So Many States, So Little Time: Verifying Memory Coherence in the Cray X1*

Dennis Abts* dabts@cray.com

Steve Scott* sscott@cray.com

David J. Lilja† lilja@ece.umn.edu

*Cray Inc.
P.O. Box 5000
Chippewa Falls, Wisconsin 54729

†University of Minnesota
Electrical and Computer Engineering
Minnesota Supercomputing Institute
Minneapolis, Minnesota 55455

Abstract

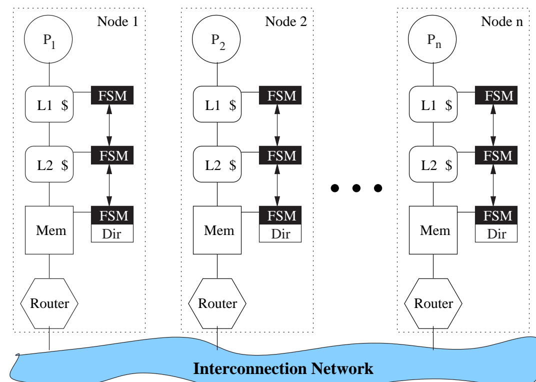
This paper investigates a complexity-effective technique for verifying a highly distributed directory-based cache coherence protocol. We develop a novel approach called "witness strings" that combines both formal and informal verification methods to expose design errors within the cache coherence protocol and its Verilog implementation. In this approach a formal execution trace is extracted during model checking of the architectural model and re-encoded to provide the input stimulus for a logic simulation of the corresponding Verilog implementation. This approach brings confidence to system architects that the logic implementation of the coherence protocol conforms to the architectural model. The feasibility of this approach is demonstrated by using it to verify the cache coherence protocol of the Cray X1. Using this approach we uncovered three architectural protocol errors and exposed several implementation errors by replaying the witness strings on the Verilog implementation.

1 Introduction

Distributed Shared Memory (DSM) multiprocessors [1, 2] are capable of scaling to large processor counts while providing a flexible programming model, allowing the programmer to treat the memory system as a large, logically shared memory. This programming abstraction, however, comes at the expense of additional hardware complexity to handle the implicit transfer of data as it migrates through the extended memory hierarchy that spans from the load-store unit of a given processor through multiple levels of cache, and possibly across multiple nodes which communicate over an interconnection network (Figure 1). This extended memory hierarchy must be kept consistent by ensuring that writes (stores) are propagated through the memory hierarchy. This *cache coherence* problem is solved by either software or hardware *cache coherence protocol* that explicitly manages the state of the memory hierarchy to ensure that data is coherent across the entire memory system. The design of an efficient coherence protocol is extremely challenging and, as Lenoski and Weber [3] point out, "...unfortunately, the *verification* of a highly parallel coherence protocol is even more challenging than its *specification*." The focus of this paper is on the verification of the directory-based hardware cache coherence mechanism employed by the Cray X1.

We treat the verification problem at two levels of abstraction: 1) architectural verification of the cache coherence protocol, and 2) verification of its corresponding Verilog implementation. Unfortunately the state space of a cache coherence protocol is enormous. Thus, random testing of the implementation provides very little confidence that the protocol state space has been sufficiently covered. Our method extracts a formal execution trace of the protocol state space during formal verification of the architectural model. Then, we replay the formal execution trace, called a "witness string," on the Verilog RTL implementation running on a logic simulator. This approach allows us to automatically generate high quality simulation traces by avoiding high fraction of redundant states visited during a random simulation. An individual witness string has no redundant states. This allows the Verilog simulations to be guided by the formal verification of the architectural model increasing confidence that the implementation conforms to the architectural model.

Figure 1: An abstraction of the hardware in a DSM multiprocessor. Control of the memory system is carried out by a finite-state machine (FSM) at each level of the memory hierarchy. The FSMs communicate by exchanging packets.



*The Cray X1 was referred to as the Cray SV2 during its research and development phase.

2 Memory Coherence

The notion of *cache coherence* is widely used, but there is no universally accepted definition of what this means. Censier and Feautrier stated that a memory system is coherent if the value returned by a read is always the value of the latest write with the same address [4]. But what is the "latest" write in a distributed system? And what guarantees are there regarding the ordering of reads and writes to different addresses? Avoiding this confusion requires a clean separation between architecture and implementation.

The *memory consistency model* (MCM) is the architectural model that defines how the memory system should behave with regard to ordering of reads and writes by different processors to different memory addresses, possibly including the effect of various synchronization primitives. There are a variety of memory consistency models, including sequential consistency [5], processor consistency [6], release consistency [7] and Total Store Order [8].

The cache coherence protocol and associated hardware structures and logic are what *implement* a given memory consistency model. Any system that allows multiple cached copies of a memory line will need some mechanism to propagate write values in order to satisfy most MCMs.

To verify that a system correctly implements its MCM, one could simulate the system looking for explicit violations of the MCM. While this is theoretically sufficient to determine correctness, it is often not practical or effective, given the enormous state space involved. By using knowledge of a specific implementation of an MCM, it is possible to detect errors in the implementation, even if they have not resulted in violations of the MCM

Consider a simple example (Figure 2) from a directory-based coherence protocol very similar to that used in the Cray X1. In this example we have two processors, P_0 and P_1 and a memory directory (MD). The MD is the data structure used to maintain the set of sharers and access permission to a cache line. The arcs in Figure 2 represent coherence messages between the L2 cache controller and the MD. All references in this example are for the same physical address, X. Each coherence message is given a timestamp. For the purpose of this example, we make a simplifying assumption that the order in which messages are sent is the order in which they are received, otherwise we would need a timestamp when the message is sent and a timestamp indicating when the message was received. When the message causes a state change at the controller the new state is recorded with a timestamp next to it to indicate which message caused the state change.

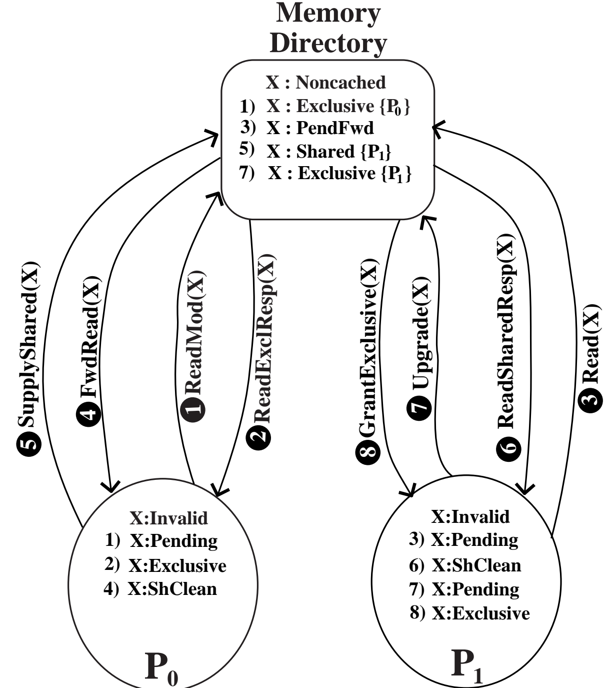


Figure 2: Two processors, P_0 and P_1, accessing a shared variable X. The memory directory (MD) erroneously removes one of the processors from the sharing set (after processing the SupplyShared(X) message) causing a potential data coherence problem.

At time ❶ processor P_0 issues a ReadMod(X), for a cache line read with intent to modify address X. As a result, P_0's cache state transitions to Pending. The MD responds at time ❷ by adding P_0 to the sharing set, sending a ReadExclResp(X) message with the cache line and transitioning to the Exclusive state. Upon receipt of the ReadExclResp(X) the L2 cache at P_0 transitions to the Exclusive state to indicate ownership of the cache line. Next, at time ❸ P_1 sends a Read(X) request and transitions to the Pending state. The MD receives the Read(X) request and transitions to the PendFwd state sending a FwdRead(X) intervention to the owner of the cache line at time ❹. Upon receipt of the FwdRead(X) message, the L2 cache controller transitions to the Shared state and at time ❺ sends a SupplyShared(X) response to the MD. The MD receives the SupplyShared(X) message and transitions to the Shared state, adds P_1 to the sharing set and sends the ReadSharedResp(X) to P_1 at time ❻.

Suppose there exists an error in the protocol specification such that after processing the SupplyShared(X) message received at time ❺, the MD erroneously removes the previous owner, P_0, from the sharing set (clearing its presence bit). The correct state of the MD *should be* X: Shared $\{P_0, P_1\}$ indicating both caches are sharing the data for address X.

At this point, after receiving the ReadSharedResp(X) at time ❻, if we ask ourselves "is the memory system coherent?" the answer is not entirely clear. Suppose that P_1 requested an Upgrade(X) request at some later time, say time ❼, to make the line writable. When the directory receives the Upgrade(X) request it will examine the sharing set and, since there appears to be no other sharers, will grant exclusive access to the line by sending a GrantExclusive(X) message to P_1's L2 cache controller at time ❽. Now, because of the protocol error that

occurred between time ❺ and time ❻, when P_1 writes to the Exclusive cache line those writes will not be propagated to P_0's cache which still has access to the line in the ShClean state.

On the other hand, it is possible for P_0 to evict X from its cache after time ❻ (or never read the value from X again) and effectively elude the malignant coherence error.

For any given execution, correct behavior requires only that the implementation conforms to the memory consistency model. However, as the example in Figure 2 illustrates, it is possible for a program execution to obey the MCM yet experience an error in the hardware that could lead to a *future* violation of the MCM. This indicates that the hardware will not be correct for *all* program executions.

Ideally, we would like to ensure that the memory consistency model is obeyed for *all possible* program executions. However, the vast search space that would entail makes exhaustive search for a violation of the MCM impractical. We can significantly increase the probability of finding an error in a design by understanding the mechanisms used to implement the MCM and searching for errors in the mechanisms before they result in violations of the MCM.

Using this approach, we define several hardware correctness properties specific to the X1 implementation; while these properties are not universal, other implementations are very likely to require a similar set of correctness properties. These correctness properties take the form of invariant expressions, which can be checked during a formal verification of the cache coherence protocol or a simulation of the Verilog implementation.

3 Overview of the Cray X1 Memory System

The Cray X1 is a DSM multiprocessor with a vector ISA, capable of scaling to thousands of processors. The X1 memory system is a significant departure from that of its predecessors and hence warrants further elaboration. Each X1 node consists of four multi-stream processors (MSPs) and 16 memory directory (M) chips. Each MSP consists of four single-stream processors (SSPs) and a shared Ecache. The MSP is implemented with four processor (P) chips and four Ecache (E) chips colocated on a multi-chip module. Any MSP may make references to any physical address in the system. However, only references made within the node are cached locally in the external cache (Ecache) and data cache (Dcache) within the SSP. Although the caching domain is restricted to a single node, the X1 memory system is globally cache coherent. All requests, local or remote, are processed by the cache coherence engine at the home node (the node where the physical memory exists).

The P chip is a semi-custom ASIC where the functional units are custom CMOS and the supporting logic for the pipeline control, dcache, load/store unit are implemented using standard cells. The E and M chips are implemented using standard cell ASIC technology with six metal layers. The P chip contains 41.5M transistors on a 16.5mm die. The E chip has 44.5M transistors on a 17.4mm die and the M chip has 31.8M transistors occupying a 16.5mm die. Each MSP, therefore, has $(41.5 + 44.5) \times 4 \approx 344$M transistors housed on a multi-chip module.

Table 1: Messages from chip-to-chip are sent as packets.

Chips	Commands
P to E	Read, ReadMod, ReadUC, ReadShared, ReadSharedUC, SWrite, SWriteNA, VWrite, VWriteNA, FetchAdd, FetchCSwap, FetchAndXor, AtomAdd, AtomAndXor
E to M	MRead, MReadMod, MReadShared, MGet, MPut, FlushAck, InvalAck, SupplySh,SupplyExclSh, SupplyDirtyInv, UpdateAck, UpdateNAck
M to E	ReadExclResp, ReadSharedResp,GetResp, FwdRead, FwdReadShared, FwdGet, FlushReq, WriteComplete, Inval, Update
E to P	PReadResp, PInvalidate

3.1 Chip-to-Chip Communication

The finite-state machines at each level of the memory hierarchy communicate by exchanging messages encoded as packets (Table 1). The packets flow on communication channels between the P chip, E chip and M chip. The physical communication channels are organized as three virtual networks, VN0, VN1 and VN2, in order to avoid deadlock in the interconnect. Requests are sent on VN0. Responses, interventions and invalidates are sent by the memory directories on VN1. Writebacks, eviction notices, and responses to interventions are sent on VN2.

3.2 Memory System Commands

The memory system commands support scalar and vector reads and writes, as well as atomic memory operations (AMOs). X1 supports two types of AMOs: result returning (FetchAdd, FetchCSwap, FetchAndXor) and non-result returning (AtomAdd, AtomAndXor). The processor request includes an *allocation hint* which allows the requestor to indicate whether the data should be allocated as shared, exclusive, or non-allocating. The commands with the *UC suffix are *uncached* in the Dcache. Commands ending in *NA (ReadNA, SWriteNA, VWriteNA) are non-allocating and do not allocate in either the Ecache or Dcache.

The memory system commands are exchanged from chip-to-chip on the virtual network (Table 1). Requests use the *mask* field of the packet to indicate which word(s) of the cache line to operate on. Each request uses a transaction identifier (TID) which is sent along with the request as it traverses the memory system. The TID is used to index into many of the auxiliary data structures used by the memory system.

Table 2: Memory system states for the Dcache, Ecache, and memory directory.

	States
Dcache	Valid, Invalid
Ecache	Invalid, ShClean, ExClean, Dirty, PendingReq, WaitForVData, WFVDInvalid
Memory Directory	Noncached, Shared, Exclusive, PendMemInvWrite, PendMemInvPut, PendMemExclusive, PendInvalPut, PendInvalWrite, PendFwd, PendDrop

3.3 Cache Organization and Memory Directory

The X1 has two levels of cache memory, Dcache and Ecache. The memory directory (MD) tracks the set of Ecaches with access to each cache line.

3.3.1 Dcache

The Dcache is a set associative write-through cache with 32-byte line size. The Dcache has only two states: Valid and Invalid. It makes no distinction between Valid and *pending* lines. Cache lines are marked Valid as soon as they are allocated. An associative match against earlier queued requests informs a new request when a valid line has not yet returned from the memory system, in which case the new request is enqueued behind the earlier request that initiated the Dcache fill.

3.3.2 Ecache

The Ecache is a set associative writeback cache with 32-byte line size and least recently used (LRU) replacement policy. The Ecache maintains inclusion over its local Dcaches using a 4-bit inclusion vector associated with the state of each line. The Ecache states are given in Table 2. The ShClean state is used only when the line is read only access. The ExClean state indicates exclusive access to the line, however, it has not yet been modified. When the line is modified it becomes Dirty. The PendingReq state is entered when waiting for a cache line response from memory. The X1 decouples the vector address and data, so the Ecache enters the WaitForVData state when it receives a VWrite request with the cache line address, but is still awaiting the vector write data packet. The WFVDInvalid state is used only when the Ecache receives an Inval packet from the directory while in the WaitForVData state. In which case, when the Ecache receives the vector data it will discard the data and transition to the Invalid state.

Ecache evictions

The X1 supports non-silent cache evictions to prune the sharing set and reduce the likelihood of a phantom invalidate. An evicted line will send an eviction notice message to the directory depending on the current state. If the state of the evicted line is Dirty a Writeback message is sent. Otherwise, a Drop or Notify message is sent if the state is ShClean or ExClean, respectively.

3.3.3 Memory Directory

Each M chip contains four memory directories. The memory directories are sufficiently large and associative to track the contents of all Ecache on the node simultaneously. This avoids the complication of evicting directory entries and invalidating the Ecache lines due to a capacity miss. Each memory directory is connected to two memory managers (MMs) which control the Rambus memory channels. Each directory entry has a *tag*, *state* and *sharing vector* associated with it. The sharing vector is a simple 4-bit vector which indicates the Ecaches that are caching the line. The directory states are given in Table 2.

Write completion

The directory collects InvalAck messages from the sharing Ecaches and sends a WriteComplete to the requesting Ecache once all invalidates have been successfully acknowledged. Only after the Ecache receives the WriteComplete message is the write deemed *globally visible*. A write is considered globally visible when no processor can read the value produced by an earlier write in the sequential order of writes to that location.

Replay queue

Both the E and M chips maintain a data structure called the *replay queue* (RQ). The RQ is used when an incoming request arrives at the Ecache or memory directory while the line is in a transient (pending) state. The new request is enqueued on the RQ, where it will be "replayed" after the transient state is resolved. The entries in the RQ are maintained as a linked list according to the requested address. So, when a request is replayed from the RQ, a simple pointer chase will find the next related entry.

Transient buffer

When a new request causes a state transition from a quiescent state to a transient state (for instance a transition from Noncached to Pending) the request is stored in a *transient buffer* (TB) indexed by the requesting TID. When the message arrives that allows the transient state to be resolved (for example, transition from Pending to Exclusive) the request is removed from the transient buffer and serviced. Each TB entry contains a buffer capable of holding one cache line. This buffer is always marked "empty" when the TB entry is allocated, and can be filled (and marked as "full") by a Writeback or Supply* message from the Ecache. Then, the RQ is "replayed" to maintain ordering of any new requests that arrived while the cache line was in a transient state. The TB line buffer may subsequently be used to provide data for a request that is being replayed from the RQ.

3.4 X1 Memory Consistency Model

X1 provides a relaxed memory consistency model (MCM) that describes how the programmer must view the shared memory in order to provide predictable program behavior. It is described in the X1 instruction

set architecture (ISA) and provides a set of guarantees and a set of synchronization instructions to the programmer. In general, the X1 MCM provides very few ordering guarantees. They include:

1. Single stream program order is preserved for scalar writes to the same word in memory. Vector and scalar writes are not ordered unless an Lsync instruction is explicitly used to provide ordering among vector and scalar references.
2. Writes to the same address are serialized. That is, no two processors can observe a *different* ordering of writes to that location.
3. A write is considered *globally visible* when no processor can read the value produced by an earlier write in the sequential order of writes to that location.
4. No SSP can read a value written by *another* MSP before that value becomes globally visible.

All other ordering between SSPs and MSPs must be provided by explicit memory synchronization instructions, Msync and Gsync. The Msync primitive is used as a lightweight synchronization operation among multiple MSPs in the same node. The Gsync is a global synchronization across all nodes in the machine.

4 Correctness Properties

Ultimately, correctness is defined by the memory consistency model (MCM) as described by the instruction set architecture (ISA). The cache coherence protocol is a critical piece of the MCM responsible for propagating writes, although other hardware such as arbiters, replay queue, and virtual network buffers, are equally important in correctly implementing the MCM since a hardware error in any one of these components could result in a violation of the MCM. For example, a hardware error in an arbiter may allow subsequent memory references to overtake earlier ones and thus violate the property of preserving individual program order. Although the properties we define are implementation-specific to the X1, it is very likely that the semantics of these properties would apply to other systems as well.

Showing that a cache coherence protocol is correct is nontrivial, as there are many aspects to "correctness," and the protocol state space is very large. Our approach is to formally model the protocol and prove that a collection of well-defined, fundamental properties hold over the state space. While these properties take into consideration the implementation details of the X1 cache coherence protocol, we expect that most coherence protocols would require similar properties. We use several predicates and functions [1] to describe the state of the caches, directory, and interconnection network.

[1] Predicates are designated by **bold** typeface and evaluate to a logical true or false. Functions return a value and are in sans serif typeface

4.1 Data Coherence

We indirectly capture the notion of data coherence by making some assertions about the state of the memory directory and caches.

Property 1 *If an address,* **a***, is in the "noncached" state at the directory and there are no messages,* **m***, in-flight from processor* **p** *to* Dir(Home(a)) *then processor* **p** *must have address* **a** *in an invalid state.*

$$\forall_a \forall_m \forall_p \, \mathbf{Noncached}(\mathsf{Dir}(\mathsf{Home}(a))) \wedge \neg \, \mathbf{InFlight}(m, p, \mathsf{Home}(a)) \Rightarrow \mathbf{Invalid}(a, p)$$

where **a** is an address, **p** is a processor cache, and **m** is a message. The function Home(**a**) returns the identity of the memory directory responsible for managing the address **a**. Likewise, the function Dir(**d**) returns the state of the memory directory (access permission and sharing set) for a given memory directory **d**.

Property 2 *If an address,* **a***, is present in cache,* **p***, then it must be included in the sharing set by the directory.*

$$\forall_a \forall_p \, \mathbf{Present}(a, p) \Rightarrow \mathbf{SharingSet}(\mathsf{Dir}(\mathsf{Home}(a)), p)$$

The **SharingSet** predicate returns true if the memory directory knows that address, **a**, is present in cache, **p**. Put another way, the set of caches with address **a** present is a subset ($\subseteq$) of the sharing set at the directory.

While these two properties do not explicitly address the read-the-latest-write aspect of memory coherence, they do ensure that the memory directory is properly maintaining the sharing set, an essential ingredient for memory coherence. Property 2 allows a cache line to be tracked by the memory directory, even if it is no longer present in the cache. For instance, if a cache line is evicted there will be some transient time between the eviction notice being sent and the memory directory removing the cache from the sharing set. As such, the cache could receive a "phantom" invalidate from the directory for a cache line that is no longer present.

4.2 Forward Progress

Ensuring *forward progress* requires every memory request to *eventually* receive a matching response. Since all coherent memory transactions occur using request-response message pairs, we can exploit this fact by formally stating:

Property 3 *Each request must have a satisfying response.*

$$\forall_x \, \mathbf{Request}(x) \Rightarrow \exists_y \, \mathbf{Response}(y) \wedge \mathbf{Satisfies}(y, x)$$

Moreover, the *forward progress* property (Property 3) encapsulates the notion of *deadlock* and *live-lock* avoidance by requiring each request to *eventually* receive a matching response. *Deadlock* is the undesirable condition where it is impossible to transition out of the

current global state. *Live-lock*, on the other hand, is a cycle of states that prevents forward progress. The predicates $\mathbf{Request}(x)$ and $\mathbf{Response}(y)$ evaluate to a logical true if x is a request and y is a response, respectively. Similarly, the predicate $\mathbf{Satisfies}(y, x)$ evaluates to a logical true if y satisfies x. For example, the predicate $\mathbf{Satisfies}(y, x)$ would consult the transition relation for the coherence protocol to determine if y was an expected response to request x. Clearly, this property ensures forward progress by ensuring that a request is never starved or indefinitely postponed.

4.3 Exclusivity

The coherence protocol enforces some access permissions over the shared-memory to ensure that there are never two or more processors with "exclusive" (write) access to the same memory block. This *single-writer* property can be stated as:

Property 4 *Two different caches,* **p** *and* **q**, *should never have write access to the same address,* **a**, *at the same time.*

$$\forall_a \forall_p \forall_q \, \mathbf{IsDirty}(a, p) \wedge q \neq p \Rightarrow \neg \mathbf{IsDirty}(a, q)$$

This property ensures that no two processors **p** and **q** are able to have memory block **a** in their local memory hierarchy in the "dirty" state [2] at the same time.

4.4 Unexpected Messages

If the coherence protocol is not fully specified it is possible to get an unexpected message making it possible for an FSM to receive an input message for which there is no corresponding entry in the specification table. For example, consider the following example encoding of an FSM:

```
...
Case State = Dirty
  Case InMsg = PrRead
    send(P, ReadResp)
  Case InMsg=PrWrite
    UpdateCache()
    send(P, WriteComplete)
    ...
  Default
    Error(UnexpectedMsg)
Case State = Shared
...
```

The "Default" case is used to trap unexpected messages, which are probably the result of an oversight in the protocol specification or some corner case that the protocol designer overlooked.

5 Verification Results

To verify the coherence protocol at an abstract level, we used the Murφ formal verification environment [9].

[2]Some protocols use the terms "dirty" or "exclusive" state. We assume that the predicate will return *true* if the cache line is either dirty or exclusive.

The coherence protocol is specified as several human-readable text files. These files are then read by an internally developed protocol compiler that automatically generates the finite-state machine descriptions in the Murφ description language. The Murφ compiler is then used to create the intermediate C++ description which is compiled and linked into the protocol verifier.

We simulated the Verilog RTL implementation using both Synopsys VCS logic simulator and Gensim cycle-based logic simulator. We replayed witness strings from the Murφ verification on the logic simulator exposing several bugs with the implementation.

5.1 Formal Verification

When constructing the formal model it is necessary to strike a balance to attain sufficient detail necessary in making the model accurate while scaling down the model to make an exhaustive search of the state space tractable. This balancing act, unfortunately, is subject to the perils of trial-and-error. We started with a fairly detailed system model and pruned away details and made abstractions where appropriate to make the size of each state relatively small. The space-complexity of the protocol verifier will be related to the size of each state.

We began with a detailed model that included the Dcache and load/store unit. However, through experimentation we arrived at a model that omitted the Dcache and load/store unit. Instead, the model included a fairly detailed representation of two slices of an MSP, each containing an Ecache, memory directory, replay queue, output request buffer, memory manager, virtual network buffers, and a simple processor model that issues requests and consumes response packets. Each Ecache had a single cache tag, with single bit of data. The virtual network model was complicated by the requirement of VN2 to always be able to sink an

Figure 3: The number of reachable states and time required to search explodes as the processors are allowed to issue more commands.

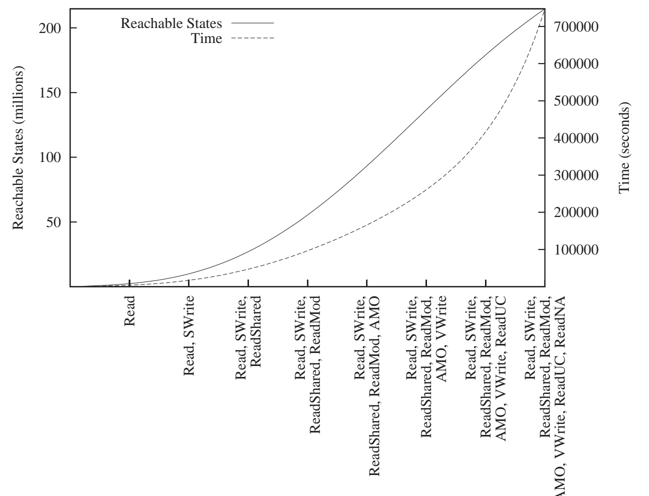
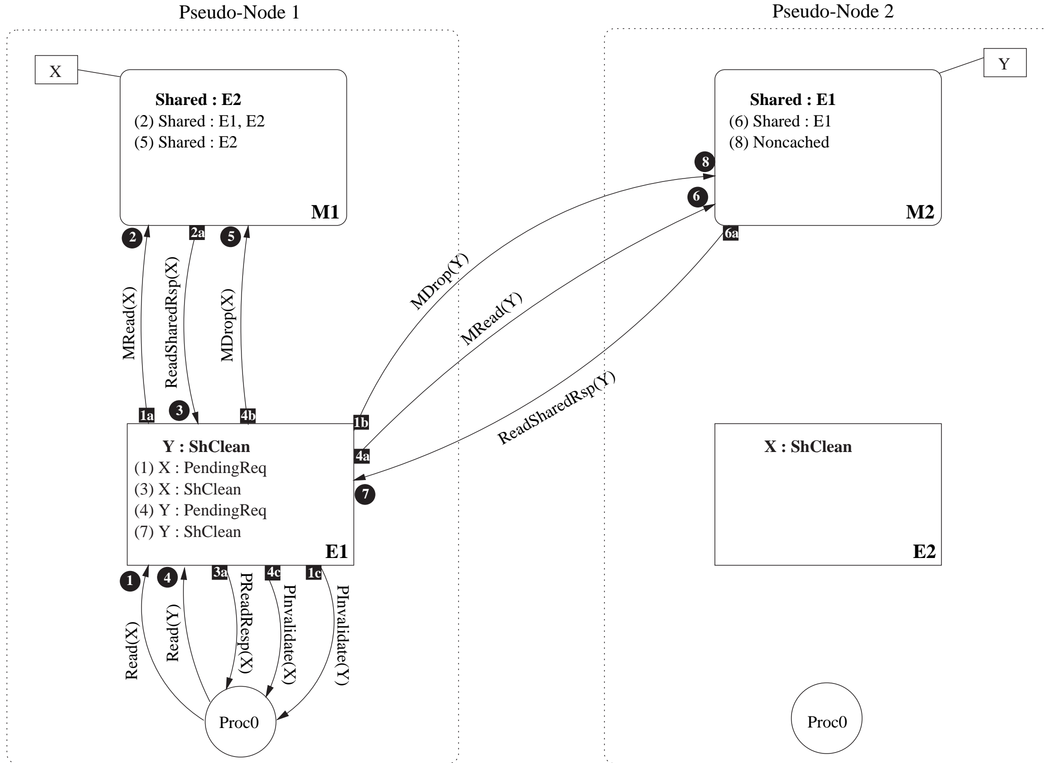


Figure 4: A protocol error that was discovered using Murφ . Had this error gone undetected it would have resulted in a loss of memory coherence.



incoming message. Otherwise, VN0 and VN1 each were a single entry buffer. However, VN2 required an additional buffer to act as a staging area to satisfy the requirement that we be able to sink a VN2 message.

The size of each state in the X1 Murφ model is 1664 bits and consists of 140 rules. At each state of the search, the verifier chooses among the 140 rules in the model to see which rules are eligible to execute (the predicate of the rule condition is true). A brute force search of every state would yield 2^{1664} states! Fortunately, a much smaller number of states are actually *reachable* (Figure 3). The protocol state space is ≈200M states, requiring almost 750000 seconds (over eight days) to explore! However, initially we did not know the upper bound on the search, so we started by only allowing each processor to issue only Read commands. This seemingly trivial case yielded a reachable state space of 8030 states and uncovered a critical error in the protocol (Figure 4).

The arrows in Figure 4 show the packets exchanged between chips, with a square marker indicating the time the packet was sent from the chip and a circular marker indicating the time the packet was received. Initially, we see that the processor at pseudo-node 1 has address Y in the ShClean state. A Read(X) request at event time 1 results in an eviction of Y from the E1 cache. The eviction causes a PInvalidate(Y) packet to be sent to the Dcache in the processor and an MDrop(Y) eviction notice to be sent to the directory. At the same time, the MRead(X) request is sent to the directory, which responds with a ReadSharedResp(X) response. Then, at event time 4 the processor issues a Read(Y) request to the E1 cache. The E1 cache evicts X and sends a PInvalidate(X) to the Dcache in the processor and an MDrop(X) eviction notice to the directory. At the same time, the E1 cache sends the MRead(Y) request to the directory, which responds with a ReadSharedResp(Y). Finally, the MDrop(Y) request from the eviction notice sent at time 1b reaches the directory at event time 8. When it receives the eviction notice, the directory removes E1 from the sharing set and checks to see if there are any remaining sharers. Since there are none, the directory transitions to the Noncached state. At this point, any stores to Y will not be propagated to the E1 cache, resulting in a loss of memory coherence. While it would have been difficult to construct a test to uncover the complex sequence of events that led to this error, our formal verification approach was able to discover it automatically.

5.2 Logic Simulation

The objective of the implementation verification is to run the "witness strings" generated by the Murφ formal verification on the Verilog RTL implementation of the hardware. The Verilog is compiled and simulated using an internally developed tool called *Gensim* [10]. The witness strings are encoded using a high-level verification environment called *Rave* [11, 12], which is

Figure 5: An example of a witness string generated by the Murφ formal verification tool.

```
-------------------------------------------------
Issuing scalar request Read from Proc_1 of Node_1
-------------------------------------------------
Quiescent: 1
E1(1:ShClean)<---Read(2) [C=0]  on vc0 from Proc_1
        Note: Evicting addr 1 from the Ecache
    E1 sending PInvalidate(1) to Proc_1
    E1 sending MDrop(1) to M1 on vc2
    E1 sending MRead(2) to M2 on vc0
-------------------------------------------------
Quiescent: 0
M1(Shared)<---MDrop(1) on vc2 from E1
-------------------------------------------------
Quiescent: 0
M2(Noncached)<---MRead(2) on vc0 from E1
-------------------------------------------------
Memory manager rule fired.
-------------------------------------------------
Quiescent: 0
M2(PendMemExclusive)<---MemRExclRsp(2) from MMGR1
    M2 sending ReadExclResp(2) to E1 on vc1
-------------------------------------------------
Quiescent: 0
E1(2:PendingReq)<---ReadExclResp(2) on vc1 from M2
        Note: Filling Ecache line...
    E1 sending PReadResp(2) to Proc_1
-------------------------------------------------
```

Figure 6: The output from Murφ (shown in Figure 5) is encoded into a format that is used by a *Rave* program as stimulus and expected results for the logic simulator.

```
Quiescent
E1(X:ShClean)<---Read(Y) on vc0 from P1 [13]
    E1 sending PInvalidate(X) to P1 [14]
    E1 sending MDrop(X) to M1 [15]
    E1 sending MRead(Y) to M2 [16]
E1(Y:PendingReq)<---ReadExclResp(Y) on vc1 from M2 [17]
    E1 sending PReadResp(Y) to P1 [18]
Quiescent
```

built around the C/C++ language. The witness strings from the Murφ formal verification are post-processed into stimulus encoded as *Rave* `apply()` and `verify()` calls.

As an example, Figure 5 shows a snippet from a witness string as generated by the Murφ formal verification tool. This string is converted into stimulus for the Verilog simulation model by choosing a "perspective" from which to observe the events (Figure 6). In this case, we will observe them from the perspective of the E chip. Alternatively, we could have chosen to observe them from the memory directory, for instance. Choosing the E chip perspective implies that we will be injecting stimulus into the E chip and verifying that the E chip produces the correct responses. The example in Figure 5 is a very simple sequence of events where a processor is simply making a Read(Y) request. However, this simple request has a total of six memory transactions associated with it. Markers are inserted into the stimulus when the memory system is quiescent, since this is often a good point to make assertions about the state of the memory system.

Memory system commands that are of the format:

```
E? sending <command>(var) to <dest>
```

are converted into a `verify()` *Rave* function call to verify the expected response from the cache coherence protocol. Memory system commands that are being injected into the E chip, from either the processor or the memory directory, are of the format:

```
E?(<var>:<st>)<---<command>(var) on <vn> from <src>
```

are converted into *Rave* `apply()` function calls to inject the packet into the E chip.

Replaying the witness strings on the E chip exposed several implementation bugs. One of them was a case where the E chip did not send the PInvalidate to the processor as prescribed by the cache coherence protocol. This is a clear case where the hardware implementation differed from the protocol specification and our witness strings approach was able to reveal this difference.

Unfortunately, the witness strings approach is limited by the speed at which the logic simulator can evaluate the logic. We ran witness strings on the E chip and performed an exhaustive depth-first search of $\approx$40M states relevant to the E chip in a few weeks worth of computing time. The M chip logic simulator, however, ran at only 1/10 the speed of the E chip and thus would have required *several months* of simulation time to completely search the set of protocol states relevant to the M chip. So, rather than trying to perform an exhaustive depth-first search, we generated witness strings from Murφ using a random search of the protocol state space.

Verifying the M chip with random witness strings yielded modest success. So, we supplemented our approach by writing additional producer-consumer shared memory programs using *Rave* that would stress the X1 coherence protocol and look for relaxations in the memory consistency model. These programs exposed additional bugs that were very subtle and required very specific timing to expose them. For instance, if the memory directory pipeline had a Read(X) request, immediately followed by an FetchAdd(X) AMO and then two clocks later a Put(X) occurred it would expose a structural dependence in the memory directory pipeline. These type of errors are very difficult to produce using witness strings because the witness string relies on determinism to be able to predict the state and values along the entire search path. With witness strings this amount of concurrent activity is limited to only a single outstanding reference at a time. One possible solution to this shortcoming is to introduce additional streams of traffic (to different addresses so that each witness string is still deterministic) that will perturb the pipeline and auxiliary data structures in an attempt to expose these subtle timing dependent errors.

Although a search of the state space using random witness strings does not provide complete coverage, it is still extremely useful. We determined that, on average, the Murφ protocol verifier would fire 35 rules in order to find a rule that produced a unique state. In other

words, 34/35 of the rules were essentially not productive for finding a new "interesting" state to explore. Assuming a similar fraction of redundant states in the logic simulation, if our logic simulation were *not* guided by the witness strings from Murφ, we would waste ≈97% of our simulation wall clock time doing redundant work!

X1 Hardware Prototype

The X1 hardware prototype was built using first revision silicon of the P, E and M chips. Within a few months of power up, the first spin silicon was able to execute many hardware diagnostics and boot the operating system from a memory image (before I/O capability). No hardware errors in the cache coherence protocol have been discovered bringing up these early systems. This level of functionality is a testament to the witness string approach as well as many other innovative ideas that went into the X1.

6 Related Work

Prior research in the area of coherence protocol verification has ranged from random tests using traditional discrete-event simulators, to formal methods. The informal approach uses a logic simulator or actual hardware prototype to determine if the implementation is correct. For instance, Dubois et al constructed a prototype [13, 14] of the Stanford DASH [15] multiprocessor using reconfigurable logic devices (FPGAs). The goal of the project was to provide a verification vehicle as well as a platform for performance evaluation. They implicitly verified aspects of the DASH multiprocessor by running the SPLASH parallel benchmarks. Unfortunately, this approach is very time-consuming, expensive, and cannot be performed until late in the design process. Furthermore, finding the design flaws is very difficult since only limited visibility and primitive tools are available for probing into the logic design.

Other efforts have proposed a hybrid approach, such as using Lamport logical clocks to study the correctness of memory consistency models [16, 17]. This timestamping technique provides a tool for reasoning about the proper event orderings in a multiprocessor memory system. However, it is unclear how this approach would be used to verify that the hardware implementation also satisfies the memory consistency model.

Recently, formal verification methods have been used to validate the coherence protocol of the SGI Origin2000 [18, 19, 20] and the Sun S3.mp (Sun Scalable Shared-Memory Multiprocessor) [21, 22]. Eiríksson used the Symbolic Model Verifier (SMV) [23] to verify an abstract model of a three node Origin2000 system (two processors and an I/O unit). Pong, et. al. used the Murφ [9] formal verification system to verify an abstracted three node S3.mp system. Each of these efforts needed to model the system at a high-level of abstraction (ignoring as much detail as possible) to make the verification tractable. Despite the unrealistically small verification model, these methods proved to be very valuable in extracting very subtle design flaws that would have been difficult, if not impossible, to detect using traditional logic simulation. Nonetheless, in both cases extensive logic simulation was performed to verify the actual RTL implementation.

7 Conclusion

It is commonplace in computer architecture to use trace-driven simulation to evaluate the performance of an architectural feature. We propose a similar idea for verifying the correctness of a cache coherence protocol based on a *formal execution trace* generated by executing the formal verification model. This formal trace file can be encoded into a practical format that allows it to be simulated on the actual RTL implementation. This technique provides a rigorous method for bridging the abstraction gap between the architectural verification and the RTL implementation verification.

We have validated this approach by successfully "replaying" the witness strings from the Murφ formal verification model on the RTL implementation of the E chip. In the process we uncovered eight implementation errors relating to the cache coherence engine in the E chip. Unfortunately, the slow simulation speed of the M chip made an exhaustive depth-first search of the M chip impractical. Future work is necessary to evaluate alternative search techniques that make this approach more practical within the confines of a tight project schedule and enormous protocol state space.

The area of coherence protocol specification and verification offers a challenging and fertile research opportunity. One area that would benefit greatly from expanded research is the area of formally specifying the coherence protocol. An example of one such effort is the Specification Language for Implementing Cache Coherence (SLICC) [24] which provides a syntax for formally specifying the protocol tables. While our approach to specifying the protocol is similar, the protocol compiler we devised is simplistic and capable of expressing only limited semantics. The goal is a more general-purpose specification language for the protocol tables with a corresponding compiler capable of generating the protocol FSMs in C++ (for a high-level performance simulator), Murφ (for formal verification), and Verilog RTL (for the implementation).

Another interesting research topic deals with potential search heuristics to prune the reachable state space, directing a search toward a likely bug. For example if we treat each state encoded as a large binary number and compute a difference between S_i and S_{i+1} to determine which branch of the search tree to explore. The idea is, rules that yield a new state that is "more different" from the current state are more likely to contain bugs. This is just one of many possible heuristics that should be examined.

Acknowledgements

We would like to thank those who contributed to this work through informal discussions and a variety of "chalk talks" to flesh out details. We would like to thank Greg Faanes, Jim Schwarzmeier, Scott Schroeder, Jim Sundet, and Brick Stephenson for recognizing and supporting innovative ideas in the presence of limited resources and a strict product development schedule. Special thanks to Mike Bye and Gerry Schwoerer for patiently sifting through many waveforms to help find and isolate errors. Also, Mike Roberts and Tyler Bennett for their support of logic simulation tools and Chris Gorzek for design synthesis metrics. This work was supported in part by National Science Foundation grants no. EIA-9971666 and CCR-9900605.

References

[1] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*, pages 655–749. Morgan Kaufmann Publishers, second edition, 1996.

[2] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*, pages 273–305 and 589–610. Morgan Kaufmann Publishers, 1998.

[3] Dan Lenoski and W.D. Weber. *Scalable Shared-Memory Multiprocessing*, pages 143–170, 134–140. Morgan Kaufmann Publishers, 1995.

[4] L. Censier and P. Feautrier. A new solution to cache coherence problems in multiprocessors systems. In *IEEE Transactions on Computer Systems*, volume C-27(12), pages 1112–1118, 1978.

[5] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[6] J.R. Goodman. Consistency and sequential consistency. Technical report, University of Wisconsin Technical Report 61, March 1989. SCI Committee.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News*, page 15, June 1990. Published as Proc. 17th Annual Int'l Symp. on Computer Architecture, ACM SIGARCH Computer Architecture News, volume 18, number 2.

[8] Inc SPARC International. The sparc architecture manual, version 8. Technical report, Prentice Hall, 1992.

[9] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design, VLSI in Computers and Processors*, pages 522–525, Los Alamitos, Ca., USA, October 1992. IEEE Computer Society Press.

[10] T. Court. Gensim user manual. Technical report, Cray Inc., 1996.

[11] D. Abts and M. Roberts. Verifying large-scale multiprocessors using an abstract verification environment. In *Proceedings of the 36th Design Automation Conference (DAC99)*, pages 163–168, June 1999.

[12] D. Abts, M. Roberts, and D. Lilja. A balanced approach to high-level verification: Performance trade-offs in verifying large-scale multiprocessors. August 2000. To appear in the Proceedings of the 2000 International Conference on Parallel Processing (ICPP-2000).

[13] L. Barroso, S. Iman, J. Jeong, K. Oner, K. Ramamurthy, and M. Dubois. RPM: A rapid prototyping engine for multiprocessor systems. *IEEE Computer*, 28(2):26–34, February 1995.

[14] M. Dubois, J. Jeong, Y. H. Song, and A. Moga. Rapid hardware prototyping on rpm-2: Methodology and experience.

[15] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of scalable shared-memory multiprocessors: The DASH approach. *CompCon Spring 1990.*

[16] M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport clocks: Verifying A directory cache-coherency protocol. In *Proc. of the 10th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'98)*, June 1998.

[17] A. Condon, M. Hill, M. Plakal, and D. Sorin. Using lamport clocks to reason about relaxed memory models. In *Proc. of the 5th International Symposium on High-Performance Computer Architecure (HPCA-5)*, Jan 1999.

[18] James Laudon and Daniel Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97)*, volume 25,2 of *Computer Architecture News*, pages 241–251, New York, June2–4 1997. ACM Press.

[19] Ásgeir Th. Eiríksson, John Keen, Alex Silbey, Swami Venkataraman, and Michael Woodacre. Origin system design methodology and experience: 1m-gate asics and beyond. In *COMPCON-97*, 1997.

[20] Ásgeir T. Eiríksson. Integrating formal verification methods with A conventional project design flow. In *33rd Design Automation Conference (DAC'96)*, pages 666–671, New York, June 1996. Association for Computing Machinery.

[21] F. Pong, M. Browne, G. Aybay, A. Nowatzyk, and M. Dubois. Design verification of the S3.mp cache-coherent shared-memory system. *IEEETC*, 47(1):135–140, 1998.

[22] Fong Pong and Michel Dubois. Formal verification of complex coherence protocols using symbolic state models. *Journal of the ACM*, 45(4):557–587, July 1998.

[23] K. L. McMillan. *Symbolic Model Checking*, pages 61–85. Kluwer Academic Publishers, 1993.

[24] M. Martin. personal correspondence, June 2000.