Verification of the Futurebus+ Cache Coherence Protocol

E. Clarke[1] O. Grumberg[2] H. Hiraishi[3]
S. Jha[1] D. Long[1] K. McMillan[1] L. Ness[4]

October 1992
CMU-CS-92-206

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]School of Computer Science, Carnegie Mellon University

[2]Computer Science Department, The Technion, Israel; currently visiting at AT&T Bell Laboratories

[3]Computer Science Department, Kyoto Sangyo University, Japan

[4]Bellcore, Morristown, NJ

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597 and in part by the National Science Foundation under Grant no. CCR-9005992 and in part by the Semiconductor Research Corporation under Contract 92-DJ-294 and in part by the U.S.-Israeli Binational Science Foundation and in part by a Japan-U.S. cooperative research grant from the Japanese Society for the Promotion of Scientific Research and in part by U.S.-Japan cooperative research grant number INT-90-16694 from the National Science Foundation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Keywords: protocol verification, model checking, logics of programs, temporal logic, standards [the computer industry], Futurebus+, multiple data stream architectures (multiprocessors), cache memories, cache coherence, binary decision diagrams

Abstract

We used a hardware description language to construct a formal model of the cache coherence protocol described in the draft IEEE Futurebus+ standard. By applying temporal logic model checking techniques, we found several errors in the standard. The result of our project is a concise, comprehensible and unambiguous model of the protocol that should be useful both to the Futurebus+ Working Group members, who are responsible for the protocol, and to actual designers of Futurebus+ boards.

1 Introduction

This paper describes the formalization and verification of the cache coherence protocol described in the draft IEEE Futurebus+ standard (IEEE Standard 896.1–1991) [8]. We constructed a precise model of the protocol in a hardware description language and then used temporal logic model checking to show that the model satisfied a formal specification of cache coherence. In the process of formalizing and verifying the protocol, we discovered a number of errors and ambiguities. We believe that this is the first time that formal methods have been used to find nontrivial errors in a proposed IEEE standard. The result of our project is a concise, comprehensible and unambiguous model of the cache coherence protocol that should be useful both to the Futurebus+ Working Group members, who are responsible for the protocol, and to actual designers of Futurebus+ boards. Our experience demonstrates that hardware description languages and model checking techniques can be used to help design real industrial standards.

Futurebus+ is a bus architecture for high-performance computers. The goal of the committee that developed Futurebus+ was to create a public standard for bus protocols that was unconstrained by the characteristics of any particular processor or device technology and that would be widely accepted and implemented by vendors. The cache coherence protocol used in Futurebus+ is designed to insure consistency of data in hierarchical systems composed of many processors and caches interconnected by multiple bus segments. Such protocols are notoriously complex and, therefore, quite difficult to debug. Futurebus+ is, in fact, the first bus standard to include this capability. Although development of the cache coherence protocol began more than four years ago, to the best of our knowledge all previous attempts to validate the protocol have been based entirely on informal techniques [7]. In particular, no attempt has been made to specify the entire protocol formally or to analyze it using an automatic verification system.

In formalizing and verifying the protocol, we used SMV [9], a temporal logic model checker based on binary decision diagrams (BDDs) [1]. SMV includes a built-in dataflow-oriented hardware description language and accepts specifications expressed in the temporal logic CTL [6]. The tool extracts a finite-state model from an SMV program and uses an exhaustive state-space search algorithm [3, 4] to determine whether the model satisfies the specifications. If the model does not satisfy some specification, SMV will produce an execution trace that shows why the specification is false. SMV represents the transition relation of the model using BDDs. This representation makes it possible to handle some examples that have several hundred state variables and more than 10^{50} reachable states.

The biggest part of the project was using the textual description of the cache coherence protocol in the standard to develop a formal model for the protocol and to derive CTL specifications for its correctness. Our model for the cache coherence protocol consists of 2300 lines of SMV code (not counting comments). The model is highly nondeterministic, both to reduce the complexity of verification (by hiding details) and to cover allowed design choices (indicated in the standard using the word *may*). We believe that one of the most important contributions of our project is the model of the *bus bridges* that connect bus segments in hierarchical system configurations. These components are not specified in detail in the standard. However, without modeling the bus bridges, it is impossible to analyze

hierarchical systems in which the most subtle and complex behaviors occur. By using SMV and our model of the bridges, we were able to find several potential errors in the hierarchical protocol. The largest configuration that we verified had three bus segments, eight processors, and over 10^{30} states.

Our paper is organized as follows: section 2 contains a brief description of the temporal logic that we use for writing specifications. The basic ideas behind symbolic model checking are also explained. Section 3 describes the SMV language and model checking tool. The design of the Futurebus+ cache coherence protocol is discussed in section 4. Several examples are given to illustrate how the protocol is supposed to work. Section 5 describes the model we constructed, and section 6 explains how we were able to specify cache coherence in temporal logic. In section 7, we describe some of the ambiguities and errors that we found in the protocol, including the ones that were discovered when the model was extended to include the bus bridges. The last section outlines some directions for future research.

2 Temporal Logic Model Checking

Temporal logic is a method for expressing the ordering of events in time without introducing time explicitly. A. Pnueli was the first to use temporal logic for reasoning about concurrent systems [11]. However, his correctness proofs were constructed by hand, and only very small systems could be verified. The introduction of temporal logic model checking algorithms in the early 1980's allowed this type of reasoning to be automated [5, 6, 12]. Since checking that a single model satisfies a formula is much easier than proving the validity of a formula for all models, this technique can be implemented very efficiently. Unlike proof-checker based methods, model checking is completely automatic. More importantly, if a formula is not true of a model, a model checker can produce a concise execution trace that shows why the formula is not satisfied.

The particular logic that we use for specifications is a branching-time temporal logic called CTL ("Computation Tree Logic") [6]. Formulas in CTL are built from three components: atomic propositions, boolean connectives, and *temporal operators*. Atomic propositions talk about the values of individual state variables. The boolean connectives are the standard ones ($\wedge$, $\vee$, $\neg$). Each temporal operator consists of two parts: a path quantifier (**A** or **E**) and a temporal modality (**F**, **G**, **X** or **U**). The quantifier indicates whether the operator denotes a property that should be true of all execution paths from a given state or whether the property need only hold on some path. The modalities describe the ordering of events in time along an execution path and have the following intuitive meanings:

1. $\mathbf{F}\,\varphi$ ("φ holds sometime in the future") is true of a path if there exists a state on the path for which the formula φ is true.

2. $\mathbf{G}\,\varphi$ ("φ holds globally") means that φ is true at every state on the path.

3. $\mathbf{X}\,\varphi$ ("φ holds in the next state") means that φ is true in the second state on the path.

4. $\varphi\,\mathbf{U}\,\psi$ ("φ holds until ψ holds") means that there exists some state on the path for which ψ is true, and for all states preceding this one, φ is true.

Each formula of the logic is either true or false in a given state. An atomic proposition is true in a state if the state variable that it refers to has the appropriate value. The truth of a formula built from boolean connectives depends on the truth of its subformulas in the usual way. A formula whose top level operator is a temporal operator with a universal (existential) path quantifier is true whenever all paths (some path) starting at the state have the property required by the operator's modality. A formula is true of a system if it is true for all the initial states of the system. The following examples illustrate the expressive power of the logic.

1. $\mathbf{AG}(\mathit{req} \rightarrow \mathbf{AF}\ \mathit{ack})$: it is always the case that if the signal *req* is high, then eventually *ack* will also be high.

2. $\mathbf{AG}\,\mathbf{AF}\ \mathit{enabled}$: *enabled* holds infinitely often on every computation path.

3. $\mathbf{AG}\,\mathbf{EF}\ \mathit{restart}$: from any state, it is possible to get to the *restart* state.

4. $\mathbf{AG}(\mathit{send} \rightarrow \mathbf{A}(\mathit{send}\,\mathbf{U}\,\mathit{recv}))$: it is always the case that if *send* occurs, then eventually *recv* is true, and until that time, *send* must remain true.

There is a model checking algorithm for CTL that is linear in the size of the state space of the system under consideration. However, the state space is usually exponential in the number of components of the system. This *state explosion* is a major problem in all methods based on exhaustive state exploration. For this reason, recent model checkers use an implicit representation for finite-state systems based on *binary decision diagrams* (BDDs) [1]. BDDs are a canonical form for boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form. Using this representation does not alter the worst case complexity of the algorithm, but in practice, it makes the procedure much more efficient [2, 3, 4]. In a number of cases, we have found that verification time scales polynomially with the number of components in the system.

Sets of states and transitions are represented with BDDs as follows. Let V be the set of state components of the system. (Here, we assume all components are boolean.) A state is determined by an assignment of either 0 or 1 to each variable in V. Given such a truth valuation, it is possible to write a boolean expression that is true for exactly that valuation. For example, given $V = \{v_0, v_1, v_2\}$ and the valuation $\{v_0 \leftarrow 1,\ v_1 \leftarrow 1,\ v_2 \leftarrow 0\}$, we obtain the boolean formula $v_0 \wedge v_1 \wedge \neg v_2$. This formula can be represented using a BDD. In general, a boolean formula may be true for many different truth valuations. If we adopt the convention that a formula represents the set of *all* valuations that make it true, then we can describe sets of states by boolean formulas and, hence, by BDDs. In addition to representing sets of system states, we must be able to represent the transitions that the system can make. To do this, we extend the previous technique. Let V' be another copy of the state variables. A valuation for the variables in V and V' can be viewed as designating a pair of states that determine a transition. We can represent sets of such valuations using BDDs as above.

The *symbolic model checking algorithm* for CTL takes a formula φ and determines the set of states (represented as a BDD) where each subformula of φ (including φ itself) is true. It does this in a bottom up fashion starting from the atomic propositions in the formula. Handling atomic propositions and logical connectives is straightforward. For the

formula $\psi = \mathbf{EX}\,\varphi$, we want to find those states having a successor for which φ is true. This is done using an *image computation* [2, 3]. For a formula such as $\mathbf{EF}\,\varphi$, we use a fixed point characterization of the temporal operator:

$$\mathbf{EF}\,\varphi = \varphi \vee \mathbf{EX}\,\mathbf{EF}\,\varphi.$$

The fixed point is computed by iterating, starting from the empty set of states. Other temporal operators are handled in similar ways.

3 SMV

SMV ("*Symbolic Model Verifier*") is a tool for checking that finite-state systems satisfy specifications given in CTL. It uses a BDD-based symbolic model checking algorithm [3, 4]. The hardware description language built into SMV has the following features.

Modules: The user can structure the description of complex systems into modules. Individual modules can be instantiated multiple times, and modules can reference variables declared in other modules. Standard visibility rules are used for naming variables in hierarchically structured designs. Modules can have parameters, which may be state components, expressions, or other modules. We used the module facility heavily when modeling the Futurebus+ protocol; each type of device described in the standard is represented by a separate module.

Synchronous and interleaving composition: Individual finite-state machines given as SMV modules can be composed either synchronously or using interleaving. In a synchronous composition, a single step in the composition corresponds to a single step in each of the components. With interleaving, a step of the composition represents a step by exactly one component. If the keyword `process` precedes an instance of a module, interleaving is used; otherwise synchronous composition is assumed. We used both types of composition in our model of the Futurebus+ protocol. The devices on a single bus run synchronously, while separate buses are composed with interleaving.

Nondeterministic transitions: The state transitions in a model may be either deterministic or *nondeterministic*. Nondeterminism can reflect actual choice in the actions of the system being modeled, or it can be used to describe a more abstract model where certain details are hidden. The ability to specify nondeterminism is missing from many hardware description languages, but it is crucial when making high-level models. Some of the ways we used this ability are described in section 5.

Transition relations: The transition relations of modules can be specified either explicitly in terms of boolean relations on the current and next state values of state variables, or implicitly as a set of parallel assignment statements. The parallel assignment statements define the values of variables in the next state in terms of their values in the current state.

Fairness constraints: A module may contain fairness constraints. These constraints are used to rule out certain infinite executions. For example, suppose we have constructed an abstract model of a device that nondeterministically responds to a request on one of its inputs. We wish to ensure that the device must eventually respond to requests. To do this, we add a fairness constraint that specifies that infinitely often, if a request is present then an acknowledgement must be given. The execution where a request is always present but is never acknowledged is eliminated since it does not satisfy this constraint.

One of the most important features of SMV is its *counterexample facility*. If the given model does not satisfy one of its specifications, then SMV produces an execution trace illustrating why the specification is false. These counterexamples are extremely useful for debugging. Moreover, for models of moderate complexity, they are generally produced within a few minutes. This makes SMV extremely effective during the design process.

We will not provide a formal syntax or semantics for the language here; these can be found in McMillan's thesis [9]. Instead, we consider a small example illustrating part of a hand-shaking protocol (figure 1). Comments begin with "`--`" and continue until the end of the line.

Module definitions begin with the keyword `MODULE`. The module `main` is the top-level module. The modules `sender` and `receiver` have the formal parameter `input`. Variables are declared using the keyword `VAR`. In the example, `strobe` is a boolean variable, while `state` is a variable whose value is one of `ready`, `sending` or `waiting`. The `VAR` statement is also used to instantiate other modules as shown on lines 3 and 4. In this case, modules `sender` and `receiver` are instantiated with names `snd` and `rec`, respectively. A module can be instantiated multiple times with different names.

Modules can refer to variables defined in other modules by prefixing the variable name with the name of its module. For example, `rec.ack` refers to the variable `ack` defined in the module instance `rec`. The modules in this example are composed in a synchronous manner, but we could use interleaving composition by instantiating them as follows:

```
VAR
snd: process sender(rec.ack);
rec: process receiver(snd.strobe);
```

The `ASSIGN` statement is used to define the initial states and transitions of the model. In this example, the initial value of the variables `state` and `strobe` are `ready` and `0` respectively. The next-state value of the variable `state` is given by a `case` statement (lines 16–22). The value of a `case` statement is determined by evaluating the clauses within the statement in sequence. Each clause consists of a condition and an expression, which are separated by a colon. If the condition in the first clause holds, the value of the corresponding expression determines the value of the `case` statement. Otherwise, the next clause is evaluated.

Expressions may represent sets of values. Sets can be written explicitly, as shown on line 17, or can be constructed using the `union` operator, as shown on line 19. When a set expression is assigned to a variable, the value of the variable is chosen nondeterministically from the set. The `DEFINE` statement can be used to define abbreviations for expressions. In the example, `busy` is defined as an abbreviation for `state in {sending, waiting}`.

```
1  MODULE main  -- Handshaking protocol example

2  VAR
3  snd: sender(rec.ack);
4  rec: receiver(snd.strobe);

5  SPEC
6  AG (snd.strobe -> AF rec.ack)

7  MODULE sender(input)

8  VAR
9  strobe: boolean;
10 state: {ready, sending, waiting};  -- An enumerated type

11 DEFINE
12 busy := state in {sending, waiting};

13 ASSIGN
14 init(state) := ready;
15 next(state) :=
16   case
17   state=ready: {ready, sending};  -- Nondeterministic choice
18   state=sending & !input: state;
19   state=sending & input: state union waiting;
20   input: waiting;
21   1: state union ready;
22   esac;

23 init(strobe) := 0;
24 next(strobe) := state=sending;

25 FAIRNESS
26 !(state=sending & input)  -- If input stays 1, must eventually not send

27 MODULE receiver(input)
28 ...
```

Figure 1: SMV code for a simple handshaking protocol

This expression is true if the value of `state` is an element of the set `{sending, waiting}`. Fairness constraints are given by `FAIRNESS` statements, and properties to be verified are given as `SPEC` statements.

4 Overview of the Protocol

The IEEE Futurebus+ Logical Layer Specification is an attempt to define a technology-independent protocol for several generations of single and multiple-bus multiprocessor systems. Part of this draft standard is a cache coherence protocol designed to work in a hierarchically structured multiple-bus system. Under the protocol, coherence is maintained on individual buses by having the individual caches *snoop*, or observe, all bus transactions. Coherence across buses is maintained using *bus bridges*. Special agents at the ends of the bridges represent remote caches and memories. In order to increase performance, the protocol uses *split transactions*. When a transaction is split, its completion is delayed and the bus is freed; at some later time, an explicit response is issued to complete the transaction. This facility makes it possible to service local requests while remote requests are being processed.

As an example of how the protocol works, we consider some example transactions for a single *cache line* in the two processor system shown in figure 2. A cache line is a series of consecutive memory locations that is treated as a unit for coherence purposes. Initially,

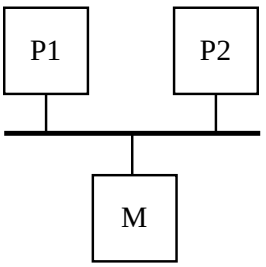

Figure 2: Single bus system

neither processor has a copy of the line in its cache; they are said to be in the *invalid* state. Processor P1 issues a *read-shared* transaction to obtain a readable copy of the data from memory M. P2 snoops this transaction, and may, if it wishes, also obtain a readable copy; this is called *snarfing*. If P2 snarfs, then at the end of the transaction, both caches contain a *shared-unmodified* copy of the data. Next, P1 decides to write to a location in the cache line. In order to maintain coherence, the copy held by P2 must be eliminated. P1 issues an *invalidate* transaction on the bus. When P2 snoops this transaction, it purges the line from its cache. At the end of the invalidate, P1 now has an *exclusive-modified* copy of the data. The standard specifies the possible states of the cache line within each processor and how this state is updated during each possible transaction.

We now consider a two-bus example to illustrate how the protocol works in hierarchical systems; see figure 3. Initially, both processor caches are in the invalid state. If processor P2 issues a *read-modified* to obtain a writable copy of the data, then the memory agent MA on bus 2 must split the transaction, since it must get the data from the memory on bus 1. The command is passed down to the cache agent CA, and CA issues the read-modified on bus 1. Memory M supplies the data to CA, which in turn passes it to MA. MA now issues a *modified-response* transaction on bus 2 to complete the original split transaction. Suppose now that P1 issues a read-shared on bus 1. CA, knowing that a remote cache has an exclusive-modified copy, *intervenes* in the transaction to indicate that it will supply the

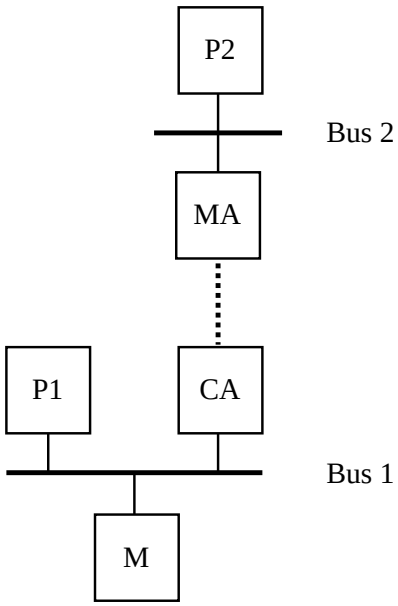

Figure 3: Two bus system

data, and splits the transaction, since it must obtain the data from the remote cache. CA passes the read-shared to MA, which issues it on bus 2. P2 intervenes and supplies the data to MA, which passes it to CA. The cache agent performs a *shared-response* transaction which completes the original read-shared issued by P1. The standard contains an English description of the hierarchical protocol, but does not specify the interaction between the cache agents and memory agents.

5 Modeling the Protocol

The IEEE Standard for Futurebus+—Logical Protocol Specification [8] contains two sections dealing with the cache coherence protocol. The first, a description section, is written in English and contains an informal and readable overview of how the protocol operates, but it does not cover all scenarios. The second, a specification section, is intended to be the real standard. This section is written using *attributes*. An attribute is essentially a boolean variable together with some rules for setting and clearing it. The attributes are more precise, but they are difficult to read. The behavior of an individual cache or memory is given in terms of roughly 300 attributes, of which about 45 deal with cache coherence. As an example, one attribute for cache modules is SHARED_UNMODIFIED:

> **SHARED_UNMODIFIED**. A CACHE or CACHE_AGENT shall set
> SHARED_UNMODIFIED and clear
> INVALID $\vee$ EXCLUSIVE_UNMODIFIED $\vee$ EXCLUSIVE_MODIFIED if
> MASTER $\wedge$ (INVALID_STATUS $\wedge$ $\neg$ADDRESS_ONLY $\wedge$ (READ_SHARED $\vee$
> READ_MODIFIED) $\vee$ KEEP_COPY $\wedge$ (COPY_BACK $\vee$ SHARED_RESPONSE)) $\vee$
> CACHED $\wedge$ (REQUESTER_SHARED $\wedge$ SHARED_RESPONSE $\wedge$
> INVALID_STATUS $\wedge$ $\neg$ADDRESS_ONLY $\wedge$ TRANSACTION_FLAG_STATUS $\vee$

SNARF_DATA ∧ ¬ADDRESS_ONLY ∨ REQUESTER_EXCLUSIVE ∧
MODIFIED_RESPONSE ∧ ¬ADDRESS_ONLY ∧ SPLIT_SATUS ∨
¬INVALID_STATUS ∧ KEEP_COPY ∧ (READ_SHARED ∨ READ_INVALID)).

A CACHE or CACHE_AGENT may set SHARED_UNMODIFIED and clear EXCLUSIVE_UNMODIFIED if EXCLUSIVE_UNMODIFIED.

A CACHE or CACHE_AGENT shall not allow modify access to the data in a cache line if SHARED_UNMODIFIED is set. A CACHE or CACHE_AGENT may allow read access to the data in a cache line if SHARED_UNMODIFIED is set.

Note that even in the specification section, some aspects of a module's allowed behavior are described informally. For example, the above attribute specifies a processor's read-write permissions in English. In addition, some aspects of the protocol, such as the bus bridges, are not completely specified using attributes.

In order to make the verification feasible, we had to use a number of abstractions. First, we eliminated a number of the low level details dealing with how modules communicate. The most significant simplification was to use a model in which one step corresponds to one complete transaction on one of the buses in the system. This allowed us to hide all of the handshaking necessary to issue a command. Another example concerns the bus arbitration. The standard specifies two arbitration schemes, but we used a model in which a bus master is chosen completely nondeterministically. In addition, the standard describes how models behave in various exceptional situations, such as when a parity error is observed on the data bus. However, we did not consider such conditions. The second class of simplifications was used to reduce the size of some parts of the system. For example, we only considered the transactions involving a single cache line. This is sufficient since transactions involving one cache line cannot affect the transactions involving a different cache line. Also, the data in each cache line was reduced to a single bit. The third class of simplifications involved eliminating the *read-invalid* and *write-invalid* commands. These commands are used in DMA transfers to and from memory. The protocol does not guarantee coherence for a cache line when a write-invalid transaction is issued for that line. The last class of abstractions involved using nondeterminism to simplify the models of some of the components. For example, processors are assumed to issue read and write requests for a given cache line nondeterministically. Responses to split transactions are assumed to be issued after arbitrary delays. Finally, our model of a bus bridge is highly nondeterministic.

Figure 4 shows a part of the SMV program used to model the processor caches. This code determines how the state of the cache line is updated. Within this code, state components with upper-case names (`CMD`, `SR`, `TF`) denote bus signals visible to the cache, and components with lower-case names (`state`, `tf`) are under the control of the cache. The first part of the code (lines 3–13) specifies what may happen when an idle cycle occurs (`CMD=none`). If the cache has a shared-unmodified copy of the line, then the line may be nondeterministically kicked out of the cache unless there is an outstanding request to change the line to exclusive-modified. If a cache has an exclusive-unmodified copy of the line, it may kick the line out of the cache or change it to exclusive-modified.

The second part of the code (lines 15–26) indicates how the cache line state is updated when the cache issues a read-shared transaction (`master` and `CMD=read-shared`). This should only happen when the cache does not have a copy of the line. If the transaction is

```
next(state) :=
  case
  CMD=none:
    case
    state=shared-unmodified:
      case
      requester=exclusive: shared-unmodified;
      1: {invalid, shared-unmodified};  -- Maybe kick line out of cache
      esac;
    state=exclusive-unmodified: {invalid, shared-unmodified,
      exclusive-unmodified, exclusive-modified};
    1: state;
    esac;
  ...
  master:
    case
    CMD=read-shared:  -- Cache issues a read-shared
      case
      state=invalid:
        case
        !SR & !TF: exclusive-unmodified;
        !SR: shared-unmodified;
        1: invalid;
        esac;
      ...
      esac;
    ...
    esac;
  ...
  CMD=read-shared:  -- Cache observes a read-shared
    case
    state in {invalid, shared-unmodified}:
      case
      !tf: invalid;
      !SR: shared-unmodified;
      1: state;
      esac;
    ...
    esac;
  ...
  esac;
```

Figure 4: A portion of the processor cache model

not split (`!SR`), then the data will be supplied to the cache. Either no other caches will snarf the data (`!TF`), in which case the cache obtains an exclusive-unmodified copy, or some other cache snarfs the data, and everyone obtains shared-unmodified copies. If the transaction is split, the cache line remains in the invalid state.

The last piece of code (lines 30–39) tells how caches respond when they observe another cache issuing a read-shared transaction. If the observing cache is either invalid or has a shared-unmodified copy, then it may indicate that it does not want a copy of the line by deasserting its `tf` output. In this case, the line becomes invalid. Alternatively, the cache may assert `tf` and try to snarf the data. In this case, if the transaction is not split (`!SR`), the cache obtains a shared-unmodified copy. Otherwise, the cache stays in its current state.

6 Specifying Cache Coherence

In this section, we discuss the specifications used in verifying the protocol. More exhaustive specifications are obviously possible; in particular, we have only tried to describe what cache coherence is, not how it is achieved. The first class of properties is used to check that no device ever observes an illegal combination of bus signals or an unexpected transaction. Each device model includes two flags *bus-error* and *error* that are used to signal these conditions. The flag *bus-error* becomes true when an illegal combination of bus signals (as defined in section 8.1.6 of the standard) is seen; *error* becomes true when a device observes a transaction which should not occur given its internal state. For example, if a processor cache has a shared-unmodified copy of a cache line, and a read-shared is issued, then no other cache should intervene in that transaction. If another cache does intervene (which can only happen when it has an exclusive-modified copy), the *error* flag in the first cache becomes true. Thus, we have the following formula for every device d:

$$\mathbf{AG}(\neg d.\textit{bus-error} \wedge \neg d.\textit{error}).$$

The next class of properties states that if a cache has an exclusive-modified copy of some cache line, then all other caches should not have copies of that line. The specification includes the formula

$$\mathbf{AG}(p1.\textit{writable} \rightarrow \neg p2.\textit{readable})$$

for each pair of caches *p1* and *p2*. Here, *p1.writable* is given in a `DEFINE` statement and is true when *p1* is in the exclusive-modified state. Similarly, *p2.readable* is true when *p2* is not in the invalid state.

Consistency is described by requiring that if two caches have copies of a cache line, then they agree on the data in that line:

$$\mathbf{AG}(p1.\textit{readable} \wedge p2.\textit{readable} \rightarrow p1.\textit{data} = p2.\textit{data}).$$

Similarly, if memory has a copy of the line, then any cache that has a copy must agree with memory on the data.

$$\mathbf{AG}(p.\textit{readable} \wedge \neg m.\textit{memory-line-modified} \rightarrow p.\textit{data} = m.\textit{data})$$

The variable *m.memory-line-modified* is false when memory has an up-to-date copy of the cache line.

The final class of properties is used to check that it is always possible for a cache to get read or write access to the line.

$$\mathbf{AG}\,\mathbf{EF}\ p.readable \wedge \mathbf{AG}\,\mathbf{EF}\ p.writable$$

We would like to give a stronger specification: if a cache issues a read or write request, it eventually obtains a readable or writable copy. Unfortunately, the model does not guarantee progress. This is due to the heavy use of nondeterminism, especially in the bus bridges. We could try to make a more precise model that would ensure progress (by adding fairness constraints or rewriting parts of the code), but this would be at the expense of an increase in verification time.

7 Some Errors Found During Verification

In this section, we describe some of the errors that we found while trying to verify the protocol. First, we describe an error in the single bus protocol. Consider the system shown in figure 5. The following scenario is not excluded by the standard. Initially, both caches

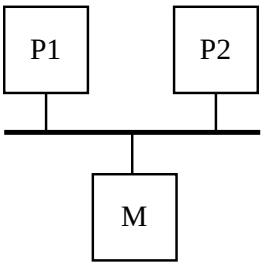

Figure 5: Single bus system

are invalid. Processor P1 obtains an exclusive-unmodified copy. Next, P2 issues a read-modified, which P1 splits for invalidation. The memory M supplies a copy of the cache line to P2, which transitions to the shared-unmodified state. At this point, P1, still having an exclusive-unmodified copy, transitions to exclusive-modified and writes the cache line. P1 and P2 are now inconsistent. This bug can be fixed by requiring that P1 transition to the shared-unmodified state when it splits the read-modified for invalidation. The change also fixes a number of related errors.

Next, we consider an error in a hierarchical configuration (figure 6). Initially, all caches are invalid. P1 obtains an exclusive-modified copy of the cache line. It then issues a *copy-back* transaction to return the updated cache line to memory, but elects to keep a shared-unmodified copy as well. In addition, P2 snarfs the data during the copyback and obtains a shared-unmodified copy. The memory agent MA obtains the data from the copyback, but does not pass it on yet. P1 now issues an invalidate, and P2 splits the transaction. Some time later, MA passes the data from the copyback to CA, and CA issues a copyback. P3 snarfs the data from the copyback and transitions to the shared-unmodified state. P2 finishes

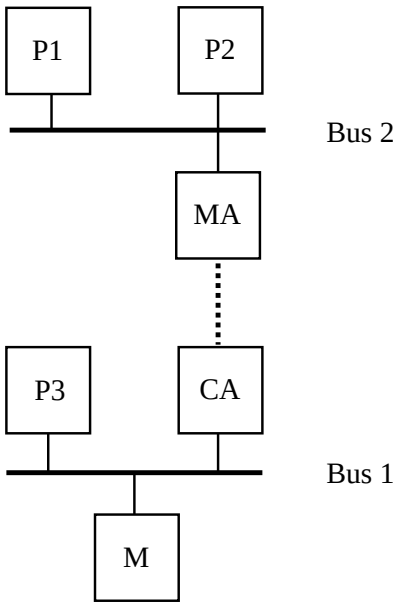

Figure 6: Two bus system

invalidating and issues a modified-response. MA must split this response, since P3 now has a copy. However, when it does this, it fails to acquire the *responder-invalidate* attribute that indicates that it must eventually issue a modified-response of its own. As a result, P1 can never obtain a writable copy of the cache line. This error can be fixed by either not allowing copybacks for lines which are in the process of becoming exclusive, or by making MA acquire the responder-invalidate attribute when splitting the modified-response from P2.

Another error in the hierarchical protocol is the following. P1, P2, and P3 all obtain shared-unmodified copies of the cache line. P1 issues an invalidate transaction that P2 and MA split. P3 issues an invalidate that CA splits. The bus bridge detects that an *invalidate-invalidate collision* has occurred. That is, P3 is trying to invalidate P1, while P1 is trying to invalidate P3. When this happens, the standard specifies that the collision should be resolved by having the memory agent invalidate P1. When the memory agent tries to issue an invalidate for this purpose, P2 sees that there is already a transaction in progress for this cache line and asserts a busy signal on the bus. MA observes this and acquires the *requester-waiting* attribute. When a module has this attribute, it will wait until it sees a completed response transaction before retrying its command. P2 now finishes invalidating and issues a modified-response. This is split by MA since P3 is still not invalid. However, MA still maintains the requester-waiting attribute. At this point, MA will not issue commands since it is waiting for a completed response, but no such response can occur. The deadlock can be avoided by having MA clear the requester-waiting attribute when it observes that P2 has finished invalidating.

We checked configurations with up to three buses and eight processors. The number of boolean state variables in our models ranged from about 75 to 250, with a corresponding number of reachable states between 10^{10} and 10^{30}. The number of BDD nodes needed to represent the model was about 150,000 nodes in the largest models that we tried. In terms of

asymptotic performance, the number of nodes needed to represent the system grew linearly with the number of components on a bus, and quadratically with the number of buses. Verification times ranged from about a minute to an hour, depending on the configuration. The most important point is that for models of moderate size, feedback from the verifier could be obtained in a matter of minutes. This made it possible to find bugs and try possible fixes very quickly.

8 Conclusions

All formal verification involves making a model of the system under consideration. Saying that the system is correct is really a claim that this model satisfies the specifications. We have attempted to make as detailed a model of the Futurebus+ cache coherence protocol as possible and to check as many system configurations as possible. Nevertheless, more remains to be done; for example, by combining model checking with induction, it should be possible to verify arbitrarily configurations. McMillan used this technique when verifying the cache coherence protocol for the Encore Gigamax [9, 10]. We plan to try to use induction with the Futurebus+ protocol, although this protocol is much more complex than the Gigamax protocol.

We believe that work on other standards would benefit by collaboration with experts in specification and automated verification throughout the design process. Use of a formal language to state requirements should result in significantly faster development of correct designs. Such a strategy would no doubt result in faster and lower cost implementations by vendors as well. Finally, model checking is not limited to finite-state models arising from hardware. Formalization and analysis of other types of systems, such as telecommunications protocols, should also be possible using SMV. Researchers must keep in mind that the ultimate test of a new formal verification technique is whether it can handle real examples like the Futurebus+ protocol.

Acknowledgements

We would like to thank Paul Dixon, of the IEEE Futurebus+ Working Group, for the time he spent discussing and reviewing our results.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[2] J. R. Burch, E. M. Clarke, and D. E. Long. Representing circuits more efficiently in symbolic model checking. In *Proceedings of the 28th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1991.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, and D. L. Dill. Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*. IEEE Computer Society Press, June 1990.

[4] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1990.

[5] E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY, May 1981*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[6] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[7] P. Dixon. Multilevel cache architectures. Minutes of the Futurebus+ Working Group meeting, December 1988.

[8] IEEE Computer Society. *IEEE Standard for Futurebus+—Logical Protocol Specification*, March 1992. IEEE Standard 896.1–1991.

[9] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* PhD thesis, Carnegie Mellon University, 1992.

[10] K. L. McMillan and J. Schwalbe. Formal verification of the Encore Gigamax cache consistency protocol. In *Proceedings of the 1991 International Symposium on Shared Memory Multiprocessors*, April 1991.

[11] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[12] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, 1981.