Using “test model-checking” to verify the Runway-PA8000 memory model

Rajnish Ghughal, Abdel Mokkedem, Ratan Nalumasu, and Ganesh Gopalakrishnan
Department of Computer Science, University of Utah,
Salt Lake City, UT 84112-9205
Contact email: {ghughal,mokkedem}@cs.utah.edu

Abstract
We have developed a formal technique called test model-checking for debugging claimed conformance to formal memory models by realistic memory systems and multiprocessor machines. Test model-checking is an embedding of a formal testing method called ARCHITECT in the model-checking framework. In this paper, we describe our technique and illustrate it on the problem of checking sequential consistency of (a model of) the HP PA8000 symmetric multiprocessing (SMP) bus called Runway. Our experiments show that test model-checking is an effective method for use in the typically iterative design cycle of complex memory systems to quickly detect ordering violations and pinpoint their cause.

1 Introduction
The fundamentally important problem [AG96] of verifying whether a given memory system conforms to a formal memory model appears in a number of contexts, including CPU design [Co92, GK94], I/O bus design [Cor97], and even in multi-threaded language design [GJS96]. As the semantics of memory orderings are too subtle to be fathomed through informal reasoning alone, formal verification methods have a natural role to play here. Unfortunately, previous solutions to this problem have either demanded valuable human expertise and time or have required complex temporal assertions to be made. Consequently, they are unsuitable for use in the typically iterative design cycle of memory systems to quickly detect ordering violations and to pinpoint the source of the violations. In this paper we describe a method that meets these objectives, called test model-checking.

Test model-checking formally adapts to the realm of model-checking a formal architectural testing method called ARCHITECT that has previously been successfully used on a number of commercial multiprocessors. While ARCHITECT is an incomplete testing method in that it does not, under all circumstances, detect violations of memory orderings [Co92], its tests have been shown to be sufficiently incisive to be useful in practice [Co9]. Being based on ARCHITECT, test model-checking is also incomplete. However, none of the (presumed) complete alternatives to date have been shown to be practical for verifying large designs. For example [PD96] involves the use of manually guided mechanical theorem proving. Even approaches based on conventional model-checking are impossible difficult to use in practice. For example, the assertions pertaining to the sequential consistency of lazy caching [Ger95], a simple memory system, expressed in various temporal logics by [Gra94] in $\mathcal{VCTL}^*$ [CES86] and [LLOR97] in TLA [Lam94] are quite complex. In [Gra94], abstract interpretation [CC77] is employed to reduce infinite-system verification to finite $\mathcal{VCTL}^*$ model-checking. They apply this technique to verify the sequential consistency of lazy caching with unbounded queues. They recognize that to get an exact characterization of sequential consistency involving only the observable event names, one needs full second order logic [Gra94]. To be able to express sequential consistency in $\mathcal{VCTL}^*$, they give a stronger characterization of sequential consistency, the temporal formula describing which is very complex. We do not believe that such descriptions will scale up. On the other hand, the test model-checking method has not only been able to comfortably handle the memory system defined by the symmetric multiprocessor (SMP) bus called Runway [BCS96, GGH97] used by Hewlett-Packard in their high-end machines, but also it discovered many subtle bugs in early Utah Runway Models (URMs) describing this bus that we created. Our URMs include a number of details such as split transactions, out of order transaction completions, and even an element of speculative execution. The errors we made in capturing these details could well have been made in an actual industrial context.

In [McM93], a $\mathcal{CTL}$ property that encodes a test for memory orderings has been discussed under the section heading ‘Sequential Consistency’. To give a historic perspective, the test model-checking idea originated in our attempt to answer the question of which memory ordering rule(s) the test of [McM93] is really verifying. We still have not found a satisfactory answer because the test of [McM93] uses only one location which then couldn’t make it a test for sequential consistency; it could plausibly be a test for coherence—which again does not correspond to what Collier formally proves in [Co92]. In this context, our contribution has been to not only answer the
above question (see Figure 1 which captures the test of
[McM93]), but also to describe formally the finite-state
abstractions that justify porting other tests from [CoB92]
to a model-checking framework.

Ahu et al [AMP96] showed that the problem of find-
ing if there is a sequentially consistent string $\sigma$ in
a regular expression $r$ is undecidable. In the model they
consider $r$ is made up of instructions $\text{read}(p, x, v)$ and
$\text{write}(p, x, v)$ where $p$ is the process that issued the in-
struction, $x$ is the shared variable, and $v$ is the value
the read instruction returns or the value to write. The
problem is undecidable as the actions that follow a read
instruction can depend on the value returned by a read
instruction. This result is not applicable in our case,
because the tests we consider do not make decisions
based on the value returned by a read instruction. This
is detailed in Section 2.1.

In [PD96], the authors use a method called aggrega-
tion on a distributed shared memory coherence protocol
used in an experimental multiprocessor, to arrive at a
simplified model of system behavior. Their technique in-
volves manually assisted theorem proving. The work in
[HM1993] as well as [DPN93] are aimed at verifying
that synchronization routines work correctly under vari-
ous memory models, where the memory models them-

selves are described using finite-state operational mod-
els. In [GK97, GK94], the authors study the problem of
deciding whether a given set of traces are sequentially
consistent. These works do not address the problem we
study, namely that of establishing that a detailed mem-
ory system model written in an HDL conforms to a for-
mal memory model.

Test model-checking has a number of desirable fea-
tures. It involves model-checking a fixed set of safety
properties for each formal memory model, that are very
nearly independent of the actual memory system model
being tested. This greatly facilitates its use in the typi-
cally iterative design cycle where debugging is most ef-
fective, design changes are frequent, and time-consuming
alterations to the properties being verified following design
changes would be frowned upon. The formal adaptation
of the tests of ARCHITECT made in test model-checking
can be verified once and for all, thanks to the fixed set of
tests used in test model-checking (we describe and argue
the correctness of these abstractions later). Finally, in
test model-checking, a memory model is viewed as a col-
clection of simpler ordering rules, and for each constituent
ordering rule, a specific property is tested on the memory
system. We found that this significantly helps com-
partimentalize errors, as opposed to producing non-intuitive
error traces that could result during conventional model-
checking, which can be very difficult to understand for
non-trivial memory systems.

Test model-checking is also a more effective debugger
for memory models than ARCHITECT in a formal sense.
The tests of ARCHITECT are straight-line programs of length
$k$, one per node. Such programs execute on various nodes
of the multiprocessor concurrently. The recommendation
accompanying ARCHITECT is that users run the tests for
as large a $k$ that is feasible, because then the chances of
being scheduled according to different interleavings (by
the underlying operating system, memory controller ar-
biter, etc.) increase. In adapting the tests of ARCHITECT,
test model-checking gives the effect of choosing $k = \infty$.
Thus, we cover all possible schedules.

2 Test Model-checking

ARCHITECT is based on the theory presented in [CoB92]
that formally defines and characterizes architectural rules
obeyed by memory subsystems of multiprocessors. Al-
though these rules are elemental, in realistic memory
systems the rules manifest in compound form. Obey-
ing a compound rule is tantamount to obeying all the
constituent elemental rules; violating a compound rule is
tantamount to violating any of the constituent elemental
rules. For read operations there is one read event per
each read operation. However, for write operations, there
is one write event per process per write operation which
captures the effect of a write operation becoming visible
to different processors at different times. Five important
elemental ordering rules are:

Rule of Computation (CMP): This is a basic rule
defining how the temporal value of each operand is
calculated from the initial values of the operands.
Some of the orderings imposed by this rule include:
(i) for statements such as $A := B$, read event for
operand $B$ happens before all write events for operand $A$
and the values of write events for operand $A$
must match the value of read event for operand $B$,
and (ii) if a process $P_i$ sees a value written for some
operand $A$ by a process $P_k$ then $P_i$’s write event
for operand $A$ (in $P_i$’s store) happens before the
read event for operand $A$ by process $P_k$ and their
values match. Though most of the literature on
memory architectures implicitly assumes this rule,
we will often keep it explicit in our discussions.

Rule of Read Order (RO): For any pair of read
events $a$ and $b$ in the same process, if $a$ comes before
$b$ in program order then $a$ happens before $b$.

Rule of Write Order (WO): For any pair of write
events $a$ and $b$ in the same process, if $a$ comes before
$b$ in program order then $a$ happens before $b$.

Rule of Program Order (PO): For any pair of events
$a$ and $b$ in the same process, if $a$ comes before $b$
in program order than $a$ happens before $b$. Event $a$
or $b$ can be either read or write events. So, both
RO and WO are special cases of PO. This is one
of the strongest ordering rules and is essential for
sequential consistency.

Rule of Write Atomicity (WA): A write operation
becomes visible to all processes instantaneously. More
precisely, one conceptual store $S_i$ is associated with
each processor node $P_i$. Then, for each write oper-
ation $W$, one write event $W_i$ is defined per store $S_i$.
Then, WA guarantees that there are no $i, j$ and no
event $e$ such that $e$ is before $W_i$ and is after $W_j$.

2.1 Assumptions about memory systems realized in
hardware

Memory systems realized in hardware as well as finite-
state models thereof are assumed to be data independent;
i.e., the control logic of the system moves data around,
and does not base its control-point settings on the data
values themselves. We also assume that the system is
address semi-dependent [HB95], i.e., the control logic can
at most compare two addresses for equality or inequality.
and base its actions on the outcome of this test. These assumptions are standard, and form the basis for defining test automata as well as memory rule safety properties, as will be discussed shortly.

2.2 Testing for architecture rules

The test $Test_{ROWO}$ of ARCHITEST for the compound rule consisting of the elemental rules CMP, RO, and WO, denoted $A_{(CMP, RO, WO)}$, is shown in Figure 1(a). $P_1$ executes a sequence of write instructions (intended to check for WO), and $P_2$ executes a sequence of read instruction (intended to check for RO). If the memory system correctly realizes $A_{(CMP, RO, WO)}$, then Condition 1 holds:

**CONDITION 1 (MONOTONIC)** The sequence of $X$ values is monotonically increasing, i.e.,

$\forall i : 1 \leq i < k : X_i \leq X_{i+1}$

ARCHITEST recommends that the parameter $k$ be made as large as possible to increase the chances that real machines running programs $P_1$ and $P_2$ interleave them in more ways. In test model-checking, we obtain the effect of $k = \infty$ by abstracting $Test_{ROWO}$ using data independence, as follows. Assume that Condition 1 is violated, i.e.,

$3i : 1 \leq i < k : X_i > X_{i+1}$

\[ (X_i > X_{i+1}) \land (X_{i+1} \leq \alpha) \]

\[ (X_i > X_{i+1}) \land \neg(X_{i+1} > \alpha) \]

Since \((\ast)\) depends only on the predicates $X_i > \alpha$ and $X_{i+1} > \alpha$, we can rewrite the test as shown in Figure 1(b), and rewrite \((\ast)\) as $X_i = 1 \land X_{i+1} = 0$. In the figure we show $\neg (X_i > \alpha)$ to show that the statement is executed only for obtaining the side effects associated with reading $A$. Since, in this figure, all reads of $A$ occur in $(A > \alpha)$, we can replace the statements $A := v$ and $\bullet := (A > \alpha)$ with $A := (v > \alpha)$ and $\bullet := A$ respectively, without affecting the correctness of the property. The effect of such a replacement is shown in Figure 1(c).

*Figure 1: $Test_{ROWO}$: A program to test for $A_{(CMP, RO, WO)}$ and its abstraction*

Figure 1(d) is essentially Figure 1(c) where every $v > \alpha$ is replaced by 0 or 1 as the case might be. Figure 1(e) is the test automata obtained from Figure 1(d) by making $P_1$ and $P_2$ guess the values of $\alpha$ and $i$ nondeterministically. MONOTONIC is checked as the memory rule safety property: $P_1$ at $S2 \Rightarrow x2 > x1$. The dotted edges in test automata indicate when the values used in memory rule safety property are assigned.

The test $Test_{WA}$ of ARCHITEST for $A_{(CMP, RO, WO, WA)}$ is shown in Figure 2(a). When this program is run on a machine that obeys $A_{(CMP, RO, WO, WA)}$, MONOTONIC must hold for arrays $U, V, X$ and $Y$, and in addition, condition ATOMIC must hold:

**CONDITION 2 (ATOMIC)** $\forall i, j : 1 \leq i, j \leq k : V_i \geq X_j \lor Y_j \geq U_i$. 

*Figure 2: $Test_{WA}$: ARCHITEST test and test automata for $A_{(CMP, RO, WO, WA)}$*
Initially $A = B = 0$

$P_1$

$L_{11} : A := 1$
$L_{12} : Y_1 := B$
$L_{21} : A := 2$
$L_{22} : Y_2 := B$

$P_2$

$L_{11} : A := B$
$L_{12} : X_1 := A$
$L_{21} : A := 2$
$L_{22} : X_2 := A$

(b) Figure 3: TestPO: ARCHTEST test and test automata for $A(CMP, PO)$

$P_1$

$P_2$

Condition 3 (PO\_Cross) \( \forall i, j : 1 \leq i, j \leq k : (X_i \geq j \lor Y_j \geq i) \land (X_i \leq j \lor Y_j \leq i) \).

Figure 3(b) shows the corresponding test automata. The memory rule safety property corresponding to condition PO\_Cross is: $P_1$ and $P_2$ in their final states $\Rightarrow (x \geq j \lor y \geq i) \land (x \leq j \lor y \leq i)$.

To show that this abstraction preserves PO\_Cross, let PO\_Cross be violated in ARCHTEST test TestPO.

$A := 0$

$E := 0$

$P_1$

$P_2$

Similar to the case of TestWA, if $A(i) : X[i] < j \land Y[j] < i$, then we can get a case in the test automata where $x = 0 \land y = 1 \land y = 0 \land y = 0$. Similarly, if $A(i) : X[i] > 0 \land Y[j] > i$, then we can get a case in the test automata where $x = 1 \land y = 0 \land y = 1 \land y = 1$. Hence, the memory rule safety property corresponding to PO\_Cross will be violated in test automata if and only if PO\_Cross will be violated in ARCHTEST test TestPO for $k = \infty$.

Summarizing test model-checking, there cannot be any real guarantees for ARCHTEST tests such as TestWA, TestPO etc. that the particular interleavings that reveal violations (such as for memory ordering rule WA watched by condition ATOMIC in TestWA) will indeed happen. To allow for as many interleavings as possible, ARCHTEST recommends that its tests be run for large values of $k$. Test model-checking effectively run ARCHTEST tests for $k = \infty$ by transforming each ARCHTEST test into a test automata using non-determinism. Also, the model-checking framework guarantees that we explore all possible interleavings rather than only particular interleavings.

3 Runway-PA8000 Memory System

Figure 4 shows a simplified view of 2 HP PA8000 CPUs and a memory controller (HOST) interconnected by HP Runway Bus[BSC96, Cam97, Kan96]. We will describe the Runway-PA8000 system in some detail to facilitate a clear description of some of the subtle bugs in URM unearthed by the test model-checking technique. Runway is a synchronous, split-transaction bus which is responsible for providing a coherent view of shared memory to the processors (clients) while still allowing the clients to maintain private copies of their memory lines in their caches. Cache Coherency is maintained by a snoopy coherency protocol described below.

Snoopy Coherency Protocol

Each cache line in a client can be in one of the four states\(^2\): invalid, shared, private-clean or dirty. If a

\(^1\)We have purposely avoided arbitration lines and other details for the sake of clarity. The actual Runway allows up to 4 CPUs and one I/O processor and also many more transactions including coherent, non-coherent and I/O transactions than we describe here. We provide a simplified view of its operation which captures the essential complexity of its behavior.

\(^2\)There are also transient states that the cache line may assume when the machine is changing from one of these clean states to another.
client suffers a read miss in cache, it generates a \textit{rsp} (read shared or private) transaction; if it suffers a write miss, it generates a \textit{rp} (read private) transaction. The transaction is broadcast on the Runway when it wins the bus mastership. All clients snoop the transaction into their CCC (cache coherency check) queues and process the entries in CCC queue at their own speed. When a transaction gets to the head of CCC of client \textit{C}, it sends a \textit{ccr} (cache coherency response) to HOST according to Figure 5, and also changes its state to reflect the transaction; for example, if the transaction is \textit{rp} generated by \textit{C}, it would assume “invalid-private-clean” transient state. If a client generates a \textit{coh\_copypout} as \textit{ccr}, it would later issue a \textit{c2cw} (cache to cache write) to supply the data. HOST enters the \textit{ccrs}'s into its CCC queue, and after all clients have responded to a transaction, the HOST determines if the data would be supplied by another client. If no client is going to supply the data, the HOST would generate a \textit{hdr} (host data return) transaction on the Runway to supply the data to the requester. It would also drive Client\_up lines to indicate whether the data must be shared (i.e., at least one of the \textit{ccrs} is \textit{coh\_shared}). When a client notices a data return (a \textit{hdr} or \textit{c2cw}) targeted towards it, it enters the information into data return (DR) queue. Note that a client might receive a data return before it generates the corresponding \textit{ccr}. In this case, the client keeps the data in data return queue until the \textit{ccr} is sent out.

**Delay in \textit{ccr} generation**

If a client has a \textit{c2cw} transaction for a line yet to go on Runway, then it delays generating any more \textit{ccrs}'s for that line. To see why this is necessary, consider the following. Suppose a client \textit{C} has a dirty line. Client \textit{C2} requests this line by issuing \textit{rsp} transaction on bus. \textit{C} will generate \textit{coh\_copypout} in response to \textit{C}’s request, invalidate its own line, and create a \textit{c2cw} transaction for \textit{C2}. Note that the most recent data for this line is with \textit{C} and not HOST. Now, a client \textit{C3} requests the same line by issuing \textit{rsp}. \textit{C2} and \textit{C3} generate respectively \textit{coh\_shared} and \textit{coh\_ok} \textit{ccrs} in response to \textit{C3}'s request. \textit{C1}'s \textit{ccr} will be \textit{coh\_ok} in response to \textit{C3}'s request. If \textit{C1} sends \textit{coh\_ok} to HOST before its \textit{c2cw} goes on the bus then HOST can provide stale data to \textit{C3} by its \textit{hdr} transaction. To avoid this, \textit{C1} delays generating \textit{ccr} until the \textit{c2cw} goes on the bus.

**Arbitration**

Runway follows a complex pipelined arbitration algorithm to determine the bus master. Here, we only present an approximation of the algorithm. Every bus user (client or HOST) must become the bus master before it can drive the bus. Bus mastership at cycle \textit{N+2} is acquired by initiating the arbitration in cycle \textit{N} by driving the request through dedicated arbitration lines (not shown in the figure). During cycle \textit{N+1}, every potential bus user evaluates the others’ drives and, in conjunction with round-robin pointers for arbitration priorities, determines who wins bus mastership for cycle \textit{N+2}. Those who do not win bus mastership keep off the bus. Bus arbitration proceeds in a pipelined manner concurrently with transaction processing.

**PA8000 Runway interface**

In addition to the Runway specifics described above, PA8000 Runway interface (PAR) also adheres to the following constraints in order to ensure Program Order and Write Atomicity [BCS96, Kann96]. PAR allows a client to initiate Runway transactions for various cache misses; it is possible that these transactions complete out of order. However, all instructions strictly complete in program order. PAR guarantees that the client will stall the coherency response for any cache line which it has an outstanding miss for (i.e., it has initiated a Runway transaction, has assumed the ownership but is still waiting for the data). The coherency response will be generated only after the client has received the data and has used it to make forward progress at least one instruction. PAR guarantees that if a client receives data for its Runway transaction before it assumed the ownership, then it will not modify or use the data until it processes its own transaction (and thus assumes ownership). PAR guarantees that if a client has a \textit{c2cw} transaction enabled to go to Runway then it gets the highest priority to go to the Runway.

**4 Verification using Test Model-checking**

To demonstrate the effectiveness of our approach, we verified three different memory systems, namely serial memory, lazy caching, and urm, all using a symbolic model-checker VIS (See [Ver] for more information). These three memory systems are described in some detail below, along with some of the subtle bugs that we could detect using test model-checking. Details of all our experiments can be obtained from the Web [Mok] or by contacting the authors.

**4.1 How do we check for sequential consistency?**

A sequentially consistent memory system [Lam93] requires that there be a single self-consistent trace \textit{t} of
memory operations that when projected onto the memory operations of each individual processor \( P_i \) (\( R_i(d,a) \) and \( W_i(d,a) \) for processor \( i \)) is according to program order for \( P_i \). As suggested in [Co]92, one’s intuition about sequential consistency matches the behavior described by \( A(CMP, PO, WA) \).

As [Co]92 does not list a single compound test to check for \( A(CMP, PO, WA) \), we can use the following two tests that are available: \textit{Test}_{WA} which tests for \( A(CMP, RO, WO, WA) \) and \textit{Test}_{PO} which tests for \( A(CMP, PO) \). This combination is exactly equivalent to testing for sequential consistency because \( PO \) implies \( RO \) and \( WO \) (as formally defined in [Co]92). While sequential consistency matches the behavior described by \( A(CMP, PO, WA) \), successful test model-checking outcomes are only necessary but not sufficient conditions to ensure sequential consistency. As part of our future work, we are exploring the ways to arrive at sufficient conditions for such tests. For every memory system we consider, these two tests are model-checked separately and summarized in Figure 8.

4.2 Serial memory and Lazy caching

The serial memory protocol for \( n \) processors and a memory is shown in Figure 6. Serial memories are often used to define SC operationally. The lazy caching protocol [ABM93, Ger95], shown in Figure 7, also implements sequential consistency, and is geared towards a bus based architecture. The memory interface still consists of reads and writes: however, caches \( C_i \) are interposed between the shared memory \( Mem \) and the processors \( P_i \). Each cache \( C_i \) contains a part of the memory \( Mem \) and has two queues associated with it: an out queue \( Out_i \) in which \( P_i \) write requests are buffered and an in queue \( IN_i \) in which the pending cache updates are stored. These queues model the asynchronous behavior of write events in a sequentially consistent memory. A write event \( W_i(d,a) \) doesn’t have an immediate effect. Instead, a request \( (d,a) \) is placed in \( Out_i \). When the write request is taken out of the queue, by an internal memory-write event \( MW_i(d,a) \), the memory is updated and a cache update request \( (d,a) \) is placed in every in queue. This cache update is eventually removed by an internal cache update event \( CU_i(d,a) \) as a result of which the cache \( C_i \) gets updated. Cache evictions are modeled by internal cache invalidate events: \( CI_i \) can arbitrarily remove locations from cache \( C_i \). Caches are filled both as the delayed result of write events and through internal memory-read events, \( MR_i(d,a) \). The latter events model the effect of a cache-miss: in that case the read event stalls until the location is copied from the memory. A read event \( R_i(d,a) \), predictably, stalls until a copy of location \( a \) is present in \( C_i \) but also until the copy contains a correct value in the following sense: SC demands that a processor \( P_i \) reads the value at a location \( a \) that was recently written by \( P_i \) unless some other processor updated \( a \) in the meantime. Hence, a read event \( R_i(d,a) \) cannot occur unless all pending writes in \( Out_i \) are processed as well as the cache updates requests from \( IN_i \), that corresponds to writes of \( P_i \). For this reason, such cache update requests are marked (with a *).

4.3 URM in VIS Verilog

We constructed a Verilog Utah Runway Model (URM) and the two abstractions of \( Test_{PO} \) and \( Test_{WA} \) to verify that its memory model is sequential consistent. The complexity of the system stems from a number of sources: (a) multiple outstanding transactions for each processor, (b) out-of-order completion of the Runway transactions, but in-order completion of instructions, (c) eager assumption of ownership without receiving the corresponding data, (d) “equivalent” states introduced by decoupled execution due to coherency queues, (e) speculative execution features of the processor to ensure performance in spite of in-order completion of the instructions, (f) an involved distributed pipelined arbitration algorithm. We did not try to model each of these features in their full glory, but we did include a modicum of these aggressive features into our URM, which in fact occupies more than 2,000 lines of VIS Verilog code (see [Mak]). For instance all essential features of (a), (b), (c), and (e) are included, (f) is abstracted by using nondeterminism. (d) is abstracted as explained below.

Abstraction of Queues Additional abstraction effort was necessary to make our URM digestible by VIS. This essentially consists in getting rid of the CCR, CCR, and DR queues which are the main cause of state explosion, but retain HDR queue in the HOST and C2CW queues in the HOST and clients.

In Runway, most of the conflicts are detected and resolved by the HOST. There is one situation where a client detects conflict: the client has a pending c2cw transaction. The client resolves this by delaying its coherence response; the net result of this delay is that the HOST would not generate hdr transactions until the c2cw goes on the Runway. Since we abstracted away the CCR queues, in our URM the clients send the coherence re-
spose for a coherent transaction immediately after its occurrence on the bus. Hence, in our URM the clients cannot resolve conflicts by delaying the coherency response; instead the HOST computes if the coherence response needs to be delayed, and if so, delays the hdrs appropriately. This is achieved as follows. A counter is associated with each HDR queue entry. If the counter is non-zero, then it is waiting for some c2cw transactions for that line from the clients, hence the hdr needs to be delayed. After all the pending c2cw transactions for that line go on the bus, the counter becomes zero, and hence the hdr transaction can go on the bus. In our URM, we used a two-bit counter, which allows up to four processors.

In Runway, all clients save the data returns (hdr and c2cw transactions) in DR queue until the corresponding request appears at the head of its CCC queue. This is necessary to enforce in-order completion of instructions. We abstract away the CCC queues and the data return queues by associating one bit of information with each cache line in each client. This bit is set for an address a whenever a data return happens for a, but a preceding instruction is not yet completed. After all preceding instructions are completed, the data is used, and the bit is reset indicating the completion of the instruction.

4.4 Verification results
The tables in figure 8 show execution time for model-checking our sequential caching and URM models for test cases of A(CMP, PO) and A(CMP, PO, WA) (recall that A(CMP, PO, WA) implies SC). The three models running separately, the test cases TestWA and TestPO are model-checked for the following conditions: (Figure 3(b) does not show some of these states)

\[
\begin{align*}
\text{TestWA:} & \quad \text{MONOTONIC:} \quad \land (P_1 \land S_2) \Rightarrow (P_2 \land V_1 \land S_2) \\
& \land (P_2 \land S_1) \Rightarrow (P_2 \land V_2) \\
& \land (P_3 \land S_2) \Rightarrow (P_3 \land V_1 \land S_2) \\
& \land (P_3 \land S_1) \Rightarrow (P_3 \land V_2) \\
\text{ATOMIC:} & \quad (P_4 \land S_2 \land P_5 \land S_1) \Rightarrow (P_4 \land V_2 \land P_5 \land Y \leq P_5 \land U)
\end{align*}
\]

\[
\begin{align*}
\text{TestPO:} & \quad \text{PO CROSS:} \quad (P_6 \land S_3 \land P_7 \land S_4) \Rightarrow (P_6 \land V_2 \land P_7 \land X \geq P_7 \land J) \\
& \land (P_8 \land Y \leq P_8 \land P_9 \land X \leq P_9 \land J)
\end{align*}
\]

As can be seen, all these conditions are safety properties, and independent of the model itself, which is a distinct advantage over other methods.

The size of the state space and number of nodes in BDDs are also reported. Note that lazy caching has more states than URM due to the queues present in the model. However, the complexity of URM is much higher, which results in large BDD size and higher run time. However, in all our experiments, whenever there was any memory ordering rule violation in our model, test model-checking detected it quickly (in the order of minutes). A desirable feature one can provide in a tool based on test model-checking is a menu of previously generated test automata for the various compound rules in [Co92], using which designers can probe their model.

Our Verilog models captures quite faithfully the cache coherence protocol and the ordering rules of the three memory systems.

After an extensive debugging using test model-checking driven by TestPO and TestWA, we have a high confidence that the memory systems built based on the Lazy caching model or URM would be sequentially consistent.

Description of a Bug found in a preliminary model of lazy caching: The following bug in our model of Lazy Caching was caught by a violation of PO_CROSS in TestPO. The bug was in the queues used by Lazy Caching, which were implemented as shift registers. We forgot to shift the s-bit in In, when the processor P_i receives a cache-update from In, queue. With this bug it is possible that In queue is not +ed when it should be, and consequently reads in P_i may bypass writes. This results in a violation of PO. This is a difficult bug to catch because its detection involves understanding the complex feedback from all components of the protocol to each other (queues, memory, and caches). Moreover, this bug is interesting because it violates PO but doesn’t violate WA. This is so because only write-read (WR) order is affected by this bug. Our technique effectively caught this bug: the PO_CROSS condition does not pass when we model-checked the model for TestPO. However, TestWA (note that it doesn’t involve PO) passed! This shows the futility of ad hoc testing methods: one could apply subjective criteria to consider a test similar to TestWA to be sufficiently incisive, when in fact it fails to account for a crucial ordering relation such as PO.

Description of a Bug found in preliminary URM: Similarly, another corner-case bug was caught by test model-checking in our URM by a violation of PO_CROSS condition using TestPO. This bug generated a long counter-example trace, due to the depth of the sequential logic of the model. The trace revealed the following situation:

(1) client_i has removed its own read transaction from the bus, then

(2) client_i sends coh_update in response to a subsequent coherent transaction for the same line before getting the data for its transaction (by hdr or c2cw).

This problem was fixed using the counter in the HOST’s HDR entries to record the pending c2cw and the one-bit information in the client’s cache lines to record whether the data is supplied, as explained in paragraph 4.3. After fixing the bug the PO condition passed.

5 Discussion and Future Plans

Using three memory systems, one of which very complex, we have evaluated a new approach to verify multiprocessor machines for formal memory models, which combines two existing powerful techniques: model-checking, and the testing method of ARCHTEST. From our results, we conclude that test model-checking can be of great value in detecting bugs during early stages of the design cycle of modern microprocessors whose memory subsystems are complex. Our results on our URM of the HP PA/Runway bus attest to this. In effect, test automata offer good specifications to check for formal memory models such as sequential consistency.

So far we have identified the rules and corresponding tests for sequential consistency. We are currently working on identifying similar rules and tests for other well-known formal memory models such as TSO, PSO, and
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>States</th>
<th>BDD Nodes</th>
<th>Conditions verified</th>
<th>Runtime [min:sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial memory</td>
<td>7220</td>
<td>7145</td>
<td>Vacuity</td>
<td>0.00:02</td>
</tr>
<tr>
<td>Lazy caching</td>
<td>7.8024×10^6</td>
<td>306092</td>
<td>Vacuity</td>
<td>01:31</td>
</tr>
<tr>
<td>UR</td>
<td>953675</td>
<td>165708</td>
<td>Vacuity</td>
<td>14:43</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>States</th>
<th>BDD Nodes</th>
<th>Conditions verified</th>
<th>Runtime [min:sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial memory</td>
<td>21242</td>
<td>10684</td>
<td>Vacuity, ATOMIC</td>
<td>00:04</td>
</tr>
<tr>
<td>Lazy caching</td>
<td>1.5074×10^6</td>
<td>813695</td>
<td>Vacuity, ATOMIC</td>
<td>02:02</td>
</tr>
<tr>
<td>UR</td>
<td>985216</td>
<td>1695092</td>
<td>Vacuity, ATOMIC</td>
<td>17:24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>States</th>
<th>BDD Nodes</th>
<th>Conditions verified</th>
<th>Runtime [min:sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial memory</td>
<td>21242</td>
<td>10684</td>
<td>Vacuity, ATOMIC</td>
<td>00:04</td>
</tr>
<tr>
<td>Lazy caching</td>
<td>1.5074×10^6</td>
<td>813695</td>
<td>Vacuity, ATOMIC</td>
<td>02:02</td>
</tr>
<tr>
<td>UR</td>
<td>985216</td>
<td>1695092</td>
<td>Vacuity, ATOMIC</td>
<td>17:24</td>
</tr>
</tbody>
</table>

Vacuity: Antecedent of $\Rightarrow$ is not always false

Figure 8: Verification results using VIS on a SPARC ULTRA-1 with 512 MB Memory

RMO [AG96] that are described in the SPARC V9 architecture manual [WG94]. This work may involve considering memory barrier operations and defining new rules as well as new tests.

We are currently working to formulate some reasonable assumptions about the memory system model under which the tests administered by our test automata can be rendered complete. Also, for a limited class of models, model-checking the test for some small value of $k$ might actually be sufficient. Our initial attempts in this direction are encouraging.

Acknowledgments. We would like to thank W. W. Collier for his help in explaining his work, his very informative emails and for providing ARCHITECT under a no-fee, research-only license. We would like to thank Dr. Paliath Narendran for many fruitful discussions. We would like to thank Dr. Al Davis and his Avalanche team for offering us the unique opportunity to work on state-of-the-art processors and buses.

References


[Cor97] Francisco Corella, April 1997. Invited talk at Computer Hardware Description Languages 1997, Toledo, Spain, on Verifying I/O Systems.


