Reducing Ownership Overhead for Load-Store Sequences in Cache-Coherent Multiprocessors

Jim Nilsson and Fredrik Dahlgren†

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
j@ce.chalmers.se

† *Ericsson Mobile Communications AB*
Mobile Phones and Terminals
SE-221 83 Lund, Sweden
fredrik.dahlgren@ecs.ericsson.se

Abstract

Parallel programs that modify shared data in a cache-coherent multiprocessor with a write-invalidate coherence protocol create ownership overhead in the form of ownership acquisitions at writes to shared data. This can have a significant impact on performance in a cache-coherent non-uniform memory architecture (NUMA) multiprocessor. By combining a read-request and an ownership acquisition, the write latency and network traffic can potentially be reduced. In this paper we propose a new hardware-based approach for performing this optimization by targeting load-store *sequences, which we show is a super-set of migratory sharing. A load-store sequence consists of a global read request followed by a global write action to the same memory location from the same processor, without any interveaning access to the same block from any other processor.*

We use detailed simulation with four benchmark programs including one on-line transaction processing (OLTP) workload and operating system execution to examine the effectiveness of the proposed technique. The results show that the technique is able to reduce write-related latency and network traffic more than previous hardware-based techniques, up to twice as much.

1 Introduction

Parallel programs that manipulate shared data on cache-coherent shared-memory multiprocessors with write-invalidate coherence protocols, create ownership overhead in the form of invalidations that can have a serious impact on performance. Such overhead can be the result of migratory sharing, shown by Gupta and Weber [4] to be the most prominent contributor to single invalidations for scientific applications in cache-coherent multiprocessors. Migratory sharing arise when shared data is modified in turn by several processors. Several techniques have been proposed to reduce the impact of ownership overhead in general [5, 6, 8, 12, 15] and to improve the performance of migratory sharing in particular [2, 16]. All techniques have in common that they combine a read request with an ownership acquisition for a memory block, making it possible to perform upcoming writes to the block locally in the cache, and thereby reducing coherence overhead.

Stenström et al. [16] and Cox et al. [2] independently presented a hardware-based solution to eliminate invalidation actions due to migratory sharing. Skeppstedt and Stenström [15] proposed a general compiler approach to reduce ownership overhead by identifying loads in the instruction-stream that were followed by a store to the same address. Furthermore, Kaxiras and Goodman [5], and Nilsson and Dahlgren [12], independently explored a similar load-exclusive technique implemented in hardware. In [12], Nilsson and Dahlgren showed these previous techniques to be only moderately effective for a transaction processing workload. The reasons for this were that only a fraction of all invalidations in this workload was due to migratory sharing and that the loads and the stores in the instruction stream were generally farther apart, leading to lower coverage for the static techniques.

This paper presents an extension to current write-invalidate cache-coherence protocols that reduce the write latency as well as network traffic for invalidations resulting from *load-store sequences*. A load-store sequence is a global read action followed by a global write action from the same processor to the same memory location, and is a super-set of migratory sharing. Specifically, migratory sharing techniques fail to detect single load-store sequences to uncached memory blocks. Loads of a memory block deemed load-store renders an exclusive copy in the requesting processor's cache, enabling the subsequent write to be performed locally, with no invalidations being sent. In order to allow for changing access behavior over time, the proposed technique includes a mechanism to de-tag a block if it ceases to be accessed by load-store sequences.

To understand the effectiveness of the technique, hereafter called *LS*, we have performed detailed simulations of several benchmark applications and compared the results with a previously proposed technique targeting migratory sharing [16]

(named *AD* further on). Our baseline system architecture is a sequentially consistent full-map directory-based multiprocessor with a write-invalidate cache-coherence protocol. We use the SimICS/sun4m program-driven simulation platform [9] to simulate the multiprocessor architecture.

The results show that *LS* is better than *AD* in reducing write stall time as well as network traffic for all applications. For some applications, the difference is significant. For OLTP, execution times were reduced by 13-14% and traffic by 14-15% for a range of cache configurations.

We continue to describe the concept of load-store sequences and how previous techniques deal with them in Section 2, and we introduce the protocol addition in more detail in Section 3. In Section 4, we present the experimental setup and results are presented in Section 5. Sections 6 and 7 relate this study to previous work and conclude this paper.

2 Background

Write-invalidate is a commonly used cache-coherence protocol in multiprocessors. In such a protocol on a NUMA multiprocessor, a processor has to acquire ownership for a memory block before it can be modified. Therefore, each memory block is designated a *home* node, where the global state information for that block is located. Information on which processors that currently hold a copy of the block is contained in the home of a memory block. At a write instruction to a shared block, all other copies of the block are invalidated, making the writing processor the only owner of the block. If implementing a sequential consistency (SC) memory model, the writing processor often has to stall a major part of the time while waiting for the invalidation transactions to complete. Processor stall time and traffic due to ownership overhead can be a significant contributor to poor performance for parallel programs on a large-scale multiprocessor.

Ownership overhead associated with modifications of shared data involves the ownership acquisition to the home node, individual invalidations sent to sharing remote nodes, and various acknowledgement replies of these invalidations to comply with the memory consistency model.

Read-modify-writes of a variable, e.g. A = A + 1, by a processor not currently caching a copy of the corresponding memory block thus involves two global memory actions in a cache-coherent multiprocessor with a write-invalidate coherence protocol. We denote the actions at the memory of such a read-modify-write sequence of a datum, a global read request followed by an ownership acquisition of the same block by the same processor, a *load-store sequence*.

If it could be known already at the load that a store to the same memory block is soon to follow, an obvious optimization is to combine the read request with the ownership acquisition, thereby reducing network traffic and write stall time by allowing the upcoming write to be performed locally in the cache. So how can we perform such an ownership-overhead reducing combination?

2.1 Previous Approaches

Cox and Fowler [2], and Stenström et al. [16], independently proposed an extension to existing write-invalidate protocols to efficiently handle migratory sharing. Migratory sharing arises when two or more processors in turn perform load-store sequences on the same piece of data. Even though successful for scientific/engineering applications, the same level of success did not show up for an OLTP workload as discovered in our previous study [12]. The reason for this is that the OLTP workload did not produce the nice migratory behavior needed for these techniques to correctly detect and tag the accessed memory blocks. Specifically, many cache misses for capacity and conflict reasons mitigated the effectiveness of the techniques when the needed sequences for detection failed to appear.

Being aware of the conditions needed to detect migratory sharing, we conclude that migratory sharing is only a subset of all possible load-store sequences. Obviously, since data accessed in a load-store sequence does not necessarily have to migrate between several processors, there exists an opportunity to eliminate more ownership overhead than that produced by migratory sharing. We have previously found [12] that only about half of all load-store sequences could be attributed to migratory sharing. With this in mind, and assuming the same baseline write-invalidate protocol as in the above studies, we present in the next section a protocol extension that optimizes load-store sequences per se, and not just in the shape of migratory sharing.

3 The Load-Store Protocol Extension

We propose a technique that optimizes load-store sequences in the way described in the previous section. It works as follows. To decide whether a load is part of a load-store sequence, we look at accesses to the home node of every memory block. If a global read action from a processor to a memory block is followed by a global write action from the same processor to the same memory block, without any interveaning accesses from another processor, that block is tagged load-store (*LS*).

The technique should adapt to the fact that data is accessed differently during the execution of a program, by allowing dynamic tagging and de-tagging of individual memory blocks. Otherwise, an increase in the number of read misses is expected if cache blocks that would have benefitted from being read-shared are unnecessarily invalidated. Furthermore, a block is also de-tagged when the home node receives a write request from a processor not holding a copy of the block in its cache. Next, we will present an example implementation of the protocol extension.

3.1 Example Implementation

In this section we present an example implementation of the *LS* optimization technique described in the previous section.

First, the ability to detect whether a global write action to a

particular memory block originates from the same processor that previously read the block is needed in order to correctly tag a block as *LS*. To detect this event, we need a *last-reader* (LR) bit-field, wide enough to hold an identifier of the node that last read the block. The width of this bit-field is $\log_2 N$ bits, where N is the number of nodes in the system. One such bit-field is needed for every memory block in the system and it is preferably stored close to the memory block itself, for example in the directory memory. When the home node receives an ownership request, its memory controller compares the source of the request message with the LR field. If they are equal, the requested block is tagged *LS*. In order to designate a block as *LS*, a one-bit flag is needed per memory block. We call this bit the *LS*-bit. If this bit is one, the block is treated *LS*, and if it is zero, it is handled in the normal write-invalidate fashion. The LR bit-field is always updated by the requestor node number at every global read access to the block. Looking at the complexity added for this protocol extension, it is clear that it is equal to the complexity added by previous migratory sharing techniques [2, 16].

As stated above, we want to achieve adaptability to changing access behavior for individual memory blocks. To do this, we need one extra temporary cache state, denoted *LStemp*. When the home node services a read request for a block that is tagged *LS*, the resulting state for that block in the requestor's cache is *LStemp*. The block remains in this state until one of three things happen:

1. The block is written to by the owning processor. The state of the cache block is then changed to *Modified*.

2. The block is read or written by another (foreign) processor. This means that the block was not accessed in a load-store fashion, and it is therefore de-tagged. As a result of a foreign read, the block changes status to *Shared* and both the requesting node as well as the home node receives an updated copy of the block. The home of the block is notified through a *NotLS* message that the block has ceased to be *LS*.

3. Due to capacity or conflict reasons, the block gets evicted or replaced from the cache before it is written to. In this case, the memory keeps the current *LS*-bit value of the block.

A finite state diagram of the protocol is showed in Figure 1. This diagram shows how state transitions take place based on coherence events to individual memory blocks at the home node. Compared to the coherence scheme of the DASH multiprocessor [7], an additional state, *Load-Store*, is used to determine if a load should return an exclusive copy or not. In Figure 1, *Repl* denotes the transition taken when the block is replaced from the only cache holding a copy.

The expected advantages of this technique as compared to previous techniques for migratory sharing are the following. First, when data is accessed in a load-store fashion without migrating between different processors. One such situation is when a block is accessed by the same processor multiple

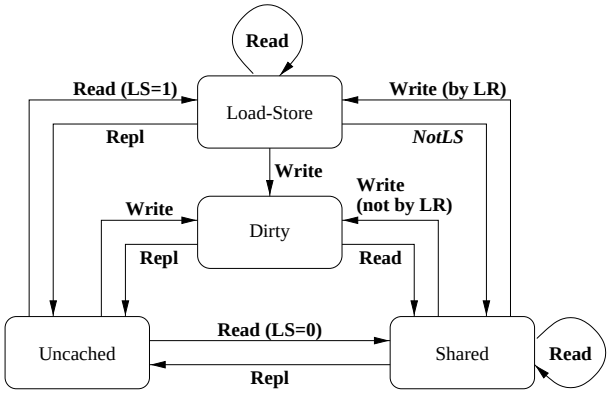

Figure 1: Memory state diagram of the new protocol

times but is replaced from the cache for conflict or capacity reasons inbetween. Second, if a block actually do migrate, but is replaced from the owning processor's cache before being accessed by a load-store sequence by another processor. This can severly limit the amount of ownership overhead that can be removed with previous techniques. The technique presented in this paper has a potential to eliminate this overhead. Furthermore, the advantages compared to static (compiler-based) methods are better coverage and avoidance of alias problems [12].

4 Experimental Methodology

For our simulations, we use the SimICS/sun4m [9] program-driven simulator. In addition, we use a memory system simulator that models the target memory system including all latencies. Every memory access produced by the workload (application, system libraries, as well as operating system) is sent to the memory system simulator which handles the access according to the contents and behavior of the caches. We model processor stall according to the behavior and latencies of the memory components, so a realistic interleaving of execution between the different processors can be maintained.

4.1 Workload

To evaluate the technique in detail as well as compare its performance to previous techniques for reducing ownership overhead, we explored its performance on several parallel benchmarks. In order to relate the obtained numbers to previous results for techniques optimizing for migratory sharing, we evaluated the MP3D application from the SPLASH [14] benchmark suite, and Cholesky and LU from the SPLASH-2 [19] benchmark suite of applications. MP3D was run for 10 time steps with 10 k particles, Cholesky with the tk15.0 configuration, and LU with a 256x256 matrix.

Perhaps of greatest interest is the susceptibility of a commercial workload to load-store optimizations. We evaluated transaction processing workload in the context of the load-store optimization technique. The transaction processing workload is the publicly available database management sys-

tem (DBMS) MySQL [17], running on top of the also publicly available operating system SparcLinux 2.0.35 operating system. Together with MySQL, we use the posix-threads package from glibc 2.0.7. The workload for MySQL is based on the TPC-B benchmark with 40 branches, equivalent to approximately 600 MB of database data.

With the exception of MySQL, all applications are executed directly on SimICS using its built-in Solaris emulation. MySQL is however totally dependent on operating system support and is therefore run on top of SparcLinux. All applications were compiled with gcc version egcs-2.91.66 and with optimizations (-O2) turned on.

4.2 Architectural Model

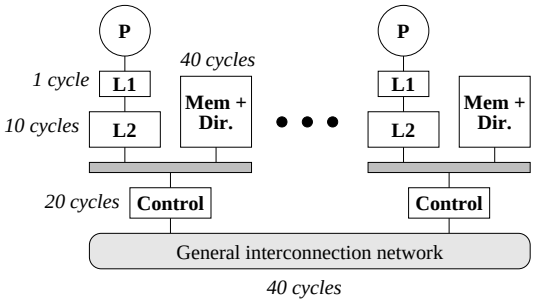

Figure 2: Simulated multiprocessor architecture with various latencies

In our experiments, we have simulated a baseline hardware architecture which is a cache-coherent, full-mapped directory-based, write-invalidate shared-memory multiprocessor with four nodes. A schematic view of the architecture with various latencies is shown in Figure 2. Every node contains a processor, a two-level cache hierarchy, a portion of the global memory, and a memory controller. The memory controller holds directory information as well as the network interface. The baseline protocol is similar to the protocol found in the DASH multiprocessor [7] with the exception that four network hops are needed to satisfy a read-on-dirty miss in our implementation. Physical memory pages are distributed in round-robin fashion among the nodes. The cache hierarchy attached to a node can have arbitrary configuration. The processor nodes are connected in a point-to-point network with a fixed delay. Contention is accurately modeled in the network.

In Table 1 we list cache parameters and the latencies of various activities of the architecture. Note that all results from the different cache configurations are not listed in this paper and a variation analysis have been made for all applications. The baseline cache configuration used for all applications except OLTP is a direct-mapped first-level cache of 4 kB and a direct-mapped second-level cache of 64 kB, with a block size of 16 bytes. The main reason for using these cache parameters is to facilitate a fair comparison between the proposed technique and previous techniques for reducing ownership overhead. For OLTP, we found that the working-set fitted well into a two-way set-associative first-level cache of 64 kB together with a direct-mapped second-level cache of 512 kB.

Cache parameter	Value(s)
L1 access time	1 cycle
L1 size	4/16/32/64 kB
L1 associativity	1/2
L1 block size	16/32/64/128
L2 access time	10 cycles
L2 size	64/512/1024/2048 kB
L2 associativity	1
L2 block size	16/32/64/128
Memory latency	**Cycles**
Memory access time	40
Network traversal	40
Memory controller	20
Local access	100
Home access	220
Remote access	420

Table 1: Cache parameters and memory system latencies used in the simulations

For larger second level cache sizes, OLTP did not experience a significant increase in performance. The default block size for OLTP is 32 bytes. The system implements a sequential consistency memory model and the processors stall on every second level cache miss, both reads and writes.

5 Results

This section presents the results of the simulations of one SPLASH application (MP3D), two SPLASH-2 applications (Cholesky, LU), and one online transaction processing (OLTP) workload. For all applications, we evaluated two techniques. The dynamic migratory optimization technique (*AD*) described in Section 2.1, and the load-store optimization technique (*LS*) presented in Section 3.1. These techniques are compared with *Baseline*, which uses the normal write-invalidate protocol.

In sections 5.1 through 5.4, we present the simulation results for the applications. The impact of changing default behavior for memory blocks is treated in Section 5.5 deals with the variation of system parameters.

5.1 MP3D

MP3D is a particle-based wind-tunnel simulator with well-known memory access behavior and a large amount of migratory sharing [4]. It is therefore highly interesting to study the techniques ability to reduce invalidation overhead for this application.

In Figure 3, the execution time, network traffic, and global read misses, respectively, are shown for MP3D with the baseline cache configuration. Execution time is divided into busy time, read stall, and write stall. Traffic is split up into two main categories; read- and write-related messages. The third category is *Other*, which includes, e.g., retry messages in case of an ongoing state change for a memory block. Global read misses are sub-divided into four groups, each denoting a separate state at the home node for the corresponding block. A

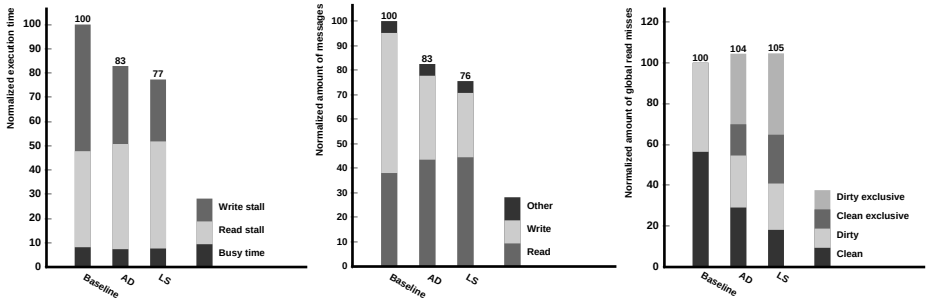

Figure 3: Behavior of MP3D

block can be either *Clean* or *Dirty*, depending on whether the home has a current copy of the block or if it is modified in a remote cache. *Exclusive* denotes a block which is either tagged migratory or load-store, depending on the technique.

As seen in the leftmost diagram, *LS* reduces the execution time somewhat more than *AD*, which is also true for total traffic, depicted in the middle diagram of Figure 3. There are two reasons why *LS* performs slightly better than *AD*. First, *LS* is able to capture all migratory accesses and tagging these, bringing the coverage level of migratory sharing accesses for *LS* to the level achieved by *AD*. Second, although the working set of MP3D fits well in a cache of size 64 kB, MP3D still experiences a non-negligable amount of conflict and capacity misses which negatively affects the migratory detection of *AD*.

No significant increase in read misses is seen for MP3D, except for a single cache configuration. This is further explored in Section 5.5. This indicates that *LS* is fairly stable and that it does not introduce a significant amount of incorrect taggings.

5.2 Cholesky

As seen in the leftmost diagram of Figure 4, *LS* successfully removes practically all ownership overhead, reducing execution time with 30%.

The ownership requests of Cholesky running on four processors with a 64 kB second level cache size do not produce any detectable amount of single invalidations, suggesting that we have little or no migration of data between the processors. In fact, single invalidations from migratory behavior only appears to a noticable degree at 16 processors or more. Invalidation traffic for Cholesky is shown for 4, 16, and 32 processors in Figure 5. In this figure, *Global Inv's* denote ownership acquisitions, i.e., global write actions to a block that is in state *Shared* in the local cache. *Invalidations* are the invalidation messages destined for caching processors, created by the coherence protocol at the home node. For four processors, ownership requests dominate the invalidation traffic. Hence the inability of *AD* to remove any write-related overhead. For 16 and 32 processors, invalidations account for 16% and 29%, respectively, of the total ownership overhead. Consequently, *AD* is increasingly successful in reducing this overhead for 16 and 32 processors. With a large number of processors, the task queue of Cholesky experiences an increased amount of contention, resulting in single invalidations created when the task queue migrates from one processor to another. As the fraction of migratory accesses of all load-store sequences increased with the number of processors, *AD* closed in on *LS* in reduction of both write stall time and total traffic. When running Cholesky with 32 processors, the execution time was reduced by 3% for *AD* and 6% for *LS*.

As Cholesky has virtually no migrating data objects at four processors, it is interesting to see the large impact of *LS* on the performance of Cholesky. The substantial decrease in write traffic is attributed to the fact that *LS* manages to eliminate the ownership request for data that is evicted from the cache for conflict or capacity reasons. At larger cache sizes, with fewer replacements, the ability of *LS* to reduce more ownership overhead than *AD* decreases. This is an example of a memory access behavior where almost all write actions are to blocks that are accessed in load-store sequences, but are not migratory. *AD* therefore lacks the ability to remove this overhead and thus fails to reduce write stall time or traffic at four processors.

5.3 LU

LU performs decompositions of dense matrices and does not contain any migratory data. Intuitively, LU should not benefit considerably from a technique such as *AD*. Looking at the leftmost diagram of Figure 6, however, we see that this statement is only supported to some extent, in that *AD* is capable of reducing write stall times with about 50%. This is equivalent to a 6% reduction in total execution time. We found that the discrepancy between the results presented here and those reported by Stenström et al. [16], arises because of a false sharing effect that was much smaller in their simulated system with 32 processors. Different processors in turn perform load-store sequences to individual parts of a memory block, thus creating an illusion of migratory behavior.

Interestingly, in the middle diagram of Figure 6, we see that *AD* manages to eliminate about half of the write stall time for LU. *LS* then reduces the write stall time with another 35%, leaving a mere 15% of the original write stall time. This is equivalent to a 20% drop in total traffic. With only about 1% increase in global read misses (rightmost diagram of Figure 6), LU experiences a 16% drop in total execution time.

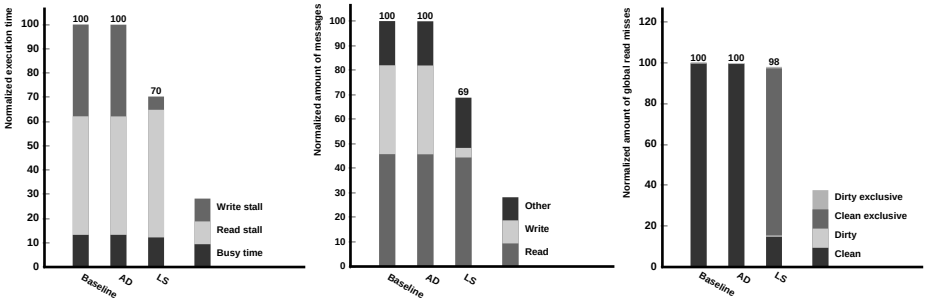

Figure 4: Behavior of Cholesky

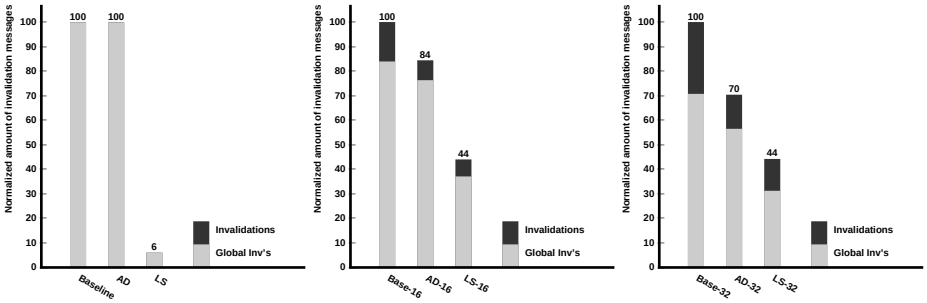

Figure 5: Invalidation traffic for Cholesky at 4, 16, and 32 processors

Again this shows how *LS* can eliminate ownership acquisitions that *AD* can not.

5.4 OLTP

In our previous study [12], we showed that *AD* has difficulties in reducing ownership-related overhead for OLTP by the same amount as for scientific applications. One of the main reasons for this is that database workloads have a significantly different system behavior than scientific workloads [10]. Specifically, OLTP workloads exhibit a large amount of conflict as well as capacity misses. The main reason for this is a substantially higher degree of cache misses to shared data [1]. We therefore expect *LS* to perform better for this type of workload.

The three diagrams in Figure 7 show execution time, total network traffic, and global read misses, respectively, for the OLTP workload. The leftmost diagram shows that *LS* cuts execution time with 13% (*AD* cuts execution time by 5%), while the middle diagram shows the decrease in traffic, which is about 15% for *LS* while only 6% for *AD*.

OLTP experiences about 1.4 invalidations on average per write to a shared block, indicating that writes to blocks which are read-shared between several processors are not uncommon. This is a difficult situation to handle for the detection mechanisms, and an increase in global read misses is expected due to an increased amount of faulty taggings. This is also shown in the rightmost diagram of Figure 7 with a 8% increase in global read misses.

As seen in the leftmost diagram of Figure 7, the OLTP workload experiences a significant reduction in busy time for both *AD* and *LS*. For *LS*, this reduction is accounted for by 49% less time spent in pthread critical sections. The remaining portion of reduced busy time is attributed to operating-system scheduling effects, where spin-lock loops on non-contended locks play a key role. These locks benefit from *LS* as well as *AD* in that they have a potential for completing faster.

Another important aspect is the relationship between load-store sequences from the application, the system libraries, and the operating system. The existence of load-store sequences in these three parts of the workload were very similar and is summarized in Table 2. The first line of Table 2 shows the fraction of all global write actions that are due to load-store sequences. The second line shows how many of the load-store sequences in the workload that are actually due to migratory sharing. On average, 42% of all global write actions are part of uninterrupted load-store sequences. Approximately 47% of these sequences are detected by *AD* as migratory sequences. This indicates that *LS* has a potential to reduce twice as much ownership related overhead than do *AD*.

Technique	Load-Store	Migratory
LS	57.6%	100.0%
AD	31.7%	47.6%

Table 3: Coverage of *LS* and *AD* for load-store and migratory sequences for the OLTP workload

In Table 3, the ability of *LS* and *AD* to remove ownership acquisitions is shown. As can be seen, *LS* removes 57.6% of

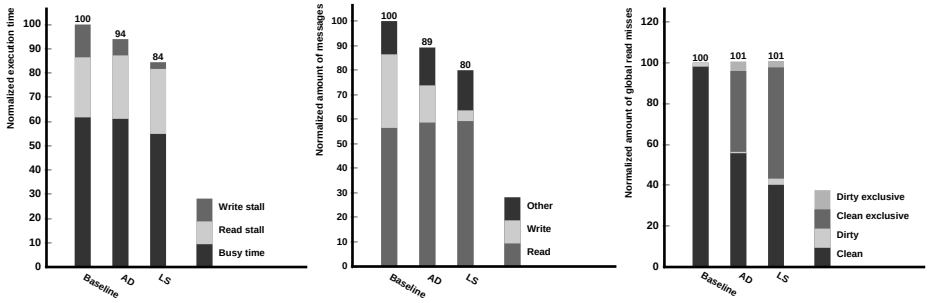

Figure 6: Behavior of LU

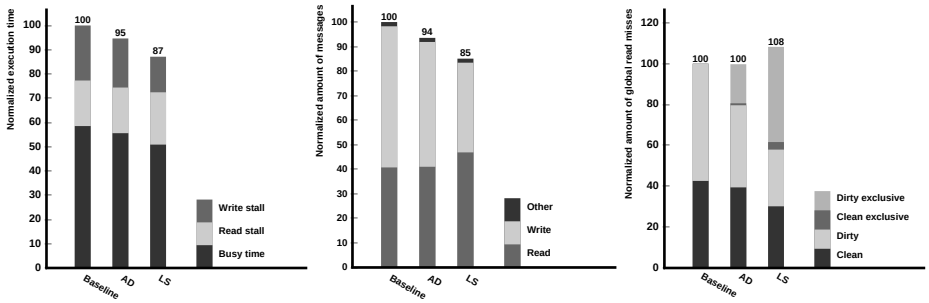

Figure 7: Behavior of OLTP

all global writes of load-store sequences, and as expected, all global writes due to migratory sequences. *AD* on the other hand is only able to remove 31.7% of the total global write actions that are part of load-store sequences. This further emphasizes *AD*'s inability to detect load-store sequences to data that is not consistently accessed in a migratory fashion. Of the global writes that are actually due to migratory access sequences, *AD* removes 47.6%, again indicating changing access behavior in the OLTP workload.

In Section 5.5, further analysis of the OLTP workload is given regarding aspects such as default tagging of memory blocks, effects of cache variations, and the possible gains achieved from introducing hysteresis in the detection algorithm for *LS*.

5.5 Variation Analysis

Private data that is accessed in load-store sequences could also benefit from being brought in exclusively at the first cold read-miss. Therefore, we evaluated the *LS* and the *AD* techniques with all memory blocks by default tagged as load-store and migratory, respectively.

We found that the application that has large amounts of migratory sharing (MP3D), benefit only little from default migratory tagging, while the other applications are not affected in a significant way.

We also evaluated a different heuristic for de-tagging blocks deemed *LS*. The heuristic was to keep the *LS*-bit value of a memory block when it was accessed by a single ownership request without being preceded by a read-request by the same processor. No noticable further improvement on performance in the form of reduction of execution time and traffic, nor an increase of read misses was experienced with these heuristics.

Table 4 shows the fraction of data misses in the OLTP workload that are due to false sharing according to the definition by Dubois et al. [3]. Clearly, the fraction of false sharing misses increases dramatically with cache block size and is substantially higher than presented for the OLTP workload studied by Barroso et al. [1]. The reduction of invalidations actually saturates at a block size of 64 bytes, diminishing the relative traffic reduction for cache block sizes of 64, 128, and 256 bytes. If proper measures were taken to reduce false sharing, we expect both the *AD* and the *LS* technique to be more successful in reducing invalidation traffic for this workload.

As a means to moderate the dynamics of the *LS* technique, it was equipped with a two step deep hysteresis, both for tagging and de-tagging, for every memory block. Such a hysteresis for tagging did however not improve performance for any application, while adding to the directory memory requirements. It was also concluded that a hysteresis on de-tagging dramatically increased the number of global read misses. This suggests that tagging and de-tagging should be done as early as possible.

6 Discussion and Related Work

Several other studies have explored both data- and instruction-centric techniques to optimize for load-store se-

Fraction of accesses	MySQL	Libraries	OS	Total
load-store of all global write actions	30.4%	25.6%	47.6%	42.0%
migratory of load-store sequences	42.9%	47.4%	51.1%	47.1%

Table 2: Occurrence of load-store sequences and migratory behavior in the OLTP workload

Block size (Bytes)	16	32	64	128	256
False sharing misses	19.9%	29.5%	37.9%	42.5%	48.5%

Table 4: Impact of cache block size on the fraction of false-sharing misses for OLTP

quences, mostly in the shape of migratory sharing. The work presented in this paper differs from these techniques in that we do not focus on migratory sharing as such, but on load-store sequences in general.

An attempt to reduce invalidation overhead for write-shared data was introduced in the Symmetry multiprocessor [8]. When a read miss is encountered to a modified block, this block is handled exclusively further on. Due to that it lacks adaptability, the probability for incurring extra read misses is increased as shown by Thakkar and Dubois [18].

Another related technique is dynamic self-invalidation by Lebeck and Wood [6]. With this technique, the cache controller tries to predict whenever a cache block is bound to be invalidated in the near future and to perform such invalidations speculatively. Ideally, this would prevent coherence actions in the form of explicit invalidation messages. The two proposed implementations of this technique are both substantially more complex and requires more memory to store state information than the implementation for the *LS* technique presented in this paper.

Load-store sequences have also been targeted by several techniques that analyze the instruction stream for loads and stores to the same memory location, rather than data being accessed in a load-store fashion. Skeppstedt and Stenström [15] proposed a compiler algorithm using dataflow analysis of a program to determine which loads were closely followed by a store to the same address. These loads were then exchanged by fictive exclusive counterparts for which the read request was combined with the ownership acquisition. This technique achieved high coverage in finding load-store sequences for the studied scientific applications. A related compiler technique with similar aim, is to insert non-binding prefetch-exclusive instructions based on locality analysis, and was introduced by Mowry in [11].

Two independent studies (Kaxiras and Goodman [5], and Nilsson and Dahlgren [12]) examined a hardware approach of detecting and tagging loads in the instruction stream that are followed by a store to the same address. While Kaxiras and Goodman not only studied migratory sharing but also other sharing behaviors such as producer-consumer sharing, their study was limited to scientific applications. The other study [12] however concluded that static instruction-centric techniques have difficulties in targeting write-related overhead for an OLTP workload. In many cases, dynamic instruction-centric techniques that needs speculative behavior to achieve performance, increased the number of read misses and network traffic.

The study in this paper has considered a conservative implementation of a sequential consistency (SC) memory model for which the proposed technique has a potential to reduce a significant portion of write stall time. Under more relaxed memory models, this reduction of write stall time is probably reduced due to these models' ability to hide remote latencies by overlapping memory references [13]. Our technique however has a potential to reduce network traffic under any memory model and thus possibly reducing memory stall time if there is much contention for the network. Another strong reason for avoiding more relaxed memory models is because the OLTP workload with MySQL and SparcLinux are to our knowledge not properly labeled to function correctly assuming a weaker memory consistency model than processor consistency, and attempting to violate this could certainly prove to be fatal in terms of correctness.

7 Conclusions

To summarize, we have presented an extension (*LS*) to existing directory-based write-invalidate cache-coherence protocols that dynamically and adaptively detects data that is accessed by load-store sequences, and optimizes access to such data by combining the read-request with the ownership acquisition. A load-store sequence consists of a global read request followed by a global write action by the same processor to the same memory location.

We employed full-system simulation with three scientific/engineering benchmarks and one OLTP workload to explore the effectiveness of the proposed technique. In order to make a fair comparison with previous techniques targeting migratory sharing, we also evaluated the adaptive protocol extension (*AD*) proposed by Stenström et al. [16] for all applications. In short, *LS* detects a global read request followed by a global write action to the same memory block from the same processor, without any interveaning accesses from other processors. For such memory blocks, further read requests renders an exclusive copy of the block in the requesting processors cache. The foremost difference between these methods is that *LS* optimizes load-store sequences in general, while *AD* only attacks migratory sharing, a sub-set of all load-store sequences. Non-migratory load-store sequences can appear

e.g., when a cached memory block of shared data is evicted from the cache for conflict or capacity reasons.

For the three scientific applications (MP3D, Cholesky, and LU), *LS* successfully removed significant amounts of write stall time and network traffic, resulting in reduced execution times by up to 30%, and a reduction of traffic by up to 31%. Furthermore, all migratory accesses are detected with *LS*, which combines the read request with the ownership request for all such sequences. Running an application such as Cholesky at four processors does not produce any migratory accesses, effectively preventing *AD* from removing any ownership overhead. *LS* however managed to reduce write-related traffic with 89% for this application. For LU, *LS* managed to not only remove the overhead related to the migratory sharing effect but in total remove 80% of the write-related stall time and 20% of the network traffic.

For the OLTP workload, having a more diverse memory access behavior and many more read misses to shared remote data than the other applications, *LS* performed significantly better than *AD*, reducing total execution times with about 15%, and total traffic with approximately 15%. The OLTP workload was subject to in average 1.4 invalidations per global write, suggesting frequent global write actions to memory blocks that are read-shared between several processors.

With its small amount of extra complexity, equal to that for *AD*, it is our belief that *LS* is a promising technique for future shared-memory multiprocessors.

Acknowledgement

We are indebted to Sun Microsystems and Virtutech (SimICS) for being instrumental in providing us with an adequate infrastructure to carry out this research. Jim Nilsson is supported by the Swedish Council for Planning and Coordination of Research with contract number 96238, and the Swedish National Board for Industrial and Technical Development (NUTEK) under grant number 97-09687. We also thank Per Stenström for his valuable discussions and comments on this paper.

References

[1] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proc. of ISCA-25*, pages 3–14, June 1998.

[2] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proc. of ISCA-20*, pages 98–108, 1993.

[3] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proc. of ISCA-20*, pages 88–97, 1993.

[4] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Trans. on Comp.*, 41(7):794–810, July 1992.

[5] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proc. of HPCA-5*, pages 161–170, Jan. 1999.

[6] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of ISCA-22*, pages 48–59, June 1995.

[7] D. E. Lenoski, J. P. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. of ISCA-17*, pages 148–159, May 1990.

[8] T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. In *Proc. of ICPP'88*, pages 303–310, Aug. 1988.

[9] P. S. Magnusson, F. Dahlgren, H. Grahn, M. Karlsson, F. Larsson, F. Lundholm, A. Moestedt, J. Nilsson, P. Stenström, and B. Werner. SimICS/sun4m: A Virtual Workstation. In *Proc. of Usenix Annual Technical Conf.*, June 1998.

[10] A. Maynard, C. Donnelly, and B. Olszewski. Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads. In *Proc. of ASPLOS-3*, pages 145–155, Oct. 1994.

[11] T. Mowry. *Tolerating Latency through Software-Controlled Prefetching*. PhD thesis, Computer Systems Laboratory, Stanford University, 1994.

[12] J. Nilsson and F. Dahlgren. Improving Performance of Load-Store Sequences for Transaction Processing Workloads on Multiprocessors. In *Proc. of ICPP'99*, pages 246–255, Sep. 1999.

[13] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proc. of SPAA-9*, June 1997.

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

[15] J. Skeppstedt and P. Stenström. Using Dataflow Analysis Techniques to Reduce Ownership Overhead in Cache Coherence Protocols. *TOPLAS*, 18(6):659–682, Nov. 1996.

[16] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proc. of ISCA-20*, pages 109–118, 1993.

[17] Detron HB TcX AB and Monty Program KB. *MySQL v3.22 Reference Manual*, Sep. 1998.

[18] S. T. Thakkar and M. Dubois, editors. *Cache and Interconnect Architectures in Multiprocessors*, chapter Performance of Symmetry multiprocessor system, pages 53–82. Kluwer Academic Publishers, 1989.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of ISCA-22*, pages 24–36, June 1995.