The Wildfire Challenge Problem

Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu

Compaq

4 Jan 2001

Abstract

We pose as a challenge to the verification community the problem of finding errors in the specification of a complicated cache-coherence protocol. It specifies a simplified version of the protocol used in an actual multiprocessor computer, except with one error deliberately introduced and another introduced by accident. The protocol and the memory model it is supposed to implement are described here; their complete specifications are posted on the Web.

Contents

1 Introduction

A few problems have become benchmarks for verification techniques. For example, Tomasulo's algorithm is a popular one for concurrent-system verification [2, 4, 6, 7, 12]. Such problems have posed the task of verifying an algorithm that is already known to be correct. A solution must be examined carefully to discover what has really been verified, since an error in formalizing the problem could render it vacuous. The Wildfire challenge problem poses a very different task: finding the error in an incorrect algorithm. As long as the solution has not been revealed in advance, we know that the problem has been solved if, and only if, the error has been found.

Wildfire was the code name for the Compaq AlphaServer GS series [5], a family of multiprocessor computers containing up to 32 Alpha processors. It has the most complicated cache-coherence protocol we know of, into which we have deliberately inserted a bug. We provide formal specifications, written in TLA^+ [10], of the incorrect protocol and of the Alpha memory model, which the protocol is supposed to implement. We challenge the verification community to find the error.

In most industrial applications, the function of verification is to find errors. It is only for certain life-critical applications that one performs the kind of complete verification that attempts to verify the correctness of the entire system. So, the problem of finding errors in a system design is a realistic one. The one unrealistic aspect of the challenge is that we announce in advance the presence of a serious error.

The problem has already been solved by Georges Gonthier. In addition to the error we had planted, he found another bug that we were not aware of—a bug in our specification, not in the actual protocol. Later, Abdelwaheb Ayari found that other bug, but not the one we had deliberately inserted. Gonthier solved the problem by inspection; Ayari used a state-enumeration tool. No other solutions have been reported to us.

This paper describes the history of the problem, the Alpha memory specification, and the Wildfire protocol specification. We do not attempt to present the specifications themselves. They are posted on the Web [11] and are thoroughly commented. For obvious reasons, we do not divulge any information that might help locate the errors. We reveal only that both errors are violations of safety properties, so they are demonstrated by finite traces.

2 The History of the Problem

Three of the authors (LL, MT, and YY) are researchers and the fourth (MS) is an engineer who was a member of the Wildfire design team. From the fall of 1996 through the summer of 1997, the three researchers engaged in a project to verify the correctness of the Wildfire cache-coherence protocol. They worked closely with the designers—especially the designer/author. The project began with the writing of a fairly low-level TLA^+ specification of the protocol, which was about

1900 lines long (exclusive of comments). The researchers then wrote a 200-line TLA$^+$ specification of the Alpha memory model, which any multiprocessor Alpha must implement. A complete verification that the low-level protocol implements the specification was impossible in the time available. Working closely with the designer, the researchers wrote a detailed hand proof for a high-level view of part of the protocol. They then wrote an even more detailed proof for certain parts of the low-level protocol that seemed most error-prone. The proofs revealed one error in the Alpha reference manual [1], the official description of the Alpha architecture, and one minor low-level error in the protocol.

In early 2000, we wrote a high-level view of the protocol that abstracts away a number of complicating details in the lower-level specification. This specification consists of about 730 lines of TLA$^+$. We hope eventually to publish a rigorous proof of its correctness.

When first writing the lower-level TLA$^+$ specification, the researchers were puzzled by one detail in the protocol. The designer had to explain to them why it was necessary. Overlooking this detail seemed easy, and it would result in the kind of subtle error that verification is supposed to catch. We decided to publish the higher-level specification with the detail omitted and challenge the verification community to find the error. We posted the problem on the Web [11] in mid-June, 2000.

In early August, Georges Gonthier sent us a solution, which he had obtained by simply thinking about the problem. We were surprised and impressed by how quickly he solved it. He found not only the bug we had planted, but another, unintentional error. That error had also been present in our original specification. It was in a part of the specification that we had not examined in either our low-level or high-level proofs; it was not present in the actual protocol. It is much more obvious than the error we planted, and we are rather surprised and embarrassed that we hadn't noticed it ourselves. Gonthier reported:

> I did spot a bug, but was a bit disappointed because it seemed too trivial, so I carried on, as I was interested in understanding how the whole thing ticked... It just dawned on me last night that there was indeed a more subtle problem with the spec.

We decided not to correct the unintentional error and to make finding it an additional part of the challenge. In late October, Abdelwaheb Ayari found this more obvious error, using a state-enumeration Haskell program that was based on ideas by David Basin [3]. Ayari reports having spent about four weeks on the problem, most of it modeling the protocol with his tool.

As of January, 2000, the problem had been downloaded from about 270 different sites. No-one else has reported finding either bug.

3 The Alpha Memory Specification

The Alpha memory model describes the result of concurrent memory accesses by different processes. To permit efficient implementation of multiprocessor memories, the Alpha designers wanted the weakest model possible that would provide the mechanisms needed for multiprocess programming. The model is specified informally in the Alpha Reference Model [1]. It allows a processor to reorder operations to different memory addresses, except where prohibited by explicit memory barrier instructions. The Alpha also provides load-locked and store-conditional instructions for interprocess synchronization.

The informal specification and our TLA$^+$ specification of the memory model differ in several ways:

- The informal specification describes the result of concurrent operations to different parts of the same word. The TLA$^+$ specification eliminates this complication by describing only operations that access a complete cache line, since that is the level of granularity at which the Wildfire protocol operates.

- The informal specification is in terms of abstract operations, with no mention of when the operations are issued. It originally allowed certain violations of causality that were only later ruled out by inserting additional restrictions. The TLA$^+$ specification is in terms of states and transitions, and issuing an operation is an explicit action. The straightforward way of writing such a specification makes it easy to ensure causality.

- We have simplified the TLA$^+$ specification for the challenge problem by omitting some features of the Alpha memory system, such as the *write memory barrier* instruction.

- The informal specification describes the memory system's interface with the program, while the TLA$^+$ specification describes its interface with the processors.

One difficulty posed by the challenge problem arises because the TLA$^+$ specification contains two "hidden" variables:

reqSeq: An array, indexed by processor, such that $reqSeq[p]$ is the sequence of requests issued by processor p, together with their execution status.

beforeOrder: A partial order on the set of all issued requests that "explains" their results.

The informal specification is mirrored in a predicate *BeforeOrderOK* on these variables that is required to hold after each step. For example, *BeforeOrderOK* asserts that the value returned by a read is obtained from the latest write that precedes it in the partial order *beforeOrder*. In TLA, variable hiding is expressed by temporal existential quantification, so the specification has the form $\exists\!\!\!\exists\, reqSeq, beforeOrder : F$ for a temporal formula F. A protocol satisfies this specification iff, for each of its possible behaviors, there exists a sequence of values for *reqSeq* and for *beforeOrder* such that F is satisfied. The large number of potential choices of such sequences makes brute-force mechanical verification infeasible.

To a first approximation, the Alpha memory model is a simple weakening of sequential consistency [9]. Sequential consistency asserts that the memory acts as if all instructions are executed in some sequence that has just one requirement: any pair of operations from the same processor appear in the order that they are issued. The Alpha memory model weakens this requirement so it applies only to instructions that access the same address or are separated by a memory barrier instruction.

4 The Wildfire Specification

As shown in Figure 1, the Wildfire system consists of a network of processors and memories connected via local switches, which are in turn connected via a single global switch. Any processor can access any memory address, but access is faster if that address resides on the processor's local switch.

The two levels of switches make the system more complicated than traditional multiprocessor memories. This complication is exacerbated by the design decision that, if a memory address is accessed only by processors on the same local switch, then no messages are sent to the global switch. Unlike conventional

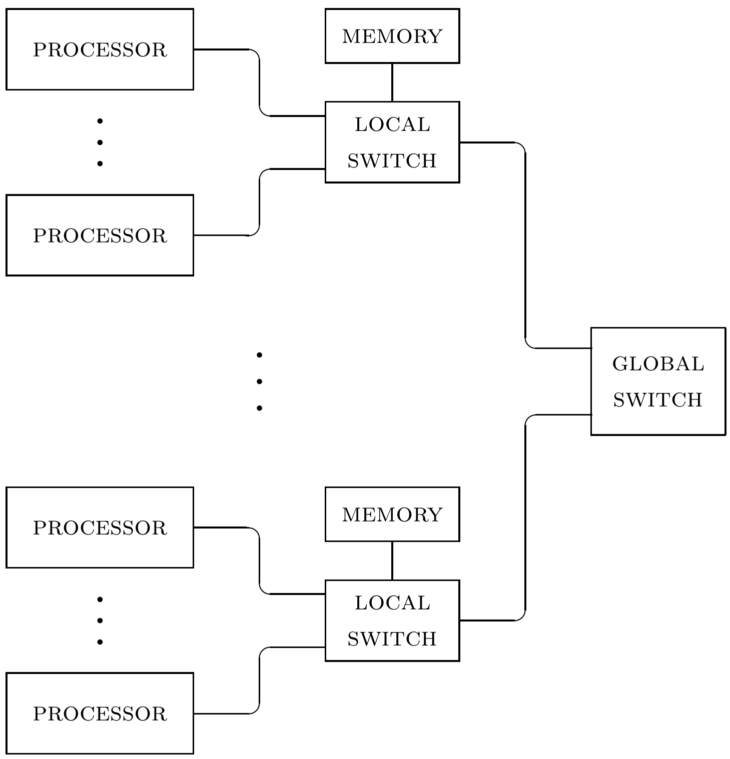

Figure 1: The structure of the Wildfire system.

bus-based protocols, the Wildfire protocol has no central point of synchronization through which all requests pass. This makes it trickier to satisfy the memory model.

The Wildfire protocol is directory based. All read and write operations are performed to data in a processor's cache. If the cache does not have a valid copy of the data in the appropriate state, it sends a request to the local switch at which the memory address resides (the "directory"). The switch either satisfies the request from memory or else sends a message to the processor that currently "owns" the address. On a *Write* request, the requester becomes the new owner, and the message causes the current owner to invalidate the copy in its cache.

In a conventional directory-based protocol, when the owner of an address receives a message invalidating its copy, it replies to the directory with a message that both acknowledges receipt of the invalidating message and returns the data. The directory then forwards the data to the new owner. In Wildfire, the old owner sends the data directly to the new owner, and it sends nothing back to the directory. In general:

- Processors do not acknowledge cache-invalidate messages. This eliminates the need to wait for those acknowledgments.
- Control signals and data are sent in separate messages. This allows a processor to perform an operation before control signals return, and to execute operations that follow a memory barrier before all operations that precede the barrier have been completed.

To illustrate how these optimization complicate the protocol, we describe one scenario. Suppose that process P1 on local switch LS1 is the current owner of an address A that resides on local switch LS2, and processes P2 and P3 on LS2 each wants to execute a partial write to address A, followed by a memory barrier, followed by additional operations. Here is one possible execution sequence:

- P2 sends a *Write* request to LS2, which receives it, records P2 as the new owner of address A, and sends a *ForwardedGet* message for P1 to the global switch.
- The global switch relays the *ForwardedGet* to LS1 and sends an acknowledgement, called a *ShadowClear*, back to LS2.
- LS1 receives the *ForwardedGet* and relays it to P1, which receives it, invalidates address A in its cache, and sends the data to P2. (The data actually travels to P2 via the switches, but our model simplifies the protocol by using separate virtual channels, not shown in Figure 1, for data.)
- P3 sends a *Write* request to LS2, which receives it, records P3 as the new owner of address A, and issues a *ForwardedGet* message for P2. Because the *ShadowClear* has not returned, LS2 does not send the *ForwardedGet* directly to P2, but instead sends it to the global switch, which relays it back to LS2.

- LS2 receives the *ShadowClear* and sends an acknowledgement, called a *Comsig* message, to P2.
- P2 receives the *Comsig*, which allows it to execute the memory barrier instruction and begin executing its next operations.
- LS2 receives the *ForwardedGet* from the global switch, sends it to P2 and sends a *Comsig* message to P3.
- P3 receives the *Comsig*, which allows it to execute the memory barrier instruction and begin executing its next operations.
- P2 receives the data from P1, which it puts in its cache and updates with the write operation.
- P2 receives the *ForwardedGet*, invalidates address A in its cache, and sends the data to P3, which will then be able to perform its write operation.

Observe that P2 and P3 execute operations that follow the memory barrier instructions before the writes that precede those memory barriers are completed. Observe also that LS2 sends a *ForwardedGet* message to P2, one of its own processors, via the global switch. Had the *ShadowClear* message arrived first, LS2 would have responded to P3's *Write* request by sending the *ForwardedGet* directly to P2 and sending a *Comsig* directly to P3.

Of course, this is just one of many possible scenarios. The protocol contains several complicating details that we have not even mentioned. Its full complexity can be appreciated only by reading the formal specification.

5 Discussion

Verification of cache-coherence protocols has been a popular activity, as the survey of Pong and Dubois [13] shows. No protocol verified thus far is as complex as Wildfire's, which is the most complicated cache-coherence protocol we know of. Even our high-level description, which abstracts away many of the details, is probably more complicated than any published protocol. As ever more complicated algorithms are implemented in hardware, we can expect this degree of complexity to become the norm. We therefore feel that our challenge problem provides a good benchmark for methods that hope to verify the concurrent algorithms in real systems.

By providing a formal specification of the Wildfire protocol, we have eliminated a major task that arises in practical verification—namely, writing a formal model of the algorithm at a suitable level of abstraction. Our specification is written in TLA$^+$, a language based on TLA (the temporal logic of actions). Various documents explaining TLA and TLA$^+$ appear on the TLA Web site [8]. (Two of those documents are distributed with the challenge problem.) However, comments describe the TLA$^+$ constructs as they occur in the specification, and most readers will probably need no further explanation of the language. We have found that engineers can learn to read the safety part of a TLA$^+$ specification in about half an hour, so understanding our specifications should pose no problem to verification researchers. (We don't have enough experience explaining liveness to engineers to say with confidence how long it takes, but we expect that researchers will have little trouble understanding the protocol's liveness properties.)

We obviously do not expect the problem to be easy. As with any complicated concurrent algorithm, the large number of possible states makes a brute-force approach unlikely to succeed. In our abstraction, the processors maintain unbounded queues of unfulfilled requests, so the state space is infinite. Since the errors are violations of safety properties, they can be found by placing a large enough bound on the number of requests. However, the number of reachable states grows very fast as a function of the number of requests. For a configuration with one-bit data words, two local switches each containing one processor and one memory address, and a fixed initial memory state, the number of reachable states is about 160 thousand with a bound of two requests and about 38 million with a bound of three requests.

Although it has been downloaded from over 275 sites, we do not know how many researchers have seriously tried to solve the problem. We suspect that a number of them began working on it but gave up because it was more difficult than they expected. We hope that by publishing the challenge here, we will encourage the verification community to try harder.

References

[1] Alpha Architecture Committee. *Alpha Architecture Reference Manual*. Digital Press, Boston, third edition, 1998.

[2] Tamarah Arons and Amir Pnueli. Verifying Tomasulo's algorithm by refinement. In *12th International Conference on VLSI Design*, 1999.

[3] David Basin. Lazy infinite-state analysis of security protocols. In *Secure Networking—CQRE [Secure] '99*, volume 1740 of *Lecture Notes in Computer Science*, pages 30–42, Düsseldorf, Germany, November 1999. Springer-Verlag.

[4] Werner Damm and Amir Pnueli. Verifying out-of-order executions. In Hon F. Li and David K. Probst, editors, *Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME)*, pages 23–47. Chapman and Hall, 1997.

[5] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of AlphaServer GS320. In Anoop Gupta, editor, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 13–24, November 2000.

[6] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. You assume, we guarantee: Methodology and case studies. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-aided Verification 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

[7] Daniel Kröning, Silvia M. Müller, and Wolfgang Paul. A rigorous correctness proof of the Tomasulo scheduling algorithm with precise interrupts. In *Proc. of the SCI'99/ISAS'99 International Conference*, 1999.

[8] Leslie Lamport. TLA—temporal logic of actions. At URL `http://www.research.digital.com/SRC/tla/` on the World Wide Web. It can also be found by searching the Web for the 21-letter string formed by concatenating `uid` and `lamporttlahomepage`.

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

[10] Leslie Lamport. Specifying concurrent systems with TLA$^+$. In Manfred Broy and Ralf Steinbrüggen, editors, *Calculational System Design*, pages 183–247, Amsterdam, 1999. IOS Press.

[11] Leslie Lamport, Madhu Sharma, Mark Tuttle, and Yuan Yu. The wildfire verification challenge problem. At URL `http://www.research.compaq.com/SRC/tla/wildfire-challenge.html` on the World Wide Web. It can also be found by searching the Web for the 24-letter string `wildfirechallengeproblem`.

[12] Kenneth L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-aided Verification 10th International Conference, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 110–121. Springer-Verlag, 1998.

[13] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *Computing Surveys*, 29(1):82–126, March 1997.