To appear in the proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA-5), Orlando, Florida, January 9-12, 1999.

Using Lamport Clocks to Reason About Relaxed Memory Models

Anne E. Condon, Mark D. Hill, Manoj Plakal, Daniel J. Sorin
Computer Sciences Department
University of Wisconsin - Madison
{condon,markhill,plakal,sorin}@cs.wisc.edu

Abstract

Cache coherence protocols of current shared-memory multiprocessors are difficult to verify. Our previous work proposed an extension of Lamport's logical clocks for showing that multiprocessors can implement sequential consistency (SC) with an SGI Origin 2000-like directory protocol and a Sun Gigaplane-like split-transaction bus protocol. Many commercial multiprocessors, however, implement more relaxed models, such as SPARC Total Store Order (TSO), a variant of processor consistency, and Compaq (DEC) Alpha, a variant of weak consistency.

This paper applies Lamport clocks to both a TSO and an Alpha implementation. Both implementations are based on the same Sun Gigaplane-like split-transaction bus protocol we previously used, but the TSO implementation places a first-in-first-out write buffer between a processor and its cache, while the Alpha implementation uses a coalescing write buffer. Both write buffers satisfy read requests for pending writes (i.e., do bypassing) without requiring the write to be immediately written to cache. Analysis shows how to apply Lamport clocks to verify TSO and Alpha specifications at the architectural level.

Keywords: memory consistency models, cache coherence protocols, protocol verification

1 Introduction

Shared-memory multiprocessor systems are increasingly employed both as servers (for computation, databases, files, and the web) and as clients. To improve performance, multiprocessor system designers use a variety of complex and interacting optimizations. These optimizations include cache coherence via snooping or directory protocols, out-of-order processors, and coalescing write buffers. These optimizations add considerable complexity at the architectural level and even more complexity at the implementation level. Directory protocols, for example, require the system to transition from many shared copies of a block to one exclusive one. Unfortunately, this transition must be implemented with many non-atomic lower-level transitions that expose additional race conditions, buffering requirements, and forward-progress concerns. Due to this complexity, industrial product groups spend more time verifying their system than actually designing and optimizing it.

This work is supported in part by the National Science Foundation with grants MIP-9225097, MIPS-9625558, CCR 9257241, and CDA-9623632, a Wisconsin Romnes Fellowship, and donations from Sun Microsystems and Intel Corporation.

To verify a system, engineers should unambiguously define what "correct" means. For a shared-memory system, "correct" is defined by a memory consistency model. A *memory consistency model* defines for programmers the allowable behavior of hardware. A commonly-assumed memory consistency model requires a shared-memory multiprocessor to appear to software as a multiprogrammed uniprocessor. This model was formalized by Lamport as *sequential consistency* (SC) [12]. Assume that each processor executes instructions and memory operations in a dynamic execution order called *program order*. An execution is SC if there exists a total order of memory operations (reads and writes) in which (a) the program orders of all processors are respected and (b) a read returns the value of the last write (to the same address) in this order. A system is SC if it only permits SC executions.

Our previous work [18,24] proved that abstractions of a SGI Origin 2000-like [5,13] directory protocol and a Sun Gigaplane-like [22] split-transaction bus protocol both implement SC. Instead of asking for the off-line existence of a total memory order, we *pretend* to augment the hardware with logical *Lamport clocks* to construct the needed order dynamically as it executes memory operations (satisfying requirement (a)). We then prove that every load (read instruction) returns the value of the last store (write instruction) in this constructed order. Thus (b) is satisfied. As with any formal method, our Lamport clocks approach cannot replace conventional testing and validation. Nevertheless, it is our premise that Lamport clocks can be valuable when reasoning about the correctness of a specification of memory ordering semantics at the architectural level, thereby aiding in the protocol design process and reducing time spent on validation later.

While work on SC is valuable, many commercial processors implement more relaxed memory consistency models in an effort to improve performance. An example is the insertion of FIFO or coalescing write buffers between the processor and the cache. Processor consistent models, such as SPARC Total Store Order (TSO) [25], relax the SC requirement (a): now, in the total ordering of memory operations, a store (ST) can appear after a load (LD) that follows it in program order. More relaxed models, such as Compaq (DEC) Alpha [23], allow a processor great free-

dom to re-order memory operations between "memory barriers."

This paper shows that Lamport clocks can be used to verify shared-memory implementations that support the TSO and Alpha relaxed memory models. Towards this end, the paper makes two primary contributions:

1. *We provide clean new memory model definitions, namely Wisconsin TSO and Wisconsin Alpha, that aid in reasoning about correctness of protocols.* We show that protocols satisfying the Wisconsin TSO and Wisconsin Alpha memory models also satisfy TSO [25] and Alpha [23], respectively. We consider the Wisconsin memory models to be more intuitive than the original definitions for the following reasons. Unlike the TSO definition, LDs always get the values of STs that occur earlier in the total order. Unlike the Alpha definition, we use a total order.

2. *We extend our Lamport timestamping scheme to protocols for both the TSO and Alpha memory models.* The key is determining at what point in the protocol an event is timestamped, and it is in this determination that the proofs of this paper differ from our previous work on SC. For example, in the Alpha protocol, a LD that gets its value from a previous ST that is still in the write buffer should be timestamped *after* the ST. But since the ST has not yet been written to the cache, the ST is not yet timestamped when the LD is issued. Our timestamping scheme handles this simply by waiting to timestamp the LD until the ST has actually been written to the cache.

While the details of the timestamping scheme are necessarily different from previous work, a strength of our approach is that, with the timestamping scheme in hand, the proofs of correctness of the protocols are almost identical to the proofs in our previous work on SC. Our protocols for TSO and Alpha are based on the same Gigaplane-like split-transaction bus protocol that we considered in previous work [24]. A similar result could easily be proved for a directory-based implementation, as in Plakal et al. [18].

In the rest of the paper, we assume a *block* to be a fixed-size, contiguous, aligned section of memory (usually equal to the cache line size). Also, LDs and STs operate on *words*, where we assume that a word is contained in a block and is aligned at a word boundary. Our scheme could be extended to handle LDs and STs on sub-units of a word (half-words or bytes) which need not be aligned. However, this makes the specification of the memory models very tedious without any gain in insight or clarity.

The rest of this paper is organized as follows. Section 2 summarizes our previous work that used Lamport clocks to reason about the correctness of shared memory systems, and discusses related work by others. We present our results for TSO and Alpha in Sections 3 and 4, respectively. Section 5 summarizes our contributions and discusses future work.

2 Related Work[1]

2.1 Our Previous Work

Our previous work [18,24] proved that implementations using a SGI Origin 2000-like [5,13] directory protocol and a Sun Gigaplane-like [22] split-transaction bus protocol both implement SC. Both implementations use three-state invalidation-based coherence protocols.

Our reasoning method associates logical timestamps with loads, stores, and coherence events. We call our method *Lamport Clocks*, because our timestamping modestly extends the logical timestamps Lamport developed for distributed systems [11]. Lamport associated a counter with each host. The counter is incremented on local events and its value is used to timestamp outgoing messages. On message receipt, a host sets its counter to one greater than the maximum of its former time and the timestamp of the incoming message. Timestamp ties are broken with host ID. In this manner, Lamport creates a total order using these logical timestamps where causality flows with increasing logical time.

Our timestamping scheme extends Lamport's 2-tuple timestamps to three-tuples: <**global . local . node-id**>, where **global** takes precedence over **local,** and **local** takes precedence over **node-id** (e.g., 3.10.11 < 4.2.1). Coherence messages, or transactions, carry global timestamps. In addition, global timestamps order LD and ST operations relative to transactions. Local timestamps are assigned to LD and ST operations in order to preserve program order in Lamport time among operations that have the same global timestamp. They enable an unbounded number of LD/ST operations between transactions. Node-ID, the third component of a Lamport timestamp, is used as an arbitrary tie-breaker between two operations with the same global and local timestamps, thus ensuring that all LD and ST operations are totally ordered.

Our prior proofs of SC use two timestamping claims that show that LDs and STs are ordered relative to transactions "as intended by the designer." One of these claims is that for every LD and ST on a given block, proper access is ensured by the most recent transaction on that block in Lamport time. (In contrast, in real time, a processor may perform a LD on a block *after* it has answered a request to relinquish the block.) Roughly, the other claim is that, in logical time, transactions are handled by processors in the order in which they are received. (In contrast, in real time, a processor may receive transaction-related messages "out of order".)

Sequential consistency is established using the concept of *coherence epochs*. An epoch is an interval of logical time during which a node has read-only or read-write access to a block of data. The life of a block in logical time consists of

1. This section borrows from material in previous work [18,24].

a sequence of such epochs. Our proof shows that, in Lamport time, operations lie within appropriate epochs. That is, each LD lies within either a read-only or a read-write epoch, and each ST lies within a read-write epoch. In addition, the "correct" value of a block is passed from one node to another between epochs. The proofs of these results build in a modular fashion upon the timestamping claims, thereby localizing arguments based on specification details. The differences between the proofs for the bus and directory protocols differ only in the details of the timestamping claims.

2.2 Other Related Work

Our Lamport clock method complements related work on proving protocols correct. First, Lamport clocks are more precise and formal than ad hoc reasoning or simulation.

Second, we find Lamport clocks easier to use and more applicable to larger systems, but less rigorous than approaches that use state-space search of finite-state machines or theorem-proving techniques. These are rigorous methods that can capture subtle errors, but they are often limited to small systems because of the state space explosion for large, complicated systems. For example, the SGI Origin 2000 coherence protocol is verified for a 4-cluster system with one cache block [7], the memory subsystem of the Sun S3.mp cache-coherent multiprocessor system is verified for one cache block [19], and the SPARC Relaxed Memory Order (RMO) memory consistency model is verified for small test programs [16]. Park and Dill [17] propose using transaction aggregation to scale beyond finite-state methods. Our approach can precisely verify the operation of a protocol in a system consisting of any number of nodes and memory blocks.

Another formal approach devised by Shen and Arvind uses term rewriting to specify and prove the correctness of coherence protocols [21]. Their technique involves showing that a system with caches and a system without caches can simulate each other. This approach lends itself to highly succinct formal proofs. We find Lamport clocks easier to grasp, while not lacking expressive power. Term rewriting relies on an ordering of rewrite rules (each of which corresponds to an event) and, as such, may benefit from the Lamport clock technique which can order events.

Third, we find Lamport clocks easier to use and of similar formal power to many of the other methods used to define and verify relaxed memory models [1, 2, 3, 6, 8, 9, 20]. Of particular note are the approaches of Collier [3] and Gharachorloo et al. [8] that model a write as *p* sub-operations to each of *p* processors. We find their approaches more general but harder to use than our approach that splits TSO stores (writes) into two components and leaves Alpha stores atomic.

Finally, Lamport Clocks have also been used in other research, including a paper by Neiger and Toueg [15]. They describe a class of problems for which, if a clock-based algorithm is proven correct assuming real-time synchronized clocks, then it must also be correct even if run with logical clocks. One difference between this work and ours is that the protocols we consider are not clock-based. Rather, we attach (logical) clocks to clock-free protocols, in order to prove correctness of the protocols

3 Total Store Order (TSO)

SPARC Total Store Order (TSO) [25] is a variant of *processor consistency* [9,10] that has been implemented on Sun multiprocessors for many years. TSO relaxes SC in that LDs can be ordered ahead of STs which precede them in program order (so long as there are no intervening memory barriers and the two operations are to different locations). We study TSO because it is formally and publicly defined, but we expect that our results can be mapped to the Intel Architecture-32 (IA-32) memory model (Section 7.2 of [4]), the other dominant processor consistency model.

We now define TSO, Wisconsin TSO, a TSO implementation, a Lamport timestamping scheme for that implementation, and its corresponding proof.

3.1 Defining TSO

TSO applies to a system with multiple processors issuing a variety of instructions. For our purposes, we are concerned with word loads (LDs), word stores (STs) and memory barriers (MBs) issued to regular memory (i.e., excluding I/O space). We consider only memory barriers at least as strong as type "MB #StoreLoad," i.e., barriers which guarantee that all prior STs are completed before any future LD, while weaker memory barriers are regarded as no-ops (e.g., "MB #LoadLoad"). Appendix D of the SPARC Architecture Manual Version 9 [25] defines TSO by defining Relaxed Memory Order (RMO) and then adding constraints to form TSO. We give the combined result.

Let $<_p$ denote *program order*. Program order totally orders all LDs, STs, and MBs at the same processor and it is thus a partial order over all processors.

Let $<_m$ be a total ordering of all LD and ST operations.

Then $<_m$ is said to be in *total store order* (TSO) if the following constraints hold. The first two constraints are called "memory order constraints." Let X and Y be a pair of LD or ST operations.

1) If X $<_p$ Y and either X is a LD or Y is a ST, then X $<_m$ Y.

2) If X $<_p$ MB $<_p$ Y then X $<_m$ Y.

The final constraint restricts possible values of LDs:

3) Let X be a LD of word *w*. Then the value of X is the value of the greatest ST, say Y, to word *w* in memory order, taken over all STs to word *w* that either occur before X in memory order or occur before X in program order (but possibly after X in memory order).

Intuitively, constraints 1 and 2 say that memory order may only violate processor order to delay a ST after a subsequent LD when there is no intervening MB. In all other cases, memory order respects program order (i.e., LD $<_p$ LD´, LD $<_p$ ST, and ST $<_p$ ST´ are preserved by memory order). Constraint 3 says that a LD should return the last value written to the same word in memory unless there is a pending ST to the same word (earlier in program order) that has not yet occurred in memory order. In this case, the value from the pending ST should be returned. So if one looks at the memory order, it *appears* as if the LD gets its value from a ST that "happens in the future."

An execution of an implementation satisfies TSO if there exists an ordering of the LDs and STs in the execution that satisfies TSO. An implementation satisfies TSO if all executions of that implementation satisfy TSO.

3.2 Wisconsin TSO

We now define some properties of an ordering which makes verification easier. TSO's condition 3 allows a load to get a value from a "future" store. Wisconsin TSO eliminates this oddity by splitting each store into a $ST_{private}$ and a ST_{public}, both of which have the same value. Each LD gets its value from the past but may return the value of a $ST_{private}$ for which the corresponding ST_{public} has not yet occurred. The goal in this case is to model write buffer bypassing where stores enter the write buffer on a $ST_{private}$ and exit with a ST_{public}.

Let $<_w$ denote an ordering of LDs, $ST_{private}$s and ST_{public}s. We say that $<_w$ is in *Wisconsin total store order* (Wisconsin TSO) if the following conditions hold.

1') The ordering ($<_w$) of LDs and $ST_{private}$s is consistent with program order. That is, if X and Y are either a LD or a $ST_{private}$, then X $<_p$ Y if and only if X $<_w$ Y.

2') For each ST, $ST_{private} <_w ST_{public}$.

3') If X and Y are STs and X $<_p$ Y then $X_{public} <_w Y_{public}$.

4') If an MB occurs between ST and LD in program order then $ST_{public} <_w$ LD.

5') Let X be a LD of word *w* at processor p_i. Then the value of X is the value of the most recent ST to *w* in $<_w$ that is either:

a) the most recent $ST_{private}$ to word *w* at p_i, if for some ST $<_p$ X to word *w*, the corresponding ST_{public} is after X in $<_w$, or

b) the most recent ST_{public} to word *w*, otherwise.

An execution of an implementation satisfies Wisconsin TSO if there exists an ordering of the LDs, $ST_{private}$s and ST_{public} s in the execution that satisfies Wisconsin TSO. An implementation satisfies Wisconsin TSO if all executions of that implementation satisfy Wisconsin TSO.

Gil Neiger [14] has developed an alternative TSO definition as a total order of LDs and STs in which a LD always get the value of the most recent ST. This is done by moving each LD that returns a value from a $ST_{private}$ to be after the corresponding ST_{public}.

Claim 1: An implementation that satisfies Wisconsin TSO also satisfies TSO.

A proof of this claim can be found in Appendix A.[1]

3.3 TSO Implementation With FIFO Write Buffers

A common TSO implementation approach separates each processor from its cache with a FIFO write buffer. Caches are kept coherent with a write-invalidate coherence protocol sufficient for implementing SC. A MB can be implemented by having a processor flush its write buffer before proceeding past a MB, without the caches or coherence protocol ever seeing MBs. We use this approach here in a manner similar to the Sun Ultra Enterprise 6000 with UltraSPARC II processors.

We begin with a brief summary of the SC implementation that Sorin et al. [24] describe for a Gigaplane-like split-transaction bus (the overall approach would be similar for the directory-based implementation described by Plakal et al. [18]). Memory blocks may be cached as *Invalid*, *Shared*, or *Exclusive*. The *A-state* (address state) records how the block is cached and is used for responding to subsequent bus transactions. The protocol seeks to maintain the expected invariants (e.g., a block is *Exclusive* in at most one cache) and provides the usual coherence transactions: *Get-Shared* (GETS), *Get-Exclusive* (GETX), *Upgrade* (UPG, for upgrading the block from Shared to Exclusive), and *Writeback* (WB). As with the Gigaplane, coherence transactions immediately change the A-state, regardless of when the data arrives. If a processor issues a GETX transaction and then sees a GETS transaction for the same block by another processor, the processor's A-state for the block will go from Invalid to Exclusive to Shared, regardless of when it obtains the data. In an SC implementation, the processor checks the A-state of a block before executing LDs and STs on that block. On a miss, the processor ensures the

1. The converse of this claim can also be proved, but it is not necessary for our proof of correctness, and we omit it here due to space constraints.

appropriate A-state for that block by sending a coherence transaction on the bus.

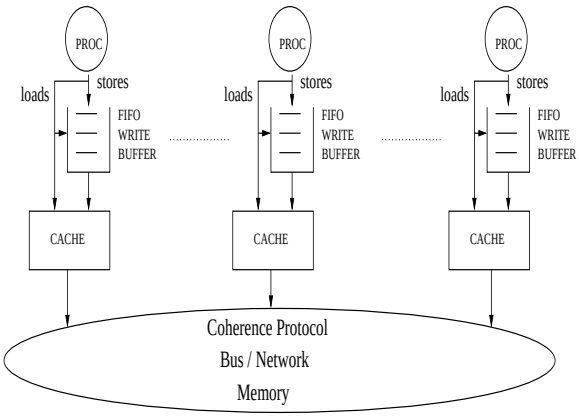

FIGURE 1. Our TSO Implementation

To convert this SC implementation into a TSO implementation, we insert a FIFO write buffer between a processor and its cache (as shown in Figure 1), and we add a MB instruction. The rest of the implementation (external to the processor and write-buffer) obeys the coherence protocol outlined above. The processor issues LDs, STs, and MBs in program order. Below, we specify exactly what happens when the processor issues one of these instructions. The processor completes issuing an instruction before proceeding to issue the next one in program order.

Stores: A ST issues into a FIFO write buffer (considered internal to the processor) in an event denoted as a $ST_{private}$. Entries in the write buffer are the size of processor words. Eventually, these entries are flushed from the write-buffer to the cache in the same order that they entered the write buffer, and this activity is independent of the issuing of STs by the processor. The event whereby an entry is flushed from the write buffer to the cache, once the processor has *established* that the corresponding block's A-state is Exclusive, is called a ST_{public}. By *establish*, we mean that the processor checks the A-state of the block and if it is not Exclusive, then the coherence protocol is invoked to change the A-state to Exclusive. Note that the Exclusive A-state is a prerequisite for a ST_{public} but not for a $ST_{private}$.

Loads: To issue a LD, the processor first checks in its write buffer for a ST to the same word. We refer to this action as a CHECK(LD). If the LD hits in the write buffer, then the LD gets the value of the most recent such $ST_{private}$ in program order. Note that a LD cannot overtake a ST to the write buffer, because the protocol does not start to issue a LD until issuing of all previous STs (in program order) has been completed. If the LD misses in the write buffer, then it is treated just like a LD in the SC protocol and has to go to the cache. That is, the processor establishes that the A-state of the block in the cache is Shared or Exclusive; if necessary, it invokes the coherence protocol (the details of which are as described by Sorin et al. [24]). In this case, the issuing of the LD completes when the processor establishes that the A-state of the block is Shared or Exclusive. We assume that LDs do not overlap with ST_{public}s to the same address, in the sense that the interval during which a LD is issued cannot overlap with the ST_{public} flushing interval, starting when the processor establishes that the A-state is Exclusive and continuing until the flush is completed.

MBs: Upon issuing a MB, our implementation simply flushes all entries in the write buffer to the cache before issuing any more operations. A more aggressive implementation could perhaps mark all the entries in some way and then ensure that subsequent coherence transactions are allowed to happen only when all marked entries have been flushed from the write-buffer.

3.4 Timestamping for TSO Implementation

We now present a scheme that assigns logical timestamps to the events of interest that occur during any execution of a program on our implementation of TSO. We define an *M-operation* (or simply an M-op) to be a LD or $ST_{private}$. M-ops are ordered by program order at a single processor. Our scheme assigns timestamps to M-ops, ST_{public}s and coherence protocol transactions (GETX, GETS, UPG, WB).

We define a notion of *binding* for M-ops and ST_{public}s which is useful for presenting the timestamping scheme. Intuitively, the binding time of an operation is the point in real time when that operation has been "committed" by the processor. $ST_{private}$s are bound when the corresponding entries enter the write buffer. ST_{public}s are bound at the time that the Exclusive A-state of the target block is established by the processor. LDs that hit in the write buffer are bound at the time that the corresponding CHECK(LD) occurs. LDs that miss in the write buffer are bound at the time that the A-state for the corresponding block is established by the processor. Both ST_{public}s and LDs that miss in the write buffer are said to be *bound to* the coherence transaction that obtained the block in the appropriate A-state.

Our timestamps are 3-tuples: <global-time.local-time.processor ID>. We give rules below for assigning global and local times to the various events that we timestamp. The processor ID acts as a tie-breaker. Conceptually, each processor has a global and a local clock which get updated in real time for transactions as well as M-ops and ST_{public}s, respectively.

Transactions are totally ordered by the bus in real time and we define the global time of a transaction to be its rank in this ordering, with the first transaction being assigned a global time of 1. At the moment that the A-state of a processor changes due to a transaction, the global clock of that processor is incremented to equal the global time of that transaction, while the local clock (and the local component of the transaction's timestamp) are set to 0.

Each M-op and ST_{public} is assigned a timestamp at the time that it is bound. If an M-op and ST_{public} happen to be bound at the same moment in real time, we assume that

they are assigned timestamps in some arbitrary (but deterministic) ordering (e.g., M-ops are always timestamped first). Note that a LD that misses in the write buffer and a ST_{public} can never be bound at the same time because of the real-time ordering properties of the protocol. The local clock is incremented by 1 to equal the local component of the timestamp assigned. The global timestamp is the value of the global clock at the moment that the M-op or ST_{public} is bound.

3.5 Proof of Correctness of TSO Implementation

We show that for any execution of our implementation, the timestamps of $ST_{private}$s, ST_{public}s, and LDs produce a Lamport ordering $<_w$ that satisfies properties 1´ to 5´ of the Wisconsin TSO definition. That properties 1´ to 4´ are satisfied follows from the real-time ordering properties of the protocol, the timestamping scheme, and the order in which events are bound. Property 5´ is proved as follows. We consider two possible situations for LD X:

1) Suppose that for some ST $<_p$ X, both to the same word, X $<_w$ ST_{public}. Let $Z_{private}$ be the most recent $ST_{private}$ to word *w* at p_i (prior to X in $<_w$). It must be that Z_{public} occurs after X in $<_w$, by property 3´ of Section 3.2. We need to show that X's value equals that of $Z_{private}$. Since instructions are issued in program order and issue intervals are non-overlapping, $Z_{private}$ is in p_i's write buffer before p_i performs CHECK(X). We claim that $Z_{private}$ is still present in the write buffer when p_i performs CHECK(X); otherwise, at the moment the check is done, Z_{public} would already be bound, causing X to be bound (to a transaction) in real time AFTER Z_{public} is bound. Since timestamps are consistent with binding order, this would contradict the fact that X $<_w$ Z_{public}. Hence, X must get the value of $Z_{private}$.

2) Suppose that for all ST $<_p$ X, both to the same word, ST_{public} $<_w$ X. It cannot be the case that X takes the value of any $ST_{private}$; if X were to take the value of a $ST_{private}$, say $Z_{private}$, then X would be bound BEFORE Z_{public}, since the interval in which X is issued does not overlap with the interval in which Z_{public} occurs. This contradicts our assumption in the previous sentence because binding order is consistent with $<_w$. Hence X gets the value of some ST_{public} and is bound to some transaction. Let Z_{public} be the most recent ST_{public} before X in $<_w$ (not necessarily at processor p_i). We need to show that X gets the value Z_{public}. The proof of this is identical to the proofs of the main theorems in our SC research [18,24], except that STs need to be replaced by ST_{public}s and the definitions of binding and timestamping there need to be replaced by the definitions of binding and timestamping in Section 3.4.

Hence all executions of the implementation satisfy Wisconsin TSO and so the implementation satisfies Wisconsin TSO. By Claim 1, the implementation also satisfies TSO.

4 Alpha

The Compaq (DEC) Alpha memory model [23] is a weakly consistent model that relaxes the ordering requirements at a given processor between any accesses to different memory locations unless ordering is explicitly stated with the use of a Memory Barrier (MB). We first define the Alpha memory model, introduce a collection of constraints on orderings which we refer to as Wisconsin Alpha, and prove the relationship between Alpha and Wisconsin Alpha. We then describe an Alpha implementation, present a timestamping scheme for the implementation, and prove that the ordering produced by the timestamping scheme satisfies Wisconsin Alpha, thus showing that the implementation correctly implements the Alpha memory model.

4.1 Defining Alpha

As with TSO, we are concerned mainly with a system containing multiple processors issuing word LDs, word STs and MBs (ordered by program order at a single processor) to regular memory (not I/O space). The Alpha memory model is formally defined through the use of two orders that must be observed with respect to memory accesses. The first order, program *issue order*, is a partial order on the memory operations (LDs, STs) issued by a given processor. Issue order relaxes program order in that there is no order between accesses to different locations without intervening MBs. Issue order enforces order between accesses to the same location, order between any access and an MB, and order between MBs. The second order, access order, is a total order of operations on a single memory location (regardless of the processors that issued them).

A third order, the "before" order, is defined to be the transitive closure over all of the issue orders and access orders. An execution of an implementation obeys the Alpha memory model if:

- for every memory location, there exists an access order for which there are no two memory operations A and B (not necessarily to the same address) such that A is before B, and B is also before A.
- a load returns the value of the most recent store to the same location in access order.

An implementation satisfies Alpha if all executions of that implementation satisfy Alpha.

4.2 Wisconsin Alpha

Although the Alpha memory model seems to have little in common with the stricter sequential consistency, we will show that the differences between the two models can be constrained to behavior internal to the processor (i.e., everything not including the cache and the rest of the memory subsystem). An execution of an implementation satisfies the Wisconsin Alpha memory model if there exists a total ordering of all loads, stores, and MBs, such that:

- all of the issue orders are respected.
- a load returns the value of the most recent store to the same location in this total order.

An implementation satisfies Wisconsin Alpha if all executions of that implementation satisfy Wisconsin Alpha.

Claim 2: An implementation that satisfies Wisconsin Alpha also satisfies Alpha.

A proof of this claim can be found in Appendix B.[1]

4.3 An Alpha Implementation Using Coalescing Write Buffers

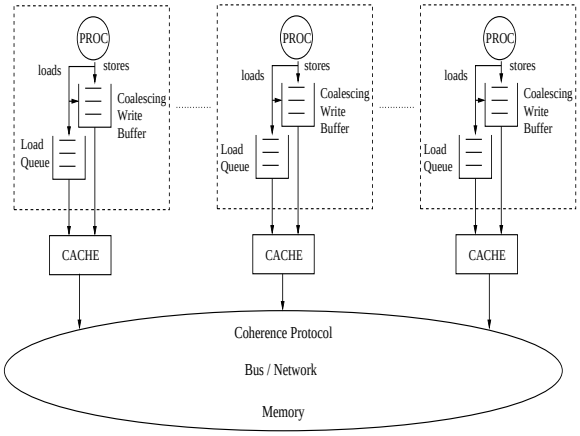

FIGURE 2. Our Alpha Implementation

Each processor in an Alpha implementation internally observes issue order. It can reorder loads and stores to different memory locations as long as there is no intervening MB. The multiprocessor implementation includes some number of these processors connected together either by a shared bus or a network. The cache coherence protocol used in either case is the same as the shared bus protocol [24] or the directory protocol [18] that we described in previous work. Our implementation is loosely modeled after a multiprocessor using the Compaq (DEC) Alpha 21264 microprocessor.

Each processor issues LDs and STs in program order. Stores are issued to a coalescing write buffer which is considered to be internal to the processor. Entries in the write buffer are the size of cache lines. Stores to the same cache line are coalesced in the same entry and if two stores write to the same word, the corresponding entry will hold the value written by the store that was issued later. Entries are eventually flushed from the write buffer to the cache, although not necessarily in the order in which they were issued to the write buffer. Exclusive permission is not required to issue a store to the write buffer, but it is required to flush the store from the write buffer to the cache.

A LD that hits in the write buffer returns the value that is found there, and this action does not require that line to be flushed from the buffer to the cache. The Alpha model, like most weak memory models, is tailored to include non-blocking caches. This optimization allows the processor to overlap read latency with other useful work, so LDs that miss in the write buffer are issued to a load queue which we consider to be internal to the processor. These LDs are handled by our existing SC coherence protocol with the following difference: a reply from the memory system satisfies all LDs to that location that are in the load queue at the moment that the processor establishes that the A-state is Shared or Exclusive. If the data was already in the cache in the appropriate A-state, then the LD can be satisfied immediately. We assume that there is no overlap between the issuing of LDs and the flushing of STs to the same address once Exclusive permission is obtained.

This implementation uses a simple mechanism for handling MBs, which is to stall the processor until the load queue and the write buffer are empty. Figure 2 illustrates our Alpha implementation, where everything outside of the dotted boxes is exactly the same as in our earlier sequentially consistent implementation.

4.4 Timestamping for Alpha Implementation

The timestamping scheme for the Alpha implementation is quite similar to that used for the TSO implementation. Coherence transactions affect the processors' global clocks in the same fashion. Each LD and ST is timestamped at the moment that it is bound, and it is in this determination of when a LD or ST is bound where Alpha differs from TSO. A ST is considered to be bound when the Exclusive A-state of the target block is established by the processor. Since an entire cache line is written at once, all of the stores in a buffer entry (including coalesced stores to the same word) are bound at the same time, but they are timestamped so as to preserve issue order. A LD that hits in the write buffer is bound exactly when that ST was bound, but it is timestamped after that ST to preserve issue order. If the LD misses in the write buffer, it is bound when the block becomes present in the appropriate A-state. At the moment that each LD or ST is bound, the local clock is incremented by 1 and the local component of the timestamp is set to the updated value. The global timestamp is the value of the global clock at the moment that the event is timestamped.

4.5 Proof of Correctness of Alpha Implementation

We show that each execution of the Alpha implementation satisfies Wisconsin Alpha. In previous work [18,24], we proved that an split-transaction bus protocol and a directory protocol obeyed sequential consistency. Parts of these proofs rely on the processors binding memory accesses in program order. To prove that our target Alpha implementa-

1. The converse of this claim can also be proved, but it is not necessary for our proof of correctness, and we omit it here due to space constraints.

tion obeys the Wisconsin Alpha memory model, we can use either proof (depending on whether our interconnect is a bus or a network) as long as we consider that binding order is now a partial order rather than a total order. Specifically, we need to modify the proofs of claims made about the binding of memory operations to coherence transactions so that references to the earliest memory operation are replaced with references to *any* of the earliest memory operations, since there could be more than one that is bound at the same time. Hence all executions of the implementation satisfy Wisconsin Alpha and so the implementation satisfies Wisconsin Alpha. By Claim 2, the implementation also satisfies Alpha.

5 Conclusions and Future Work

High performance shared-memory multiprocessors often incorporate relaxed memory consistency models. These implementations may use many hardware optimizations, such as write buffers and out-of-order issue, and it is difficult to verify that a complex implementation satisfies a given relaxed consistency model. We have extended our Lamport clock verification technique to handle two relaxed consistency models: processor consistency and weak consistency. Reasoning with Lamport clocks, we have shown that two sample implementations satisfy a processor consistent model (Total Store Order) and a weakly consistent model (Alpha), respectively.

Future work with Lamport clocks will extend the method to reason about consistent I/O and the detection of deadlock and livelock. We are interested in automating the verification process.

6 Acknowledgments

This work has benefited from feedback from many people, including Robert Cypher, James Goodman, Erik Hagersten, Daniel Lenoski, Paul Loewenstein, Gil Neiger, and David Wood.

7 References

[1] Sarita V. Adve and Mark D. Hill. Weak Ordering—A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 28–31, 1990.

[2] Hagit Attiya and Roy Friedman. A Correctness Condition for High-performance Multiprocessors. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, May 1992.

[3] William W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Inc., 1992.

[4] Intel Corporation. *Pentium Pro Family Developer's Manual, Version 3: Operating System Writer's Manual*. January 1996.

[5] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Draft of Parallel Computer Architecture: A Hardware/ Software Approach*, chapter 8: Directory-based Cache Coherence. Morgan Kaufmann, 1997.

[6] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[7] Asgeir Th. Eiriksson and Ken L. McMillan. Using Formal Verification/Analysis Methods on the Critical Path in Systems Design: A Case Study. In *Proceedings of the Computer Aided Verification Conference*, Liege, Belgium, 1995. Appears as LNCS 939, Springer Verlag.

[8] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report CS-TR-1199, University of Wisconsin – Madison, December 1993.

[9] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[10] J. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, 1989.

[11] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[12] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):241–248, September 1979.

[13] James P. Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, CO, June 1997.

[14] Gil Neiger. Private communication, October 1998.

[15] Gil Neiger and Sam Toueg. Simulating Synchronized Clocks and Common Knowledge in Distributed Systems. *Journal of the Association for Computing Machinery*, 40(2):334–367, April 1993.

[16] Seungjoon Park and David L. Dill. An Executable Specification, Analyzer and Verifier for RMO (Relaxed Memory Order). In *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 34–41, Santa Barbara, California, July 17–19, 1995.

[17] Seungjoon Park and David L. Dill. Verification of FLASH Cache Coherence Protocol by Aggregation of Distributed Transactions. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 288–296, Padua, Italy, June 24–26, 1996.

[18] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Architectures and Algorithms*, Puerto Vallarta, Mexico, June 28–July 2 1998.

[19] Fong Pong, Michael Browne, Andreas Nowatzyk, and Michel Dubois. Design Verification of the S3.mp Cache-Coherent Shared-Memory System. *IEEE Transactions on Computers*, 47(1):135–140, January 1998.

[20] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[21] Xiaowei Shen and Arvind. Specification of Memory Models and Design of Provably Correct Cache Coherence Protocols. Group Memo 398, Massachusetts Institute of Technology, June 1997.

[22] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres. Gigaplane: A High Performance Bus for Large SMPs. *Hot Interconnects IV*, pages 41–52, 1996.

[23] Richard L. Sites, editor. *Alpha Architecture Reference*

Manual. Digital Press, 1992.

[24] Daniel J. Sorin, Manoj Plakal, Mark D. Hill, and Anne E. Condon. Lamport Clocks: Reasoning About Shared-Memory Correctness. Technical Report CS-TR-1367, University of Wisconsin-Madison, March 1998.

[25] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1994. SPARC International, Inc.

Appendix A: Proof of relationship between Wisconsin TSO and TSO

Claim 1: An implementation that satisfies Wisconsin TSO also satisfies TSO.

Proof: Suppose that an implementation satisfies Wisconsin TSO, i.e., for every execution on that implementation, there exists a total ordering $<_w$ of the LDs, $ST_{private}$s, and ST_{public}s satisfying Wisconsin TSO. We claim that the implementation satisfies TSO. To show this, we show that each execution that satisfies Wisconsin TSO also satisfies TSO. This is done by defining a new ordering $<_m$ of just LDs and STs by removing all $ST_{private}$s and using the order of ST_{public} to define the order of each ST. We claim that the resulting ordering $<_m$ satisfies TSO. To see this, consider the requirements of TSO:

1. If $X <_p Y$ and X is a LD or Y is a ST, then $X <_m Y$.

- First, suppose that X is a LD. There are two possibilities for Y: (a) Y is a LD. This follows from 1´. (b) Y is a ST. This follows from 1´ and 2´, since by 1´, $X <_w Y_{private}$ and by 2´, $Y_{private} <_w Y_{public}$.
- The other possibility is that X and Y are STs. In this case, $X_{public} <_w Y_{public}$ by property 2' and hence $X <_m Y$.

2. If an MB occurs between X and Y in program order, then $X <_m Y$.

Again, we have separate cases depending what X and Y are:

- X is a LD. Then $X <_p Y$ and so by our argument in 1, $X <_m Y$.
- X is a ST and Y is a ST. Follows from 3´.
- X is a ST and Y is a LD. Follows from 4´.

3. Let X be a LD of word, and Y be the ST to word *w* in memory order ($<_m$) satisfying the constraints of property 3. Let W be the ST (either a ST_{public} or a $ST_{private}$) to word *w* in Wisconsin order ($<_w$) satisfying the constraints of property 5'. We need to show that Y = W.

- Suppose that W is a ST_{public}, call it W_{public}. Then, from the constraints in 5´ on W, no ST before X in program order has its ST_{public} after X in Wisconsin order. Therefore, W_{public} is the greatest ST_{public} in Wisconsin order (and hence W is the greatest ST in memory order), taken over all ST_{public}s Z_{public} to word *w* for which either (i) Z_{public} occurs before X in Wisconsin order (i.e. Z occurs before X in memory order) or (ii) Z occurs before X in program order (since there are no STs Z in category (ii) that are not already in category (i)). Hence Y = W.
- Suppose that W is a $ST_{private}$, call it $W_{private}$. Since $W_{private}$ satisfies the constraints of 5´, $W_{private}$ must be the most recent $ST_{private}$ at processor p before X in Wisconsin order (and so W must be the most recent ST before X in program order by 1´), and W_{public} must occur after X in Wisconsin order. Since the timestamps of ST_{public}s agree with the order of the corresponding STs in program order (by 3´), W_{public} is the greatest ST_{public} in Wisconsin order, taken over all ST_{public}s Z_{public} to word *w* for which either (i) Z_{public} occurs before X in Wisconsin order or (ii) Z occurs before X in program order. Therefore, Y = W.

Appendix B: Proof of relationship between Wisconsin Alpha and Alpha

Claim 2: An implementation that satisfies Wisconsin Alpha model also satisfies Alpha.

Proof: Suppose that an implementation satisfies Wisconsin Alpha i.e., for each execution of that implementation, there exists a total ordering of LDs, STs and MBs that satisfies the constraints of Wisconsin Alpha. We show that the implementation also satisfies Alpha by showing that each such execution also satisfies the constraints of Alpha. Given an ordering $<_w$ of LDs, STs and MBs in an execution that satisfies Wisconsin Alpha, let us define the access order for word *w* to be the ordering of LDs and STs on that word in $<_w$, and the issue order at a processor to be the ordering of LDs, STs and MBs issued at that processor in $<_w$. The "before" ordering is the transitive closure of issue order and access order. We now show that the two constraints of Alpha are met by these definitions of access order and "before":

- Let A and B be any 2 memory operations in the execution. Without loss of generality, suppose that operation A is before operation B. Since the before order is the transitive closure of the access and issue orders, and since $<_w$ respects both access and issue orders, then $A <_w B$. Hence, it cannot be that B is also before A, because otherwise $B <_w A$, which is impossible since Wisconsin Alpha order is a total order.
- A LD returns the value of the most recent store to the same location in the $<_w$ ordering which, from our definition of access order above, is also the most recent store to the same location in access order.