State Pruning for Test Vector Generation for a Multiprocessor Cache Coherence Protocol

Ying Chen† Dennis Abts* David J. Lilja†
† Electrical and Computer Engineering
University of Minnesota
Minneapolis, Minnesota 55455
{wildfire, lilja}@ece.umn.edu

*Cray Inc.
P.O. Box 5000
Chippewa Falls, Wisconsin 54729
dabts@cray.com

Abstract

Verification is extremely important in designing digital systems such as a cache coherence protocol. Generating traces for system verification using a model checker and then using the traces to drive the RTL logic design simulation is an efficient method for debugging. This method has been called the witness string method [2], which is based on DFS. We investigate a state pruning method that exploits multiple search heuristics in simultaneous DFS searches to choose the most efficient traces. We distribute the hash table of the entire state space among the simultaneous searches so that they cooperate to avoid redundant state exploration. To evaluate this new search algorithm, we implant several protocol bugs in the Stanford DASH cache coherence protocol. Using an IBM Power4 system with the Berkeley Active Message library, we show an improvement in witness strings generation through the state pruning method over a pure DFS and a guided DFS.

1. Introduction

System verification in the pre-silicon state of development is crucial for controlling the budget and the time to market. Safety critical systems, such as a cache coherence protocol, are not trivial to verify because there are numerous correctness properties that result in a very large state space that causes the state explosion problem when enumerating states for formal verification.

In order to alleviate this problem, the cache coherency models are usually "down-scaled." Verification of the "down-scaled" models cannot guarantee the correctness of the designs, but is useful as a debugging tool [1]. The witness string approach [2] is such a method that captures bug traces from depth-first-search (DFS) of a state space to help debug the RTL design of a cache coherence protocol. The efficiency of the resulting witness strings is very important. For instance, in the Cray X1 cache coherence protocol, a single witness string can be on the order of a few thousands states. It takes a considerable amount of time to execute all of the necessary witness strings in the RTL logic simulation [3].

One method to determine efficient witness strings is through bug-oriented heuristic in a DFS search of the state space [4]. Different heuristics in a single exhaustive DFS search can improve bug discovery, but the efficiency of the resulting witness strings is not consistent for different kind of bugs. In this study we propose a method of pruning out redundant states using multiple simultaneous DFS searches that each exploits a different heuristic. To implement the multiple DFS searches, we use a parallel algorithm. Similar to the parallel BFS [5], the hash table for the whole state space is distributed among the processors with each state sent to its owner processor according to a hash function. First, a master DFS search uses a breadth fist search (BFS) to generate the frontier state, which is similar to [6]. Then each DFS search uses the frontier state it receives as the start state to begin its search. To reduce redundant searches, we use communication among the multiple DFS searches to coordinate the searches and thereby prune out redundant state exploration.

In the remainder of this paper, Section 2 discusses the problems in test vector generation. In Section 3, we explain the state pruning method. The experimental setup and benchmarks are described in Section 4 with the results shown in Section 5. Section 6 discusses related work. Finally, Section 7 summarizes and concludes.

2. Test vector generation

To verify the implementation of a digital system, manually generated test vectors are typically simulated

on a hardware language description of the design. However, manually generated test vectors cannot guarantee equivalence between the logic design and the specification.

To provide a solid link between the specification and the implementation, the witness string method [2,4] has been proposed. This method records the events, which are called witness strings, during a DFS of the state space of a formal model as shown in Figure 1. The witness strings are later executed as test vectors on the RTL implementation simulation. In this manner, high quality test vectors are automatically generated by avoiding a large fraction of the redundant states that would be visited during random simulations.

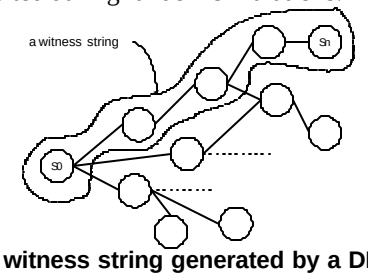

Figure 1. A witness string generated by a DFS of the coherence protocol state space.

As mentioned, in real systems, the number and sizes of the generated witness strings could be very large. We tried to improve the efficiency by exploring several DFS-based heuristics to guide the DFS to generate "better" witness strings first [4] but ended up with "one size does not fit all" heuristic problem, which leads to sub-optimal performance when used across a broad spectrum of error types.

3. State pruning

To solve the "one size does not fit all" problem, we propose a state pruning method that uses multiple heuristics, of which each targets different portions of the state space with different types of bugs, simultaneously in several separate DFS searches. Each search maintains its own hash table and uses a different DFS-based heuristic. Communication among the multiple searches prunes out redundant state exploration to thereby choose more efficient witness strings than those obtained with both straight DFS and guided DFS [3].

3.1. DFS-based heuristics

Previous work [4] explored the benefit of applying a heuristic to guide the depth-first search (DFS) of a model checker verifying a cache coherence protocol. This work compared four different heuristics (i.e., max Hamming distance, min Hamming distance, max cache_score, min cache_score) with the goal of improving the efficiency by reducing the number of searched states before discovering the error. The details of the heuristics are described in [4].

3.2. Basic algorithm

We use multiple DFSs concurrently. Each DFS uses a different one of the four heuristics mentioned above to guide its search. The hash table for the whole state space is distributed among the DFSs using a hash function. Each DFS is the owner of the states distributed in its hash table. The basic algorithm for state pruning is shown in the following steps:

Step 1: A BFS on a master search (DFS search0) generates enough initial frontier states.
Step 2: The frontier states are distributed among the DFS searches (DFS search 0,1,2,3) using a hash function. Each DFS search receives one start state.
Step 3: Upon receiving its start state, each search begins a DFS search. There is communication among the DFS searches to avoid redundant state exploration.

> **Step 3.1:** In each current state a DFS search chooses its next state according to its local heuristic algorithm described in Section 3.1 [4].
> **Step 3.2:** Each DFS search sends its chosen next state to its owner according to a hash function.
> **Step 3.3:** On receiving a state sent by a sender to an owner, each DFS search checks whether the state is in its hash table. If it is not, the state is added into the owner's hash table. Then the owner sends a message to the sender telling whether the state has already been explored.
> **Step 3.4:** Each DFS receives a message from the next state's host. If the next state has not been explored, the DFS search continues and sets the next state to be the current state.
> **Step 3.5:** Whenever a DFS search reaches a bug in the design being tested, or finishes searching, a final report for this search is printed.

Since the types of bugs that may exist in an actual system are usually unknown, this new algorithm solves this problem of not knowing which search heuristic to use in a specific situation by using several different heuristics simultaneously.

4. Experimental setup

4.1. Model checker

We implemented the state pruning method into a model checker built on our previous work [4] using Murphi [9], the parallel Murphi model-checker [5] written for the Berkeley Active Messages library, and the parallel RW algorithm [6].

4.2. Benchmark

The Stanford DASH multiprocessor implemented a directory-based coherence protocol that uses NAKing. It is the predecessor to the SGI Origin2000 [10] cache coherence protocol. The formal model of DASH has three types of requesters: the *local* or *remote* cluster, and the *home* cluster. It also has two classes of

memory references to local or remote memory. The total state space is 10,466 states. We implanted bugs in the system by inspecting state diagrams of the protocol given in [9]. The specific bugs implanted are described below.

BUG1 Handle read exclusive request to home.
No invalidation is sent to the master copy requesting cluster, which has already invalidated the cache.
Invariant violated: ***Consistency of data***
BUG2 Send reply message instead of NAK.
Instead of a negative acknowledgement (NAK) message, an acknowledgement (ACK) message is sent in reply.
Invariant violated: ***Explicit writeback for non-dirty remote***
BUG3 Handle read request to remote cluster.
Instead of changing the cache block in the remote cluster to be locally shared after sending the data block to the requesting remote cluster, the cache block remains locally exclusive.
Invariant violated: ***Consistency of data***
BUG4 Handle read exclusive request to remote cluster.
Instead of changing the cache block in the remote cluster to be not locally cached, the cache block remains locally exclusive.
Invariant violated: ***Only a single master copy exists***

4.3. Experimental environment

There are many ways to implement this state pruning method. It can be multithread on either a single processor or on a multiprocessor, for instance; or it can be implemented as a coarse-grained parallel algorithm on a multiprocessor system using message passing, which is how we implement it. We use one pSeries 690+ node with 8 processors in an IBM Power4 system. The Berkeley Active Message library [8] is used to communicate among the DFS search processors.

5. Performance evaluation

We experiment with four unique DFS searches in four separate processes where each DFS search uses one of the following heuristics: max Hamming distance, min Hamming distance, max cache_score, min cache_score. We also experiment with 8 DFS searches in 8 processes where each of two DFS searches use the same heuristic. We use one-step BFS, which generates 5 frontier states, for four DFS searches and two-step BFS, which generates 19 frontier states, for 8 DFS searches.

The communication among the multiple DFSs is non-deterministic since the communication timing and ordering is different each time the algorithm is executed. Consequently, the results shown in Table 1.b are the mean and standard deviation of 20 separate runs.

5.1 Multiple DFS vs. single DFS

As we can see from Table 1.a, different heuristics are better at finding different types of bugs. The min-max-predict method [4] was proposed as a compromise among the different heuristics. As was shown in [4], min-max-predict is the most efficient DFS-based heuristic for a single exhaustive search for these tests.

By exploiting multiple heuristics in multiple simultaneous DFSs, we can choose the shortest trace (witness string) from the traces produced by the different heuristics. Thus, the final trace is close to the one that would be generated by the most efficient heuristic for each bug.

When there is no communication among the multiple DFS searches, Table 1.b shows that the four DFS searches provide an improvement for most of the bugs even when compared with the min-max-predict approach. When we use eight DFS searches with no communication, some results improve compared with the results of four DFS searches and some do not. This inconsistent behavior occurs because of different start states for four and eight DFS searches. Some start states may be randomly closer to the bug, while others may be further.

Table 1. States explored using the witness string method for the Stanford DASH coherence protocol.

(a) Length of the witness strings using one DFS-based heuristic in a single DFS search

	DFS	Hamming Distance		Cache Score		Min-Max Predict
		MIN	MAX	MIN	MAX	
BUG1	4,719	9,093	4,958	932	8,553	2,411
BUG2	421	25	437	54	409	124
BUG3	610	3,825	755	1,027	413	593
BUG4	533	10,265	504	924	78	410

(b) Length of the witness strings using four different DFS-based heuristics in four and eight DFS searches. "Independent" means there is no communication among the multiple DFS searches, while "State Pruning" allows communication among the multiple DFS searches.

	4 searches		8 searches	
	Indepen-dent	State Pruning	Indepen-Dent	State Pruning
BUG1	1,426	1,131±378	1,221	960±244
BUG2	70	33±14	38	18±7
BUG3	720	449±136	791	429±125
BUG4	363	386±135	362	226±104

5.2. State pruning vs. multiple independent DFS

The results for the state pruning method are generated from 20 separate runs and are shown in the form of *mean* ± *standard deviation* in Table 1.b.

In the state pruning method using four DFS searches, most of the results are better than the case of multiple DFS with no communication. For bug 2 and bug 3, even the upper bounds of the results for state pruning are better than the results of multiple DFS searches with no communication.

In the case of eight DFS searches, we show further improvements with the state pruning method even

when compared to the experiment with eight DFS processes with no communication, which showed no improvement. For all of the bugs, the upper bounds of the results for state pruning are better than the results with no communication. This occurs because with more DFS searches, the communication among the searches increases, thereby pruning out more redundant states.

6. Related works

The previous work most closely related to this study can be categorized into two classes: search heuristics and parallel algorithms.

Yang and Dill [7] examined the efficiency of several BFS based heuristics. They concluded that Minimum Hamming distance is easy to implement but has inconsistent efficiency. Tracks and Guidepost are more robust, while requiring more manual labor. Our previous study [4] examined several DFS-based heuristics. The proposed min-max-predict heuristic quickly discovered the embedded bugs in several different coherence protocols. All of these search heuristics were studied using a single exhaustive search that targeted a single embedded bug at a time.

Some parallel algorithms are related to our approach for implementing the state pruning method. Stern and Dill [5] studied a parallel BFS that generates short paths to states, requiring a large amount of memory per node to maintain the queue without being able to get quickly into the deeper state space as can DFS or a random walk (RW). Sivaraj et al. [6] combined BFS with RW. Although they vastly reduced the message exchange rate and the memory requirements, this approach is not suitable for witness string generation because RW generates huge witness strings.

Unlike previous studies, we exploit multiple heuristics to choose the most efficient witness strings by pruning out the redundant states through communications among the multiple DFS searches.

7. Conclusions

The state pruning method presented in this study uses multiple concurrent DFS processes. This approach gives the flexibility to choose the most efficient witness strings from all of the witness strings generated by the multiple DFS searches. To prune out the redundant state searches, we use communication among the multiple searches. Our results show a significant improvement when using this multiple DFS approach compared to the previously proposed min-max-predict approach. As the number of simultaneous searches increases, the state pruning benefits tend to increase compared to using multiple DFS searches with no communication. In summary, instead of a "one size fits all" heuristic, the state pruning method exploits multiple heuristics in different regions of the global state space to choose the most efficient overall witness strings. By pruning out redundant states in a DFS, the state pruning method enhances the efficiency of the witness strings generated.

Acknowledgements

This work was supported in part IBM, Compaq's Alpha Development Group, the University of Minnesota DigitalTechnology Center, and the Minnesota Supercomputing Institute.

References

[1] U. Stern and D. L. Dill, " Automatic Verification of the SCI Cache Coherence Protocol", In Correct Hardware Design and Verification Methods: IFIP WG10.5 Advanced Research Working Conference Proceedings, 1995

[2] D. Abts and M. Roberts, "Verifying large-scale multiprocessors using an abstract verification environment", In Proceedings of the 36th Design Automation Conference (DAC 99), pages 163-68, June 1999.

[3] D. Abts, S. Scott, and D. J. Lilja, "So Many States, So Little Time: Verifying Memory Coherence in the Cray X1", International Parallel and Distributed Processing Symposium (IPDPS), April, 2003

[4] D. Abts, Y. Chen and David J Lilja, "Heuristics for Complexity-Effective Verification of a Cache Coherence Protocol Implementation", Laboratory for Advanced Research in Computing Technology and Compilers Technical Report No. ARCTiC 03-04, November 2003.

[5] U. Stern and D.L. Dill, "Parallelizing the Murphi Verifier", *Formal Methods in System Design*, vol. 18, no. 2, pp. 117-129, 2001, (Journal version of their CAV 1997 paper).

[6] H. Sivaraj and G. Gopalakrishnan, "Random Walk Based Heuristic Algorithms for Distributed Memory Model Checking", 2nd International Workshop on Parallel and Distributed Model Checking (PDMC'03), Boulder, Colorado, USA, July 2003

[7] C. H. Yang and D. L. Dill, "Validation with Guided Search of the State Space", In Proceedings of the 35th Annual Design Automation Conference (DAC98), June 1998.

[8] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation", In 19th Annual International Symposium on Computer Architectures, 1992.

[9] D. L. Dill, "The Murphi verification system", in *Computer Aided Verification*, Lecture Notes in Computer Science, pp. 390-393, 1996. Springer-Verlag.

[10] James Laudon and Daniel Lenoski, “The SGI origin: A ccNUMA highly scalable server”, In Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), volume 25,2 of Computer Architecture News, page 241-251, New York, 2-4 1997. ACM Press.

[11] IEEE Std 1596-1992, IEEE Standard for Scalable Coherent Interface (SCI).