

State Pruning for Generating Efficient Test Vectors

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

Dennis Abts*

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

Abstract

The previously proposed witness string method [2] generates traces for system verification using a model checker and then uses the traces to drive the RTL logic design simulation. This paper extends the depth first search (DFS) used in the original witness string method with a state pruning method that exploits multiple search heuristics in simultaneous searches where each DFS uses a different heuristic. The hash table of the entire state space is distributed among the simultaneous searches so that they cooperate to avoid redundant state exploration. To evaluate this new search algorithm, we implanted several protocol bugs in the Stanford DASH cache coherence protocol model. We also evaluated the Counter benchmark that models the state space of a general application. Using an IBM Power4 multiprocessor system with the Berkeley Active Message library, we show that this new method of state pruning produces quantitatively better witness strings compared to both pure and guided DFS. The state pruning method proposed in this study is motivated from the problems in digital system design and then extended to general verification.

Keywords: verification, witness string method, state pruning method, Stanford DASH, Counter benchmark.

1 Introduction

State space traversal is a well-understood technique for verifying the abstract specification of a complex system with cooperating finite-state machines. Unfortunately, traversal of a large state space is mired in computational complexity admitting solutions that are exponential in the number of states. This “state explosion problem” has motivated techniques to make model checking more practical in an industrial setting. One approach is to apply heuristics to the search process. However, as Abts and Chen [4] discovered, there is not a single heuristic that is efficient across a broad range of error types.

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 they consist of multiple cooperating finite state machines (FSMs) which yield a very large state space. Abstraction and hierarchical verification techniques are used alleviate this problem which often entails construction of a “scaled down” abstract system model. 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] uses the notion of simulation containment to show that the RTL design is a refinement of the abstract cache coherence protocol specification. Traces, or *witness strings*, are captured from a depth-first-search (DFS) of a state space of the abstracted formal model (smaller state space) and provided as stimulus to the RTL design (with a much larger state space) of the system being verified. The efficiency of the resulting witness strings captured through this DFS process 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 generate efficient witness strings is through a bug-oriented DFS search of the state space. To enhance the bug-finding capability of a model checker, several heuristic algorithms have been proposed [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.

To generate efficient witness strings, we propose a method of pruning out redundant states using multiple simultaneous DFS searches each of which exploits a different heuristic. With this state pruning method, each DFS search targets a different type of bug in the state space. The state pruning method proposed in this study is applicable to general verification including both hardware and software.

To implement the multiple DFS searches in the state pruning method, we use a parallel algorithm executed on an IBM Power4 multiprocessor system. 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 first search (BFS) to generate the frontier states, similar to what was done in [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. We tested the state pruning method on the Stanford DASH cache coherence protocol model [7], where we implanted several protocol bugs, and a benchmark from the Counter benchmark suite [8], which is a benchmark for the model checking of general applications. The chosen Counter benchmark models an abstract state space, which can be applied to any application that can be abstracted into a finite state space.

In the remainder of this paper, Section 2 discusses the problems in test vector generation. In Section 3, we explain the new 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. The time required for the verification depends heavily on the efficiency of the test vectors. However, manually generated test vectors cannot guarantee equivalence between the logic design and the specification.

The witness string method [2, 4] has been proposed to provide a solid link between the specification and the implementation. This approach efficiently exposes design errors within both a formal model and its RTL implementation. The witness strings generated in the formal model are used as a verification certificate between the formal verifier and the RTL logic implementation.

The witness string 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, a trace recorded from the start state S_0 to a leaf state S_n is a witness string. 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.

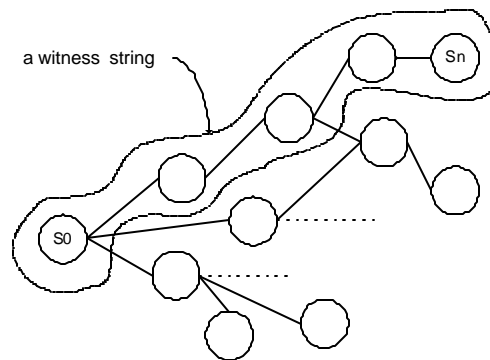


Figure 1. A witness string generated by a DFS of the coherence protocol state space.

In real systems, however, the witness string can be on the order of a few thousand states, and there are thousands of witness strings generated. As a result, it takes a long time to execute the witness strings in the RTL logic simulation [3]. In order to improve the efficiency of the witness strings, our previous work [4] explored several DFS-based heuristics to guide the depth-first search so that the bug-prone regions are searched first. In this manner, “better” witness strings are generated first. These previous results showed, however, that different heuristics produce better witness strings for different kinds of bugs. Unfortunately, a “one size fits all” heuristic is impractical and sub-optimal 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 simultaneously in several separate DFS searches. Each simultaneous search is targeted towards different portions of the state space with different types of bugs. 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]. We study this new approach for generating witness strings in the context of verifying the implementation of a multiprocessor cache coherence protocol and a Counter benchmark for the model checking of general applications

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 with the goal of improving the efficiency by reducing the number of

searched states before discovering the error. The DFS heuristics included max Hamming distance, min Hamming distance, max *cache_score*, min *cache_score*. The heuristic is used to choose the next state to explore among the current frontier of the search space.

Hamming distance is defined to be the number of bits whose values differ between the two states. By using the max or min Hamming distance heuristic, the search goes toward state spaces of the most or least different states, respectively. We use Hamming distance for both the DASH cache coherence protocol and a Counter benchmark.

The *cache_score* heuristic assigns a scoring function to the cache states using the basic observation that a memory reference will transition from a quiescent state, to a pending state, and finally return to a quiescent state after the request is satisfied. Having multiple outstanding requests for the same cache line from different processors creates “interesting” reference streams capable of exposing race conditions in the coherence protocol. The score gives a metric of the number of outstanding coherent memory references. The max or min *cache_score* heuristic guides the search into state spaces with the most or least concurrent coherence traffic, respectively. We use *cache_score* for the DASH cache coherence protocol exclusively.

3.2 Basic algorithm

We use multiple DFS searches concurrently. Each DFS search uses a different one of the four heuristics described above to guide its search. The hash table for the whole state space is distributed among the DFS searches using a hash function. Each DFS search is the owner of the states distributed in its hash table.

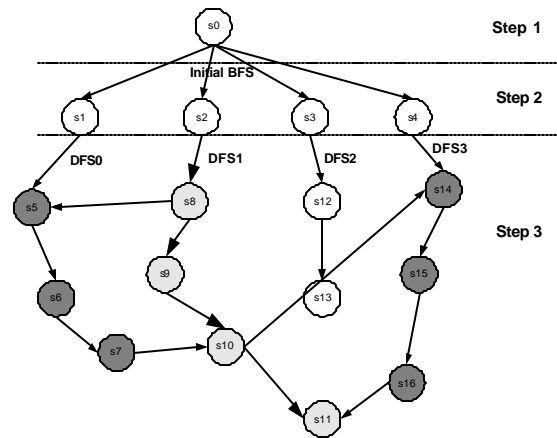
The basic algorithm for state pruning is shown in Figure 2. First, a breadth-first search (BFS) is used to generate the frontier states. BFS executes enough steps to ensure that there are just enough BFS frontier states to provide one state for every DFS search. Then these BFS frontier states are distributed onto all of the DFS searches by sending each state to its owner using a hash function. On receiving a frontier state each DFS search begins.

Each DFS search traverses from a current state to the next state using one of the heuristics described in Section 3.1 [4]. Once the next state is chosen, it is sent to its owner to check whether it has already been explored, that is, if it already exists in the owner’s hash table. If it has not been explored before, it is confirmed to be a new state and is added into its owner’s hash table. Whenever a DFS search reaches a bug in the system being tested, or finishes its search, a report is generated.

The state pruning effect is shown in step 3 of Figure 2. As DFS1 traverses to s8, it finds out s5 has already been explored by DFS0. Thus DFS1 goes from s8 to s9 instead

of s5. It turns out the path $s8 \rightarrow s9 \rightarrow s10$ is a shorter path than from $s8 \rightarrow s5 \rightarrow s6 \rightarrow s7 \rightarrow s10$. In this manner, the number states in a witness string is pruned. The similar situation happens when DFS1 goes from $s10 \rightarrow s11$ instead of from $s10 \rightarrow s14 \rightarrow s15 \rightarrow s16 \rightarrow s11$ since DFS1 finds out s14 has been explored by DFS3 already when DFS1 is in s10.

As shown in our previous study [4], the four different heuristics used in the witness string method are appropriate for finding different types of bugs. However, 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.



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 FS 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.

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.

Figure 2. The basic state pruning algorithm using multiple DFS searches (e.g., DFS search 0,1,2,3) for generating witness strings.

4 Experimental setup

We implement the state pruning method into the Murphi [11] model checker and use one pSeries 690+ node in an IBM Power4 system with the Berkeley Active Message library [10] to run the program. To test the efficiency of the witness strings generated with the state pruning method, we randomly implant bugs into the formal model of the Stanford DASH cache coherence protocol.

4.1 Model checker

Our model checker is built on our previous work [4] using Murphi [11], the parallel Murphi model-checker [5] written for the Berkeley Active Messages library, and the parallel RW algorithm [6]. Murphi uses explicit state enumeration to exhaustively search the reachable state space of a system of interacting finite state machines. We add new functions to implement the heuristics and the state pruning algorithm. We also added new message handlers for the communications among the DFS processes.

4.2 Benchmark

We used the Stanford DASH multiprocessor model [ref?] and a Counter benchmark [ref?] for this study.

The Stanford DASH multiprocessor implemented a directory-based coherence protocol that uses NAKing to defer incoming requests to a pending cache block. The DASH protocol is the predecessor to the SGI Origin2000 [12] cache coherence protocol. The formal model of DASH has one home node and two remote nodes with a total state space size of 10,466 states.

DASH has three types of requesters: the *local* or *remote* cluster, and *home* cluster. It also has two classes of memory references to local or remote memory. We implanted bugs in the system by inspecting state diagrams of the protocol given in [11] and randomly modifying some transitions to create an error. The implanted specific bugs, which are not obvious to discover, 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*

The Counter benchmark suite is a set of benchmarks to test and compare different model checking tools. It simulates different shapes of state spaces (i.e., bi-directional, unidirectional, branch, two diamonds) and, for each shape, different locations and distributions of errors (i.e., deep-dense, deep-sparse, shallow-dense, shallow-sparse). We choose the bi-directional deep-sparse benchmark because it models the state space shape of a general application. Each state of bi-directional has some forward edges and backward edges. Sparse errors are modeled by making every value of array to be equal to a certain value. Since it is a rare condition, the errors are sparse. Deep errors are obtained by setting the error deep into the state space. The total state space size of the bi-directional deep-sparse benchmark is 132,651 states.

4.3 Experimental environment

There are many ways to implement this state pruning method. It can be multithreaded on either a single processor or on a multiprocessor, or it can be implemented as a coarse-grained parallel algorithm on a multiprocessor system using message-passing.

We use one pSeries 690+ node with 8 processors on an IBM Power4 system. The Berkeley Active Message library [10] is used to communicate among the DFS search processors. After sending a message, the sender continues execution without waiting for the message to arrive at the receiver. The receiver needs to periodically call the *poll()* command to execute the handlers for the received messages. To synchronize all the processes, the

`barrier()` command is used to keep processes waiting until every process reaches the barrier.

Since the sender does not wait for the receiver, a sender process continues execution without the information about whether the next state already exists in its owner process's hash table. Therefore, we use a `poll()` command to check the hash table in the owner process. In this manner, we guarantee the correct communication among the processes to avoid extra work.

To demonstrate the effectiveness of the state pruning communication among the multiple DFS searches, we compare against a scheme where there is no communication among the DFS searches. In this case, each DFS exhaustively searches the whole state space.

5 Performance evaluation

Our experimental results for the DASH system and the bi-directional deep-sparse benchmark are shown in Table 1 and Figure 3.

For the DASH system 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.

For the bi-directional deep-sparse Counter benchmark we experiment with four unique DFS searches in four separate processes where each of two DFS searches use one of either the max Hamming distance or min Hamming distance heuristics. We also experiment with 8 DFS searches in 8 processes where each of four DFS searches use the same heuristic. We use a two-step BFS, which generates 6 frontier states, for four DFS searches and a three-step BFS, which generates 10 frontier states, for 8 DFS searches.

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

We compare the results from the new state pruning algorithm with the results using a single DFS-based heuristics, shown in Table 1.a. We also use a scheme where there is no communication among the multiple DFS searches as shown in Table 1.b and Figure 3. The results show the efficiencies of the generated witness strings in terms of the lengths of the witness strings. The length of a witness string is the number of states it contains. The shorter the length of a witness string the more efficient it is.

5.1 Multiple DFS searches vs. single DFS search

For DASH system, 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. *Min-max-predict* combines the min/max Hamming distances with the `cache_score` heuristic and a 3-bit saturating counter to decide which region of the global state space to search next. As was shown in [4], *min-max-predict* is the most efficient DFS-based heuristic for a single exhaustive search.

For the bi-directional deep-sparse benchmark, both min and max Hamming distance are more efficient than the DFS.

By exploiting multiple heuristics in multiple simultaneous DFS searches, 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. As a result, this method produces good traces for all of the various types of bugs.

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 DASH bugs even when compared with the *min-max-predict* approach, which combines the four heuristics. 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 some of the start states for the DFS searches are different from the ones used by the four DFS searches. Some start states may be randomly closer to the bug, while others may be further.

In the case for the bi-directional deep-sparse Counter benchmark, when there is no communication among the multiple DFS searches the results are more efficient than the original DFS but are less efficient than the min/man Hamming distance. This is because the start states of each DFS search in the multiple DFS searches are generated by the initial BFS search and thus different from the start state of the original DFS search. When there is communication among the multiple DFS searches, the results are more efficient than the min/man Hamming results and the multiple independent DFS searches results. And the eight DFS searches state pruning approach generates shorter witness strings than the four DFS searches approach.

5.2 State pruning vs. multiple independent DFS searches

The results for the state pruning method are generated from 20 separate runs and are shown in the form of (mean \pm 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 DASH 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, which shows the efficiency of the state pruning method for witness string generation.

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 thus pruning out more redundant states.

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

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

	DFS	Hamming Distance		Cache Score		Min-Max Predict
		MIN	MAX	MIN	MAX	
DASH BUG1	4,719	9,093	4,958	932	8,553	2,411
DASH BUG2	421	25	437	54	409	124
DASH BUG3	610	3,825	755	1,027	413	593
DASH BUG4	533	10,265	504	924	78	410
Bi-directional Deep-Sparse	79,221	1,611	2,412	N/A	N/A	N/A

(b) The lengths of the witness strings generated using four different DFS-based heuristics (min Hamming, max Hamming, min *cache_score*, max *cache_score*) in four and eight DFS searches. Each DFS search uses a different heuristic. “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	
	Independent	State Pruning	Independent	State Pruning
DASH BUG1	1,426	1,131 \pm 378	1,221	960 \pm 244
DASH BUG2	70	33 \pm 14	38	18 \pm 7
DASH BUG3	720	449 \pm 136	791	429 \pm 125

DASH BUG4	363	386 \pm 135	362	226 \pm 104
Bi-directional Deep-Sparse	57,702	1,336 \pm 190	57,774	1,288 \pm 118

5.3 Effect of the number of searches in state pruning

In order to further explain the effect of the number of searches in the state pruning method, we generate the probability diagrams (Figure 3) because each result for state pruning experiments is generated from 20 runs. These diagrams show the probabilities of finding a bug (i.e., bug1, 2, 3, 4) using different numbers of searches (i.e., 4, 8) in the state pruning method. The probabilities are calculated using the following form:

$$p(x) = \frac{N_x}{N}$$

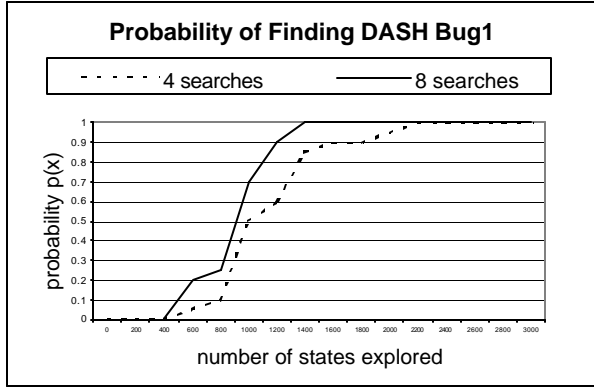
N_x : total number of runs that find the bug before reaching the x th states

N : the number of total runs, which is 20 in this study

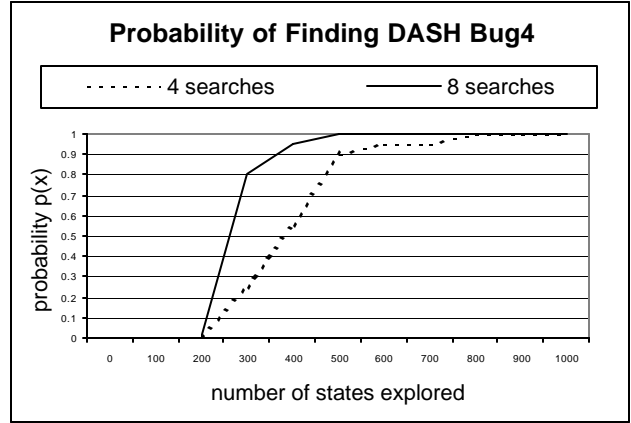
The number of states where the probability is 0.5 is the median case for finding a bug. The number of states where the probability is 1 is the worst case for finding a bug.

Each result for state pruning experiments is generated from 20 runs. In order to further explain the effect of the number of searches in the state pruning method, we generate the probability diagrams shown in Figure 3. These diagrams show the probabilities of finding a bug (i.e., DASH bug1, 2, 3, 4, and the bug in the bi-directional deep-sparse Counter benchmark) using different numbers of searches (i.e., 4, 8) in the state pruning method. The number of states where the probability is 0.5 is the median case for finding a bug. The number of states where the probability is 1 is the worst case for finding a bug.

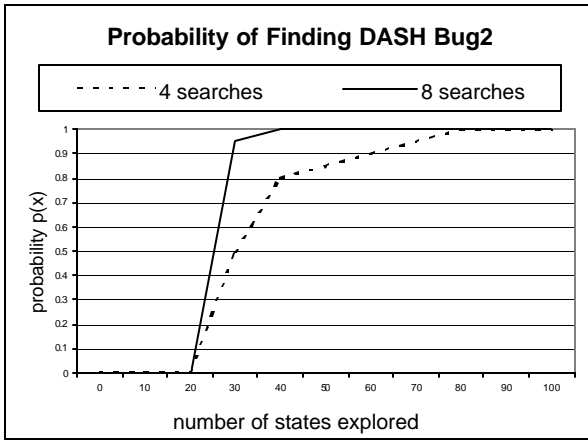
As shown in Figure 3, the curve for the 8 searches case is always above the curve for the 4 searches case, except for DASH bug3 and the bi-directional deep-sparse benchmark. This relationship means that state pruning with 8 searches is more efficient than state pruning with only 4 searches. Furthermore, in the worst case, state pruning with 8 searches finds a DASH bug faster than with only 4 searches for all of the DASH bugs except bug3. The improvement can be up to 100% (DASH bug2). Although the two curves in DASH bug3 are very close, the curve for 8 searches is above that of 4 searches most of the time. It is the same for the bi-directional deep-sparse benchmark. Therefore, in the median cases, state pruning with 8 searches always finds a DASH bug faster than with 4 searches. The improvement is up to 30% (DASH bug4).



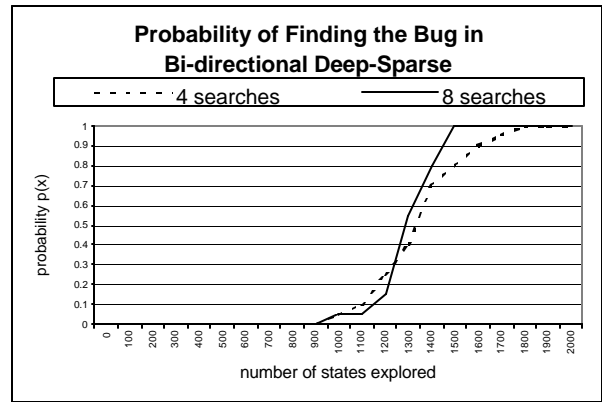
(a)



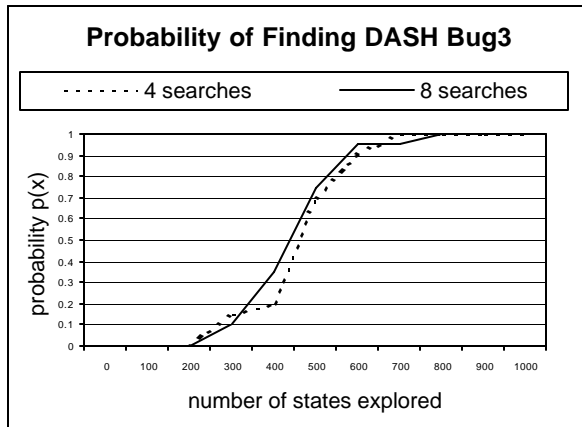
(d)



(b)



(e)



(c)

Figure 3. The probabilities of finding a bug in the Stanford DASH cache coherence protocol and the bi-directional deep-sparse Counter benchmark using different numbers of searches (i.e., 4, 8) in the state pruning method.

6 Related work

Two classes of previous work are most closely related to this study: search heuristics and parallel algorithms that are related to the algorithm we use to implement the state pruning method.

Since usually a large portion of the state space satisfies the specification, model checking wastes much effort to verify this correct portion. Thus, search heuristics are invented to guide the search for the error-prone states. Yang and Dill [9] examined the efficiency of several BFS based heuristics: minimum Hamming distance, Tracks, and Guidepost. 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 DFS-based heuristics: max Hamming distance, min Hamming distance, max *cache_score*, min *cache_score*, and *min-max-predict*. Our results confirmed that Hamming distance is inconsistent with DFS. This work also showed that different kinds of bugs are more easily found using different heuristics. By exploiting the benefits of minimum and maximum Hamming distance, the *min-max-predict* heuristic quickly discovered the embedded coherence errors in several different coherence protocols with a range of embedded bugs. *Min-max-predict* produced efficient witness strings for both the Cray X1 and the Stanford DASH protocols. All of these search heuristics were studied using a single exhaustive search that targeted a single embedded bug at a time.

In order to implement the state pruning method, we develop a parallel algorithm for the multiple DFS searches with communication. The following parallel algorithms are related to ours. They all use multiple concurrent searches among processors for model checking. Stern and Dill [5] studied the performance of a parallel BFS that distributed the states in the state space to an owner thread. Although parallel BFS generates short paths to states, and speeds-up the complete search, it still requires a large amount of memory per node to maintain the queue. Furthermore, BFS cannot quickly move into the deeper state space as can DFS or a random walk (RW), which is important for debugging purpose. However, DFS or RW alone may get trapped in state areas. 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 does not use a hash table. As a result, it generates huge witness strings.

Unlike previous studies, we exploit multiple heuristics to choose the most efficient witness strings. Implemented in a parallel algorithm, our method prunes out the redundant states through communications among the multiple DFS searches to further enhance the efficiency of the witness strings generated for a cache coherence protocol. Tested on a model of an abstract state space, the state pruning method is furthermore shown to be efficient for the test vector generation for the verification of general applications including both hardware and software.

7 Conclusions

The state pruning method presented in this study uses multiple concurrent depth-first search (DFS) processes with a different search heuristic in each process to expose bugs in different areas of the target state space. This approach gives the flexibility to choose the most efficient witness strings from all of the witness string 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 for cache coherence protocols, which combines multiple heuristics in a single exhaustive DFS. Furthermore, this state pruning method is efficient for general verification. 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 depth-first search, the state pruning method enhances the efficiency of the witness strings generated for the witness strings approach in the digital design approach and is furthermore efficient for automatic generation of test vectors using model checking for general applications.

Acknowledgements

This work was supported in part IBM, Compaq's Alpha Development Group, the University of Minnesota Digital Technology 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] Lenoski, D., J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. S. Lam., "The Stanford DASH Multiprocessor", *IEEE Computer* (Mar 1992), 63-79.
- [8] <http://vv.cs.byu.edu/~tonga/>
- [9] 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.
- [10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer, "Active messages: a mechanism for integrated communication and computation", In *19th Annual International Symposium on Computer Architectures*, 1992.
- [11] D. L. Dill, "The Murphi verification system", in *Computer Aided Verification*, Lecture Notes in Computer Science, pp. 390-393, 1996. Springer-Verlag.
- [12] 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, 24 1997. ACM Press.