Industrial Strength Distributed Explicit State Model Checking

Brad Bingham*, Jesse Bingham†, Flavio M. de Paula*, John Erickson†, Gaurav Singh†, Mark Reitblatt‡
*Department of Computer Science
University of British Columbia
Vancouver, Canada
{binghamb, depaulfm}@cs.ubc.ca
†Intel, U.S.A.
{jesse.d.bingham, john.erickson, gaurav.2.singh}@intel.com
‡ Department of Computer Science
Cornell University
mark@reitblatt.com

***Abstract*—We present PREACH, an industrial strength distributed explicit state model checker based on Murphi. The goal of this project was to develop a reliable, easy to maintain, scalable model checker that was compatible with the Murphi specification language. PREACH is implemented in the concurrent functional language Erlang, chosen for its parallel programming elegance. We use the original Murphi front-end to parse the model description, a layer written in Erlang to handle the communication aspects of the algorithm, and also use Murphi as a back-end for state expansion and to store the hash table. This allowed a clean and simple implementation, with the core parallel algorithms written in under 1000 lines of code. This paper describes the PREACH implementation including the various features that are necessary for the large models we target. We have used PREACH to model check an industrial cache coherence protocol with approximately 30 billion states. To our knowledge, this is the largest number published for a distributed explicit state model checker. PREACH has been released to the public under an open source BSD license.**

***Keywords*-Explicit State Model Checking, Distributed Model Checking, Murphi**

I. 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. Unfortunately, the *state-space explosion problem*, which says that the number of states grows exponentially with the state variables, renders EMC of large systems intractable. For nondeterministic, high-level models of hardware protocols, it has previously been argued that EMC is better than symbolic model checking [1]–[3]; this is because the communication mechanisms inherent in protocols tend to cause the BDDs in symbolic model checking to blow up. Also, EMC is amenable to techniques like *symmetry reduction* [4] and *hash compaction* [5], which allow the verification to complete without explicitly storing all reachable states. Still, the *size* (number of reachable states) of models that can be handled by EMC is bounded by the amount of memory resources 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 disk to store reachable states; this is done by several EMC tools, e.g. TLC [6] and Disk-Murφ [3]. However, in spite of advanced schemes to hide disk latencies and allow for quick random access, disk-based approaches are inherently slower than checking a table in RAM. For instance, In Stern & Dill's experiments [3], the slow-down factor when using disk appears to grow linearly with the proportion of disk space used vs. RAM; for some models using $50\times$ more disk space than RAM caused a slow down of a factor of 30.

An orthogonal approach to increase the capacity of EMC 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 [7], Divine [8], and PSpin [9]. Most experiments in the DEMC literature pertain to the speed up of DEMC over sequential EMC. However, we contend that for industrial applications, a more important focus is the ability to scale to very large models. It was a lack of tools addressing this problem which led us to develop PREACH (Parallel REACHability). During our evaluation of Divine, it failed to check a toy model with half a billion states when given 16 machines on which to run, when the same model ran to completion on a single machine with standard Murphi. PSpin, although purportedly developed with scalability in mind, listed 2.8 million states as the largest model checked. Our initial experiments with Eddy were only slightly better. We had problems with the tool randomly crashing. Although eventually some improvements were made which improved the stability of Eddy, as of this date, Eddy reports inconsistent state counts for even small models and there is no active maintainer of the software. The complex 4500+ line C++ codebase was a deterrent for us to try to fix Eddy itself. We have no dedicated tools person, and need to be able to keep our tools running "in our spare time", while also performing our usual silicon validation duties on schedule. This ease of maintainability requirement drove us to keep PREACH's code base simple. A nice side benefit of this is that it allows PREACH to function as a platform for DEMC

researchers to quickly implement and evaluate new ideas.

PREACH is based on the DEMC algorithm of Stern and Dill [10], which we review in Sect. II. Architecturally, PREACH is split into two levels: a high level Erlang program and low level C++ code. The high level Erlang program handles the distributed aspects of the algorithm and the low level C++ uses the existing Murφ code base to handle routines such as parsing and state manipulation. Erlang was "designed for programming large-scale distributed soft real-time control applications" [11], and supports a simple and intuitive message-passing model of concurrency, making the high level code especially easy to understand and manipulate. This modular architecture of PREACH offers a clean division between communication aspects and model checking computations. Furthermore, Erlang's expressiveness allowed us to write the core algorithms in the Erlang layer using fewer than 1000 lines of code. This is particularly important for maintainability of the code. PREACH's front-end is also inherited from Murφ and hence its input language is the well-established Murφ modeling language. We have used PREACH to model check an industrial cache coherence protocol with approximately 30 billion states. As far as we know, this is the largest reachable state space ever explored using any explicit state model checker[1]. On a per thread basis, PREACH runs at about 50 percent of the speed of Murphi for the industrial protocols we analyze the most. We find this overhead is far outweighed by the benefits of being able to distribute the load across many machines. PREACH is publicly available under an open source BSD license [13].

The paper is organized as follows. Sect. II gives a high-level overview of PREACH. Sect. III outlines some of the key features of PREACH beyond the basic Stern-Dill algorithm; these features were driven by our analysis of large models. Sect. IV presents experimental results showing the effectiveness of PREACH and its features. Related work is presented in Sect. V and the paper concludes in Sect. VI.

II. Overview of PREACH

PREACH is based on the Stern-Dill DEMC algorithm [10], a distributed breadth-first search[2] that partitions the space across the compute workers (a.k.a. *nodes*) [3] using a uniform random hash function that associates an *owner* node with each state. Pseudocode for Stern-Dill is given in Algorithm 1; standard details such as termination detection, error trace generation, as well as all PREACH-specific features have been omitted for simplicity; also we have done some minor code reformatting compared to the original paper.

Each worker has two main data structures T and WQ, which, mathematically are a set of states and a list of states, respectively (declared on lines 1 and 2). Set T is maintained to contain the states owned by the worker that have been visited, while WQ are those states that have been visited but not yet expanded (i.e. had its successors generated). The computation begins with each worker executing the main routine SEARCH. After initializing T and WQ to be empty and synchronizing with a barrier, SEARCH sends the initial states to their respective owners in lines 11 to 15. Next each worker enters a while loop (line 16) that iterates until termination has been detected; termination detection is done as in [14] and we leave the details out of this paper. In the body of the while loop, we first invoke the procedure GETSTATES, shown on line 30, which iteratively pulls incoming states from the runtime. Each state s in GETSTATES that is not in T is inserted into T and appended to the tail of WQ. Once there are no more states to receive from the runtime, control is returned to line 18 where it is checked if WQ is nonempty. If so, we pop a state s from WQ and compute its successors. Each successor is first canonicalized for symmetry reduction [4] on line 21, and then the resulting state s'_c is sent to its owner.

We now discuss PREACH's implementation. To avoid wheel invention and to harness fast and reliable code, PREACH uses existing Murφ code for many key functions involved in EMC, such as

- front end parsing of murphi models
- state expansion and initial state generation
- symmetry reduction
- state hash table T look-ups and insertions
- invariant and assertion violation detection
- state pretty printing (for error traces)

To facilitate Erlang calling of Murφ functions, we had to write some light-weight wrapping of the existing Murφ C code, we call the resulting code the *Murφ Engine*. We also employ the Murφ front-end that compiles the Murφ model into C++.

Aside from space and time efficiency during model checking, using the Murφ Engine had another clear benefit: the Erlang code need only handle management of the distributed aspects of the algorithm, while the Murφ Engine handles key computations that pre-existing Murφ code can do quickly and correctly. The Erlang code, at a high level, resembles Algorithm 1, with key functions such as $Successors()$, $Canonicalize()$, and $Insert()$ calling into the Murφ Engine. One notable exception to this paradigm is the work queue WQ; this is handled in Erlang code and uses an optimized disk-file to allow for very large lists to be handled.

III. Features

Here we discuss the key features that contribute to PREACH's scalability and robustness. This section is concluded with pseudocode of the PREACH DEMC algorithm, which shows how these features are integrated into the basic Stern-Dill DEMC algorithm. We note that in our experience throughout the development of PREACH, network bandwidth was never a performance bottleneck (at Intel, we typically used a 1 Gbit/s network).

[1] An exception is SPIN, which has handled comparably sized models [12], but with a higher error rate than PREACH affords.

[2] Strictly speaking, the search order is not breadth-first because the timing of communication actions results in various visitation interleavings. However, the algorithm is "breadth-first" in that each worker operates on a FIFO of states.

[3] The workers can run on distinct hosts or some can seamlessly sit on the same host.

Algorithm 1 Stern Dill DEMC

```
1:  T : set of states
2:  WQ : list of states
3:
4:  procedure SEARCH( )
5:      T := ∅
6:      WQ := []
7:      barrier()
8:      if I am the root then
9:          for each startstate s do
10:             Send s to Owner(s)
11:         end for
12:     end if
13:     while ¬Terminated() do
14:         GETSTATES()
15:         if WQ ≠ [] then
16:             s := Pop(WQ)
17:             for all s' ∈ Successors(s) do
18:                 s'_c := Canonicalize(s')
19:                 Send s'_c to Owner(s'_c)
20:             end for
21:         end if
22:     end while
23: end procedure
24:
25: procedure GETSTATES( )
26:     while there is an incoming state s do
27:         Receive(s)
28:         if s ∉ T then
29:             Insert(s, T)
30:             Append(s, WQ)
31:         end if
32:     end while
33: end procedure
```

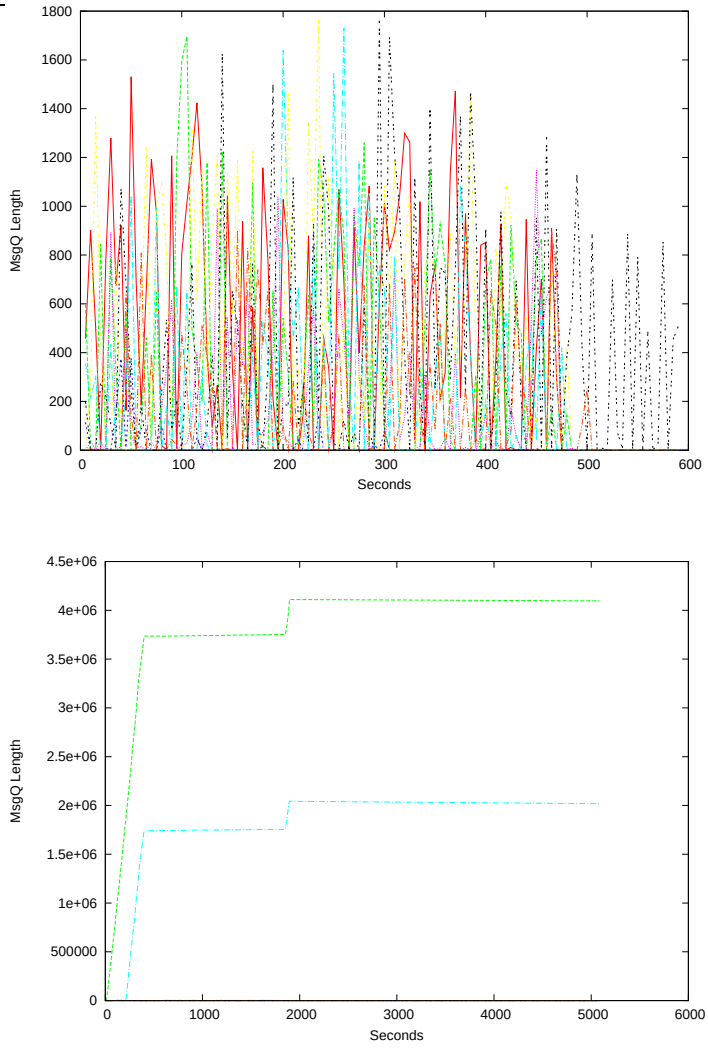


Fig. 1. LDASH protocol with 8 workers. Message queue lengths with crediting on (top) and off (bottom). Credit based on number of messages equal to 1000.

A. Crediting

An important problem we noticed when running big models with early versions of PREACH was that some workers would mysteriously grind down to almost a halt or completely crash. In a naive implementation of DEMC, each node sends all the successors of a state to the owners of the successors immediately upon expanding a state. This is how we originally implemented PREACH, and it worked sufficiently well on some small models. However, we noticed that for bigger models some nodes would inexplicably fail at random times. Upon closer inspection, these nodes would typically balloon to several gigabytes of memory usage before this happened. After analyzing Erlang mailbox[4] size statistics, we found that the problematic nodes were getting a disproportionate number of messages in the mailbox. What was happening was that a large number of messages were sent to a single node simultaneously by many other nodes. The original node would not be able to quickly receive these messages into WQ. As these messages piled up, the node would eventually have to allocate more memory and this would result in slower processing of messages. This effect cascaded until the node started paging and was unable to do anything at a reasonable speed at all.

To solve this problem, we first introduced a backoff mechanism where a node would send a *backoff* message to all other nodes whenever its message queue reached a certain bound. The other nodes would stop sending until an *unbackoff* message was received. The overloaded node would send unbackoff when it had processed all or most of the messages in its queue. This mechanism worked better, but not perfectly. The problem here was that Erlang's communication semantics dictate that these backoff messages travel in the same FIFO mailbox as states and other messages and hence it could be a non-negligible amount of time from when a node broadcasts

[4]The *mailbox* stores the incoming messages in the Erlang runtime that have yet to be received by the Erlang program.

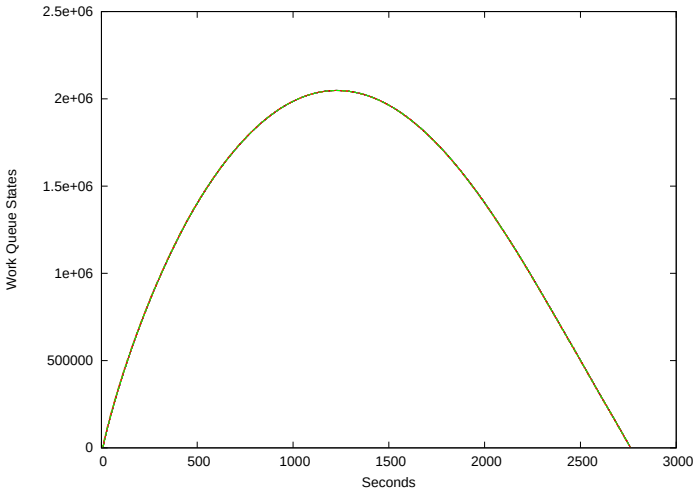


Fig. 2. Load balancing with Kumar and Mercer's algorithm.

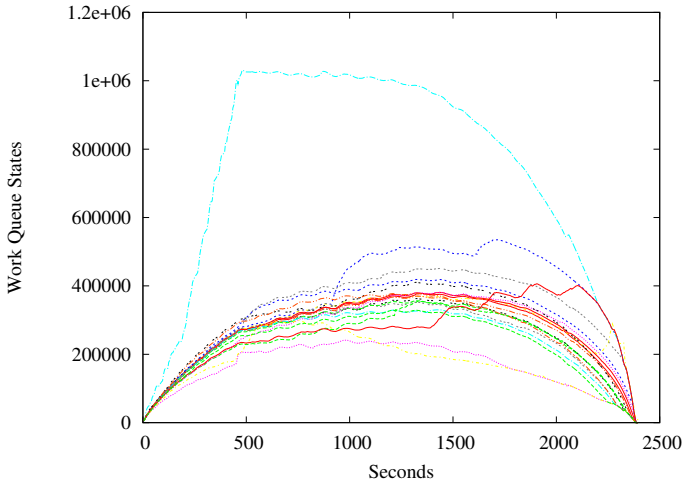
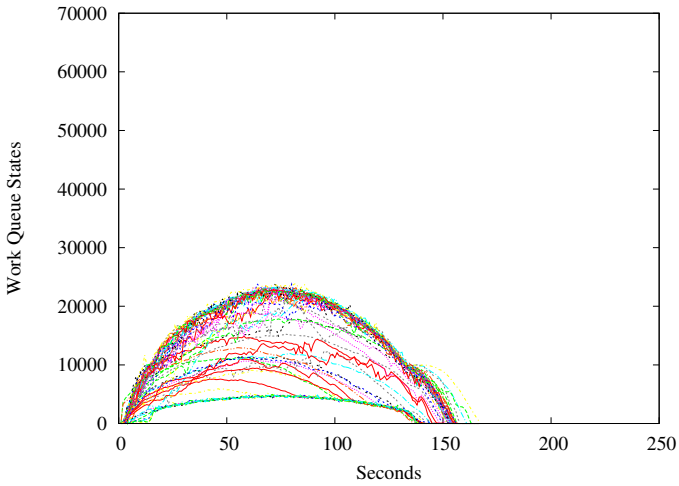


Fig. 3. Light weight load balancing.



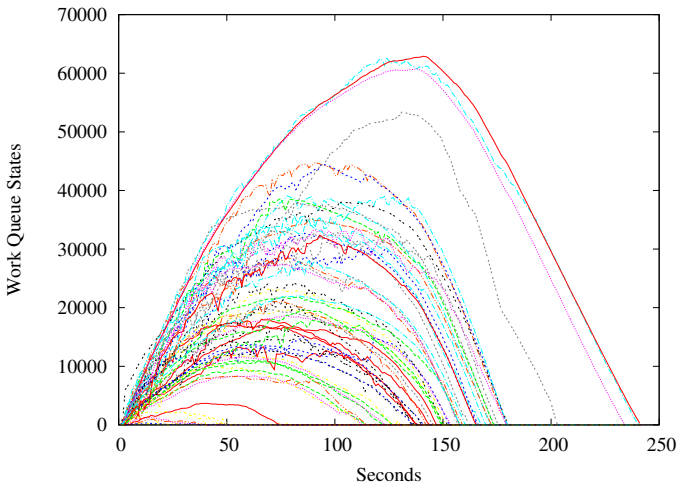


Fig. 4. LDash protocol with 60 workers work queue lengths with load balancing on (top) and off (bottom)

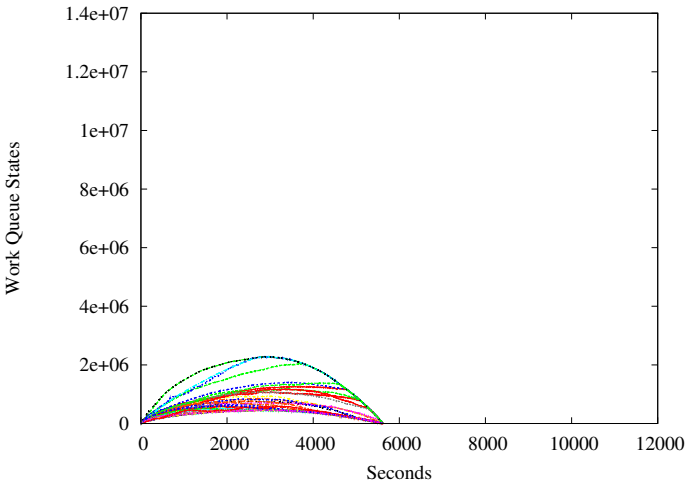
backoff to the point where peer nodes get the message.

Our final solution was to introduce a crediting mechanism. Each node gets a number of credits C to send state messages to each peer node. When a state message gets received by another node, an acknowledgment is returned. This provides a hard bound on the total number of messages a node may have in its message queue at any one time, i.e. NC where N is the number of nodes. We note that we did not notice performance bottleneck due the additional messages (i.e., the acknowledgment messages).

In Figure 1, we compare two PREACH runs for the same model (the LDASH protocol with $HomeCount = 1$ and $RemoteCount = 4$, running with 8 workers. The plot on the top shows the results for crediting. The spiking nature of the plot is expected. We want to avoid the message queues to grow unreasonably large like in the plot on the bottom. When not using crediting, two workers have their message queues grow roughly 3 orders of magnitude larger than the other node's message queues. In fact, we can barely see the message queue of the other 6 workers. Since crediting controls the number of messages for each worker, the overall algorithm works faster. In this particular example, PREACH-with-crediting finished computing in about 600 sec; without crediting the computation timed-out at 21600 sec (we only display the first 5000 sec since the message queue grows very rapidly in the beginning of the runs).

B. Light Weight Load Balancing

In a distributed, heterogeneous computing environment, the total runtime is determined by the runtime of the slowest node. Thus, if one node takes much longer than the others to finish its work, it does not matter how fast the others were. This is the reason why load balancing is important. Previous work by Behrmann [15] and Kumar and Mercer [16] observed the problem of an unbalanced work load. Even though at the end of the computation, all nodes have stored roughly the same number of states (due to uniform hashing) [10], the



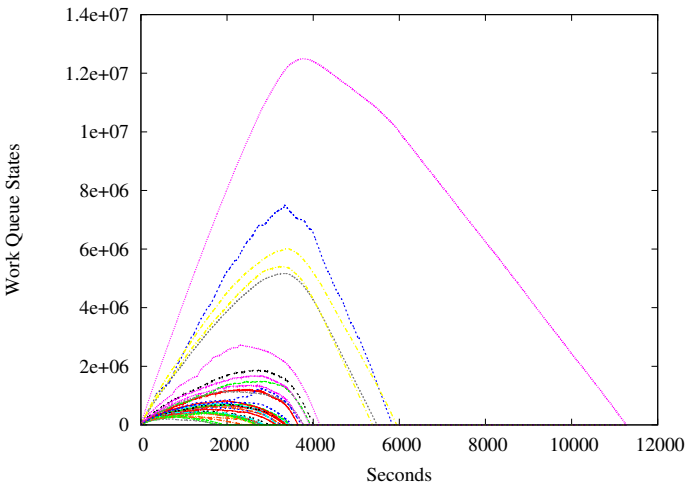
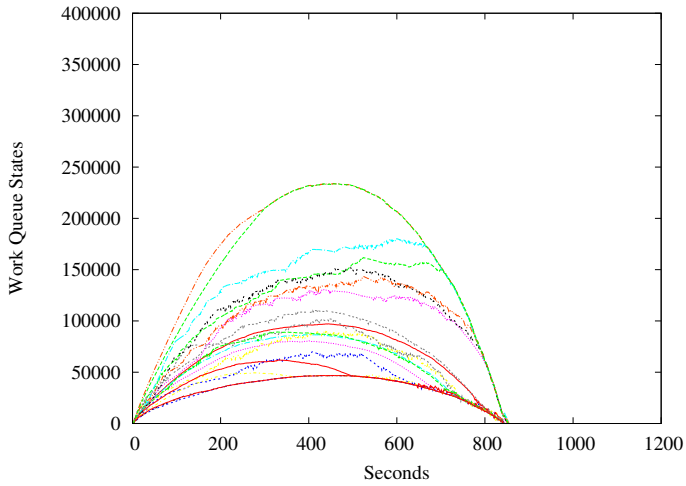


Fig. 5. SCI protocol with 40 workers work queue lengths with load balancing on (top) and off (bottom)



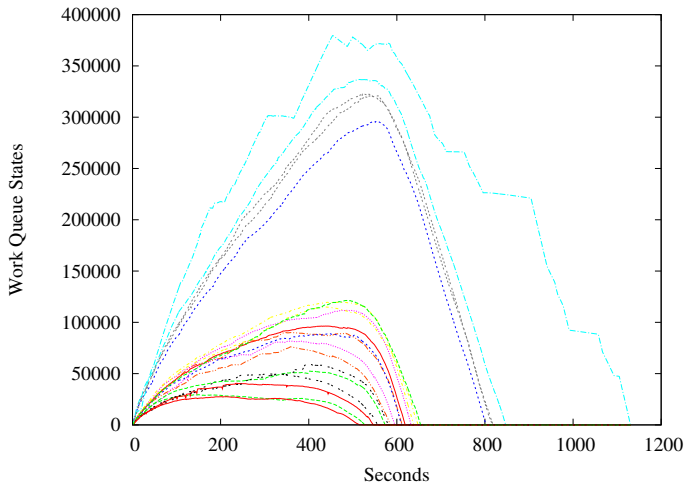


Fig. 6. German9 with 20 workers work queue lengths with load balancing on (top) and off (bottom)

dynamic work queue sizes during model checking can vary wildly between workers.

To address this problem, Kumar and Mercer proposed a rebalancing scheme that attempts to keep the work queues of all nodes equal throughout the computation [16]. This is achieved by comparing work queue sizes between hypercube adjacent nodes and passing states to neighboring nodes with smaller work queues. This is done in an aggressive manner, with the goal of keeping all work queues roughly the same lengths. In PREACH, since work queues are kept on disk, we are not concerned about minimizing the maximum work queue size (as Kumar and Mercer achieve). Rather, we simply desire to never have a node sitting idle with an empty work queue (until DEMC completion, of course). Our scheme, which we simply call *light weight* load balancing, reduces the amount of overhead. Each node tracks the sizes of all other nodes' work queues by sending work queue size information along with state messages. When one node notices that a peer node has a work queue that is $LBFactor$ times smaller than its own (for a fixed parameter $LBFactor$), it sends some of its states to the peer. Empirically, we found $LBFactor = 5$ to allow enough load balancing to occur so that nodes would complete around the same time without doing too much unnecessary rebalancing. If disk usage were an issue, one could use a smaller factor to keep the maximum work queues smaller at the expense of some extra load balancing.

To show the difference between the two schemes, we ran a simple 10^7 state counter model using PREACH with both schemes. In Figure 2 we see strict balancing that keeps the work queue sizes of all nodes identical. In Figure 3 we see light weight load balancing, which allows the work queues to vary somewhat. In both cases, however, all nodes complete around the same time. The benefit of the light weight scheme is that it is able to process states faster because it is doing less

load balancing and it completes sooner, at 2404 seconds as opposed to the stricter scheme which finishes in 2768 seconds.

To show the improvement between a load balanced run and a non-load balanced run, we ran several well known protocols: SCI, LDASH and German. These protocols are included in the Murφ distribution. We ran them on 60, 40, and 20 nodes, respectively. The results are in Figs. 4-6, which clearly demonstrate the efficacy of load balancing.

C. Batching of States

Stern and Dill's algorithm, and consequently PREACH, involves communicating successor states to their respective owners (see line 19 in Algorithm 1). The number of reachable transitions is typically at least a factor of 4 larger than the number of reachable states; thus, we could easily end up communicating billions of states in the large models we target. This motivates us to look at reducing the number of messages sent, which reduces the number of calls to Erlang's *Send* and *Receive* communication primitives, along with utilizing the network bandwidth more efficiently. Indeed, in simple Erlang benchmarking tests we have observed a factor of 10 to 20 speedup by sending states in batched lists of length 100 to 1000 as opposed to sending the states individually.

Each PREACH worker batches generated states in separate queue for each peer node before sending. This mechanism is controlled by two parameters: $MaxBatch$ and $FlushInterval$. A batch of states for a peer node is sent when $MaxBatch$ states have accumulated for the peer, or of $FlushInterval$ seconds have passed since the last batch to the peer, whichever happens earlier. We found that setting $MaxBatch = 1000$ and $FlushInterval = 1$ second achieved good performance. However, for very large models it is possible that different values would be needed, or an adaptive scheme where their values vary between nodes and over time. This was not explored in depth because while batching helps reduce runtime, some models suffered serious performance issues where a small number of nodes would become overwhelmed with states; this problem was described in Sect. III-A. Fortunately, grouping of messages is compatible with both methods that alleviate this problem: load balancing and crediting. When load balancing is activated, it is immediately known how many states should be sent from one node's work queue to the other node's work queue, so neither of the parameters are needed. With crediting, we must choose a value for $MaxBatch$, but the timeout $FlushInterval$ is unnecessary because $MaxBatch$ states are sent to node p whenever a credit is available for p and the round-robin scheduler selects p.

The idea of batching states for communication in DEMC has been considered before. Stern and Dill [10] batched states into groups of 10 before sending, as long as the current node had at least 20 states queued for expansion. In the algorithm implemented by Eddy Murφ [7], each node statically allocates a "communication queue" for each of the other $N-1$ nodes. A communication queue is essentially a two-dimensional array of states: 8 "lines" of 1024 states each. A line accumulates expanded states until it it is full. Then, it is marked to be sent to the node that owns the states, and an empty line is selected to next accumulate states. This approach allows multiple calls to MPI's nonblocking send function to be in-flight, effectively overlapping their execution. Also, the communication queue acts as an interface between the two MPI threads that run in concert on a single node in Eddy Murφ: one for state expansion and one for communication. In contrast, PREACH does not maintain multiple "lines"; instead, one large queue for each other node is maintained. Another major difference is that state expansion in Eddy Murφ will halt if a communication queue entirely fills up. PREACH, on the other hand, will not halt state expansion. If a sufficiently large number of states are waiting to be sent, and temporarily write the waiting states to disk to avoid too much memory usage.

D. Pseudocode

In Algorithm 2 we show pseudocode for the PREACH algorithm. The basic outline of the code follows closely the original Stern-Dill algorithm. However, the key features of batching, crediting, and load balancing all require changes. First, in order to implement batching, the outgoing states are placed in a queue (line 19) instead of being sent directly to their destination. These states are actually sent out later, in line 22, for the current destination. We visit the outgoing queues in a round robin manner and attempt to send to one node after each state is expanded. Next, for crediting, we return acknowledgments (line 30) from states sent to us in GETSTATES. Each node keeps track of the credits it has for sending to every other node. The $SendQ$ function contains the rest of the changes. It implements the logic that decides when we should send to a particular destination. It first checks to see that credits are available (line 39). Then it makes sure that we don't try to send states to a node that is currently sending us load balancing states (line 40). Next, we check to see if the node is eligible to receive load balancing states (line 41). If not, we check to see if we should send and states from its outgoing queue. We try to wait until there are at least $MaxBatch$ messages in a batch (typically $MaxBatch = 100$), but if the destination's work queue is low or if we don't have any work ourselves, then we will send smaller batches (lines 43-46).

IV. RESULTS

Here we present a number of experiments detailing the performance of PREACH. Most of these experiments were run on machines from a heterogeneous computing pool consisting primarily of 2.5-3.5Ghz Core 2 and Nehalem class Intel machines. Typically 4-8GB of memory was available per worker, although most experiments did not use all of this.

A. Scalability

Table I presents a few of the larger models we have verified with PREACH. All of the features discussed in Sect. III combined to allow us to achieve these results.

Algorithm 2 Preach DEMC

```
1: T : set of state
2: WQ : list of state
3: Cred : array of int
4: OutQ : array of list of state
5: WQEst : array of int
6:
7: procedure SEARCH
8:     T := ∅
9:     WQ := []
10:    CurDest := 0
11:    if I am the root then
12:        for each startstate s do
13:            Send s to Owner(s)
14:        end for
15:    end if
16:    while ¬Terminated() do
17:        GETSTATES()
18:        if WQ ≠ [] then
19:            s := Pop(WQ)
20:            for all s' ∈ Successors(s) do
21:                s'_c := Canonicalize(s')
22:                Enqueue s'_c in Owner(s'_c)'s out queue
23:            end for
24:        end if
25:        SENDQ(CurDest)
26:        CurDest := (CurDest + 1) mod NumNodes
27:    end while
28: end procedure
29:
30: function GETSTATES()
31:    while there is an incoming state s do
32:        Receive(s)
33:        SendAck()
34:        if s ∉ T then
35:            Insert(s, T)
36:            Append(s, WQ)
37:        end if
38:    end while
39: end function
40:
41: function SENDQ(dest)
42:    if Cred[dest] > 0 ∧
43:      WQEst[dest] < LBFactor * length(WQ) then
44:        if LBFactor*WQEst[dest] < length(WQ) then
45:            Dequeue min{length(WQ), MaxBatch}
46:          states from WQ and send to dest
47:        else if OutQ[dest] ≥ MaxBatch ∨
48:          (OutQ[dest] > 0 ∧ (length(WQ) = 0 ∨
49:          WQEst[dest] < MaxBatch)) then
50:            Dequeue min{length OutQ[dest], MaxBatch}
51:          states from OutQ[dest] and send them to dest
52:        end if
53:    end if
54: end function
```

Model	States ($\times 10^9$)	Nodes	Time (hours)	States per Sec per Node
Pet8	15.3	100	29.6	1493
Intel3_5	10.1	61	24.7	1860
Intel3_7	28.2	92	90.2	945

TABLE I
LARGE MODEL RUNS. HERE PET8 IS PETERSON'S MUTUAL EXCLUSION ALGORITHM OVER 8 CLIENTS (WITH NO SYMMETRY REDUCTION), AND INTEL3 IS AN INTEL PROPRIETARY CACHE PROTOCOL (WITH SYMMETRY REDUCTION ENABLED). THE LAST TWO ROWS ARE FOR INTEL3 WITH RESPECTIVELY 5 AND 7 TRANSACTION TYPES ENABLED.

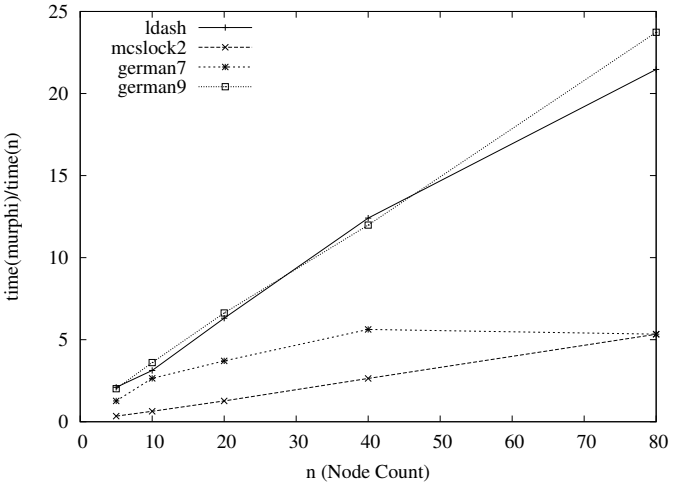


Fig. 7. Speed up experimental results.

B. Speed Up

Though previous implementations of the Stern-Dill DEMC algorithm report linear speed-up [7], [10], it is important to show that such speed up is also achieved in our implementation, especially given that a high level language (Erlang) is handling communication details. Fig. 7 shows the speed-up for n nodes against Murφ for a few public domain models. In all cases except german7 we see near linear speed-up. German7, which is German's protocol over 7 caches, shows the diminishing returns for smaller models; the reachable states (after symmetry reduction) are less than 2×10^6, and Murφ completes in only 315 seconds. Murφ took an hour or more for the three other protocols (German9: 6242 seconds, mcslock2: 8386 seconds, ldash: 3583 seconds).

V. RELATED WORK

Two most common methods of performing model checking are explicit enumeration of states and the use of symbolic methods. For some industrial protocols, explicit model checking is considered to be more effective verification technique [10], and hence is employed by numerous tools like Murφ [2], SPIN [17], TLC [6] and Java PathFinder [18]. Most of these model checkers explore the state of the system to be verified in a sequential manner which can be a hindrance (both in terms of memory usage and runtime) during the verification of systems

with large state space, thus making the use of parallel model checkers preferable.

Some examples of well-known explicit-state parallel model-checkers are - Murφ, DiViNe and PSPIN. Parallel Murφ [10] is a parallel version of Murφ model checker [2] based on parallel and distributed programming paradigms. Eddy Murφ [7] improves (in terms of speed) on Parallel Murφ [10] by providing the separation of concerns between next-state generation and communication during distributed model checking. DiVinE [19], [20] is a distributed model checker for explicit state LTL model checking and is known to handle large systems consisting of as many as 419 million states [21]. PSPIN has also been used for performing distributed model checking with the capability of handling up to around 2.8 million states [9]. One of the earliest work on distributed symbolic model-checking is presented in [22]. As previously mentioned, the need for load balancing in DEMC has previously been identified by first Behrmann in the context of timed automata model checking [15], and later Kumar and Mercer [16]. The latter's solution was contrasted with PREACH in Sect. III-B.

Being in a friendly competition with Eddy [7], we have exchanged ideas with Professor Gopalakrishnan and his students. Eddy's latest version implements more robust message passing that prevents slower nodes from getting overwhelmed with work. This improvement was the result of a collaboration with our team at Intel and was influenced by lessons learned while developing PREACH's backoff mechanism.

VI. Conclusions

We have presented a new industrial strength Murφ-based distributed explicit state model checker called PREACH. It was created to fill the need for a reliable, maintainable, scalable distributed model checker for industrial cache coherence protocols. PREACH uses the concurrent functional language Erlang to distribute the model checking load across many machines while relying on the proven Murphi codebase for low level routines. This results in source code that is simple while also being reasonably efficient. It has been proven to scale to real world models with nearly 30 billion states, and is freely available under an open source BSD license [13].

Acknowledgement

We thank Flemming Andersen (Intel Oregon) and Prof. Mark Greenstreet (U. British Columbia) for their ongoing support of the PREACH project.

References

[1] A. Hu, "Techniques for efficient formal verification using binary decision diagrams," Ph.D. dissertation, Stanford University, 1995.

[2] D. L. Dill, "The murphi verification system," in *International Conference on Computer Aided Verification*, London, UK, 1996, pp. 390–393.

[3] U. Stern and D. L. Dill, "Using magnetic disk instead of main memory in the murphi verifier," in *10th International Conference on Computer Aided Verification (CAV)*, 1998.

[4] C. N. Ip and D. L. Dill, "Better verification through symmetry," *Formal Methods in System Design*, vol. 9, no. 1/2, pp. 41–75, 1996.

[5] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *Correct Hardware Design and Verification Methods, IFIP WG 10.5 Advanced Research Working Conference, CHARME '95*, 1995, pp. 206–224.

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

[7] 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.*, vol. 11, no. 1, pp. 13–25, 2009.

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

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

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

[11] J. Armstrong, "The development of erlang," in *ACM SIGPLAN international conference on Functional programming*, 1997, pp. 196–203.

[12] G. J. Holzmann, "personal correspondence," 2010.

[13] Intel and U. of British Columbia, "Open-source PREACH." [Online]. Available: http://bitbucket.org/jderick/preach

[14] U. Stern and D. L. Dill, "Parallelizing the murphi verifier," *Formal Methods in System Design*, vol. 18, no. 2, pp. 117–129, 2001.

[15] G. Behrmann, "A performance study of distributed timed automata reachability analysis," *Electr. Notes Theor. Comput. Sci.*, vol. 68, no. 4, 2002.

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

[17] G. J. Holzmann, "The model checker spin," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, 1997.

[18] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," in *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2(4). Springer-Verlag, March 2000, pp. 366–381.

[19] K. Verstoep, H. Bal, J. Barnat, and L. Brim, "Efficient Large-Scale Model Checking," in *International Parallel and Distributed Processing Symposium*, 2009.

[20] J. Barnat, L. Brim, M. Češka, and T. Lamr, "CUDA accelerated LTL Model Checking," in *To appear in proceedings of ICPADS 2009.* IEEE, 2009.

[21] 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)*. Springer, 2008, pp. 48–62.

[22] S. Ben-David, T. Heyman, O. Grumberg, and A. Schuster, "Scalable distributed on-the-fly symbolic model checking," in *International Conference on Formal Methods in Computer-Aided Design*, 2000, pp. 390–404.