Using Formal Specifications to Monitor and Guide Simulation: Verifying the Cache Coherence Engine of the Alpha 21364 Microprocessor

Serdar Tasiran
Compaq Systems Research Center
Palo Alto, CA

Yuan Yu
Microsoft Research
Mountain View, CA

Brannon Batson
Intel Corporation
Santa Clara, CA

Scott Kreider
Intel Corporation
Santa Clara, CA

Abstract

We describe a technique for verifying that a hardware design correctly implements a protocol. The application of this technique to verifying the cache and memory controllers of the Alpha 21364 EV7 microprocessor against a formal specification of the EV7 cache coherence protocol is presented.

1 Introduction

For hardware implementing a complex protocol, verification of consistency with high-level specifications is a very labor intensive process that is never entirely completed in practice. Simulation using random patterns or hand-written test programs is the only tool available for this purpose. Typically, the design is described in a hardware description language and the high-level specification is a text document. Code is written to check for violations of the high-level specification during simulation. This approach is not satisfactory for several reasons. First, since the specification is informal, it is difficult to verify whether it is consistent and complete. Second, the code checking for specification violations may itself contain errors. Third, and most important, it is difficult to quantify how well different aspects of the specification have been explored, and to direct simulation runs towards unexplored areas.

We present a technique to improve this process. Our technique requires a high-level specification written in a formal language and a mapping that relates simulation steps in the implementation to state transitions in the specification. A model checker checks each such state transition for consistency with the formal specification, and stores the specification states visited. This approach provides many significant benefits:

(i) Using formal verification techniques, logical errors can be eliminated from the specification and it can be ensured that the specification is complete.

(ii) Each step of the simulation run is checked against a verified formal specification using a formal verification tool, which is better debugged than design- and protocol-specific code.

(iii) Coverage analysis of the specification points out gaps in validation. We measure state coverage for a user-defined set of specification variables and report unexplored assignments to these variables.

(iv) A model checker is used to improve coverage of the specification. Traces to interesting unvisited specification states are generated using the model checker. These traces are then used to direct simulation towards coverage gaps.

The techniques described in this paper were developed as part of the verification of the cache coherence engine of the Alpha 21364 EV7 microprocessor.

2 Our Approach

2.1 The Formal Specification and Verification Framework

TLA$^+$ [3] is a formal language for writing high-level specifications of concurrent and reactive systems. TLA$^+$ is based on the temporal logic of actions (TLA) [2] and incorporates first order logic, set theory, and temporal operators, and is therefore very expressive. TLA$^+$ supports high-level constructs, such as sets, queues, records, and tuples, which arise naturally in high-level specifications but are difficult to express in verification input languages aimed at the register-transfer level. TLA$^+$ has been used successfully inside Digital Equipment Corporation and Compaq for specifying and formally reasoning about complex protocols such as the EV6 Wildfire, and EV7 cache coherence protocols [1, 4].

Several formal verification tools have been developed to reason on TLA$^+$ specifications. In our work, we make use of the TLA$^+$ model checker TLC [4]. TLC is an explicit-state model checker for verifying safety properties of TLA$^+$ specifications. It has been used to check the consistency and completeness of TLA$^+$ specifications, model check relatively large system specifications, and simulate larger ones. For practical systems, even specifications have very large state spaces, and it is not feasible to perform exhaustive exploration of the state-space. TLC has two key features that are helpful for dealing with large state spaces, which we make use of in coverage analysis. The first one is support for *views*. A view v is an (often many-to-one) function from a large state space to a smaller one, expressed in terms of the state variables in a TLA$^+$ specification. When a state s is visited while exploring the state-space, instead of storing the entire set of variable assignments that define s, TLC can store $v(s)$ and, in this way, visits only one state for each value in the range of v. v can be used to define a coverage metric on the specification state-space by assigning the same value of v to states that are qualitatively the same. The second feature of TLC is taking advantage of symmetries. Symmetries in a specification can be described as a set Π of permutations of state variables. A specification is symmetric with respect to Π if for each $\pi \in \Pi$, permuting the state variables in the specification using π results in the same specification. Given Π, TLC avoids exploring states that are symmetric with respect to Π. The EV7 cache coherence protocol is symmetric for all permutations of processors and memory addresses, therefore the state space explored is reduced by $p!a!$, where p is the number of processors and a is the number of memory addresses in the multi-processor configuration being verified.

2.2 The EV7 Cache Coherence Hardware

The EV7 microprocessor provides support for single-chip multiprocessing. It contains hardware that enables EV7 processors connected in a two-dimensional torus to act as a cache-coherent shared-memory multiprocessor. A directory-based cache coherence protocol is implemented by means of

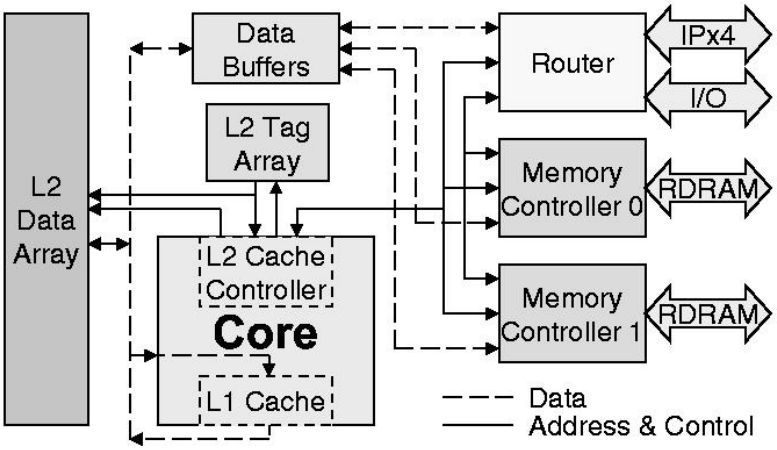

Figure 1: EV7 chip block diagram

an on-chip protocol engine which collaborates with the level 2 cache controller (Figure 1). The protocol engine is part of the memory controller and can handle 64 outstanding protocol transactions simultaneously. Both the cache and the memory controllers are deeply pipelined (10-20 stages each), highly optimized, complex circuits. Queues, address and data arrays often have more than one read and write port. There are large (16 or more entries) victim buffers between level 1 and level 2 caches, the level 2 cache and the rest of the system. The network protocol between processors can reorder messages. All of these amount to a hardware implementation that is hard to debug and is far beyond the reach of formal verification tools. Moreover, it has been found that to exercise certain aspects of the hardware for an EV7 multi-processor system, configurations with more than six processors may need to be simulated. Simulating such configurations is itself very computationally demanding and often necessitates a distributed simulator running on several machines.

2.3 Specifying the EV7 Cache Coherence Protocol

The specification for the EV7 cache coherence implementation was written in TLA$^+$ by architects who were consulted by formal verification researchers. The protocol, excluding comments, is around 2000 lines of TLA$^+$ code. Writing the specification required translating several high-level descriptions of the protocol into one formal description in TLA$^+$ at roughly the same level of abstraction. This process required rigorous thought about the protocol and the implementation, and was found by the architects to be well worth the effort. TLC was used to check the specification for consistency, completeness and for proving properties on small configurations for the protocol, i.e. few processors and addresses. For larger systems, certain safety properties were proven by hand on the specification. By the time we started applying our approach to verifying the protocol implementation, both the implementation and the specification were relatively mature.

The specification is parametrized in the number of processors and memory addresses (cache lines) in the system. The specification state variables can be grouped into three following the structure of the design: variables that represent the cache controller state (the $CBox$), memory controller state (the $ZBox$) and variables that represent messages in flight between processors (the Net). An n

processor configuration can be viewed as an interconnection of n $ZBox$'es, n $CBox$'es, and a Net , where Net[i] represents the set of protocol messages that have been sent to processor i. Frequent use is made of queues, arrays, and sets in describing the $CBox$, the $ZBox$ and the Net . TLA$^+$ specifications consist of *actions*, which can be viewed as guarded commands: if a certain condition holds of the current state, the TLA$^+$ action specifies how the state variables are to be updated. Each action in the EV7 protocol pertains to the $ZBox$ or $CBox$ of one processor and represents how one protocol transaction is to be executed given the packets at the input ports and the current state of the $ZBox$ or $CBox$. A protocol transaction updates the input state of the $ZBox$ or the $CBox$ and sends messages to the Net or other $CBox$'esor $ZBox$'es. This description style mimics the original high-level textual description of the EV7 protocol.

Due to the complexity of the protocol, even after abstracting away all data and reducing the address space to very few addresses, because of the large number of protocol transactions that can be pending simultaneously, the specification for any non-trivial configuration involves thousands of state variables. Even if some smaller configurations can be model-checked automatically, there are many larger important configurations for which the only kinds of automatic formal checks that are feasible are whether the specification is complete and consistent.

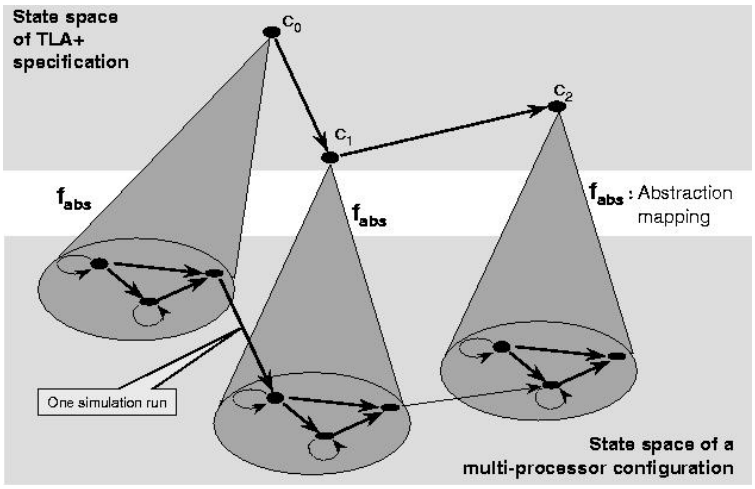

Figure 2: The mapping from the state space of a multi-processor system to that of the corresponding TLA$^+$ specification.

2.4 The Mapping from EV7 to the TLA$^+$ Protocol Description

The prohibitive complexities of the EV7 hardware and the protocol leave simulation as the only validation tool for practical multi-processor configurations. We aim to make the most of simulation resources by (i) making the correctness checking during simulation runs formal and rigorous, and (ii) directing simulation runs to portions of the specification that have not been yet explored. (ii) is discussed in Sections 2.5. To achieve (i), we define a mapping from simulation runs of the implementation to runs in the state space of the specification. This mapping has the form of a state abstraction function f_{abs} (Figure 2) where each implementation state s is mapped to a state of the specification $c = f_{abs}$. For each transition in the implementation $s_i \rightarrow s_j$ taken during a

simulation run, we compute $c_i = f_{abs}(s_i)$ and $c_j = f_{abs}(s_j)$, and use TLC to check whether the transition $c_i \rightarrow c_j$ is allowed by the TLA$^+$ specification for the EV7 design. This mapping can reveal implementation errors, i.e. conflicts with the specification, as well as gaps in the specification where no next transition is allowed from a certain state. The latter form of inconsistency is serious if it is not due to an error in formalizing the specification. It may mean that a possible system state was not considered during the protocol design, and the hardware is doing something random.

The mapping was implemented as a C++ module that was integrated into the simulator. At each clock phase, from the implementation state s_i, the mapping module computes a specification state c_i, and invokes TLC to check if the transition from the previous specification state c_{i-1} to the current one was legal.

The TLA$^+$ specification is written at a much higher level and much coarser granularity of time than the hardware implementation. Packets arriving at or leaving the $CBox$, $ZBox$ and the Net can take a large number of clock ticks. Directory and cache state, which appear readily available and instantly updated in the TLA$^+$ specification, require separate requests and responses to access and update in the actual hardware. The hardware makes use of speculation, pipelining, parallelism and redundancy, and a number of other optimizations that are below the level of detail of the specification. Therefore, relating runs in the implementation to runs in the EV7 specification is a complex task.

To bridge the gap between the specification and the implementation, the mapping module adopts a two-phase approach to mapping implementation runs to specification runs. The implementation state is first mapped to an intermediate, *auxiliary state*. This first phase of the mapping involves monitoring the interfaces between major data structures in the $CBox$ and the $ZBox$, packets being sent between the $CBox$, $ZBox$ and the Net and state updates of important control state machines inside the hardware modules and recording them as part of the auxiliary state. The second phase of the mapping computes the abstract state using the auxiliary state only. The auxiliary state contains all the information required to deduce if all the events that are the preconditions of a TLA$^+$ protocol action have taken place and all the state updates required by the TLA$^+$ action have been observed. The second phase of the mapping then updates the abstract state. Implementation details such as long packets that are in the process of leaving or entering a hardware module and speculative execution update the auxiliary state but leave the abstract state unchanged unless all conditions required by a TLA$^+$ action have been met.

The updates to auxiliary state can be viewed as tokens, some of which, upon an update of the abstract state, are consumed by a TLA$^+$ action. If a sequence of auxiliary state updates correspond to no TLA$^+$ action, no update of the abstract state occurs, and tokens keep accumulating. The mapping module detects if the number of unconsumed tokens is larger than can be consumed by any applicable action and issues a warning if this is the case.

Using this two-phase approach, all implementation events that are relevant to the EV7 specification are recorded in the auxiliary state and transitions in the abstract state are synchronized with the way the specification updates its state. We believe that this way of constructing mapping functions is especially well suited to complex designs where a large team of designers implement hardware sub-blocks. Each hardware designer can easily construct the part of the mapping that extracts the auxiliary state relating to his sub-block. Then architects responsible for the global design can construct the mapping from the auxiliary state to the abstract protocol state. This separation allows conceptual errors in the protocol errors to be distinguished from implementation errors, and makes detection and correction of such errors easier.

The portion of hardware involved in the mapping was described using approximately 20 thousand lines of HDL code. The mapping itself took eight thousand lines of C++ code, including comments. The mapping was constructed by formal verification researcher after the design was complete, which made the process time consuming. The cache coherence hardware of a future design was first specified using TLA$^+$ and it was expected that the task of extracting the information from the design required for such a mapping would be passed down to the implementors of hardware components as described above.

2.5 Simulation Guided by Formal Specification State Coverage

Formal specifications used as simulation monitors offer important benefits at the cost of writing abstraction mappings. However, their real value lies in their use for automating coverage-guided validation. During simulation, coverage data, such as state or transition coverage, can be collected on a formal specification. Gaps in specification coverage then serve as targets for formal verification tools that operate on the specification. For instance, a model checking or guided state-space search tool can generate a path to an interesting, unvisited state of the specification. Such a path is intuitive to understand, and provides a very useful starting point for simulation input generation, whether it is being done by hand or by the aid of automatic tools such as constraint solvers. Due to the complexity of the implementation, it is infeasible to achieve this directly at the hardware level. When the specification state space is too large to use automatic formal verification tools on, a more relaxed version of the specification such as a projection of it to a smaller number of variables can be used.

A major difficulty in making use of coverage information is identifying which coverage gaps are due to insufficient simulation and which ones are scenarios that cannot happen. The use of formal specifications for coverage measurement alleviates this difficulty. Formal verification tools can determine or conservatively estimate whether a coverage target is reachable or not. Coverage gaps that appear due to insufficient simulation can then be given priority in test generation. While this is not a complete solution, it is a very useful aid to verification engineers.

In our framework, we use TLC to store the set of specification states which are visited during simulation runs. To keep coverage information managable, we make use of symmetry reductions and views as mentioned in Section 2.1. We are experimenting with several view functions. Our starting point is the set of coverage goals that are already being used by verification engineers. These are at the same level of abstraction as the EV7 specification and are easily expressed in terms of the specification state variables. Using views and symmetries reduces the set of states we aim to cover. We then identify an unvisited state c in this reduced space and use the counterexample generation capability of TLC to generate a path to c. Since the abstract specification closely reflects the structure of the EV7 design, translating this path into a simulation run of the EV7 is a manageable task.

3 Conclusions

We presented a technique for using formal specifications of hardware as simulation monitors, coverage analysis, and coverage-guided generation of simulation input vectors. Our approach makes the process of checking functional correctness during simulation formal and rigorous, and enables

formal coverage analysis, and automation for directing simulation runs towards coverage gaps. We demonstrate the efficacy of our approach on verifying the cache coherence engine of the Compaq Alpha 21364 microprocessor.

References

[1] H. Akhiani, D. Doligez, P. Harter, L. Lamport, M. Tuttle, and Y. Yu. `http://research.microsoft.com/users/lamport/tla/fm99.pz.Z`. 1999.

[2] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[3] L. Lamport. *Specifying Systems (Preliminary draft)* `http://research.microsoft.com/users/lamport/tla/book.html`.

[4] Y. Yu, P. Manolios, and L. Lamport. Model checking tla$^+$ specifications. In *Proceedings of the IFIP Working Conference on Correct Hardware Design and Verification Methods, CHARME*, Lecture Notes in Computer Science 1703, pages 54–66, 1999.