PReach: A Distributed Explicit State Model Checker*

University of British Columbia, Computer Science Technical Report TR-2010-05

Flavio M. de Paula
University of British Columbia, Canada

Brad Bingham
University of British Columbia, Canada

Jesse Bingham
Intel Corporation, U.S.A.

John Erickson
Intel Corporation, U.S.A.

Mark Reitblatt
Intel Corporation, U.S.A.

Gaurav Singh
Intel Corporation, U.S.A.

April 6, 2010

Abstract

We present PReach, a distributed explicit state model checker based on Murφ. PReach is implemented in the concurrent functional language Erlang. This allowed a clean and simple implementation, with the core algorithms under 1000 lines of code. Additionally, the PReach implementation is targeted to deal with very large models. PReach is able to check an industrial cache coherence protocol with approximately 30 billion states. To our knowledge, this is the largest number published to date for a distributed explicit state model checker.

1 Introduction

Explicit-state model checking (EMC) is an important technique for verifying properties of hardware designs. Using a formal description of the system, EMC explores the reachable states looking for specification violations. For nondeterministic, high level models of hardware protocols, it has previously been argued that EMC is better than symbolic model checking [6]. Still, the *size* (number of reachable states) of models that can be handled by EMC is bounded by the amount of memory available to the EMC program; this has certainly been our experience with industrial-sized examples. One obvious approach to expanding the memory resource is to use the much larger disk to store reachable states; this is done by several EMC tools, e.g. TLC [11]. An orthogonal approach is to harness the memory resources of multiple computers in a distributed computing environment. In the past decade, several *distributed* EMC (DEMC) tools have arose, e.g. Eddy [9], Divine [2], and PSpin [8]. Most experiments in the DEMC literature pertain to the speed up of DEMC over sequential EMC.

We take the view that another important return delivered by DEMC is an increase in the model size.[1] With this observation in mind, we have developed a DEMC tool called PReach. A secondary goal of PReach is to provide a simple code base; this allows PReach to function as a platform for DEMC researchers to quickly implement and evaluate new ideas.

Top-level algorithms of PReach are implemented using *Erlang*, a concurrent and functional language [1], whereas low-level operations are handled

*This technical report describes preliminary work and is expected to be superceded by a conference paper soon. That is, if the year is not 2010, this report is probably obsolete. The PReach tool is available for download at `http://bitbucket.org/jderick/preach`

[1] We note the original PSpin paper [8] also took this view, however their focus is LTL and the largest automaton they handle is 2.8 million states.

by pre-existing C code of the Murφ model checker [5]. PREACH's input language is the well-known Murφ modelling language. We have used PREACH to model check a system with 30 billion states, using about 100 machines. As far as we know, this is the largest reachable state space ever explored using EMC.

Related Work. Some examples of DEMC tools are Eddy Murφ, DiViNe and SPIN. Eddy Murφ [9] improves the original Parallel Murφ [10] (in terms of speed) by using separate threads for next-state generation and communication. DiVinE (e.g. [2]) is an DEMC tool that has sophisticated algorithms for LTL model checking. Most papers on DiVinE do not consider large models, however it has been reported to handle an automaton with 419×10^6 states [3]. SPIN has also been used for performing distributed model checking with the capability of handling up to 2.8×10^6 states [8].

2 Implementation

PREACH is based on the Stern-Dill DEMC algorithm [10], a distributed depth-first search that partitions the space across the compute nodes using a uniform random hash function that associates an *owner* node with each state. The computation begins by sending the initial states to their respective owners. Each node maintains a set of states `ss` containing the states it owns that have been visited. Upon receipt of a state s, a node checks to see if $s \in$ `ss`. If not, s is added to `ss` and also appended to a *work queue* `wq`. Once there are no more pending states to receive, the head of `wq` is popped and its successors computed, which are then sent to their respective owners. PREACH's termination detection also follows [10].

Murφ Engine Interface. To avoid wheel invention and to harness fast and reliable code, PREACH uses existing Murφ code for several key functions involved in EMC: state hash table look-ups and insertions, state expansion, symmetry reduction, invariant and assertion violation detection, and state pretty printing. To facilitate Erlang calling of Murφ functions, we had to write some light-weight wrapping of the C code, we call the resulting code the *Murφ Engine*. We also employ the Murφ front end that compiles the Murφ model into C++.

Backoff Mechanism. A fundamental problem we faced when running PREACH on large models is that on some compute node, the number of messages (states) piling up in the Erlang runtime would explode, causing the PREACH process (and hence the whole model check) to crash. This phenomenon can be caused by myriad factors such as heterogeneous compute nodes, sporadic network conditions, and also dynamic loading effects observed previously [7, 4]. Our solution is based on the observation that it is better to slow down the DEMC algorithm and allow overloaded nodes to catch up than it is to crash, and involves a conceptually simple yet effective *backoff* mechanism. The mechanism sends a *backoff* message to all other nodes when the message queue exceeds a fixed size. When other nodes receive *backoff* messages, they stop sending any states to the originating node. This is acheived by recycling states on `wq` if any successor is owned by a node from which *backoff* has been received. The overloaded node then gets a chance to "catch-up" and sends an *unbackoff* message when the runtime queue falls below a lower limit; the other nodes then resume sending to the node.

Load Balancing. Despite even assignment of states to nodes, dynamic state queue lengths across nodes can be extremely uneven. This has been observed with other DEMC tools [7, 4]. We have implemented a load balancing scheme inspired by that of Kumar and Mercer [7]. Periodically, each thread will broadcast to all other threads to report its current state queue length q_s. When these messages are received, the receiver's state queue length q_r is compared with q_s. If q_r is sufficiently greater than q_s, then a number of states equal to some fraction of $q_r - q_s$ is sent from the receiver's state queue to the original sender. Unlike regular states that are sent among threads as a result of state expansion, load balancing states are *not* owned by the thread that receives them. Thus, such states are not queried in `ss` when received. Rather they are *always* enqueued into `wq`. We note that backoff and load balancing address somewhat different problems; back-off slows down computation to avoid congestion-related crashes, while load balancing is a performance opti-

mization.

Disk Files. We can optionally store the set of visited states on disk. Our original implementation stored both the set of visited states and the work queue in memory. We quickly found that storing the work queue on disk saved memory without compromising performance. Storing the states on disk, however is more difficult due to the random access pattern. We used a technique similar to Stern and Dill's [10] that processes states in batches to avoid random accesses. The basic idea is rather than looking up new states in a hash table, we accumulate them in a *filter queue*. Once the filter queue reaches a predefined size, we search the *hash file* for any matches, discarding states that have been seen before and adding any new states to the file. Only after states have been filtered in this manner are they added to the work queue.

One technique we found useful that was not mentioned in Stern and Dill's paper was to keep the hash file sorted. This allows a single pass to be done on the filter queue, cutting down on processing time. Another optimization was to keep a separate *unhash file* that maps hashes to full states for the states currently in the filter queue. Since the filter queue was stored in memory, this increases the maximum capacity of the filter queue, allowing scans of the hash file to be amortized over more states. Another idea that we believe may work, but have not yet implemented, is to store the filter queue itself on disk.

3 Results

Table 1 presents a few of the largest models we have verified with PReach. All of the features discussed in Sect. 2 combined to allow us to achieve these results. The largest model we have checked thus far is 28.2B states, about ten times larger than the largest model we had known to have been checked previously with any other DEMC or EMC tool within Intel.

Recently, we collaborated with Ganesh Gopalakrishnan's group to run Eddy [9] on some of our internal models. After some improvements to Eddy, we were able to check a 10.1×10^9 state model. It is important to note that currently Eddy runs 4 times faster than PReach for some models. However Eddy adds many more lines of code to the Murφ code base than PReach (4700 vs 1500, respectively). Hence we believe PReach has simpler source code, thanks to Erlang's expressiveness.

Model	States ($\times 10^9$)	Nodes	Time (hours)	States per Sec per Node
Peterson8	15.3	100	29.6	1493
Intel3 (5 txns)	10.1	61	24.7	1860
Intel3 (7 txns)	28.2	92	90.2	945

Table 1: Large model runs. Here Peterson8 is Peterson's mutual exclusion algorithm over 8 clients, and Intel3 is an Intel proprietary cache protocol. The last two rows are for Intel3 with respectively 5 and 7 transaction types enabled.

We also attempted to compare against DiVinE. For a simple model that nondeterministically increments 4 counters (having 500 Million states), DiVinE crashed after allocating 3 GB on each of 16 nodes whereas PReach verified the model on a single machine using only 3 GB. However, DiVinE handles a much richer specification language (LTL), making direct comparison difficult.

References

[1] J. Armstrong. The development of erlang. In *ACM SIGPLAN international conference on Functional programming*, pages 196–203, 1997.

[2] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. DiVinE – a tool for distributed verification. In *Computer Aided Verification*, pages 278–281, 2006.

[3] J. Barnat, L. Brim, P. Simecek, and M. Weber. Revisiting Resistance Speeds Up I/O-Efficient LTL Model Checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 48–62. Springer, 2008.

[4] G. Behrmann. A performance study of distributed timed automata reachability analysis. *Electr. Notes Theor. Comput. Sci.*, 68(4), 2002.

[5] D. L. Dill. The murphi verification system. In *International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996.

[6] A. Hu. *Techniques for Efficient Formal Verification Using Binary Decision Diagrams.* PhD thesis, Stanford University, 1995.

[7] R. Kumar and E. G. Mercer. Load balancing parallel explicit state model checking. In *Parallel and Distributed Model Checking*, 2004.

[8] F. Lerda and R. Sisto. Distributed-memory model checking with spin. In *Proc. of SPIN 1999, volume 1680 of LNCS.*, pages 22–39. Springer-Verlag, 1999.

[9] I. Melatti, R. Palmer, G. Sawaya, Y. Yang, R. M. Kirby, and G. Gopalakrishnan. Parallel and distributed model checking in eddy. *Int. J. Softw. Tools Technol. Transf.*, 11(1):13–25, 2009.

[10] U. Stern and D. L. Dill. Parallelizing the murphi verifier. In *International Conference on Computer Aided Verification*, pages 256–278, 1997.

[11] Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 54–66, 1999.