Conditional Memory Ordering

Christoph von Praun and Harold W. Cain
IBM T.J. Watson Research Center
Yorktown Heights, NY
{praun,tcain}@us.ibm.com

Abstract

Current techniques for memory ordering in multiprocessor systems with weakly consistent memory follow an eager push model: at a synchronization point, a processor makes its own updates to memory available to other processors by executing a memory barrier instruction, effectively pushing the values of its recent writes to other processors in the system. We argue that this model can lead to superfluous memory barriers in programs with acquire-release style synchronization. We present a new model for memory synchronization called *conditional memory ordering* that avoids this problem. The key idea is that a processor can initiate a memory synchronization at another processor and hence memory ordering can be done in a lazy manner only if necessary. We describe the design and implementation of this mechanism in the context of a PowerPC-based multiprocessor. Experience with a software prototype that implements this synchronization model shows that a significant fraction of memory ordering operations can be avoided, and that a runtime speedup for multi-threaded Java workloads with frequent locking can be achieved.

1 Introduction

Modern multiprocessor systems sometimes provide a weakly consistent view of memory to multi-threaded programs. This means that the order of memory operations performed by one or more processors in the system may appear to have occurred out of sequence with respect to the order specified by each processor's program. When communication among processors necessitates establishing a well-defined ordering of operations, memory barrier instructions must be explicitly added by the programmer to specify the ordering.

1.1 Current model

In current processor architectures, these memory barrier instructions perform ordering with a *processor-centric* view. This means that memory ordering instructions control only the processing and visibility of memory accesses of the processor that performs the memory ordering operation. This model implies, e.g., that if two processors want to communicate in a reliable producer-consumer mode, then both processors have to use appropriate memory ordering instructions. Typically, processors offer multiple variants of memory ordering instructions with different ordering guarantees. These variants are useful to tune the cost of memory ordering for processors with different roles in the synchronization (sender, receiver) [22] but do not address the principal problem of the processor-centric memory ordering mechanism.

1.2 Cost of memory synchronization

When the processor-centric mechanism of memory ordering is used for the implementation of higher level synchronization constructs like locks or barriers, a significant number of memory ordering operations occur superfluously [28]. Although this does not affect correctness, it can degrade application performance because memory ordering operations are relatively costly compared to other instructions, and limit the amount of instruction-level parallelism that may be exploited by the processor. Table 2 shows the frequency of synchronization performed by a commercial Java virtual machine while running several single and multi-threaded Java workloads, in terms of dynamic instructions executed per lock acquisition, rounded to the nearest five. Because each lock acquisition requires one memory ordering instruction at the acquire point and one memory ordering instruction at the release point, the frequency of memory ordering instructions is actually doubled, ranging from approximately 1 memory ordering operation per 80 instructions in the most frequent case to

1 per 520 instructions in the least frequent case. Also, this data does not include system-level locking, another source of locking overhead that can potentially be addressed by conditional memory ordering (although our current implementation does not address these locks). In any case, memory ordering operations occur frequently enough that their performance should not be ignored.

Table 1 shows the corresponding speedups of single-threaded programs on IBM Power4 and Power5 multiprocessor systems using a modified commercial virtual machine, where the JIT code generator omits memory barrier instructions (PowerPC *lwsync* and *isync*) when emitting code for lock and unlock operations. The programs stem from the SPEC [23] and Java Grande Forum [13] benchmarks and make frequent use of locking. Although these operations in single-threaded programs are obviously superfluous, prior work has demonstrated the potential of omitting memory barrier operations for multi-threaded applications [28]. The techniques presented in this paper leverage this potential.

benchmark	*Power4*	*Power5*
jvm98_db	1.23	1.17
jvm98_jack	1.19	1.04
jvm98_javac	1.09	1.02
jgf_monte (size B, 1 thr)	1.17	1.03
jbb (1 wh)	1.09	1.04

Table 1: Speedup of single-threaded benchmarks after removing memory barrier instructions (PowerPC *lwsync* and *isync*) in the JIT code generator.

1.3 New approach

We present the design of a hardware mechanism called *conditional memory ordering* (CMO) that reduces the cost of memory ordering by determining circumstances in which a memory ordering operation is unnecessary, and avoiding the overheads of these operations by reducing the frequency of invoking hardware memory ordering mechanisms.

CMO is not processor-centric; it allows one processor to control the ordering of operations performed by another processor in the system. The necessity of invoking memory ordering mechanisms is determined dynamically according to information about the previous memory ordering events in a multiprocessor system.

1.4 Overview

Section 2 provides background material on acquire-release synchronization and illustrates a common lock implementation including mechanisms of memory ordering that are available in current multiprocessor architectures. Section 3 introduces CMO and its programming interface on the example of locks with acquire-release memory semantics. Section 4 discusses the hardware support necessary for an efficient implementation of CMO. Finally, Section 5 reports on our experience with a software prototype based on a commercial virtual machine.

2 Acquire-release synchronization

Inter-thread synchronization mechanisms are available to a programmer in the form of locks, monitors, barriers, etc.. These constructs combine two aspects of synchronization:

***Flow synchronization*:** The control flows of all synchronizing threads meet at some synchronization point. A thread might delay or temporarily suspend execution at a synchronization point.

***Acquire-release memory ordering*:** As defined by Gharachorloo et al. [6], an *acquire* operation is necessary to correctly observe the most recent value of shared variables *after* a synchronization point. Updates to shared memory are guaranteed to be visible to other threads only after a *release* operation. A *release* operation is issued *before* a synchronization point.

Figures 1 and 2 illustrate the implementation of a mutex lock with acquire-release memory semantics (e.g., Java locks [19]). Let l be a word sized variable that holds the unique identifier of the thread holding the lock when the lock is taken and 0 otherwise. Provisions for queued waiting and a counter for reentrant acquire operations, which are typically found in lock implementations, are omitted for simplicity.

In Figure 1, lines 2 to 9 implement the aspect of flow synchronization, i.e., a thread is delayed until it finds the lock to be free and succeeds to take it in an atomic step. *sync_acquire* in line 9 implements the memory ordering aspect of the lock operation and ensures that subsequent reads correctly observe write operations of the thread that previously released the lock. Depending on the memory model and the instruction set architecture, the *sync_acquire* operation can be a *no-op* (e.g. in sequentially consistent systems), can be implied by the preceding atomic read-modify-write (e.g., Intel IA32 [12]), or is an explicit instruction (e.g. the PowerPC *isync* instruction prevents instructions following the *isync* from executing before instructions that precede the *isync*, thus satisfying this criterion [22]).

In the unlock code (Figure 2), *sync_release* guarantees that other threads can observe previous writes issued by the current thread. The synchronization model

underlying *sync_release* is a push model, i.e., a processor makes memory consistent eagerly, immediately before releasing a lock. Depending on the memory model of the architecture, the *sync_release* operation might be a *no-op* or an explicit instruction (e.g., the PowerPC *lwsync* instruction, which forces reads and writes preceding the instruction to be ordered before any writes following the instruction [22]). As argued in [28], this proactive model of memory ordering can lead to significant redundancy, e.g., when a subsequent lock operation executes on the same processor as the preceding unlock operation. In architectures that include explicit instructions for performing either *sync_acquire* or *sync_release* (e.g. Alpha, IA-64, PowerPC, SPARC RMO), these operations may have a longer latency than other instructions, causing a performance penalty with each use. Conditional memory ordering addresses this problem, by reducing the number of *sync_acquire* and *sync_release* operations that must be performed.

```
1.  tid ⇐ ⟨unique id of current thread⟩
2.  while true do
3.      atomic
4.          if l = 0 then
5.              l ⇐ tid
6.              break
7.          end if
8.      end atomic
9.  end while
10. sync_acquire
```

Figure 1: Lock operation on lock l.

```
1.  sync_release
2.  l ⇐ 0
```

Figure 2: Unlock operation on lock l.

3 Programming model of CMO

This section illustrates CMO and describes how it can be used to implement acquire-release memory semantics.

Figures 3 and 4 correspond to the lock and unlock operations in Figures 1 and 2, the model of memory synchronization is however different: the release synchronization is omitted at the unlock operation and "recovered" at the lock operation – only if necessary. Necessity is determined according to a *release number* that is communicated between the thread that unlocks l and the thread that subsequently locks l.

3.1 Release numbers

A *release number* is a word-sized value (details on the format are given in Section 4), which contains a combination of a *processor id* and a counter of the release synchronization operations (*release counter*) that the respective processor performed at a certain stage during the execution of a program. From the programmer's point of view, the value and the interpretation of the bit sequence is not of interest; the release number is treated as an opaque value that is stored in a user-level synchronization data structure to convey information about the memory consistency and synchronization state of the system from a release to the subsequent acquire point.

```
1.  tid ⇐ ⟨unique id of current thread⟩
2.  while true do
3.      atomic
4.          if THREAD(l) = 0 then
5.              relnum ⇐ RELNUM(l)
6.              THREAD(l) ⇐ tid
7.              break
8.          end if
9.      end atomic
10. end while
11. sync_conditional(relnum)
```

Figure 3: Acquire of lock l with conditional memory ordering.

```
1.  relnum ⇐ ⟨id and release ctr. of current proc.⟩
2.  atomic
3.      THREAD(l) ⇐ 0
4.      RELNUM(l) ⇐ relnum
5.  end atomic
```

Figure 4: Release of lock l with conditional memory ordering.

In the CMO programming model, the interpretation of the lockword l (again a word-sized variable) is as follows: l holds either the unique id of a thread, or a release number. The following selectors make these different fields accessible: $THREAD(l)$ refers to the unique identifier of the thread holding lock l if the lock is taken, 0 otherwise. $RELNUM(l)$ is the release number left by processor that previously released l, or 0 if the lock is taken. Note that 'previously' is well defined because lock and unlock actions on the same lock are totally ordered.

The flow synchronization aspect in the CMO variant of the lock and unlock operations is the same as in Figures 3 and 4. The focus of the discussion is hence

on the handling of the release number and the memory synchronization. The release number is provided by the processor (line 1 in Figure 4) and stored in the lockword l (line 4 in Figure 4). Note that *sync_release* is omitted at unlock. The pseudo code explicitly specifies that the update of l shall occur atomically; an implementation typically performs this unlock step in a single store operation to a word-sized variable, for which atomicity is naturally given on common processor platforms.

3.2 Conditional memory ordering

At the lock operation, after successfully passing the flow synchronization (lines 2–10 in Figure 3), the release number found in the lockword is passed as argument to *sync_conditional* in line 11. This operation is the core of CMO: based on the release number, the system arranges that release synchronization is recovered at the processor that previously released the lock, but only if necessary. Conceptually, *sync_conditional* implies *sync_acquire*.

Figure 5 shows the logic performed during the execution of the *sync_conditional* instruction. The selectors $PROCID(r)$ and $RELCTR(r)$ reflect access to the individual fields of a release number r. In the common-case (the condition in line 5 evaluates to false), no synchronization is necessary. However, in some circumstances the *sync_remote* operation must be performed on the designated remote processor (line 7). Both operations (*sync_conditional* and *sync_remote*) shall be implemented in hardware. Their logic and implementation are described in more detail in Section 4.

3.3 Self-consistency and thread-switching

We assume that a processor in the system is *self-consistent*, i.e., no memory synchronization is necessary to guarantee that a read operation on a processor can observe the value of a preceding write operation on the same processor. A software thread is an abstraction of a hardware processor, and as such it must be *self-consistent*. A thread may execute on different physical processors during its lifetime; it is assumed that memory synchronization that would be necessary due to a processor switch is applied automatically by the system (typically the operating system or hypervisor).

In the pseudo code in Figures 1 to 5, the switch of a thread to a different processor may occur at any time, however, not during the successful execution of an atomic section. *sync_conditional* and *sync_remote* shall, as they are implemented as instructions by the hardware, occur without processor switch.

As the release number encodes information that is specific to a hardware processor, the acquire and release synchronization protocol must be correct, even if the executing thread is switched to a different hardware processor during protocol execution. The key insight is that a "stale" release number will conservatively lead to memory synchronization. First, in the case of the unlock operation, a processor switch between line 1 and 2 causes a "stale" release number to be written to l. The release number is, however, still correct with respect to the protocol, because it pertains to a processor that has a consistent view of the updates that the thread previously performed on memory. A future acquire will correctly use that processor as a reference when evaluating the need to apply memory synchronization (*sync_conditional*).

```
      global relvector[⟨total number of procs⟩]

1.    procedure sync_conditional(relnum)
2.        pid ⇐ ⟨unique id of current proc⟩
3.        r_pid ⇐ PROCID(relnum)
4.        r_relctr ⇐ RELCTR(relnum)
5.        if r_pid ≠ pid then
6.            if r_relctr = relvector[r_pid] then
7.                sync_remote(r_pid)
8.            end if
9.            sync_acquire
10.       end if
11.   end procedure

      /* executes at processor pid */
12.   procedure sync_remote(pid)
13.       sync_release
14.       relvector[pid]++
15.   end procedure
```

Figure 5: Logical implementation of *sync_conditional* instruction, and remote synchronization operation, which is performed (at the designated remote processor) conditionally based on the release number value.

4 Hardware support

In this section, we describe the logical operation of hardware used in the conditional memory ordering protocol, followed by an implementation description applicable to a PowerPC-based multiprocessor.

4.1 Logical operation

Figure 5 contains pseudo-code describing the logical operations performed in hardware at the execution of a *sync_conditional* instruction. In support of the CMO mechanism, there exists a release number per processor, organized as a vector and logically shared system-

wide (labeled *relvector* in Figure 5). This vector cannot be written explicitly, but only as a side-effect of the *sync_conditional* operation. Only the release vector entry corresponding to the local processor may be explicitly read, to be embedded in the lockword when performing a lock release.

When performing an acquire operation, the lockword is read and passed to the *sync_conditional* instruction as a single register operand. When the *sync_conditional* instruction is executed, this register operand is compared to the release vector entry for the local processor (line 5). If the processor id components of the two release numbers are equal, memory ordering is unnecessary, and the instruction completes execution.

If the two processor ids are not equal, meaning the lock was most recently released at a different processor, the release counter portion of the register operand is compared (line 6) to the release vector entry for the remote processor. If the two are equal, it is possible that the remote processor may not have performed a memory ordering operation since it wrote the lockword, and therefore memory ordering may be necessary at the remote processor. If such is the case, the actions specified in the *sync_remote* portion of Figure 5 are performed.

A release counter at processor *pid* allows other processors to detect the occurrence of a *sync_release* operation on processor *pid* since the moment when processor *pid* last released a certain lock. The comparison of release counters is approximate due to their finite size. Wrap-around of a release counter value may cause some redundant remote memory ordering but does not affect the correctness of the protocol. Regardless of the outcome of the release counter comparison, a *sync_acquire* operation is performed (line 9).

4.2 Hardware implementation

An implementation of this logical design is fairly straightforward in the context of PowerPC-based multiprocessors. An additional *sync_conditional* instruction is easily supported as a new variant of the *sync* instruction (which already has three variants, and whose encoding contains sufficient bits for specifying the single register operand). In PowerPC, the *mfspr* instruction is used to copy special-purpose register values to general purpose registers; an additional variant of the *mfspr* instruction can be added to read the current value of the local release number.

The most challenging part of the implementation is maintenance of the release vector and the remote synchronization operation. Because reads of the release vector, performed during execution of the *sync_conditional* instruction, should have a low latency, a copy of the release vector is stored locally and mirrored at each processor. We use 32-bits of storage for each release number, resulting in $32n$-bits of storage per processor in an n-processor system. In a multi-threaded processor design, only a single copy of the release vector is necessary per processor (because it should appear identical at each processor), assuming write-buffers are shared among hardware threads.

Remote synchronization is conditionally performed at the execution of the *sync_conditional* instruction, causing a *sync_release* operation to be performed at the designated processor, and the release vector entry corresponding to the designated processor to be incremented in each processor's local mirrored copy of the release vector.

In order to inform the designated processor that a *sync_release* operation should be performed, a point-to-point message must be sent from the initiating processor to the designated processor. Upon reception of this message the designated processor suspends execution of memory operations until its local write queue drains (analogous to the execution of an *lwsync* instruction in PowerPC). At this point, the designated processor sends a broadcast message to every other processor in the system, indicating that the release vector entry corresponding to the designated processor should be incremented at each mirrored copy. At the reception of this message by the originating processor, a *sync_acquire* operation is performed (line 9 in Figure 5), and the *sync_conditional* operation may complete. Because remote synchronization is rarely performed per *sync_conditional* operation (as shown in Section 5), and the frequency of *sync_conditional* operations is less than the frequency of load and store misses (another source of broadcast traffic in snoop-based multiprocessors), the overall fraction of address fabric bandwidth consumed by remote synchronization will be quite small.

It should be obvious that this implementation will be backwards compatible with existing binaries because only additions have been made to the architecture; the semantics of existing instructions are unchanged. With minor modifications, it is also possible to write software that is forward compatible with CMO while remaining backwards compatible with systems that do not support CMO.

5 Software prototype

We modified a commercial Java virtual machine and JIT compiler to implement a simplified variant of CMO.

5.1 Simple CMO

The simplified CMO (S-CMO) synchronization model relies entirely on abstractions that are available in a

Java virtual machine; S-CMO does not require any hardware or OS support. The notion of a physical processor is replaced by a virtual processor, which is a simple software thread inside the VM. There is no tracking of release counters per virtual processor, i.e., the release number degenerates to the unique thread identifier.

For the protocol in Figure 5, this means that the processor id *pid* is replaced with a unique thread id. The comparison of release counters (line 6) is omitted, which is conservative, since *sync_remote* is, due to the lack of release counter information, invoked more often than actually necessary.

Summarizing, the S-CMO protocol issues a *sync_remote* operation if subsequent unlock and lock operations on the same lock are executed by different VM threads.

The implementation of remote synchronization relies on a highly efficient inter-thread signaling mechanism. Our implementation takes precautions that VM threads (virtual processors) do not "disappear" during the program execution, and that they remain responsive (e.g., if the corresponding thread performs I/O or terminates). In situations where there exist more VM threads than physical processors, the latency of a *sync_remote* operation can increase dramatically, because the receiver thread might, at the time when the operation is issued, not be scheduled for execution on a physical processor. This is a limitation of the software prototype and for such scenarios (WAS/trade6 and jbb (16wh)), we do not specify the execution times.

5.2 Experience

We use single and multi-threaded programs to evaluate the effectiveness of S-CMO, most of which are taken from the Java Grande and SPEC benchmarks suites [23, 13, 24]. hedc is the kernel of a warehouse for scientific data developed at ETH Zürich [25]. jigsaw is a http-server implementation [29] (version 2.2.4). hedc and jigsaw are not CPU-bound in the configuration that we tested, and hence we do not report speedup numbers for these programs.

Table 2, column *sync avoided [%]* reports the frequency of release synchronizations (*sync_release*) that could be omitted by S-CMO and that have no corresponding remote synchronization (*sync_remote*) for recovery. The cases correspond to redundancy through 'thread locality' in the original "push" model of memory synchronization [28]. For single-threaded executions, the S-CMO protocol eliminates all *sync_release* operations.

Columns *speedup* report the speedup obtained on Power4 (6-way) and Power5 (4-way + SMT) multiprocessor systems. To compensate for the slight variations in execution time across different runs, the speedup is computed from the average execution time of five runs in each category. For single-threaded benchmarks, the results are close to the speedups reported in Table 1. For multi-threaded programs, a naive omission of memory synchronization (methodology for single-threaded programs in Table 1), could have resulted in incorrect program behavior on the multiprocessor systems we used. Not so for the software prototype that selectively omits and recovers memory synchronization according to the S-CMO protocol. As the benchmarks frequently perform lock operations, the speedups are significant. The improvement on Power4 is more pronounced than on Power5 – which is consistent with our observation that memory barrier instructions on Power4 are relatively more expensive than on Power5.

The software prototype provides a conservative estimation, i.e., a lower bound, on the performance gains that can be achieved by a full CMO implementation in hardware. First, the addition of the release counting mechanism to S-CMO would increase the number of opportunities for omitting the release synchronization. Secondly, the software implementation involves software overheads due the additional instructions in the fast-path of the lock protocol and the cost of the thread-thread signaling mechanism that can be significantly higher than a hardware-based inter-processor communication (*sync_remote*). Furthermore, this data reflects the performance benefits from CMO being applied solely to the lock implementations used within VM, ignoring potential performance benefits from utilizing CMO within other parts of the system.

6 Related work

Prior work related to the optimization of memory ordering instructions exists in both software optimizations that minimize the frequency of memory ordering operations in the dynamic instruction stream, and hardware implementations that minimize the per-operation cost of memory ordering.

6.1 Hardware proposals

Adve and Hill present a counter-based memory ordering instruction implementation, which stalls instruction processing only while outstanding cache misses exist [1]. This mechanism was implemented in the AlphaServer GS320 [7], augmented with support for early detection of the ordering of writes. Because these implementations rely on the presence of a centralized ordering point within the memory system, the cost of each memory barrier is relatively low, whereas the completely asynchronous nature of the interconnect in IBM pSeries sys-

benchmark	*sync frequency [instr/sync]*	*sync avoided [%]*	*speedup*	
			Power4	*Power5*
single-threaded				
jvm98_db	165	100.0	1.22	1.16
jvm98_jack	405	100.0	1.19	1.03
jvm98_javac	530	100.0	1.06	1.01
jgf_monte (size B, 1 thr)	435	100.0	1.15	1.01
jbb (1 wh)	430	100.0	1.09	1.01
multi-threaded				
jvm98_mtrt	915	99.9	1.01	1.01
jgf_monte (size B, 4 thr)	450	99.8	1.11	1.02
jigsaw	1045	84.8	-	-
hedc	590	97.2	-	-
jbb (4 wh)	440	99.7	1.03	1.02
jbb (16 wh)	385	99.7	n/a	n/a
WAS/trade6	760	74.5	n/a	n/a

Table 2: Frequency of synchronization, fraction of avoided memory ordering operations, and speedup through S-CMO software prototype. Values 'n/a' reflect a limitation of the software prototype (see text).

tems requires the occasional broadcast of memory ordering operations to properly flush various memory system queues.

There has already been significant work within IBM to reduce the performance impact of memory ordering instructions, reflected in a number of patents. Arimilli et al. describe system implementations that avoid the broadcast of memory ordering operations in situations where it can be inferred that it is unnecessary based on the lack of cache misses by the local processor [4] or the lack of external coherence requests observed from the interconnect [3]. Guthrie et al. describe a read-set tracking implementation that allows instructions after a memory ordering instruction to be speculatively executed before prior memory references have been ordered [8]. In the event that the speculation is incorrect, the instructions after the memory ordering instruction are squashed and re-executed. This implementation is similar to the original speculative ordering implementation described by Gharachorloo et al. in the context of sequentially consistent systems [5]. Guthrie at al. also describe variants of the speculative protocol that will reduce the cost of ordering when there are multiple ordering instructions simultaneously in the pipeline. If all of the coherence permissions for the cache blocks touched between the two memory ordering operations can be obtained before the first memory ordering operation completes, the second may be treated as a no-op [9, 10]. Despite this work, memory ordering instructions can still be a performance hindrance in PowerPC-based multiprocessors, as shown in Section 1.

6.2 Software proposals

Shasha and Snir [21] developed an algorithm that minimizes the number of memory ordering operations that are required in a program execution to guarantee sequential consistency [18]. The technique is based on the static analysis of execution traces. The theory developed by Shasha and Snir is the foundation of several static program analyses and transformation techniques [16, 27, 26] that strive to achieve sequential consistency through barrier insertion at compile-time. Such analysis is conservative and typically adds overhead to the program execution. Whereas these techniques aim to strengthen the shared memory available to the programmer, the goal of CMO is to optimize and reduce the cost of the known acquire-release memory synchronization protocol using a pure runtime technique.

Others have developed dynamic techniques to reduce the overhead of locking in the context of the Java programming language [14, 15, 20]. The principal idea is to reserve a lock for a particular thread and then allow that reserving thread to acquire and release the lock with a highly efficient lock protocol. This work focuses on reducing the frequency of atomic read-modify-write operations, and the proposals differ in adaptivity of the protocol to changes the reservation status of a lock. CMO's goal differs in that it strives to reduce the cost of memory ordering operations, not atomic operations. The idea to convey reservation information from a release point to a subsequent acquire for the purpose of optimization is similar to the techniques in [14, 15, 20]. Although CMO can be implemented to optimize the operation of locking in software, it is general enough to optimize memory synchronization associated with other synchronization mechanisms (e.g., barriers or volatile variables [19]) and can be implemented entirely in hardware.

Logical clocks have been extensively used in the distributed systems community. Lamport [17] and Schwarz et al. [11] developed conceptual models for capturing a partial order of events in distributed systems. The release vector that CMO keeps in each processor

can be understood as a logical clock according to these models, where events correspond to memory accesses (read and update). This logical clock (i.e. the release vector in each processor) allows a program executing at processor i to determine if its view on a certain part of shared memory is current with respect to updates done by some (other) processor j. [17] and [11] provide a very general theory that is useful to define a notion of ordering and consistency in a parallel system, which is useful to define and reason about the correctness of CMO itself. [17] and [11] are however not mechanisms for shared memory per se.

TreadMarks [2] is a distributed software shared memory system that implements a weak consistency model called Lazy Release Consistency (LRC). In LRC, the propagation of memory updates after a synchronization point is deferred. At a synchronization point (barrier), only a digest of updates is broadcast in terms of so called write notices for memory pages that have been modified. While the principle of deferring memory synchronization is similar to CMO, TreadMarks is a software system and is designed to operate in a distributed multi-computer environment, not, like CMO, in a closely coupled multiprocessor. Also, write notices are recorded for selected parts of the memory (page granularity), whereas release numbers in CMO reflect the consistency of the entire memory updated by a processor.

7 Conclusions

We have developed a new model of memory synchronization in multiprocessor systems called conditional memory ordering. The new model avoids significant redundancy in the memory synchronization and therefore better matches the requirements of software with frequent acquire-release synchronization. Although the latency of initiating synchronization on a remote processor may be higher than the latency of conventional *sync_acquire* and *sync_release* operation, because remote synchronization is rarely necessary, CMO can significantly improve the performance of multiprocessor systems. We demonstrate the opportunity and actual runtime speedups with a software prototype. In future work, we plan to perform a detailed performance evaluation of our proposed hardware implementation of conditional memory ordering in the context of a full-system multiprocessor simulator.

Although this work has described and evaluated CMO's benefits in an acquire/release lock implementation, we also believe that it will be beneficial to the performance of other shared-memory synchronization primitives. We plan to explore its applicability to these primitives in future work.

Acknowledgments

We thank Calin Cascaval, Jong-Deok Choi, Manish Gupta, Pratap Pattnaik, and Kyung Ryu for discussions and their detailed and insightful comments.

References

[1] S. V. Adve and M. D. Hill. Weak ordering - a new definition. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, June 1990.

[2] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[3] R. Arimilli, J. Dodson, D. Williams, and J. Lewis. Demand based sync bus operation. US Patent 6065086, May 2000.

[4] R. Arimilli, J. Dodson, D. Williams, and J. Lewis. Demand based sync bus operation. US Patent 6175930, January 2001.

[5] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. of the 1991 Intl. Conf. on Parallel Processing*, pages 355–364, August 1991.

[6] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Intl. Symposium on Computer Architecture*, pages 15–26, 1990.

[7] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and design of AlphaServer GS 320. In *Proc. of the Ninth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[8] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Multiprocessor speculation mechanism via a barrier speculation flag. US Patent 6691220, February 2002.

[9] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Multiprocessor speculation mechanism for efficiently managing multiple barrier operations. US Patent 6625660, September 2003.

[10] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Mechanism for folding storage barrier operations in a multiprocessor system. US Patent 6725340, April 2004.

[11] Intel Corporation. IA-32 Intel architecture software developer's manual, volume 3: System programming guide. http://developer.intel.com/design/pentiumIV/manuals, 2005.

[12] JGF. Java Grande Forum multi-threaded benchmark suite, 1999.

[13] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 292–310, Nov. 2002.

[14] T. O. K. Kawachiya and A. Koseki. Lock reservation for java reconsidered. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'04)*, pages 560–584, June 2004.

[15] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Journal of Parallel and Distributed Computing*, 38(2):130–144, Nov. 1996.

[16] L. Lamport. Time, clock and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[17] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.

[18] J. Manson, W. Pugh, and S. Adve. The java memory model. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'05)*, pages 378–391, 2005.

[19] T. Ogasawara, H. Komatsu, and T. Nakatani. To-lock: Removing lock overhead using the owners' temporal locality. In *Proceedings of the Conference on Parallel Architectures and Compilation Techniques (PACT'04)*, pages 255–266, Oct. 2004.

[20] R. Schwarz and F. Mattern. Detecting causal relationships in dirstibuted computations: In search of the holy grail. *Journal of Distributed Computing*, 7(3):149–174, 1994.

[21] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.

[22] E. Silha, C. May, and B. Frey. PowerPC User Instruction Set Architecture (Book I), 2003.

[23] SPEC. Standard Performance Evaluation Corporation - SPECjvm98, 1998.

[24] SPEC. Standard Performance Evaluation Corporation - SPECjbb2000, 2000.

[25] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific data repositories – designing for a moving target. In *Proceedings on the International Conference on Management of Data and Symposium on Principles of Database Systems (SIGMOD/PODS'03)*, pages 349–360, June 2003.

[26] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler techniques for high performance sequentially consistent java programs. In *Proceedings of the Symposium Principles and Practice of Parallel Programming (PPoPP'05)*, pages 2–13, June 2005.

[27] Z. Sura, C.-L. Wong, X. Fang, J. Lee, S. Midkiff, and D. Padua. Automatic implementation of programming language consistency models. In *Record of the Workshop on Compilers for Parallel Computers (CPC'03)*, Jan. 2003.

[28] C. von Praun. Deconstructing redundant memory synchronization. In *Proceedings of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD'05)*, June 2005.

[29] W3C. World wide web consortium: Jigsaw - W3C's web server. http://www.w3.org/Jigsaw, 1998.