Evaluation of Design Alternatives for a Directory-Based Cache Coherence Protocol in Shared-Memory Multiprocessors

Håkan Grahn

Doctoral thesis submitted for partial fulfilment of the requirements for the degree of Doctor of Philosophy in Computer Engineering.

Lund October, 1995

CODEN: SE-LUTEDX/TEDT-95/1007+18p

Abstract

In shared-memory multiprocessors, caches are attached to the processors in order to reduce the memory access latency. To keep the memory consistent, a cache coherence protocol is needed. A well known approach is to record which caches have copies of a memory block in a directory and only notify the caches having a copy when a processor modifies the block. Such a protocol is called a directory-based cache coherence protocol. This thesis, which is a summary of seven papers, identifies three problems in a directory-based protocol, and evaluates implementation and performance aspects of some design alternatives. The evaluation methodology is based on program-driven simulation.

The write-invalidate policy, which is used in the baseline protocol, forces all other copies of a block to be invalidated when a processor modifies the block. This leads to a cache miss each time a processor accesses an invalidated block. To reduce the number of cache misses, a competitive-update policy is proposed in this thesis. The competitive-update policy is shown to reduce both the read stall and execution times as compared to write-invalidate under a relaxed memory consistency model. However, update-based policies need more buffering and hardware support in the caches.

In the baseline protocol, the implementation cost of the directory is proportional to the number of caches. To reduce this cost, an alternative directory organization is proposed which distributes the directory information among the caches sharing the same memory block. To achieve a low write latency, the caches sharing a block are organized in a tree. The caches are linked into the tree in parallel with application execution to achieve a low read latency.

The hardware-implemented directory controller in the baseline protocol may lead to high design complexity and implementation cost. This thesis evaluates a design alternative where the controller is implemented using software handlers executed on the compute processor. By using efficient strategies and proper architectural support, this design alternative is shown to be competitive with the baseline protocol. However, the performance of this alternative is more sensitive to other design choices, e.g., block size and latency tolerating techniques, than the baseline protocol.

Evaluation of Design Alternatives for a Directory-Based Cache Coherence Protocol in Shared-Memory Multiprocessors

Håkan Grahn

This thesis is a summary of the following papers. References to the papers will be made using the roman numbers associated with the papers.

I. H. Grahn, P. Stenström, and M. Dubois: "Implementation and Evaluation of Update-Based Cache Protocols Under Relaxed Memory Consistency Models," *Future Generation Computer Systems*, Vol. 11, No. 3, pages 247-271, June 1995.

II. H. Nilsson[1] 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 817, Springer-Verlag, pages 363-374, July 1994. The paper received Best Paper Award at the conference.

III. H. Nilsson[1] and P. Stenström: "The Scalable Tree Protocol — A Cache Coherence Approach for Large-Scale Multiprocessors," In *Proceedings of Fourth IEEE Symposium on Parallel and Distributed Processing*, pages 498-506, December 1992.

IV. H. Nilsson[1] and P. Stenström: "Performance Evaluation of Link-Based Cache Coherence Schemes," In *Proceedings of the 26th Annual Hawaii International Conference on System Sciences*, pages 486-495, January 1993.

V. 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.

VI. 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. Presented at the Fifth Workshop on Scalable Shared-Memory Multiprocessors, June 1995.

VII. H. Grahn and P. Stenström: "*Relative Performance of Hardware and Software-Only Directory Protocols Under Latency Tolerating and Reducing Techniques*," Technical Report, Dept. of Computer Engineering, Lund University, August 1995.

[1]In 1994, I changed my last name from Nilsson to Grahn.

1 Introduction

Parallelism is a way to meet the ever increasing demand for computational power. One approach to achieve higher performance is to connect several processors together in a *multiprocessor*. In a *shared-memory* multiprocessor, all memory in the machine is presented to the programmer as one single memory unit. However, even though the memory often is physically distributed over several memory modules, a memory access takes a significant amount of time to complete, referred to as the *memory access latency*. To reduce this latency, caches are attached to each processor in order to bring the data closer to the processors. Since caches allow multiple copies of the same datum to be present in the machine, a mechanism is required to ensure a coherent view of the memory. This mechanism is referred to as the *cache coherence protocol* [24].

Given that there exist several copies of a datum and one processor wants to modify the datum, the cache coherence protocol is responsible for notifying all other caches having copies where to obtain the new value. Therefore, the protocol must handle two problems. The first is *how* to notify the other caches of a new value. The second is how to *locate* the caches to be notified. In this thesis we consider a protocol, that is similar to the one implemented in the Stanford DASH multiprocessor [20] and constitutes our baseline protocol, which solves the two problems above as follows. When a processor modifies a datum, the protocol notifies all other caches having copies of this datum by invalidating these copies. In order to locate which copies to invalidate, a *directory* records which caches have a copy of a datum. Such a protocol is called a *directory-based cache coherence protocol* [2, 5]. For each memory block, the directory has one bit vector with as many bits as there are caches in the system. Associated with the directory is a hardware implemented *directory controller* which is responsible for managing the directory.

This thesis, which is a summary of seven papers referred to as **I-VII**, addresses three problems with the above protocol, and evaluates performance and implementation aspects of some design alternatives for it. In the performance evaluations, the baseline protocol is used as a reference protocol. The first problem arises when a processor wants to access an earlier invalidated datum. Then, it has to fetch a fresh copy which results in a long access latency. An alternative is to update the other copies with the new value upon processor writes resulting in shorter read access latencies. In **I** and **II**, which are summarized in Section 2, this design alternative is evaluated. The second problem concerns the implementation cost of the directory, which increases proportionally with the number of caches. To reduce this cost, a novel directory organization is proposed and evaluated in Section 3, which summarizes **III** and **IV**. The third problem is related to the hardware implemented directory controller; a large design effort is required in order to engineer it correctly. By migrating the directory controller functionality to software, the hardware design effort can be reduced. Section 4, which summarizes **V**, **VI**, and **VII**, addresses the issues involved when designing a software implemented directory controller.

2 Coherence Policy Alternatives in Shared-Memory Multiprocessors

When multiple caches have copies of the same datum and one processor modifies it, the *coherence policy* [24] defines how to notify the other caches of the modification. In the rest of the thesis, I will use the term *block* instead of datum, since a block which contains multiple data is the unit that the cache coherence protocol handles. The baseline protocol uses a *write-invalidate policy* [13], which forces all other copies of the block to be invalidated upon a processor write, resulting in an exclusive copy of this block in the cache of the writing processor. Unfortunately, when a processor accesses an invalidated block it encounters a cache miss, known as a *coherence miss*. All misses including the coherence misses have a long *read miss latency*. The main motivation in **I** and **II** is to reduce the *read stall time*, which is the product of the total number of misses and the average read miss latency.

2.1 Performance of the Write-Update Policy

Under the *write-update policy* [21], the total number of cache misses is reduced since coherence misses are eliminated; upon a processor write, all copies of a block are updated. Unfortunately, the write-update policy generates more network traffic than the write-invalidate policy. One of the objectives in **I** is to evaluate if the write-update policy actually can reduce the read stall time as compared to the write-invalidate policy despite the larger amount of network traffic. This issue has been addressed earlier in the context of bus-based multiprocessors [10, 11] but not for multiprocessors with a general interconnection network and a directory-based protocol.

Using program-driven simulations of a detailed multiprocessor model [3], I find in **I** that for two of four studied applications, the write-update policy reduced the read stall time significantly; up to two thirds in the best case, as compared to the baseline protocol. Unfortunately, for the other two applications the read stall time was longer under the write-update policy than the write-invalidate policy; in the worst case almost five times longer. For these two applications, the write-update policy generates between eight and ten times as much network traffic as the write-invalidate policy.

There are mainly two reasons why the write-update policy generates more traffic than the write-invalidate policy [9]. First, if one processor writes several times to the same block with no intervening access by another processor, the overhead for all updates but the last one is unnecessary. Second, a block may by present in a cache and continue to be updated even though it is no longer accessed by the processor. This higher traffic causes contention in the network which increases the average miss latency for the remaining read misses, and may result in a longer read stall time for the write-update policy than for the write-invalidate policy.

2.2 The Competitive-Update Policy

To cope with the high communication demand under the write-update policy but still benefit from a low cache miss rate, a technique called competitive snooping has been proposed in the context of bus-based multiprocessors [19]. The basic idea is to only update those copies that are regularly accessed by the processors. When a copy of a block has been updated a predefined number of times with no intervening local processor access, the block is invalidated.

Based on the idea of competitive snooping, I propose in **I** a *competitive-update policy.* While competitive snooping is defined in the context of a bus-based system, the competitive-update policy is applicable to so called CC-NUMA multiprocessors with a directory-based cache coherence protocol. In addition to updating the cached copies, competitive-update offers a big advantage for CC-NUMA multiprocessors as follows. In CC-NUMA multiprocessors under the write-invalidate policy, a read miss initiates a request for a fresh copy from memory. The memory needs to forward the request to the cache with the exclusive copy, which updates the memory. Then, the memory supplies a fresh copy to the requesting cache. Under the competitive-update policy, the memory is kept up-to-date and can supply a fresh copy at once as long as at least two caches have a copy of the block, which results in a significantly lower read latency.

To implement the competitive-update policy, a counter is associated with each block frame in the cache. When the local processor accesses a block, the counter is set to a predefined threshold. For each update of the block by another processor, the counter is decremented. When the counter reaches zero, the block is invalidated and the updates to it cease.

Regarding performance, I show that the competitive-update policy reduces the read stall times as compared to the write-invalidate and the write-update policies for the four applications that we use. However, the network traffic for the competitive-update policy is still higher, between 25% and 85%, than for the write-invalidate policy. I evaluate various thresholds and find that four establishes a good trade-off between miss rate reduction and network traffic for the studied applications.

The total execution time of an application depends not only on the read stall time but also on the *write latency.* The write latency is the time it takes to complete one write request, i.e., to invalidate or update all other copies upon a processor write. A write request to a block that is exclusive in the local cache can complete locally, resulting in a short write latency. By contrast, the write latency is longer if other copies need to be invalidated or updated. When a processor writes several times to a block without any intervening access by another processor to the same block, only the first write incurs a long write latency for the write-invalidate policy while succeeding writes can complete locally. By contrast, for the write-update policy, all writes encounter the longer write latency to update remote copies.

In a system where the processors stall on writes until they have completed, the write latency can contribute significantly to the total execution time. To reduce the execution time, some researchers have proposed techniques that allow the processors to continue their program execution without waiting for writes to complete [1, 8, 12]. As a result, the write latency can be overlapped with both computation and other latencies, e.g., read miss latency.

To allow the processor to continue its execution without waiting for a write to complete, we need to buffer write requests. Another issue examined in **I** is to quantify how much buffer space update-based policies need as compared to the write-invalidate policy. While only a few buffer entries is enough for the write-invalidate policy, sixteen entries seems more reasonable for the update-based policies. Further, the results in **I** also show that while the write-invalidate policy only needs support for one global pending invalidation request in the cache, update-based policies need support for multiple pending updates to hide the write latency effectively.

2.3 An Adaptive Update-Based Protocol

Even though the competitive-update policy both has lower read stall time and execution time for the applications studied, it is suboptimal for applications with *migratory data objects* [14]. Migratory data objects, which are not uncommon in parallel programs, are accessed by only one processor at the time, usually in a read-modify-write manner, but by many processors in the long run. Unfortunately, this behavior generates unnecessary traffic for the competitive-update policy in the following way. Consider a situation where N processors read and modify a block in a circular fashion and the competitive threshold T is smaller than $N-1$. When processor i has modified the block, $N-1$ other processors read and modify the block before processor i reads it again. When processor i reads the block it has been updated $N-1$ times, but since $T<N-1$, it has also been invalidated for processor i. As a result, the updates to processor i are useless and only generate unnecessary network traffic.

The objective in **II** is to reduce the unnecessary traffic generated for migratory data blocks under the competitive-update policy by using the *migratory detection mechanism* proposed in [25] in the context of the write-invalidate policy. Blocks detected as migratory are handled differently than other blocks; instead of a read miss request followed by an invalidation request, the read miss and the invalidation requests are merged resulting in only one request, a *read-exclusive* request.

In **II**, I modify the migratory detection mechanism from [25] to detect migratory data blocks under an update-based coherence policy. The modified detection mechanism works as follows. When a processor wants to modify a block, updates are sent to all other caches with a copy. However, these updates are tagged to indicate that the block potentially is migratory. Each cache that receives a tagged update decides locally whether it can agree that the block is migratory or not. If all caches agree that the block is migratory, the block is deemed migratory and is handled with read-exclusive requests as proposed in [25].

For one application, I detected a problem with the detection mechanism. When two processors share the same block but read and write different variables in it, referred to in the literature as *false sharing*, the detection mechanism classifies the block as migratory. However, in this situation it turns out to be better to resort to the competitive-update policy. Therefore, the detection mechanism is extended to only classify a block as migratory if it has migrated among at least three caches. Further, to reclassify a block when it stops to be migratory, a mechanism is included which detects when a block starts to be read-only instead of migratory.

In **II**, a performance evaluation of an adaptive update-based protocol with the modified migratory detection mechanism incorporated is carried out. The results show that the adaptive protocol can significantly reduce the cache miss rate by up to 71% and the network traffic by up to 26% as compared to a write-invalidate protocol.

3 The Scalable Tree Protocol — An Alternative Directory Organization

In the baseline protocol the directory is implemented with bit vectors [4]. One bit vector is associated with each memory block and each bit vector is N bits long, given that the system has N caches. Such a directory is called a *full-map* directory. Unfortunately, the implementation cost of the directory becomes very high for large-scale systems since the length of the bit vectors increases proportionally with the number of caches. In Section 3.1, I discuss two alternative directory organizations, and then in Section 3.2, evaluate the implementation cost and performance of the different organizations.

3.1 A Novel Directory Organization

To reduce the cost of the directory, one possibility is to distribute the directory. One such proposal is the Scalable Coherent Interface (SCI), which is an approved IEEE standard [18]. In the SCI, the caches sharing the same block are organized in a double-linked list. Unfortunately, the time it takes to traverse and invalidate or update the list of caches upon a processor write is O(n), where n is the number of caches in the list, i.e., the write latency is proportional to the number of caches sharing the block. If many caches share the same block, the write latency can become prohibitive.

In **III**, I propose a new directory-based cache coherence protocol, called *Scalable Tree Protocol* (STP), with an alternative directory organization. When designing the protocol, I followed two guidelines; the directory should have an organization with a low implementation cost, i.e., a low memory overhead, and coherence actions must be handled efficiently. To achieve a low implementation cost, the directory is distributed among the caches as it is in the SCI. In contrast to the SCI, however, the caches sharing the same block are organized in a tree structure instead of a linear list. By organizing the caches in a tree, the write latency becomes O(log n) instead of O(n), where n is the number of caches in the tree.

To achieve low read latencies, data is supplied from memory for non-exclusive blocks. Further, linking the cache into the tree is done after the cache has received data and in parallel with computation. The caches are linked into the tree at the leaf-level of the tree, and each level is filled before caches are linked into a new level. Thus, a balanced tree is obtained and a logarithmic write latency is achieved. When a cache does a block replacement, an important issue to address in a tree-based protocol is how to maintain a balanced tree, even if the cache doing the replacement is in the middle of the tree. I solve the problem by moving a cache from the leaf-level in the tree to the place in the tree where the cache doing the block replacement resides, thus a logarithmic write latency is always obtained.

3.2 Evaluation of Different Directory Organizations

Although the STP is a promising cache coherence approach for large-scale multiprocessors, it is important to evaluate its implementation cost and performance as compared to the SCI and the baseline protocol. The implementation cost is evaluated in **III** and the performance in **IV**.

I show in **III**, that the implementation cost of the STP is three pointers of length $\log_2 N$ bits for each memory block and three plus K pointers, also of length $\log_2 N$ bits, for each cache block, where N is the number of caches in the system and K is the fan-out in the tree. Thus, the STP has significantly lower implementation cost than a full-map protocol. The SCI meets the goal of a low implementation cost for the directory since one pointer of size $\log_2 N$ bits, where N is the number of caches in the system, is associated with each memory block. In addition, the SCI associates two pointers of size $\log_2 N$ bits with each cache block in order to build the double-linked list. As a result, the SCI has slightly lower implementation cost than the STP.

The performance results in **IV** show that the read stall time is the same for all three directory organizations. The write latency for the STP and the SCI is competitive with the write latency for the full-map protocol when only a few processors share the same block. However, when many processors share a block, the SCI suffers from long write latencies while the STP is still competitive with the full-map protocol. In **IV**, I do not evaluate the impact of block replacements on the performance. Since replacements take more time in the STP than the other two protocols, the STP is expected to suffer more than the others if replacement misses dominate over coherence misses.

4 Design and Performance of a Software-Only Directory Protocol

In the baseline protocol, a *directory controller* implemented in hardware manages the directory. However, the complexity of a hardware implemented controller requires a large design effort to engineer it correctly, leading to a high implementation cost. In addition, a hardware implemented controller does not provide any flexibility in its design. To address these issues, some research projects have suggested to migrate the controller functionality from hardware to software [6, 16, 17, 23]. While these proposals only have migrated parts of the directory controller to software or have included a special protocol processor, **V-VII** evaluate a design alternative where the controller functionality is implemented by software handlers executed on the compute processor and the directory is allocated in main memory. Such a protocol is referred to as a *software-only directory protocol* [6]. A software-only directory protocol has lower performance than the baseline protocol, but **V-VII** propose several ways to increase the performance of such protocols.

4.1 Basic Properties of Software-Only Directory Protocols

To build intuition into the performance problems associated with software-only directory protocols, I describe how a read miss to a memory block that is exclusive in one processor's cache is handled. The protocol actions and latencies involved for a read miss to an exclusive block are shown in Figure 1. When a processor experiences a cache miss, a miss request is sent over the network to the node where the block is allocated (1). When the miss request arrives at the memory, the directory controller checks whether the memory has an up-to-date copy (2). In this scenario, the block is exclusive in another cache, so the controller sends a write-back request to the cache with the exclusive copy (3). The cache with the exclusive copy sends a fresh copy to the memory (4 and 5). When the block is written back to memory, the directory controller updates the directory (6) and forwards a copy to the requesting cache (7). Finally, the requesting cache loads the block and the processor continues its execution (8). As a result, a read miss incurs the latency of four network hops (T_{net}) plus the latency of two interactions with the directory controller (T_{sw1} and T_{sw2}) plus the time to retrieve the block from the remote cache (T_C).

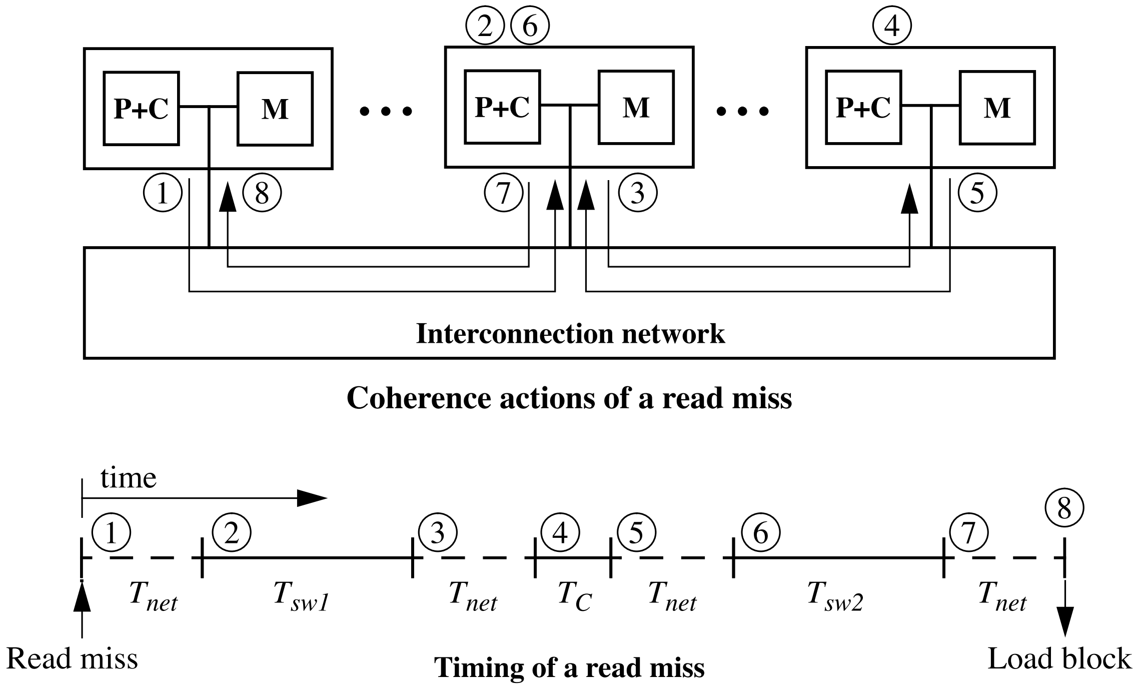

Figure 1: Coherence actions and the timing of a read miss to an exclusive memory block.

In a software-only directory protocol, each interaction with the directory controller causes an interrupt on the compute processor. In **V**, I show that these interrupts can have a devastating effect on the performance since the time it takes to execute software handlers, i.e., T_{sw1} and T_{sw2}, is included in the memory access latency of a read miss or write request. If the number of handler invocations is reduced or if they are handled in parallel with the message transfer over the network, T_{sw1} and T_{sw2} can be removed from the memory access latency. Among the objectives in **V** and **VI** is to find out whether it is possible to reduce, remove, or hide T_{sw1} and T_{sw2} from the read or write latency seen by the requesting processor.

In **V**, I describe the basic functionality of a processor node to enable software handler execution on the compute processors. This functionality includes a fast mechanism for switching from execution of application code to software handler execution, buffering of requests from other nodes that require handler invocations, e.g., read miss and write requests, buffering of software handler initiated messages such as write acknowledgments, and mechanisms to ensure forward progress of the application program.

4.2 Efficient Handling of Read Misses and Architectural Support in the Processor Node

In **V**, a strategy is proposed that removes or hides T_{sw1} and T_{sw2} from the read miss latency. This strategy includes forwarding of requests to blocks that are exclusive without handler invocation, and the ability to supply data to the requesting cache before interrupting the compute processor. To accomplish this, the hardware needs access to the state of a block, and efficient data transfer mechanisms so the compute processor can handle the directory at a slower pace in the background. Using this strategy, the read latency in the software-only directory protocol is shown to be virtually the same as in the baseline protocol. However, the handler latency is more difficult to remove from the write latency since a directory lookup must be done in order to know which copies to send invalidations to. Note that even though the handler latency can be removed from the memory access latency, it still burdens the compute processor.

In **V**, read misses to up-to-date blocks stored in the local memory, referred to as local misses, involve software handler invocations. This is a severe performance limitation since read misses to for example private data should not incur any software handler invocation. Therefore, one of the objectives in **VI** is to remove the handler latency for local misses. I show that handler invocations can be avoided for local misses by not keeping track of whether the local cache has a copy of a memory block or not. Instead, the system relies on snooping mechanisms within the processor node to maintain coherence.

Another objective in **VI** is to identify the necessary architectural support for software-only directory protocols. To address this issue, I show in **VI** a possible organization of a processor node with support for efficient handling of read misses, i.e., a node supporting the strategies proposed in **V** and **VI**. Further, I also identify and discuss several deadlock situations present in software-only directory protocols. Even though some situations are unique to software-only directory protocols, there exists published solutions to these problems.

A third objective in **VI** is to evaluate the performance effects of caching the directory information when a software handler is executed. If there exists locality in the directory accesses, caching the directory is expected to increase the performance. On the other hand, when a block with directory information is cached a block needed by the application program may be evicted from the cache, possibly resulting in a performance loss. The results presented in **VI** show that there is little interference between application data and directory data, and that caching the directory information consistently results in better performance.

4.3 Effects of Latency Tolerating and Reducing Techniques

In software-only directory protocols, as well as in hardware-based directory protocols such as the baseline protocol, the performance is often limited by processor stall times resulting from memory access latencies. To address this problem, techniques to either reduce or tolerate these latencies have been proposed and evaluated in the context of hardware-based directory protocols [1, 7, 8, 12, 15, 22, 25]. One of the objectives in **VII** is to evaluate the effectiveness of such techniques in the context of software-only directory protocols.

For software-only directory protocols, the total execution time is prolonged not only by processor stall times, but also by protocol execution overhead resulting from the handler invocations. Even though T_{sw1} and T_{sw2} do not affect the memory access latency seen by a requesting processor, they can still burden the compute processor in the node where a memory block is allocated. Unfortunately, the results in **V** indicate that there is very limited possibilities to overlap this protocol execution overhead, referred to as *p-time*, by other processor stall times. As a result, the effectiveness of latency tolerating or reducing techniques in software-only directory protocols depends on how they impact p-time. They can be divided into three classes depending on whether p-time *increases*, *decreases*, or is *invariant* when a technique is applied. As representatives for the three classes I study adaptive sequential prefetching [7], a migratory optimization technique [25], and release consistency [12], respectively.

Adaptive sequential prefetching, which has been shown effective for the baseline protocol [7], uses special prefetch requests to bring a block into the cache prior to its use, thus reducing the read stall time. Unfortunately, some prefetches are useless and do not reduce the read stall time, e.g., because the prediction of which block to prefetch is not perfect. The results presented in **VII** indicate that this can be a severe problem since each useless prefetch increases p-time without reducing the read stall time. In **VII**, I find that adaptive sequential prefetching often *degrades* the performance of software-only directory protocols due to a too high number of useless prefetches, which increases p-time so much that the read stall time reduction is outweighed. Therefore, larger attention must be turned to the number useless prefetches when choosing a prefetch scheme for software-only directory protocols than for a hardware-based protocol. An alternative to prefetching is to increase the block size. The results in **VII** indicate that a larger block size has a potential to reduce both the read stall time and p-time at the same time. However, the performance of software-only directory protocols is more sensitive to the block size choice than the baseline protocol.

The migratory optimization technique reduces the number global write requests, and thus reduces both p-time and the write stall time. As a result, the total execution time can be reduced relatively more for software-only directory protocols than for the baseline protocol. In **VII**, I show that the performance of the software-only directory protocol increases as compared to the baseline protocol for three of four applications. In general, software-only directory protocols gain relatively more than the baseline protocol when techniques that reduce p-time are applied. This fact is confirmed both with the migratory optimization technique and when the block size is varied.

Finally, the results in **VII** show that release consistency manages to hide all write latency for both the software-only directory and the baseline protocols, virtually without affecting the protocol execution overhead. However, since the protocol execution overhead is unaffected, it becomes a relatively larger part of the total execution time under release consistency. As a result, for three of our applications, the execution time ratio between the software-only directory and the baseline protocols increases when release consistency is applied. In total, **VII** shows that the efficiency of a latency tolerating technique not only depends on how well it attacks the latency as seen by the requesting processor, but also on how it interacts with the node where a memory block is allocated. As a result, more attention is needed when choosing the appropriate latency tolerating technique for software-only directory protocols than for the baseline protocol.

Acknowledgments

First, I would like to thank my supervisor Professor Per Stenström for his guidance through the work presented in this thesis, his enthusiastic support, and for all valuable discussions we have had. From discussions with graduate students at other departments, I understand that Per's commitment has been far above standard.

Next, I would like to thank Professor Lars Philipson for his continuous support during my time as a graduate student. I also thank all my colleagues at the department, especially the members of the Cachemire group: Mats Brorsson, Fredrik Dahlgren, Magnus Karlsson, and Jonas Skeppstedt. They have all provided a stimulating and creative atmosphere during my years at the department.

I also would like to thank Professor Michel Dubois from University of Southern California with whom I wrote one of the papers in this thesis. It was a very inspiring experience to work with him.

Finally, but definitely not least, I would like to thank my wife Marie for all her love and encouragement during my graduate studies. She has supported me in innumerable ways all these years, and without her inspiration this thesis probably would have taken much longer time to accomplish. Thank you Marie, we did this together!

This work was supported by the Swedish National Board for Industrial and Technical Development (Nutek) under the contract numbers 9001797 and P855.

References

[1] S.V. Adve and M.D. Hill, “Weak Ordering-A New Definition,” In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 2-14, May 1990.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, “An Evaluation of Directory Schemes for Cache Coherence,” In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 280-289, May 1988.

[3] 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.

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

[5] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal, “Directory-Based Cache Coherence in Large-Scale Multiprocessors,” *IEEE Computer*, 23(6):49-58, June 1990.

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

[7] 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.

[8] M. Dubois and C. Scheurich, “Memory Access Dependencies in Shared Memory Multiprocessors,” *IEEE Transactions on Software Engineering*, 16(6):660-674, June 1990.

[9] 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).

[10] S. Eggers and R. Katz, “A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation,” In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373-382, May 1988.

[11] S. Eggers and R. Katz, “Evaluating the Performance of Four Snooping Cache Coherency Protocols,” In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2-15, May 1989.

[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] J. R. Goodman, “Using Cache Memory to Reduce Processor-Memory Traffic,” In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124-131, June 1983.

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

[15] 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.

[16] 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.

[17] 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.

[18] IEEE. IEEE - P1596 Draft Document, Scalable Coherent Interface Draft 2.0, March 1992.

[19] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator "Competitive Snoopy Caching," In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 244-254, 1986.

[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] L. Monier and P. Sindhu, "The Architecture of the Dragon," In *Proceedings of the 30th IEEE Computer Society International Conference*, pages 118-121, 1985.

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

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

[24] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, 23(6):12-24, June 1990.

[25] 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.