

Conditional Memory Ordering

Christoph von Praun, Harold W. Cain, Jong-Deok Choi, Kyung Dong Ryu

IBM T.J. Watson Research Center
Yorktown Heights, NY 10598, USA
{praun, tcain, jdchoi, kryu}@us.ibm.com

Abstract

Conventional relaxed memory ordering techniques follow a proactive model: at a synchronization point, a processor makes its own updates to memory available to other processors by executing a memory barrier instruction, ensuring that recent writes have been ordered with respect to other processors in the system. We show that this model leads to superfluous memory barriers in programs with acquire-release style synchronization, and present a combined hardware/software synchronization mechanism called conditional memory ordering (CMO) that reduces memory ordering overhead. CMO is demonstrated on a lock algorithm that identifies those dynamic lock/unlock operations for which memory ordering is unnecessary, and speculatively omits the associated memory ordering instructions. When ordering is required, this algorithm relies on a hardware mechanism for initiating a memory ordering operation on another processor.

Based on evaluation using a software-only CMO prototype, we show that CMO avoids memory ordering operations for the vast majority of dynamic acquire and release operations across a set of multithreaded Java workloads, leading to significant speedups for many. However, performance improvements in the software prototype are hindered by the high cost of remote memory ordering. Using empirical data, we construct an analytical model demonstrating the benefits of a combined hardware-software implementation.

1 Introduction

Modern multiprocessor systems sometimes provide a weakly consistent view of memory to multi-threaded programs, in which 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.

In current processor architectures, these memory barrier instructions perform ordering with a *processor-centric* view, i.e., memory ordering instructions control only the processing and visibility of memory accesses for 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) but do not address the principal problem of the processor-centric memory ordering mechanism.

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 [35]. Although this does not affect correctness, it can degrade application performance because memory ordering operations are relatively costly compared to other instructions. Prior work has demonstrated efficient memory consistency implementations that use aggressive speculation combined with large amounts of buffering [10, 28], however the cost of buffering and speculation verification hardware can be prohibitively expensive.

We present the design of a combined hardware/software mechanism called *conditional memory ordering* (CMO) that improves performance 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 a new memory consistency model, but a new programming interface and implementation that can be used to optimize synchronization algorithms in existing memory models.

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 previous memory ordering events in a multiprocessor system.

This paper makes the following contributions:

- We present a detailed characterization of synchronization and memory ordering operations in lock intensive Java workloads and demonstrate that a large majority of memory ordering operations occur redundantly.
- We describe a programming interface, called conditional memory ordering (CMO), that eliminates virtually all inefficiencies due to redundant memory ordering operations.
- We describe a possible hardware implementation of CMO.
- We evaluate the performance potential of CMO and its hardware implementation using a software prototype and an analytical model.

2 Characterizing the redundancies of memory ordering in conventional locking algorithms

In this section, we characterize the locking behavior of a set of Java applications and describe and measure several circumstances in which memory ordering is redundant. We begin with some background material on locking protocols under various memory models.

```

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 variable l .

```

1.  sync_release
2.  l ← 0

```

Figure 2. Unlock operation on lock variable l .

2.1 Memory ordering in conventional lock implementations

Figures 1 and 2 show implementations of lock acquisition and the lock release operations. Due to potential re-ordering of memory operations by the processor or memory system, an *acquire* operation is necessary in many architectures 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 [8].

Let l be a lock variable that contains 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 lock operations, which are often included in lock implementations, are omitted for simplicity.

In Figure 1, lines 1 to 9 implement a lock operation, 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 10 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 [15]), or is an explicit instruction (e.g. the PowerPC *isync* instruction, which prevents instructions following the *isync* from executing before instructions that precede the *isync*, thus satisfying this criterion [31]).

In Figure 2, line 2 implements the unlock operation, and *sync_release* guarantees that other threads will 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 [31]). 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. CMO addresses this problem, by reducing the number of *sync_acquire* and *sync_release* operations that must be performed.

2.2 Experimental methodology

Our characterization and evaluation of CMO uses a set of single and multi-threaded Java benchmarks from the Java

<i>benchmark</i>	<i>comments</i>
<i>single-threaded</i>	
jvm98_db	SPECjvm98 benchmark
jvm98_jack	” ”
jvm98_javac	” ”
jgf_monte (B, 1 th)	Java Grande Forum benchmark
jbb2000 (1 wh)	SPECjbb2000 benchmark, 120s warmup, 120s steady-state, one warehouse
<i>multi-threaded</i>	
jvm98_mtrt	
jgf_monte (B, 4 th)	
jigsaw	2500 random HTTP requests to Jigsaw http server v.2.2.4
hedc	kernel of a warehouse for scientific data [32]
jbb2000 (4 wh)	SPECjbb2000 benchmark, 120s warmup, 120s steady-state, four warehouses
jbb2000 (16 wh)	” ”, 16 warehouses
WAS/trade6	internal IBM benchmark, 3-tier setup of WAS 6.0 and DB2 v.8.01 with 240s warmup and 240s steady-state
WAS/jas	SPECjAppServer2004, “ “

Table 1. Benchmark characteristics

Grande [16] and SPEC benchmark suites selected for their high frequency of lock operations. Lock operations are particularly frequent in Java programs due to the language’s *synchronized* keyword. We run these applications on IBM’s J9 production virtual machine [11], a state-of-the-art JVM that already aggressively reduces locking frequency using lock coarsening [33] and a limited form of escape analysis [6].

Our experiments are performed on both Power4 and Power5 multiprocessor systems running AIX, with four and six processors respectively. We present data for previous generation Power4-based systems because the higher cost of memory ordering instructions on this system may be used to project performance improvement on other types of systems (e.g. directory-based systems) where the costs of memory ordering instructions may be higher. Table 1 presents our benchmark suite with setup parameters.

Because *hedc* and *jigsaw* are not CPU-bound in the configuration that we tested, we do not report any performance improvement for these application, however they provide additional evidence of the locking redundancies that are identified and exploited by CMO.

2.3 Frequency of memory ordering

The overheads caused by memory ordering instructions are a function of their execution frequency. In this subsection, we quantify the frequency and performance costs of these instructions for a set of Java applications.

Table 2 shows the frequency of lock acquire operations performed by a commercial Java virtual machine when executing the benchmarks programs; this data includes locking that is performed on behalf of the benchmark program and

also internal locking by the virtual machine. Notice that the frequency of memory ordering operations is at least twice the locking frequency, as every lock acquire is followed by a corresponding lock release operations. In addition, explicit memory ordering can also occur with module initialization, recompilation, garbage collection, and volatile variable access. We don’t account for these incidents in the table.

Column *usr* specifies the fraction of instructions that are executed in user mode. The characterization that follows refers only to locking that occurs in this mode. Column *freq* reports the number of lock operations that occur per thousand user mode instructions, [% -total] specifies how much this locking contributes to the overall (user + system) locking in the system. Most benchmarks spend a large majority of their execution time in user mode, yet for some programs, a significant fraction of total lock events occurs in system mode. This indicates that the frequency of locking in the AIX kernel is higher than our set of applications. Although this work focuses on lock optimization at the JVM level, there may also be opportunity using CMO in kernel lock routines.

Column *contended* characterizes lock usage patterns and specifies how many of 1000 lock acquire operations find a lock being owned or waited on by another thread. In single-threaded programs, such a situation can never occur (*jbb2000 (1 wh)* is not entirely single threaded). We observe that even for multi-threaded programs, this situation is rather rare but not uncommon for well-optimized server programs with high-degrees of concurrency (*jigsaw*, *WAS/trade6*, *WAS/jas*).

The J9 virtual machine implements bimodal locks, where the lock implementation adapts itself to the usage pattern [27]. Acquire and release on uncontended locks – the common case according to Table 2 – require only a few instructions, including memory ordering operations, which can contribute significantly to the overall cost. For some percentage of contended locks (reported in the *inflated* column), excessive contentions cause them to be *inflated*. Lock and unlock operations on inflated locks are more costly and hence memory ordering operations constitute only a small fraction of the total cost.

2.4 Sources of memory ordering redundancy

In an implementation of the acquire-release model that executes memory ordering operations at every acquire and release point (Section 2.1), some of the ordering operations can occur redundantly in a specific execution. The design of CMO is motivated by different classes of redundancy that are defined and quantified in our earlier work [35] – we give a brief recap in the following paragraphs.

<i>benchmark</i>	<i>usr</i> [%]	<i>freq</i> [%instr]	<i>frac</i> [%-total]	<i>contended</i> [%-user]	<i>inflated</i> [%-contended]
<i>single-threaded</i>					
jvm98_db	97.3	4.7	96.4	0	-
jvm98_jack	92.5	1.1	66.5	0	-
jvm98_javac	96.7	1.0	92.7	0	-
jgf_monte (B, 1 th)	96.6	1.8	93.3	0	-
jbb2000 (1 wh)	98.4	1.4	96.8	< 0.01	100.0
<i>multi-threaded</i>					
jvm98_mtrt	91.2	0.1	83.6	0.01	77.8
jgf_monte (B, 4 th)	96.6	1.8	94.0	0.01	5.3
jigsaw	97.3	< 0.1	52.6	9.71	96.7
hedc	97.1	< 0.1	60.1	2.6	82.4
jbb2000 (4 wh)	99.9	1.2	99.4	< 0.01	4.9
jbb2000 (16 wh)	99.7	1.1	99.8	0.01	90.1
WAS/trade6	77.7	1.2	49.7	51.01	87.8
WAS/jas04	79.0	1.1	40.4	9.75	64.0

Table 2. Characterization of locking in user mode.

Thread-confinement of lock variables. Memory ordering that occurs for lock variables that are solely accessed by a single thread are redundant [24, 35]. Table 3, column *thr-conf* quantifies the occurrence of thread-confined synchronization for multi-threaded benchmarks. Thread-confined synchronization is mainly an idiosyncrasy of the Java programming model and its standard library implementation. A single-threaded application would report 100% redundancy. Despite the aggressive elimination of thread-confined locking by the compiler, a large fraction of thread-confined lock use and memory synchronization remains.¹

Thread locality of locking. Locality of locking is a situation where consecutive acquires of a lock variable are made by the same thread. Since no other thread is acquiring the lock in the interim, there is no communication and therefore no need to perform ordering at the first release and second acquire. Thread-confinement is a special case of thread locality, where the locality of access extends over the whole lifetime of a lock. Table 3, column *thr-loc* quantifies the locality of locking. Memory ordering operations that occur with a lock access that experiences locality are redundant [35].

Eager releases and repetitive acquires. Implementations of memory ordering operations are commonly oblivious to the lock variable *l* with which these operations occur. A result is that memory ordering instructions order memory accesses to any part of the memory, not only to variables that are accessed inside the critical section protected by *l*. This gives rise to further redundancies: a *sync_release*

operation might occur redundantly (eagerly) if the subsequent *sync_acquire* operation to that lock is separated in the happens before relation by another *sync_release* and *sync_acquire*. For example, if $unlock(x) \rightarrow unlock(y) \rightarrow lock(y) \rightarrow lock(x)$, where \rightarrow is the happens before relation, the memory ordering operations performed by $unlock(x)$ are redundant in this execution.

The dual redundancy can occur for a *sync_acquire* (repetitive acquire) that is preceded by another acquire that had already established the memory ordering [35]. Eager releases and repetitive acquires can be avoided by associating shared data with specific lock objects (as in [5]), or through logical clock systems [19], which we also use in our implementation. Table 3, column *thr-loc not e/r* quantifies the overall redundancy found in our experiments.

<i>benchmark</i>	<i>thr-conf</i>	<i>thr-loc</i>	<i>thr-loc not e/r</i>
jvm98_mtrt	99.3	99.9	99.9
jgf_monte (B, 4 th)	99.6	99.8	99.9
hedc	89.6	96.3	99.7
jigsaw	45.5	89.1	98.0
jbb2000 (4 wh)	33.5	99.7	99.9
jbb2000 (16 wh)	18.9	99.7	99.9
WAS/trade6	30.3	71.9	93.7
WAS/jas04	44.2	82.7	98.3
<i>average</i>	57.6	92.4	98.6

Table 3. Fraction of redundant memory ordering operations according to redundancy classes in percent [%].

3 Programming model of CMO

This section illustrates CMO and describes how it can be used to implement mutual exclusion using locks.

¹The methodology used for the collection of data in Table 3 was limited to the Java application itself (thus excludes locking performed internally to the JVM/GC/JIT and native libraries).

```

1.  tid  $\leftarrow$  (unique id of current thread)
2.  while true do
3.    atomic
4.      if THREAD(l) = 0 then
5.        relnum  $\leftarrow$  RELNUM(l)
6.        THREAD(l)  $\leftarrow$  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.

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 value (details on the format are given in Section 4) that reflects 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 the synchronization variable to convey information about the memory consistency and synchronization state of the system from a lock release to the subsequent lock acquire of the same lock variable.

```

1.  relnum  $\leftarrow$  (id & release ctr. of current proc)
2.  atomic
3.    THREAD(l)  $\leftarrow$  0
4.    RELNUM(l)  $\leftarrow$  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 is as follows: l holds either the unique id of a thread, or a release number. We use the following

notation when referring to the individual release number fields: *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 the processor that previously released l , or 0 if the lock is taken. Note that ‘previously’ is well defined because lock and unlock actions on a lock are totally ordered.

The flow synchronization aspect in the CMO variant of the lock and unlock operations is the same as in Figures 1 and 2. The focus of the discussion is therefore on the handling of the release number and the explicit ordering of memory accesses. 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

After successfully acquiring the lock (lines 2–10 in Figure 3), the previous release number found in the lockword is passed as an argument to the *sync_conditional* instruction 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* at the processor that issues the instruction.

Figure 5 shows the logic performed during the execution of the *sync_conditional* instruction. The notation *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, *sync_conditional* and *sync_remote*, shall be implemented in hardware. Their logic and implementation are described in more detail in Section 4.

A thread may execute on different physical processors during its lifetime due to migration; it is assumed that the operating system context switch handler includes a conventional memory ordering instruction, which is already the case for most operating systems.

Because the release number includes processor-specific information, it may not be obvious why this protocol metadata itself is consistent in the event of thread migration. The key insight is that a “stale” release number will conservatively cause memory synchronization. During an unlock operation, if the unlocking thread is migrated between executing lines 1 and 2, a “stale” release number may be

```

global relvector[(total number of procs)]

1. procedure sync_conditional(relnum)
2.   pid  $\leftarrow$  (unique id of current proc)
3.   r_pid  $\leftarrow$  PROCID(relnum)
4.   r_relctr  $\leftarrow$  RELCTR(relnum)
5.   if r_pid  $\neq$  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 conditionally performed (at the designated remote processor) based on the release number value.

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*). A thread migration during an acquire does not create a new release number, and the release number that is read does not pertain to the processor that is performing the acquire (it pertains to the processor that last released the lock), therefore it cannot be affected by migration.

Although it is possible to completely implement the protocol in software using either a polling mechanism (as is done in our software prototype described in Section 5.1), or an interrupt mechanism, these options are unwieldy due to performance overhead. Polling is inherently low-performance, as the polling itself consumes CPU time. Because the operating system must also isolate running applications from malicious code, the construction of low-overhead user-level interruption mechanisms is also difficult. In the next section, we describe hardware for efficiently supporting the *sync_conditional* instruction.

4 Hardware support for CMO

At minimum, an implementation of CMO must include support for the *sync_conditional* instruction to provide a mechanism with which one processor may initiate memory ordering at another processor. Since the only architecturally visible aspect of CMO is the addition of this single instruction, it should be obvious that CMO is backwards compatible with respect to existing software.

A very simple implementation may unconditionally send a *sync_remote* message to the designated processor, which would in turn perform the necessary memory ordering. Such an implementation would avoid the complexity of the release counter mechanism altogether. However, many applications perform remote synchronization frequently such that the cost of these operations will incur a performance penalty. The release counter mechanism (Section 4.1) provides a means of reducing the frequency of these operations.

4.1 Logical operation

Figure 5 contains pseudo-code describing the logical operations performed in hardware at the execution of a *sync_conditional* instruction. A release number is maintained per processor, organized as a vector and logically shared system-wide (labeled *relvector* in Figure 5). This vector cannot be written explicitly; it is incremented as a side-effect of a memory ordering operation. Only the release vector entry corresponding to the local processor may be explicitly read, which is stored by software 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. The *sync_conditional* instruction compares this operand to the release vector entry for the local processor (shown at line 5 of Figure 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 identifiers 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. Consequently, the actions specified in the *sync_remote* portion of Figure 5 are performed. Otherwise, it is known that the remote processor performed already a *sync_release* operation since it unlocked the lock, and *sync_remote* is therefore unnecessary.

The release counter for processor *pid* allows other processors to detect the occurrence of a *sync_release* operation

by processor *pid* since the moment that *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 Release vector support

The most challenging part of the implementation is the maintenance of the release vector, which is updated at the execution of memory ordering instructions. Reads of the release vector should have low latency (since the release vector entry is read at the execution of every *sync_conditional* instruction). To support low latency reads, a copy of the release vector is mirrored in local storage at each processor. Since updates to the release vector have fewer constraints relative to traditional coherence protocols (e.g. relaxed atomicity requirements), they can be maintained fairly simply. We believe that 32-bits of storage for each release number should be adequate, resulting in $32n$ -bits of storage per processor in an n -processor system.²

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 causing 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 is sent from the initiating processor to the designated processor. Upon reception of this message, the designated processor suspends execution of memory operations until its outstanding writes have been ordered with respect to other processors in the system (analogous to the execution of an *lwsync* instruction in PowerPC). The designated processor then 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. A similar implementation is used for maintaining the barrier synchronization register supported by Power5 systems [23].

Although this protocol depends on a broadcast operation, the additional system bandwidth consumed is extremely low, because remote synchronization is rarely per-

²Since the individual release numbers associated with each lockword are embedded in the lockword itself, this release vector represents the only storage overhead added by the CMO protocol.

formed per *sync_conditional* operation (as shown in Section 5.1). Since the frequency of *sync_conditional* operations is already much lower than the frequency of load and store misses (another source of broadcast traffic in snooper-based multiprocessors), the overall fraction of address fabric bandwidth consumed by remote synchronization will be quite small. In future work, we plan to design a more scalable solution applicable to directory-based multiprocessors.

4.3 Release hints

So far, we have described an implementation (Figure 5) where a processor increments its release counter only in response to a remote synchronization event (line 14), which we evaluate in the next section. In future work, we also plan to explore an optimization of this protocol that proactively increments release counters at points where memory is naturally ordered. For example, in most systems, write misses are naturally ordered at some point in time regardless of the existence of memory ordering instructions, as the write queue drains. By proactively updating release counters at moments when ordering occurs, subsequent stalls for memory ordering instructions may be reduced.

Release counters may be proactively incremented by issuing a *release hint* before an unlock operation (e.g. at line 1 in Figure 2) that would instruct a processor to increment its release counter as soon as the conditions of memory ordering according to a *sync_release* are met. We plan to explore the tradeoff between aggressively issuing release hints and conserving limited interconnect bandwidth, potentially using a bandwidth-adaptation mechanism similar to [25].

5 Evaluation

The CMO evaluation is split into three parts. In the first, we present the implementation of a software-only CMO prototype, which implements the *sync_remote* operation using a high-latency inter-thread signaling mechanism. Considering the cost of this signaling, the software prototype performs quite well as presented in Section 5.2, illustrating CMO's effectiveness at avoiding memory ordering operations. However, in some benchmarks the frequency of *sync_remote* operations reduces performance. In Section 5.3, we present CMO performance when including hardware support for the *sync_conditional* instruction.

5.1 S-CMO: A software CMO prototype

We have initially implemented a simplified version of CMO in the context of the IBM J9 virtual machine, which we call S-CMO. S-CMO relies entirely on abstractions that are available in a Java virtual machine; S-CMO does not require any hardware support. The notion of a physical processor is replaced by a virtual processor, which corresponds

to each software thread running inside the VM. There is no tracking of release counters per virtual processor, i.e. the release number degenerates to the unique thread identifier.

In lieu of the protocol described in Figure 5, a processor id *pid* is replaced with a unique thread id. The comparison of release counters (line 6) is omitted, which is conservative because *sync_remote* is invoked more often than actually necessary due to the lack of release counter information. The implementation of remote synchronization relies on a user-level inter-thread polling/signaling mechanism.

A *sync_remote* operation can be omitted if it targets a thread that is not currently scheduled on a CPU (e.g. if it has been context-switched out). These operations are avoidable because the operating system provides the necessary memory ordering at the time the thread is de-scheduled. The S-CMO prototype exploits this property as follows: to determine whether the destination thread of a *sync_remote* operation is currently scheduled on a processor, we use a version of AIX extended to provide information about the status of VM threads scheduled at each physical processor. We refer to this information as the *active-thread scoreboard*. The active-thread scoreboard is efficiently accessible to the VM. This scoreboard is only implemented on a version of AIX that runs on our Power5 systems; when experimenting using Power4 systems, we omit data (in Table 4) for benchmarks that have more threads than processors. The active-thread scoreboard is not necessary in a hardware CMO implementation.

Although S-CMO does not include release counters, the information provided by the active-thread scoreboard can be regarded as a conservative approximation of release counter information: the fact that a thread is not scheduled on a CPU can be interpreted as an increment of that thread’s release counter.

5.2 S-CMO performance

Table 4 reports the relative frequency of *sync_remote* operations and the speedup of execution times. We have evaluated two variants of the S-CMO prototype, one that follows the S-CMO protocol for all operations on flat locks (columns *all S-CMO*), and one that can temporarily revert to the standard memory synchronization protocol for locks that observe contention and hence are likely to cause *sync_remote* operations (columns *adaptive S-CMO*). The adaptive protocol was primarily motivated by the relatively high overhead of *sync_remote* in the software prototype. The frequency of *sync_remote* operations in the adaptive variant of the protocol is generally lower than in the *all S-CMO* case.

The frequency of *sync_remote* operations is generally low and reflects the protocol’s ability to identify the redundancies found in the conventional implementation

of acquire-release (as reported in Section 2.4). Obviously, for single-threaded executions, S-CMO eliminates all *sync_acquire* and *sync_release* operations. The right columns (*sync_remote - score [%]*) specify the frequency of operations if some *sync_remote* instances are avoided thanks to the active-thread scoreboard. This mechanism is quite effective in configurations with a larger number of threads than processors (jigsaw, hedc, jbb2000 (16 wh)). WAS-based programs benefit as well, although the effect we observed lagged behind our initial expectations – in the light of about 80 software threads that are scheduled to 8 hardware threads. The cause of this behavior is that *sync_remote* operations occur on a relatively small set of highly contended locks that each thread acquires and releases several times during one scheduling quantum. We expect a release counter implementation to further reduce the frequency of *sync_remote* operations, because it can detect redundant synchronization when threads are scheduled in addition to when they are not.

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 the best four out of five runs in each category. 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 speedups obtained with the adaptive scheme are slightly better for most benchmarks and demonstrate the significance of the overhead due to the software implementation of *sync_remote*. Notice that overheads actually cause a slowdown for the WAS/trade6. hedc and jigsaw are not CPU-bound in the configuration that we tested, and hence we observe no speedup numbers for these programs.

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. Two main factors contribute to the overhead of the software prototype: First, release counters are omitted – which would require even fewer *sync_remote* operations than reported in columns *sync_remote* and *score [%]*. Secondly, the software implementation involves software overheads due to additional instructions on the fast-path of the locking protocol, and the cost of the inter-thread signaling mechanism is significantly higher than hardware-based inter-processor communication would be. Furthermore, this data reflects the performance benefits from CMO being applied solely to the lock implementations within the J9 virtual machine, ignoring potential performance benefits from utilizing CMO within other software components.

benchmark	all S-CMO				adaptive S-CMO			
	sync_remote		speedup		sync_remote		speedup	
	[%]	score [%]	Power4	Power5	[%]	score [%]	Power4	Power5
<i>single-threaded</i>								
jvm98_db	0	0	1.18	1.11	0	0	1.19	1.11
jvm98_jack	0	0	1.15	1.08	0	0	1.12	1.07
jvm98_javac	0	0	1.05	1.01	0	0	1.06	1.02
jgf_monte (B, 1 th)	0	0	1.11	1.04	0	0	1.13	1.04
jbb2000 (1 wh)	< 0.01	< 0.01	1.12	1.06	< 0.01	< 0.01	1.14	1.06
<i>multi-threaded</i>								
jvm98_mrt	< 0.01	< 0.01	1.02	1.04	0.01	< 0.01	1.05	1.02
jgf_monte (B, 4 th)	0.20	0.15	1.10	1.01	< 0.01	< 0.01	1.12	1.00
jigsaw	9.92	1.04	-	-	5.65	0.68	-	-
hedc	3.54	0.34	-	-	2.50	0.22	-	-
jbb2000 (4 wh)	0.28	0.27	1.08	1.02	0.24	0.23	1.11	1.03
jbb2000 (16 wh)	0.29	0.11	n/a	1.02	0.25	0.08	n/a	1.02
WAS/trade6	23.64	11.16	n/a	0.83	8.90	4.06	n/a	0.89
WAS/jas04	16.65	6.67	n/a	1.00	6.61	1.00	n/a	1.01

Table 4. Percentage of lock acquire operations that led to a *sync_remote* operations and speedups observed for software prototypes of the S-CMO and adaptive S-CMO protocols. The first column labeled *sync_remote* specifies the percentage when omitting the active-thread scoreboard, the second column indicates the fraction including the active-thread scoreboard. Data in column *speedup* is obtained when including the scoreboard.

5.3 A simple analytical model

As demonstrated in the last section, CMO offers significant performance improvement for many applications, however some applications suffer due to the high cost of our software-based *sync_remote* signaling implementation. Given a hardware-based *sync_conditional* and *sync_remote* implementation, CMO performance will improve dramatically. In this section, we present a simple analytical model that predicts this improvement given various *sync_remote* latencies. Our model, shown below, predicts CMO speedup for a particular application as a function of five variables: the application’s *CPI*, the number of conventional memory ordering instructions avoided per instruction (MOI_{freq}), the average performance penalty of each conventional memory ordering instruction (MOI_{lat}), the number of *sync_remote* operations per instruction (SR_{freq}), and the average performance penalty of each *sync_remote* operation ($SR_{latency}$).

$$CMO_{speedup} = \frac{CPI}{CPI - (MOI_{freq} * MOI_{lat}) + (SR_{freq} * SR_{latency})}$$

This model is quite basic, as it simply subtracts the the expected *CPI* component due to acquire/release-time memory ordering instruction stalls, and adds a *CPI* component due to expected *sync_remote* stalls. Although there are many potential sources of variation between this simple model and an actual system (e.g. non-uniform stall latencies for memory ordering instructions), we find that the model works quite well in practice, as it successfully predicts the performance of the S-CMO prototype within a small margin of error.

We have experimentally measured *CPI*, MOI_{freq} , and SR_{freq} for each application. Figure 6 and Figure 7 present CMO speedup when using a MOI_{lat} value empirically measured on an IBM Power4-based system and an IBM Power5-based system, respectively, while varying $SR_{latency}$ across the *x*-axis. The S-CMO implementation evaluated in the prior section applied the CMO optimization to only those lock routines within the Java virtual machine, and consequently underestimates the potential performance improvement for applications in which significant locking occurs in the operating system and native libraries. Because CMO can be applied to all locks system-wide, our MOI_{freq} value is based on the frequency of system-wide locking for each application (shown in Table 2).

As shown earlier, CMO improves performance significantly more on the Power4 system than the Power5 system. However, when including a lightweight remote synchronization mechanism, CMO improves the performance of *all* applications even on the Power5 where MOI_{lat} is relatively low (28 cycles), whereas the cost of *sync_remote* degraded WAS/trade6 performance in the software prototype. As $SR_{latency}$ increases, performance of course decreases, but this decrease is very gradual for all applications other than WAS/trade6, where CMO begins to slow down the application at 700 cycles. In a real system, the latency of the average remote synchronization operation should be comparable to the latency of average cache-to-cache transfers between processors. This latency will be much less than 500 cycles for current and near-future medium-scale (4-64 processor) systems, considering the trend toward chip-

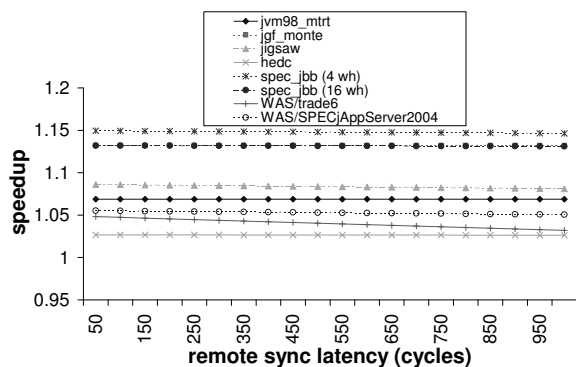


Figure 6. CMO performance while varying remote sync latency in high-cost (Power4) memory ordering implementation.

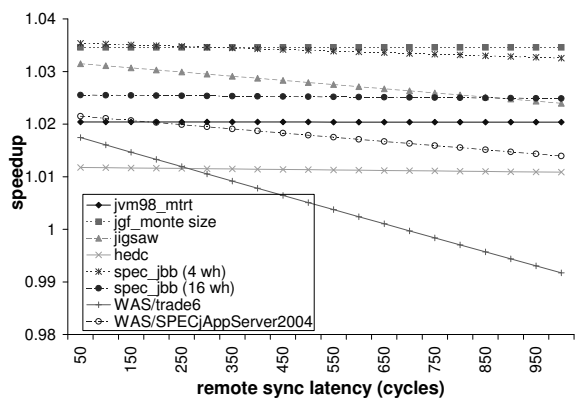


Figure 7. CMO performance while varying remote sync latency in low-cost (Power5) memory ordering implementation. (Note different y-axis scale.)

multiprocessor architectures where communication latencies are low. At a conservative 500 cycle value of $SR_{latency}$, CMO improves the performance of these applications on average 11% for high MOI_{lat} (130 cycle) systems, and 3% for low MOI_{lat} (28 cycle) systems.

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 implementa-

tions 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 [9], augmented with support for early detection of the ordering of writes. Since 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 unordered nature of the interconnect in IBM pSeries systems 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 [12]. 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 [7]. Guthrie et 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 [13, 14]. Despite this work, memory ordering instructions remain a performance hindrance in PowerPC-based multiprocessors, as shown in Section 5.2.

6.2 Software proposals

Shasha and Snir [30] developed an algorithm that minimizes the number of memory ordering operations that are required in a program execution to guarantee sequential consistency [22]. 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 [20, 34] 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 model available to the programmer, the goal of CMO is to optimize and reduce the cost of the 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 [27, 17, 18, 26]. 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. CMO's goal is different 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 [17, 18, 26]. 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 [24]) and can be implemented entirely in hardware.

Logical clocks have been extensively used in the distributed systems community. Lamport [21] and Schwarz et al. [29] 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 . [21] and [29] 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.

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 multiprocessor synchronization algorithm called conditional memory ordering, which avoids significant redundancy of memory ordering 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. With hardware support, CMO offers significant performance benefits across our set of Java benchmarks when assuming a reasonable remote synchronization latency. In future work, we plan to explore the effectiveness of CMO across a wider range of synchronization algorithms and workloads.

Acknowledgments

We thank Marc Auslander for bringing the idea of release hints to our attention. We thank Calin Cascaval, Manish Gupta, and Pratap Pattnaik 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] B. Bershad and M. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, Sept. 1991.
- [6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*, pages 1–19. ACM Press, Nov. 1999.
- [7] 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.

- [8] 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.
- [9] 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.
- [10] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Intl. Symp. on Computer Architecture*, pages 162–171, May 1999.
- [11] N. Grcevski, A. Kielstra, K. Stoodley, M. G. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, pages 151–162, May 2004.
- [12] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Multiprocessor speculation mechanism via a barrier speculation flag. US Patent 6691220, February 2002.
- [13] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Multiprocessor speculation mechanism for efficiently managing multiple barrier operations. US Patent 6625660, September 2003.
- [14] G. Guthrie, R. Arimilli, J. Dodson, and D. Williams. Mechanism for folding storage barrier operations in a multiprocessor system. US Patent 6725340, April 2004.
- [15] Intel Corporation. IA-32 Intel architecture software developer’s manual, volume 3: System programming guide. <http://developer.intel.com/design/pentiumIV/manuals>, 2005.
- [16] JGF. Java Grande Forum multi-threaded benchmark suite, 1999.
- [17] 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.
- [18] 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.
- [19] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA’92)*, pages 13–21, May 1992.
- [20] 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.
- [21] L. Lamport. Time, clock and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.
- [22] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Trans. on Computers*, 46(7):779–782, July 1997.
- [23] P. Mackerras, T. Matthews, and R. Swanberg. Operating system exploitation of the power5 system. *IBM Journal of Research and Development*, 49(4/5):533–541, 2005.
- [24] 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.
- [25] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. In *Proceedings of the Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, pages 251–262, 2002.
- [26] 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.
- [27] T. Onodera and K. Kawachiya. A study of locking with bimodal fields. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA’99)*, pages 223–237, Nov. 1999.
- [28] 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 the Ninth Symp. on Parallel algorithms and Architectures*, pages 199–210, 1997.
- [29] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Journal of Distributed Computing*, 7(3):149–174, 1994.
- [30] 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.
- [31] E. Silha, C. May, and B. Frey. PowerPC User Instruction Set Architecture (Book I), 2003.
- [32] 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.
- [33] M. G. Stoodley and V. Sundaresan. Automatically reducing repetitive synchronization with a just-in-time compiler for java. In *Proceedings of the 3rd Annual Symposium on Code Generation and Optimization*, pages 27–36, March 2005.
- [34] 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 (PpPP’05)*, pages 2–13, June 2005.
- [35] C. von Praun. Deconstructing redundant memory synchronization. In *Reader of the Workshop on Duplicating, Deconstructing, and Debunking (WDDD’05)*, June 2005.