: Normalized execution times for hardware-only and software-only directory protocols without (HW and SW) and with adaptive sequential prefetching (HW-P and SW-P).

(LU), respectively, under SW-P as compared to SW as a result of longer queuing delays in the home node. The average number of messages in the interrupt buffer at the time a new request is deposited in the buffer increases under prefetching. For example, for LU the average number of messages in the interrupt buffers upon deposit of a new message is increased from three to four when prefetching is applied.

By examining the protocol execution overhead, we observe a significant increase of P_{SW} under SW-P as compared to SW; P_{SW} has increased by between 12% (MP3D) and 34% (Ocean). This higher protocol execution overhead is expected according to the discussion in Section 3.2. The increase of P_{SW} stems from a higher number of coherence requests to home, and our simulation results show that SW-P generates between 9% (MP3D) and 33% (LU) more software handler invocations than SW.

Finally, by adding all the execution time components under SW and SW-P, we conclude that the total execution time, E_{SW}, has increased by between 3% (Water) and 13% (Ocean) when adaptive sequential prefetching is applied to the software-only directory protocol. As a result, the execution time ratios (*ETR*) between SW-P and HW-P are higher than between SW and HW for all applications. We summarize the resulting *ETR* values in Table 2.

One reason why the adaptive sequential prefetching scheme incurs so much protocol execution overhead can be tracked down to a low prefetch efficiency, which turns out to be between 25% (MP3D) and 36% (LU). In other words, for each successful prefetch, i.e., a prefetched block

Table 2: Execution time ratios between software-only and hardware-only directory protocols with and without prefetching.

	Water	LU	Ocean	MP3D
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.16	1.37	1.43	1.76
$ETR = E_{sw}(\text{SW-P})/E_{hw}(\text{HW-P})$	1.23	1.58	1.64	1.88

that the processor accesses before the block is evicted from the cache, between two and three useless prefetches are issued. Even though useless prefetches increase contention also in a hardware-only directory protocol, they can have a devastating effect on the performance of software-only directory protocols. Therefore, a high prefetch efficiency in a prefetching scheme is more important in software-only than in hardware-only directory protocols.

In Figure 4 we see that prefetching is able to reduce R_{sw} but at the cost of higher P_{sw} under SW-P. An interesting question is whether the reduction of R_{sw} can outweigh the increase of P_{sw} for reasonable network latencies. Therefore, we simulate network latencies of 60 pclocks and 120 pclocks, which is 2 and 4 times the baseline latency, respectively. Four times the default network latency might seem long at a first glance, but considering contemporary multiple-issue processors running at 300 MHz, a ratio between the instruction rate and the network frequency of eight is a realistic design point to examine.

In Figure 5, we show the normalized execution times for the software-only directory protocol with (SW-X-P) and without (SW-X) adaptive sequential prefetching, where X is the network latency counted in processor clock cycles and has the values 30, 60, and 120. We start by concluding that prefetching reduces R_{sw} for all applications and network latencies, although for Ocean the reduction is negligible. For Water and MP3D, we see that as the network latency increases, the read stall time reduction actually is larger than the increase in protocol execution overhead. As a result, adaptive sequential prefetching has a potential to reduce the total execution time when the speed gap between the processor and the network increases. However, for LU we observe that even though prefetching reduces R_{sw} significantly, the performance gain is outweighed mainly by longer write stall times and higher protocol execution overhead. As a result, for the network latencies we consider in this study, LU does not benefit from adaptive sequential prefetching under software-only directory protocols. Finally, for Ocean the overhead resulting from prefetching totally outweighs the small gains achieved in the read stall time. Therefore, we speculate that adaptive sequential prefetching does not reduce the execution time of Ocean for any reasonable network latency.

Since adaptive sequential prefetching exploits spatial locality, an important question to address is whether adaptive sequential prefetching is more effective in reducing the execution times for block sizes smaller than the default size of 64 bytes. Therefore, we evaluated the benefits of adaptive sequential prefetching in software-only directory protocols for block sizes of 16 bytes and 32 bytes with network latencies of 30, 60, and 120 pclocks. The resulting execution

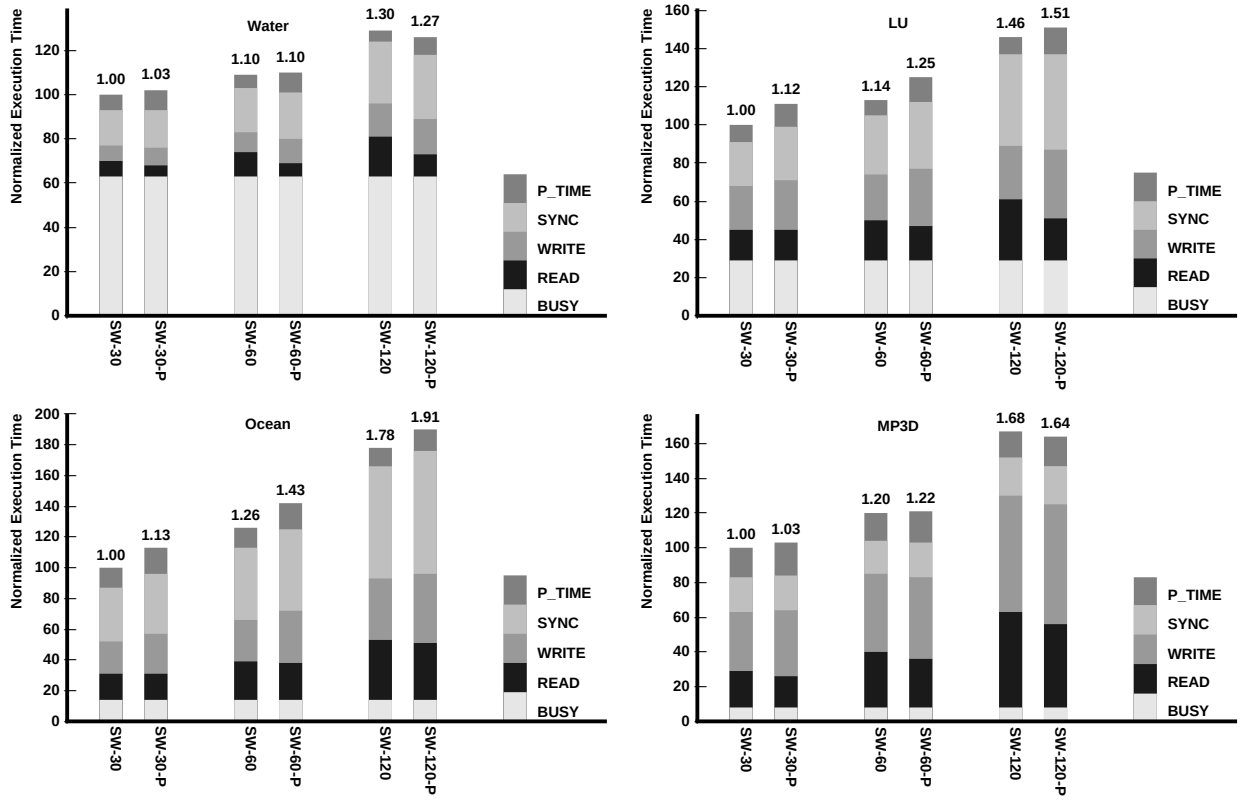

Figure 5: Normalized execution times of the software-only directory protocol with (SW-P) and without (SW) prefetching for various network latencies.

time ratios between the software-only directory protocol with and without adaptive sequential prefetching are summarized in Table 3.

Table 3: Execution time ratios between software-only directory protocols with and without sequential adaptive prefetching for different block sizes and network latencies.

Block size	Network latency	*ETR* = *E*(SW-P)/*E*(SW)			
		Water	LU	Ocean	MP3D
16 bytes	30 pclocks	1.00	1.00	1.27	1.13
	60 pclocks	0.93	0.99	1.06	1.07
	120 pclocks	0.84	0.99	1.08	1.12
32 bytes	30 pclocks	1.01	1.05	1.05	1.08
	60 pclocks	0.97	1.04	1.08	1.04
	120 pclocks	0.92	0.99	1.04	0.96

We begin by concluding that the execution time ratios between the software-only and the hardware-only directory protocols increase when adaptive sequential prefetching is applied for all block size and network latency combinations we have evaluated (not shown in Table 3). As expected, simulation results show that adaptive sequential prefetching generally reduces the read stall times more for smaller block sizes. Unfortunately, similar to the results for 64 bytes blocks,

the gains in execution time obtained by the read stall reduction is often outweighed by larger protocol execution overhead and/or by longer write and synchronization stall times. Therefore, even though adaptive sequential prefetching seems useful for applications with high computation-to-communication ratio such Water, the overall usefulness of it is questionable in software-only directory protocols.

In summary, even though a prefetching scheme is efficient when applied under a hardware-only directory protocol, it might increase the execution time when applied under a software-only directory protocol. Since software-only directory protocols are more sensitive to useless prefetches than hardware-only directory protocol, a low prefetch efficiency can have a devastating effect on the performance of software-only directory protocols. Simulation results show that the execution time ratio between software-only and hardware-only directory increases by between 6% and 15% for our default architectural parameters. However, as the speed difference between the processor and the network increases, prefetching has a potential to reduce the execution time also under software-only directory protocols. Since adaptive sequential prefetching does not provide any consistent performance improvement for software-only directory protocols, other prefetching schemes with higher prefetch efficiency seems to be a more promising alternative to consider.

5.2 Migratory Optimization

In this section we evaluate the relative performance between a software-only and a hardware-only directory protocol when the migratory optimization technique is applied. Migratory optimization reduces the protocol execution overhead as discussed in Section 3.3 and have, as we will see, other effects on the *ETR* than the prefetching technique in the previous section. In Figure 6, we show the resulting execution times when the migratory optimization technique is applied to hardware-only and software-only directory protocols, referred to as HW-M and SW-M, respectively. We first conclude that the migratory optimization reduces execution times for all applications under both hardware-only and software-only directory protocols. In order to understand the effects the migratory optimization has on the different execution time components, we go through each of them starting with the write stall time component.

By looking at the write stall times in Figure 6, we see that both W_{hw} and W_{sw} are significantly reduced for Water and MP3D under HW-M and SW-M, respectively. This result is expected from the discussion in Section 3.3 and in accordance with the results presented in [26]. By looking at the write stall time reduction in more detail, we find that W_{hw} is reduced by 73% and 75% for Water and MP3D, respectively, while the corresponding numbers for W_{sw} are 78% for both Water and MP3D. In other words, W_{sw} is reduced relatively more than W_{hw} as a result of shorter queuing delays in home; the reduced number of coherence requests reduces the average size of the interrupt buffer. For LU and Ocean, only small decreases in the write penalty are observed. In total,

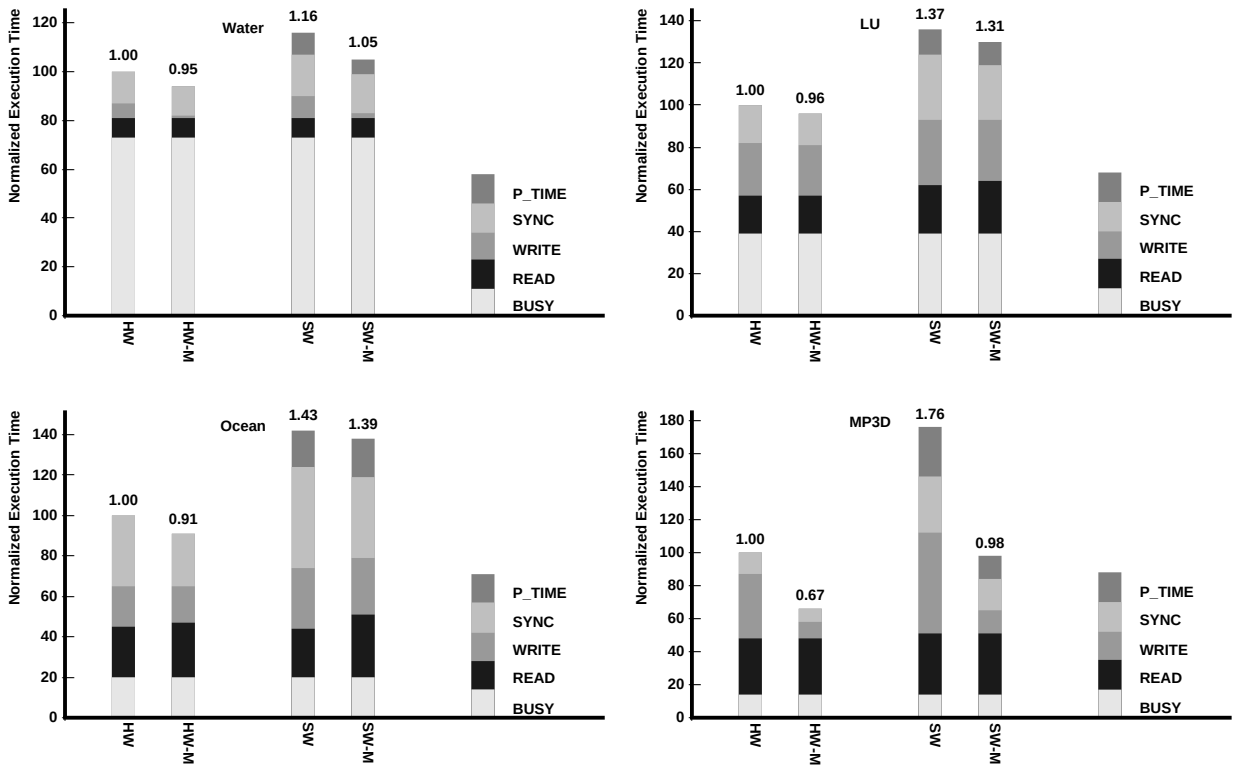

Figure 6: Normalized execution times for hardware-only and software-only directory protocols without (HW and SW) and with (HW-M and SW-M) migratory optimization.

this corresponds well to the application behavior; in Water and MP3D, migratory sharing dominates while producer-consumer sharing dominates in LU and Ocean.

Continuing with the synchronization stall times, we observe that they are also reduced when the migratory optimization is applied. This effect, which is most pronounced for LU, Ocean, and MP3D, arises mainly because of more efficient barrier synchronizations. In our barrier implementation, a counter variable is used to keep track of the current number of processors waiting at the barrier. This counter variable exhibits migratory sharing and, of course, benefits from the migratory optimization. However, for Ocean, S_{sw} is reduced relatively less than S_{hw}, 19% and 26%, respectively, when the migratory optimization is applied which might increase the *ETR*. This surprising observation is explained as follows. In [13], S_{sw} was found to be larger than S_{hw} due to imbalance in the number of coherence interrupts; some processors encounter up to ten times as many interrupts as other processors. Even though the barrier itself is more efficient with the migratory optimization, the inherent larger overhead due to the interrupt imbalance in software-only directory protocols are not affected.

The next execution time component to examine is the protocol execution overhead, P_{sw}, which is reduced by 35% and 56% for Water and MP3D, respectively. This reduction stems from a significantly lower number of coherence interrupts in the home nodes as we predicted in Section 3.3. For Water, the number of coherence interrupts has decreased by 34% and for MP3D by 52%. For the other two applications, LU and Ocean, P_{sw} is virtually unaffected as expected.

Finally, both R_{hw} and R_{sw} are virtually unaffected for three of the applications (Water, LU, and MP3D) when we apply the migratory optimization. However, for Ocean we observe that R_{hw} and R_{sw} are increased by 11% and 27%, respectively, when the migratory optimization is applied. Ocean has a non-negligible amount of false sharing [11], which causes many blocks to be deemed as migratory even though they are not. As a result, the miss rate increases with 12% under HW-M and 29% under SW-M. This effect was as far as we know first observed and reported in [23].

In Table 4, we show the resulting execution time ratios between SW and HW, and between SW-M and HW-M, respectively. Using the migratory optimization results in lower *ETR* for applications with a high degree of migratory sharing (Water and MP3D). The lower *ETR* results both from a relatively larger reduction of W_{sw} than of W_{hw} and from a reduction of P_{sw}. For LU, we find that the *ETR* is virtually the same with and without the migratory optimization. Finally, for Ocean the *ETR* increases, mainly as a result of a relatively smaller reduction of S_{sw} than S_{hw} and a larger increase of R_{sw} than of R_{hw} when migratory optimization is applied.

Table 4: Execution time ratios between software-only and hardware-only directory protocols with and without the migratory optimization.

	Water	LU	Ocean	MP3D
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.16	1.37	1.43	1.76
$ETR = E_{sw}(\text{SW-M})/E_{hw}(\text{HW-M})$	1.11	1.36	1.53	1.46

In summary, we have found the migratory optimization effective in reducing both the write stall time and the protocol execution overhead for applications with a high degree of migratory sharing. Since the write and synchronization stall times are relatively more reduced under SW-M than under HW-M, the execution time ratio decreases. Further, the migratory optimization has a potential to reduce the synchronization stall times also for other applications as a result of more efficient barrier synchronizations. Finally, the migratory optimization can increase the *ETR* for applications with a high degree of false sharing which penalizes SW more than HW.

5.3 Release Consistency

In this section we evaluate the relative performance of hardware-only and software-only directory protocols when release consistency is applied. In Figure 7, we show the relative execution times of hardware-only and software-only directory protocols under sequential consistency (HW and SW, respectively) and release consistency (HW-RC and SW-RC, respectively). The execution times are normalized to the execution time of a hardware-only directory protocol under sequential consistency.

A comparison between the execution times of HW and HW-RC, and between the execution times of SW and SW-RC in Figure 7 shows that release consistency gives a consistent performance improvement for both hardware-only and software-only directory protocols. This is consistent with results presented in earlier studies [13, 16].

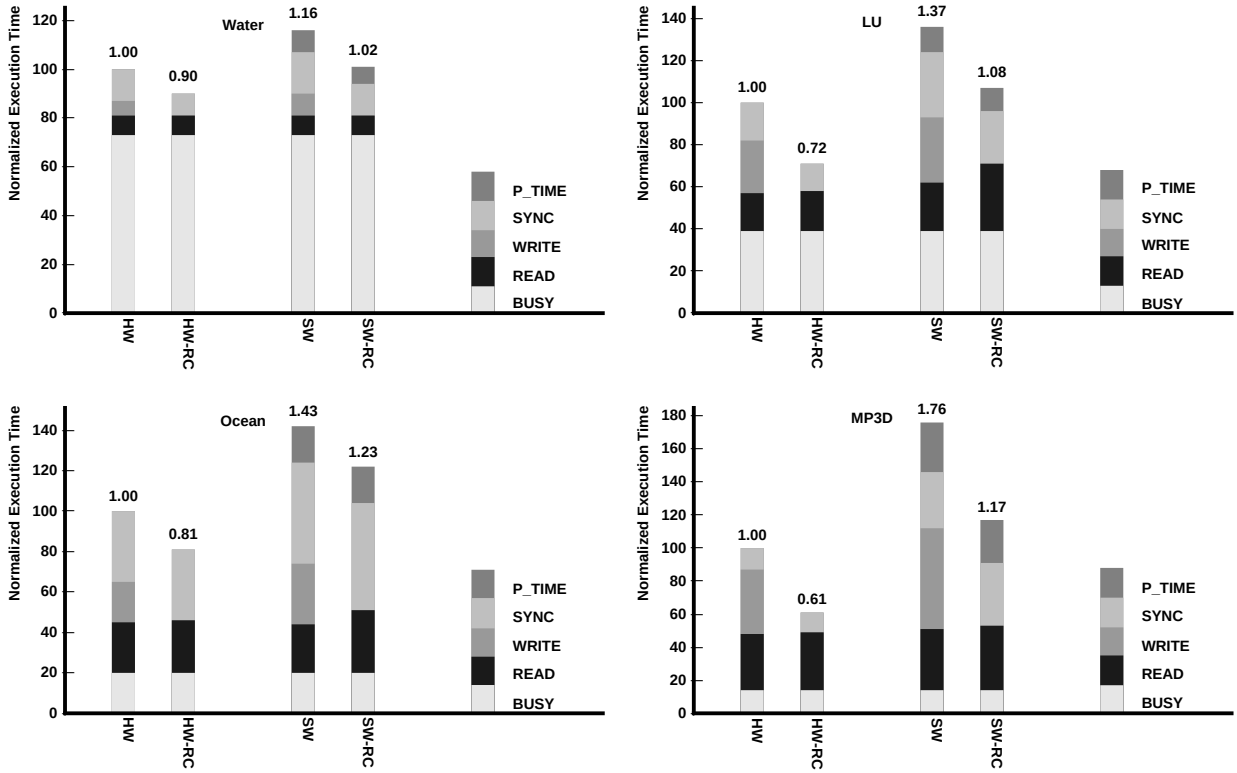

Figure 7: Normalized execution times for hardware-only and software-only directory protocols under sequential consistency (HW and SW) and under release consistency (HW-RC and SW-RC).

By comparing the protocol execution overhead for SW and SW-RC, we see that the intuition from Section 3.4 seems to be correct; since the number of coherence requests are the same under sequential and release consistency, P_{sw} should not be affected. As the results presented in Figure 7 show, P_{sw} is virtually unaffected for all applications which confirms our intuition.

As we can see in Figure 7, release consistency removes the write stall time for all applications. Under the hardware-only directory protocol, this is achieved with virtually no increase in read and synchronization stall times. By contrast, under the software-only directory protocol the read stall time is increased for two of the applications, LU and Ocean. For LU, the main reason is longer delays in the first-level write buffer. Under sequential consistency, a *FLC* read miss gets served by the *SLC* almost at once, while under release consistency it has to wait in average seven cycles. So, why does this happen in a software-only protocol? In a software-only directory protocol, each global write request takes a longer time than in a hardware-only directory protocol. Therefore, the second-level write buffer with its 16 entries is filled up much more often in a software-only directory protocol which blocks requests from the first-level write buffer. For Ocean, the longer read stall time originates from a combination of longer delays in the first-level write buffer and higher contention in the network interface of the home nodes.

The resulting execution time ratios between SW and HW, and between SW-RC and HW-RC are presented in Table 5. For three of the applications, LU, Ocean, and MP3D, the *ETR* increases when release consistency is applied as we predicted in Section 3.4. The relatively larger contribu-

tion from P_{sw} to the total execution time under release consistency results in a higher *ETR* for MP3D. For LU and Ocean, a combination of a relatively larger P_{sw} and a longer read stall time increases *ETR*. Surprisingly, the *ETR* slightly decreases for Water when release consistency is applied. This is caused by slightly shorter S_{sw} and P_{sw} under release consistency as compared to under sequential consistency.

Table 5: Execution time ratios between software-only and hardware-only directory protocols under sequential consistency and release consistency.

	Water	LU	Ocean	MP3D
$ETR = E_{sw}(\text{SW})/E_{hw}(\text{HW})$	1.16	1.37	1.43	1.76
$ETR = E_{sw}(\text{SW-RC})/E_{hw}(\text{HW-RC})$	1.13	1.50	1.52	1.92

The results presented in this section show that release consistency hides all write penalty for both hardware-only and software-only directory protocols. As expected, the execution time ratio between software-only and hardware-only directory protocols increases for three of the application when release consistency is applied. However, for the fourth application, Water, which is an application with high computation-to-communication ratio, the *ETR* slightly decreases when release consistency is applied.

5.4 Effects of Block Size Variations

As we discussed in the introduction, larger block sizes has a potential to reduce the read stall time and at the same time reduce the protocol execution overhead. To understand whether this intuition is true and also to understand how the block size choice impacts the *ETR* between software-only and hardware-only directory protocols, we simulated block sizes of 16, 32, 64, 128, and 256 bytes. The resulting execution times are shown in Figure 8, where HW-X and SW-X denotes the hardware-only and the software-only directory protocols, respectively, and X is the block size in bytes. All execution times are normalized to the execution time for the hardware-only directory protocol with a block size of 64 bytes, i.e., HW-64.

We start by looking at the execution times for the hardware-only directory protocol. For three of the applications (Water, LU, and MP3D) we observe a constant decrease of the execution time as the block size increases from 16 bytes up to 256 bytes. However, remember that we do not model contention in the interconnection network, and thus, the results are biased towards larger block sizes. If contention were modelled, we expect that the best block size choice should be less than 256 bytes. For Ocean the best block size, as indicated by our results, seems to be 64 bytes. Larger block sizes increase the execution time as a result of false sharing [11].

By looking at the execution times for the software-only directory protocol, similar effects as for the hardware-only protocol can be observed but they are more pronounced. For two of the applications (Water and MP3D) the execution time decreases more rapidly than for the hardware-only directory protocol as the block size increases from 16 bytes to 256 bytes. For LU, the execu-

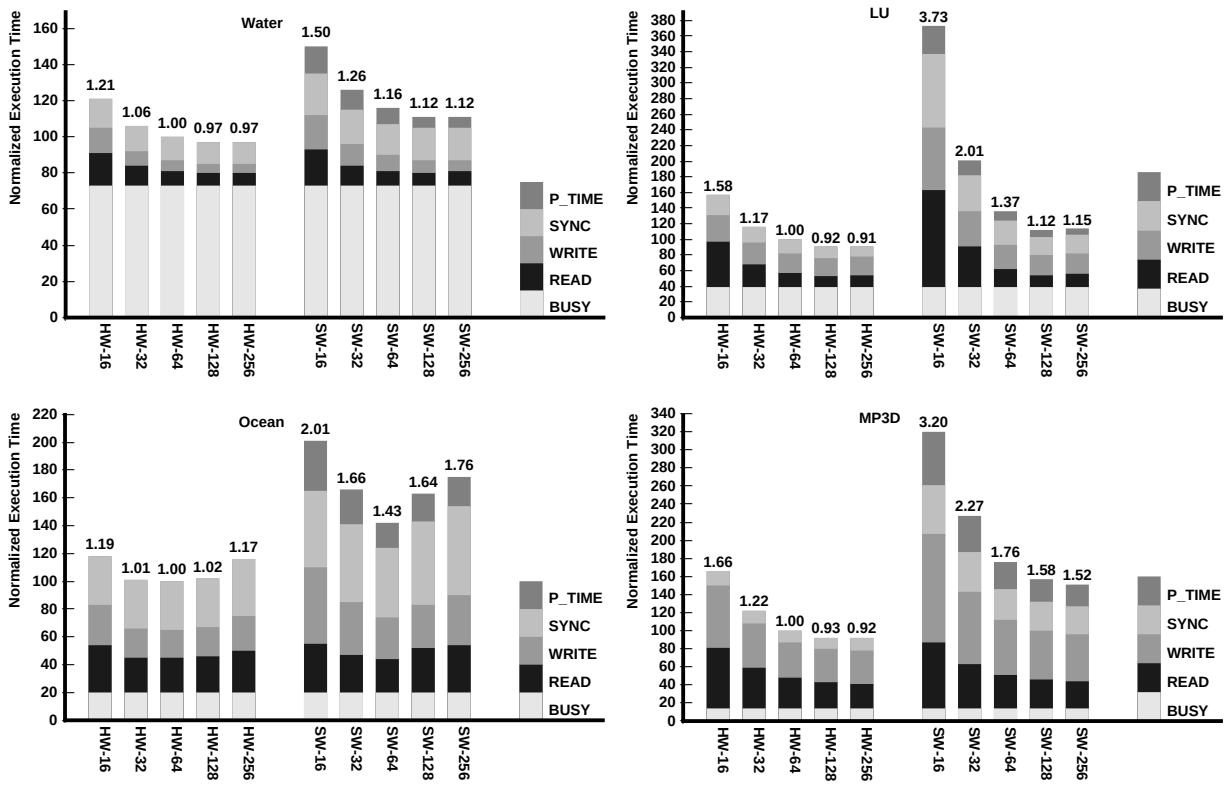

Figure 8: Normalized execution times of the applications for 16, 32, 64, 128, and 256 bytes blocks.

tion time dramatically decreases up to a block size of 128 bytes, and then it increases slightly again. Finally, for Ocean, we see a steep decrease in execution time for increasing block sizes up to 64 bytes, and then the execution time increases rapidly as a result of false sharing. As expected, we find by looking at the results in Figure 8 that the protocol execution overhead decreases with increasing block sizes for all applications except for Ocean, where P_{sw} slightly increases for 128 bytes and 256 bytes blocks. Intuitively, P_{sw} decreases as a result of fewer coherence interrupts.

In contrast to the execution time variations between different block sizes for the hardware-only directory protocol, the execution time variations are much more dramatic for the software-only directory protocol. In other words, the performance of the software-only directory protocol is more sensitive to the block size choice than the hardware-only directory protocol performance is. The ratios between the execution times for the worst block size choice and for the best block size choice are 1.25, 1.74, 1.19, and 1.80 for Water, LU, Ocean, and MP3D, respectively, for the hardware-only directory protocol. The corresponding ratios for the software-only directory protocol are 1.34, 3.33, 1.41, and 2.11, which indicates the higher sensitivity to the block size choice.

The resulting execution time ratios between software-only and hardware-only directory protocols for the various block sizes are summarized in Table 6. Since the execution time varies relatively more for the software-only than for the hardware-only directory protocol, the *ETR* values also vary. For Water and MP3D, the *ETR* decreases as the block size increases, i.e., the execution time decreases faster for the software-only than for the hardware-only directory protocol for increasing block sizes. For LU, the *ETR* also follows the variations in the software-only directory

protocol execution, i.e., the *ETR* decreases for block sizes up to 128 bytes and then increases slightly for 256 bytes blocks. Finally, for Ocean, the *ETR* decreases up to 64 bytes block, and then starts to increase again.

Table 6: Execution time ratios between the software-only and the hardware-only directory protocols for block sizes of 16, 32, 64, 128, and 256 bytes.

	Water	LU	Ocean	MP3D
ETR = *E*(SW-16)/*E*(HW-16)	1.24	2.36	1.69	1.92
ETR = *E*(SW-32)/*E*(HW-32)	1.19	1.72	1.64	1.86
ETR = *E*(SW-64)/*E*(HW-64)	1.16	1.37	1.43	1.76
ETR = *E*(SW-128)/*E*(HW-128)	1.15	1.22	1.61	1.70
ETR = *E*(SW-256)/*E*(HW-256)	1.15	1.26	1.50	1.65

6 Discussion and Generalizations

In this study we have only considered read-shared prefetching, i.e., the block is prefetched in a shared mode. An alternative is to allow read-exclusive prefetching [21], i.e., when a block is prefetched, it is obtained in an exclusive mode. A successful read-exclusive prefetch not only reduces the read stall time, possibly to zero, but also removes the coherence actions associated with a separate ownership acquisition. Thus, the total number of coherence requests is expected to decrease when read-exclusive prefetching is used, and as a result, the protocol execution overhead is also expected to decrease. Our experience from Section 5.2 indicates that techniques reducing the protocol execution overhead is expected to decrease the execution time ratio between software-only and hardware-only directory protocols. Therefore, we expect software-only directory protocols to be more competitive with hardware-only directory protocols when read-exclusive prefetching is used.

In Section 5.1, we found rather small gains of adaptive sequential prefetching under the software-only directory protocol. Since adaptive sequential prefetching exploits spatial locality, and in Section 5.4 we showed that with 64 bytes blocks we exploit most of the spatial locality in the applications, the gains are limited. Adaptive sequential prefetching also suffers from a too low prefetch efficiency to be really effective in a software-only directory protocol. Therefore, we expect software-controlled prefetching schemes [3, 19, 21, 22] to have a larger potential to increase the performance of software-only directory protocols since they can exploit knowledge about the application behavior.

We have also studied a stride prefetching scheme [9] as an alternative to adaptive sequential prefetching. Since stride prefetching is more selective than adaptive sequential prefetching when issuing prefetches, we believed that stride prefetching should be better than adaptive sequential prefetching for a software-only directory protocol. Unfortunately, our results indicate that stride prefetching either issues too few prefetches to reduce the read stall time or when the read stall

time actually is reduced, the protocol execution overhead has increased to the same levels as for adaptive sequential prefetching. This is not surprising since the results in [9] show that the most common stride is one, and then stride and adaptive sequential prefetching behave similarly in terms of read stall time reduction.

Multithreading is a latency tolerating technique used in, e.g., the MIT Alewife [1]. It has been shown to be effective in hiding processor stall times for accesses that require global actions by switching to another thread of computation [16]. However, since multiple threads run on the same processor, the number of global actions originating from each processor is likely to increase. As a result, the protocol execution overhead in a software-only directory protocol is expected to increase. Therefore, building on our experience from Section 5.1, we expect the execution time ratio between software-only and hardware-only directory protocol to increase when multithreading is used.

As we have seen is the number of coherence interrupts on a processor a critical parameter in software-only directory protocols. A way to reduce the number of coherence interrupts for each processor would be to arrange the processors in clusters. Thus, coherence actions are only needed for accesses to blocks allocated in other clusters; within each cluster a snoopy cache protocol can be used. A simple back-on-the-envelope calculation give us the expected number of coherence interrupts for each processor. Assume that we have P processors organized in K clusters and each processor generates M misses evenly distributed over the clusters. Thus the total number of misses in the system is M*P. For each cluster M*P/K misses can be served locally, thus totally M*P*(K-1)/K misses require service in another cluster. Assuming that each processor services equally many requests, it turns out that each processor responds to (M*P*(K-1)/K)/P = M*(K-1)/K misses. Consider for example a system with 16 processors. If each cluster only has one processor, each processor has to respond to M*15/16 misses. On the other hand, if the processors are organized in four clusters each processor only has to respond to M*3/4 misses, i.e., about 19% fewer misses for each processor. Therefore, we believe that software-only directory protocols can be an interesting design alternative for cluster-based multiprocessors.

7 Conclusions

In both software-only directory protocols, i.e., protocols where the directory management is done by software handlers executed on the compute processor, and hardware-only directory protocols, performance is often limited by processor stall time due to memory accesses. In addition to these stall times, the total execution time for a software-only directory protocol might be prolonged due protocol execution overhead resulting from the handler invocations. To cope with these latencies, many latency tolerating and reducing techniques have been proposed and evaluated in the context of hardware-only directory protocols. However, the effectiveness of such techniques in the context of software-only directory protocols has been unexplored.

In this study we have evaluated how the relative performance between software-only and hardware-only directory protocols is affected when different latency tolerating and reducing techniques are applied. The techniques can be divided into three classes depending on how they impact the protocol execution overhead in software-only directory protocols; a technique either increases, decreases, or does not affect the protocol execution overhead. As representatives for the three classes we have chosen adaptive sequential prefetching, a migratory optimization technique, and finally, release consistency as techniques that increase, decrease, and do not affect the protocol execution overhead, respectively.

Based on architectural simulations we have found that software-only directory protocols are more sensitive to useless prefetches than hardware-only directory protocols are. Each useless prefetch, i.e., a prefetch that fetches a block that is evicted from cache before the processor accesses it, incurs protocol execution overhead in the node where the memory block is allocated which potentially prolongs the total execution time. Our results show that even though adaptive sequential prefetching reduces the read stall time by between 2% and 34% in a software-only directory protocol, the execution time is prolonged by between 3% and 13% mainly due to too many useless prefetches. Therefore, larger attention must be paid to the prefetch efficiency when choosing a prefetch scheme for a software-only than for a hardware-only directory protocol.

In contrast, software-only directory protocols generally gain relatively more than hardware-only directory protocols when techniques that reduce the number of coherence actions are applied. This fact was confirmed both with the migratory optimization technique and when the block size was varied. Further, software-only directory protocols is more sensitive to the choice of the block size than hardware-only directory protocols are. As seen from a programmer’s point of view, software-only directory protocols suffer more than hardware-only directory protocols when the best block size for an application does not match the block size of the cache.

Release consistency, a technique which does not affect the protocol execution overhead, manages to hide all write stall time for both software-only and hardware-only directory protocols. However, since release consistency does not reduce the protocol execution overhead, it becomes a relatively larger part of the total execution time. For three of our applications, this fact results in a higher execution time ratio between software-only and hardware-only directory protocols when release consistency is applied.

Overall, this study shows that the efficiency of a latency tolerating technique not only depends on how well it tolerates the latency as seen by the local node, but also on how it interacts with the node where a memory block is allocated. Therefore, more care has to be taken when choosing an appropriate latency tolerating and reducing technique for software-only directory protocols than needed for hardware-only directory protocols.

Acknowledgments

This research has been funded in part by the Swedish National Board for Industrial and Technical Development (NUTEK) under contract number P855.

References

[1] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," in *Proceedings of 22nd International Symposium on Computer Architecture*, pages 2-13, June 1995.

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

[3] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 40-52, April 1991.

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

[5] D. Chaiken and A. Agarwal, "Software-Extended Coherent Shared Memory: Performance and Cost," In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 314-324, April 1994.

[6] T-F. Chen and J-L. Baer, "Effective Hardware-Based Data Prefetching for High-Performance Processors," *IEEE Transactions on Computers*, 44(5):609-623, May 1995.

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

[8] A.L. Cox and R.J. Fowler, "Adaptive Cache Coherency for Detecting Migratory Shared Data," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 98-108, May 1993.

[9] F. Dahlgren and P. Stenström, "Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," In *Proceedings of the First IEEE Symposium on High Performance Computer Architecture*, pages 68-77, January 1994.

[10] F. Dahlgren, M. Dubois, and P. Stenström, "Sequential Hardware Prefetching in Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 4(7):733-746, July 1995.

[11] M. Dubois, J. Skeppstedt, and P. Stenström, "Essential Misses and Memory Traffic in Coherence Protocols," accepted as a regular paper in *Journal of Parallel and Distributed Processing*, October 1995 (in press).

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 15-26, May 1990.

[13] H. Grahn and P. Stenström, "Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors," In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 38-47, June 1995.

[14] H. Grahn and P. Stenström, “Architectural Support for an Efficient Implementation of a Software-Only Directory Cache Coherence Protocol,” Technical Report, Dept. of Computer Engineering, Lund University, June 1995.

[15] A. Gupta and W-D. Weber, “Cache Invalidation Patterns in Shared-Memory Multiprocessors,” *IEEE Transactions on Computers*, 41(7):794-810, July 1992.

[16] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W-D. Weber, “Comparative Evaluation of Latency Reducing and Tolerating Techniques,” In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 254-263, May 1991.

[17] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy, “The Performance Impact of Flexibility in the Stanford FLASH Multiprocessor,” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 274-285, October, 1994.

[18] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood, “Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,” *ACM Transactions on Computer Systems*, 11(4):300-318, November 1993.

[19] A.C. Klaiber and H.M. Levy, “An Architecture for Software-Controlled Data Prefetching,” In *Proceedings of the 18th International Symposium on Computer Architecture*, pages 43-53, May 1991.

[20] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, “The DASH Prototype: Logic Overhead and Performance,” *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41-61, January 1993.

[21] T. Mowry and A. Gupta, “Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors,” *Journal of Parallel and Distributed Computing*, 12(2):87-106, June 1991.

[22] T. Mowry, “Tolerating Latency Through Software-Controlled Data Prefetching,” Ph.D. Thesis, Stanford University, March 1994.

[23] H. Nilsson and P. Stenström, “An Adaptive Update-Based Cache Coherence Protocol for Reduction of Miss Rate and Traffic,” In *Proceedings of Parallel Architectures and Languages Europe (PARLE*), Lecture Notes in Computer Science, No. 817, Springer-Verlag, Berlin, pages 363-374, July 1994.

[24] S. K. Reinhardt, J. R. Larus, and D. A. Wood, “Tempest and Typhoon: User-Level Shared-Memory,” In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325-336, April 1994.

[25] J-P. Singh, W-D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared-memory,” *Computer Architecture News*, 20(1):5-44, March 1992.

[26] P. Stenström, M. Brorsson, and L. Sandberg, “An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing,” In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 109-118, May 1993.