Checking Cache-Coherence Protocols with TLA$^+$

Rajeev Joshi
HP Labs, Systems Research Center, Palo Alto, CA.

Leslie Lamport
Microsoft Research, Mountain View, CA.

John Matthews
HP Labs, Cambridge Research Lab, Cambridge, MA.

Serdar Tasiran
HP Labs, Systems Research Center, Palo Alto, CA.

Mark Tuttle
HP Labs, Cambridge Research Lab, Cambridge, MA.

Yuan Yu
Microsoft Research, Mountain View, CA.

Abstract. We have a great deal of experience using the specification language TLA$^+$ and its model checker TLC to analyze protocols designed at Digital and Compaq (both now part of HP). The tools and techniques we have developed apply equally well to software and hardware designs. In this paper, we describe our experience using TLA$^+$ and TLC to verify cache-coherence protocols.

Keywords: TLA$^+$, TLC, model checking, cache coherence.

1. Introduction

TLA$^+$ is a specification language for describing and reasoning about asynchronous, nondeterministic, concurrent systems [4, 5]. A specification in this language is a single formula that describes a state machine in terms of an initial condition, a next state relation, and possibly some liveness conditions. TLA$^+$, however, is a high-level language with the full power of set theory, predicate logic, and temporal logic, so this formula is a more compact and often more readable specification than is possible in typical specification languages. TLC is a model checker for specifications written in TLA$^+$ [5, 7]. No model checker can handle every specification written in a language as expressive as TLA$^+$, but TLC has accepted the specifications that have arisen in practice. It is an explicit-state, on-the-fly model checker written in Java that runs on any sequential, parallel, or distributed system with Java support.

Our experience is that TLA$^+$ and TLC can be quite useful in an industrial setting when applied at the right level of abstraction. This level of abstraction, in fact, might be one of the distinguishing features of our work. TLA$^+$ is best suited for reasoning about protocols, in contrast to other languages designed for reasoning about implementations. We get the most value from TLA$^+$ when it is used as early as possible in the design phase, even as early as sketches on a white board. In our personal use, we have used TLA$^+$ and TLC to debug protocol ideas in a matter of hours. This has kept us from pursuing dead ends for days before realizing our mistakes. In our industrial applications, we have usually entered the picture quite late in the engineering cycle, well after the protocol has been designed, but engineers who have used TLA$^+$ early in the design phase report experiences similar to ours.

In this paper, we describe our experience applying TLA$^+$ and TLC to several cache-coherence protocols. In a shared-memory multiprocessor, the processors communicate by reading and writing values in memory locations. Each processor has a cache in which it stores local copies of these values for easy access. The *memory model* defines the relationship among the values read and written by the processors. The *cache-coherence protocol* is the algorithm that is responsible for implementing the memory model by preserving suitable relationships among the cached values. Cache coherence protocols are at the heart of multiprocessor designs, and aggressive optimizations for performance and scalability make modern protocols quite tricky, so they are attractive targets for the application of formal methods.

2. Alpha cache coherence

Our first application was to cache-coherence protocols for servers based on two generations of the Alpha processor [1], the EV6 (Alpha 21264) and the EV7 (Alpha 21364).

2.1. The EV6 protocol

Our first project was the EV6 cache-coherence protocol [3]. Heavy optimization makes this the most complicated cache-coherence protocol any of us has ever seen. It is a hierarchical protocol, in that the system consists of four-processor nodes connected by a cross-bar switch, and the protocol does everything it can to satisfy requests for memory locally within a node without generating any coherence traffic on the switch. Making verification even more difficult, the protocol actually separates the generation of commit events used to order memory requests from the generation of responses to requests. The protocol makes use of this separation in two ways. First, it allows commit events sent to a processor to bypass other messages to the processor, decreasing the amount of time the processor spends at memory barriers waiting for prior memory requests to become ordered. Second, it can generate the commit event well before it formulates the reply to a request, allowing a processor to move past memory barriers even before the data sent in response to the memory request has been determined. Even the designers found it intuitively surprising that this optimization worked [3]. This ability to generate early commit events depends heavily on ordering properties of the switch, and several unusual data structures in each node are required to make it work.

2.1.1. *The coherence protocol*

We began by writing a precise description of the protocol.

Our first obstacle was obtaining a coherent understanding of the protocol. Coming in after most of the design had been completed, we received a two-inch stack of a dozen documents describing the system, documents written at different stages of the design and therefore sometimes inconsistent with one another. One artifact that was crucial to understanding the design was a simulator that one designer had written in Lisp.

Our second obstacle was the complexity of the protocol. Since some messages had slightly different semantics in different message queues, there were effectively over 60 kinds of messages in the system. One of the important abstractions we made was to break those messages down into 15 units of functionality that we called *quarks*, and to model messages as sets of quarks. This dramatically simplified the description of the protocol.

The specification wound up being 1900 lines long, which is a large specification in a language as expressive as TLA$^+$.

2.1.2. *The Alpha memory model*

We wanted to show that the cache-coherence protocol implemented the Alpha memory model with an invariance proof, and such proofs involve reasoning about the transitions of a state machine. The Alpha memory model [1], however, is defined in terms of sequences of operations like loads and stores. We needed a definition of the memory model in terms of state transitions, so we decided to write our own in TLA$^+$, making a few simplifications such as considering only loads and stores to whole cache lines. Our model is a state machine that receives memory requests, determines return values, and generates responses. It guarantees that all memory operations appear to be performed in some order that we call the *Before* order, and that all return values are consistent with this order. The heart of the model is a predicate called *GoodExecutionOrder* that describes the properties this *Before* order must satisfy. The entire specification is only 200 lines, and the *GoodExecutionOrder* predicate is only 40 lines, demonstrating the expressiveness and compactness of TLA$^+$ specifications.

2.1.3. *The proof*

After spending months just to understand the protocol and describe it in TLA$^+$, it became obvious that we did not have the time or the people to write a complete correctness proof by hand, and TLC had not yet been written. As a way of finding bugs, we focused on writing the protocol invariant—the invariant we would have used for a refinement proof that the coherence protocol implements the memory model. We wrote about 1000 lines of an informal invariant, and we focused on two conjuncts of about 150 lines each that described what we considered to be the most error-prone parts of the system. These conjuncts described the sequences of messages that could appear in two message queues separating three important data structures.

This turned out to be hard work, although it was made easier by the fact that the specification and invariant were both written in TLA$^+$, in contrast to other systems where the state machine is written in one language and the correctness properties in another. We completed the proof for the first conjunct, which amounted to about 2000 lines of proof with 13 nested levels of case splits. We completed an equal amount of the proof for the second conjunct, but the chance of finding additional errors appeared too small to justify completing the proof.

Based on our past experience writing correctness proofs for concurrent algorithms, we expected to find dozens of errors, but we found only two of interest. The first was an easily-fixed error in an entry of a table. The simplest scenario exhibiting it required four processors, two memory locations, and over fifteen messages. We believe that this error could have been found only by writing a proof, since simulation was unlikely to find it (in fact, it didn't) and the protocol's complexity put it beyond the ability of any model-checking technology we were aware of at the time. The second error was a behavior of the protocol that violated the memory model. In the end, the architects decided this was an error in the model and not the protocol, and they revised the official Alpha memory model accordingly.

2.2. The EV7 protocol

The level of success achieved on the EV6 protocol convinced designers of the EV7, the next generation of the Alpha processor, to apply formal methods to their protocol. By this time, the TLC model checker had just been written, and its first major application was to the EV7 protocol. In contrast to the EV6 project, a great deal of the formal verification of the EV7 protocol was carried out by the official EV7 verification team. In fact, the TLA$^+$ specification of the EV7 protocol was written by one of the engineers after about eight hours of instruction and some periodic consultation. It was about 1800 lines long.

2.2.1. *Model checking*

We were pleased to discover that TLC could accept the specification as written by the engineer, without modification. The complexity of the EV7 protocol, however, generated a huge state space. To make the state space tractable, we ran the model checker on a configuration having one cache line, two data values, and three processors, resulting in six to twelve million reachable states. By adding fifteen lines to the specification to encode some symmetry-breaking constraints, we were able to reduce the state space to half this size. TLC could check this configuration in three to six days on what was then a fast workstation.

Engineers running the RTL simulations of the EV7 protocol surprised us by discovering two unexpected applications of TLC and the TLA$^+$ specification. First, they forced their RTL simulator into interesting corner cases by taking traces generated by the model checker and translating them into input stimuli for their simulator. Second, since the next-state relation in a TLA$^+$ specification contains explicit assertions about which variables a transition leaves unchanged, they added these assertions to the list for the RTL simulator to check.

Model checking found about 70 errors. Most of them were specification errors caused by ambiguity in the English specification from which the TLA$^+$ specification was derived. Five were actual implementation errors. One of these was found by TLC itself, and four were found using TLC error traces to generate input to the RTL simulators, showing that the RTL model violated the protocol as described by the TLA$^+$ model.

2.2.2. *The proof*

Because the EV7 protocol is simpler than the EV6 protocol, we were actually able to sketch a complete invariance proof of its correctness. The model checker dramatically simplified the process of writing the proof. We were able to formulate invariants and check them with TLC on small configurations before wasting our time trying to prove them.

2.2.3. *The implementation*

In addition to verifying the correctness of the EV7 protocol, we are in the process of using its TLA$^+$ specification to verify the RTL implementation [6]. While verification of the protocol itself was relatively simple, verification of the actual implementation is complicated by a very aggressive implementation strategy. Our approach is to check that the RTL implementation is a refinement of the TLA$^+$ specification by defining a refinement mapping from the RTL to the TLA$^+$.

This mapping is used in two ways. First, we can check that traces generated by the RTL simulator are allowed by the TLA$^+$ specification. For each transition in a trace, we map the pair of RTL states to a pair of TLA$^+$ states, and then use TLC to check that the TLA$^+$ specification allows a transition between these states. Second, we can keep track of what abstract TLA$^+$ states have been visited by the simulator, identify interesting ones that have not been visited yet, use TLC to generate traces to these states, and translate (as mechanically as possible) these traces into input stimuli to force the RTL simulator into these states.

3. Itanium cache coherence

We also applied TLA$^+$ and TLC to a cache-coherence protocol for the Itanium processor family. This protocol relied on proprietary hardware components from other companies, so we had to write abstract models of their external behavior. Simply writing the protocol specification uncovered ambiguities in the English descriptions and suggested two small design changes.

TLC could not check the TLA$^+$ specification of these protocols on large enough instances to give us adequate confidence in the design. TLC took too much time. This was because the abstract models of the proprietary components allowed many more behaviors than any actual implementation would exhibit, and because the design appeared to require at least four processors to produce interesting scenarios. However, we were able to run meaningful tests by having TLC generate random simulations of some larger instances of the specification.

Perhaps the most interesting artifact to come out of this project is a TLA$^+$ specification of the Itanium memory model [2], initially intended to serve as the correctness condition for the coherence protocol. The Itanium model is more subtle than our version of the Alpha model. For example, unlike the *Before* order in the Alpha model, there may be no single total order on memory requests. The model allows some memory requests to be ordered using acquire/release semantics, and other memory requests to be unordered. These unordered operations are not required to be atomic, and different processors can see unordered operations happen in different orders. One of our current projects is using this formulation of the memory model to check the correctness of hardware implementations of the memory model, and to verify the correctness of software algorithms running on top of the memory model.

4. Conclusions

On the whole, we are satisfied that TLA^+ and TLC can be a valuable pair of tools for debugging industrial-sized protocols when applied at the right level of abstraction and at the right time in the design phase. Designing cache-coherence protocols is a mature discipline now, and the coherence protocols we studied were written by some of the best protocol designers in the world, so we were not surprised that we did not find a large number of bugs in their work. In fact, the designers were generally pleased with the results of our analysis.

We cannot emphasize enough, however, the importance of using these tools early in the design phase. On a number of much smaller projects, where these tools were used early on, we have found a satisfying number of bugs. In standards work, for example, we have used TLA^+ and TLC to find bugs in proposals submitted to the working group for the PCI-X bus protocol. For a database system project, we have used TLC to check database recovery and cache-management protocols. We also routinely use TLC to check the concurrent algorithms we write in the course of our own research. In processor design, EV7 engineers working on what was to be the EV8 believed enough in the value of early specification to write a TLA^+ model of the cache-coherence protocol before any hardware had been designed. One of these engineers, now at Intel, reports that TLA^+ and TLC are starting to become more widely used within the former Alpha group at Intel, as well as generating significant interest in other groups.

Given the power of formal methods early in the design phase, and given that it is nearly impossible to participate in the early design phase without being one of the designers, industry's current approach of dividing teams into the protocol design engineers and the formal verification engineers is a serious mistake. However, the pressure of production environments makes it difficult for designers to find time to learn about formal verification tools. One of the most pressing problems today, therefore, is teaching the next crop of designers to be comfortable with formal verification tools while they are still in school. With this background, designers might think to call on verifiers earlier in the design cycle, and might participate more deeply in the use of the tools.

Our experience drives home the tension between language expressiveness and tool performance, the same tension found with programming languages. TLA^+ is an extremely expressive language, yet we have seen engineers become comfortable reading TLA^+ specifications after fifteen minutes and writing them within a week. TLA^+ was designed to make specifications easy to write, but this comes at the expense of efficient tools for TLA^+. On the other hand, the tools currently demonstrating such impressive performance typically have input languages that are too low-level for the kinds of protocol verification that we want to do. A second pressing problem, therefore, is understanding the extent to which the algorithms underlying these powerful tools can be applied to higher-level languages like TLA$^+$. One promising approach is the use of satisfiability checking to implement bounded model checking with TLA$^+$.

Acknowledgements

We were assisted by many colleagues in this work. On the EV6 project: Madhumitra Sharma helped us understand the protocol, and Paul Harter helped write the proof. On the EV7 project: Joshua Scheid wrote the specification, Homayoon Akhiani and Jonathan Nall implemented the TLA$^+$-to-RTL translator, Damien Doligez wrote the invariance proof, and Scott Kreider, Scott Taylor, and Brannon Batson helped with the RTL refinement checking. On the Itanium project: Jae Yang helped specify the cache-coherence protocol, and Gil Neiger helped specify the Itanium memory model. On other projects: Thomas Rodeheffer worked on the PCI-X bus protocol, and David Lomet worked on the database protocols. We thank three referees for their comments on this paper.

References

1. Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.
2. Intel Corporation. *IA-64 System Architecture*, volume 2 of *Intel IA-64 Architecture Software Developers Manual*. Intel, July 2000.
3. Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.
4. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
5. Leslie Lamport. *Specifying Systems*. Addison-Wesley Publishing Company, 2002.
6. Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *In Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, June 2002.
7. Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer-Verlag, September 1999.