Fault Tolerance for Distributed Explicit-State Model Checking

by

Valerie Lynn Ishida

B.A., University of California, Berkeley, 2008

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

October 2014

Abstract

PReach, developed at the University of British Columbia and Intel, is a state of the art parallel model checker. However, like many model checkers, it faces reliability problems. A single crash causes the loss of all progress in checking a model. For computations that can take days, restarting from the beginning is a problem. To solve this, we have developed PReachDB, a modified version of PReach. PReachDB maintains the state of the model checking computation even across program crashes by storing key data structures in a database. PReachDB uses the Mnesia distributed database management system for Erlang. PReachDB replicates data to allow the continuation of the computation after a node failure. This project provides a proof-of-concept implementation with performance measurements.

Preface

This thesis is an original, unpublished, independent work by the author, Valerie Lynn Ishida. Some of the LaTeX for the algorithms of this document is adapted from [5].

PReachDB was written independently by myself. PReachDB was written starting from the code of the UBC PReach model checker written by Brad Bingham and Flavio dePaula as a course project for CPSC 538E: Parallel Computing, Spring 2009.

Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh and Mark Reitblatt presented the Intel PReach model checker, a rewrite of UBC PReach at Intel, in [5].

Table of Contents

List of Tables

List of Figures

Acknowledgments

Thanks to my supervisor, Mark Greenstreet, for helping me through the program. Thanks to my second reader, Rachel Pottinger, for teaching me some neat skills. Thanks to my labmates for their mentorship, to my lab and department for providing a great place to be, and to friends and family for moral support. Thanks to Michael Constant for companionship and guidance.

Dedication

Dedicated to my mother, Connie Ishida.

Chapter 1

Introduction

1.1 Motivation

Explicit-state model checking is a graph search problem on a finite state space, checking invariants at each reachable state. An example of when model checking is used is in the verification of cache coherency protocols, where an example invariant is that no two processors have the same cache line in the exclusive state at the same time. The model defines a set of possible states which correspond to configurations of the cache system at any point in time. In the example, a state could represent which cache lines are resident on which processors and in what cache state those lines are in.

Large state spaces can be described succinctly using high level state transition languages like Murphi [11], but the verification of the state space still requires visiting each reachable state. This exploration pushes the limits of both traditional and distributed computation. Distributed model checkers to date have not focused on handling faults, and thus tend to be problematic when used at an industrial scale.

PReach, a system for parallel reachability written in Erlang [3], implements the Stern and Dill distributed model checking algorithm [35] and has been successfully used for the verification of large models (billions of states), but it has no features to continue computation in the event of a system failure. If a failure occurs, the computation is ceased with no way to resume using previously computed values. Likewise, if a compute node loses connectivity, its workload is not dynamically shifted to an online node, and the system cannot make progress.

This document presents PReachDB, a new model checker based on PReach with the key features of computation restart on program crash and data replication with hot-swappable nodes, to address the reliability problems of a distributed model checker. The goal of this project is to determine if using database disk and replication techniques for distributed model checker fault recovery are feasible.

1.2 Model Checking

Model checking is the automatic verification of concurrent systems. A concurrent system has multiple units working simultaneously. Concurrent systems tend to have a protocol meant to guide the system towards the goal or desired functionality. Verifying that a protocol matches the desired functionality can be stated as a computational problem, where a model of the system is described in a high level language and the desired functionality is described in a specification language. A verifier takes the model and specification files as input, and outputs whether the model meets the specification. We will shortly discuss what kinds of properties can be specified, but first an example.

1.2.1 Hardware Designs and Specifications

An example of applied model checking is the verification of hardware designs. Modern digital designs consist of many concurrently working units. In the creation of hardware designs, modularity is essential; and interface design between modules is considered very difficult. Different modules may be connected to different clocks, and thus beat to different drums, or be connected to no clock at all. Protocol bugs found after fabrication can be extremely costly. Hardware specifications lay out properties about the item being checked that, if true, should guarantee that the item performs as desired. Note that it is possible for the specification to incorrectly describe what "working" means to the designer. For this reason, hardware manufacturers use the test-redesign cycle frequently.

1.2.2 Safety Properties, Invariants, Assertions

Specifications describe what the model must do to guarantee correctness. Correctness includes both safety and liveness properties. A safety property in a specification states that nothing bad happens, more specifically, that the system never reaches a failure state, such as computing a wrong answer, granting two processes simultaneous access to an exclusive resource, etc. Liveness properties state that something good happens; in other words that the system eventually reaches a good state. An example of a liveness property is responsiveness every request is eventually acknowledged. PReachDB verifies safety properties.

A safety property is invariant if it holds for all reachable states. When using a model checker, the user is usually interested in whether safety prop-

erties are invariant. One way invariant checking is written in model checking languages is to use assertions, which are program statements that cause an error if the condition being asserted is false.

1.2.3 Explicit and Symbolic Model Checking

Finite-state systems can be algorithmically checked as a graph search problem. The set of all states reachable by following any path starting from the set of initial states is called the *reachable state space*. In computing the reachable state space, we build the set of reachable states by iteratively adding states to the set of states known to be reachable from the initial states. A state transition is a change in the model state. A state transition in the model is represented in the graph as a path from one state to another. A state reachable by a state transition from another state is a successor of that state. The computation is finished either when a reachable state is found to violate the specification or if we have examined all reachable states and found all safety properties invariant.

A different approach uses the implicit graph and symbolic logic to decide whether states that violate safety properties are reachable. Ordered Binary Decision Diagrams (OBDDs) [8, 9] are often used to represent the set of reachable states as a boolean function. For protocol verification, explicit-state model checking often outperforms symbolic model checking [20]. For this document, we focus on explicit-state model checking nearly exclusively.

1.3 Overview

Our goal in this project is to implement a distributed, reliable system for use with model checking with the specific features of disk persisted data and computation resumption in the event of a system error using a database management system (DBMS) designed for use with functional languages. The research goals are to show that the system can recover in the event of the system going down and to collect measurements on state exploration, memory usage, message queue sizes, and machine time.

The main contribution of this project is a proof-of-concept implementation of a distributed, crash-tolerant model checker capable of computation resumption or continuation on the event of a node failure. Resumption is possible in the case of system crash when computation state is stored on disk. Continuation is possible through the swap in of a hot-spare node for a disconnected node and through the use of data replication among the node

pool. In addition we consider the theoretical likelihood of computation completion for this system compared with that of a traditional, fault-intolerant model checker.

This work does not address methods for making this system's runtime performance compare well with that of a traditional, fault-intolerant model checker. This work is also a proof-of-concept, and some features are not robustly implemented. For example, hot-spare swap-in is implemented for a single node failure only; the system could easily be engineered further to allow additional hot-spares, and the free use of hot-spares is assumed for the theoretical failure handling model.

1.4 Organization

This document is organized as follows. Related work in explicit-state model checking and in fault tolerance for distributed computing is discussed in Chapter 2. Chapter 3 contains full details of the PReachDB model checking system, including fault handling procedures. Chapter 4 shows the results of the system responding to failures and contains the performance analysis. Conclusions and future work are discussed in Chapter 5.

Chapter 2

Related Work

This chapter will discuss previous work relevant to the understanding of a distributed explicit-state model checker using disk and aiming to provide some fault tolerance. It will cover explicit-state model checkers and work done on the two orthogonal features of using disk and distributing the computation. It will also cover basic fault tolerance measures in distributed computing.

2.1 Explicit-State Model Checking

For concurrent systems with a finite number of protocol states, one method of verifying the safety properties is to enumerate all reachable protocol states. We can do this as a graph search problem, of which Breadth First Search (BFS) and Depth First Search (DFS) are algorithmic solutions.

Definition 1. *A state graph is a triple [21]:*

$$A = (Q, S, \Delta)$$

where Q *is the set of states.*
S *is the set of start states,* $S \subset Q$.
Δ *is the set of transition rules* $r_i : Q \to Q \cup \{Error\}$.

Reachable states include S and all states reachable from any application sequence of the transition rules from S.

In practice, model checkers implement either Breadth First Search or Depth First Search and keep a table of protocol states visited in random-access memory. Two well known model checkers are Murphi [11] and SPIN [26]. This document focuses on Murphi because both PReach and PReachDB use Murphi description files and both are geared towards the Murphi way of working.

2.1.1 Murphi

Murphi is both a description language and a verification system. The description language is a high-level programming language that can describe a concurrent system with protocol state type and variable declarations, transition rules and invariant descriptions. The Murphi compiler generates an executable C++ program from a Murphi description file. The C++ program enumerates all reachable states and checks the safety properties at each reachable state.

PReach uses the Murphi language for describing the model to be checked, but PReach parallelizes the verification process.

2.1.2 State Explosion

Time and memory usage are two major concerns when verifying a model. With the basic graph search approach, these both increase linearly with increasing reachable state space size. However, reachable state space sizes tend to increase much more rapidly than linearly with the model description file. This is known as the *state explosion problem* [21]. As explicit-state model checkers are used to verify larger and larger models, a point is reached at which the set of reachable states no longer fits in a table in memory.

Several techniques have been developed to reduce memory usage, which is doubly beneficial in that it allows us to check larger models that would not otherwise fit in memory and it decreases the amount of time the machine spends reading and writing memory. One method is to store in the visited state table only a hash signature of the state rather than the full state descriptor. This is known as hash compaction [37]. Improvements such as using ordered hashing [2] have been examined in [34]. Using the hash instead of the full descriptor may cause a problem if there is a hash collision. If a newly generated state hashes to the same signature as a state we have already visited, the state and all its descendant states may not be visited. If all protocol error states go undetected, then the verifier is said to produce false positives. Hash compaction provides bounds on the probability of missing even one state. In practice, the probability of missing a state can be made very small, and reduced even further by repeating the verification with a different random-seed for producing the hash function [37].

Another method is to create a table of bits. The bits are initially zero. When a state is inserted, two (or more) hash values for the state are generated and the corresponding bits are set to one. A lookup on the table works in the same fashion, and the state is considered present if the appropriate

bits are set to one. This method is called bitstate hashing [18, 19]. Bitstate hashing provides an average for the percentage of reachable states visited, but not a bound on omission of an error state.

Another approach exploits the inherent *symmetry* in typical protocols. Cache coherence and network protocols tend to be designed so that any agent may perform actions (such as enter a critical section), where *which* agent is acting may not matter. In [21] the authors add a data type called a *scalarset* to Murphi that allows for pruning of symmetric state graphs when the only difference between state descriptors is a permutation of a scalarset. This requires the users to change their Murphi description files, but if the user requires verification over a highly symmetric protocol, a great deal of memory and runtime savings can be achieved.

2.2 Disk-Based Explicit-State Model Checking

In the previous section we discussed *memory saving* approaches to handling state explosion. In this and the next section we discuss approaches of *auxillary storage*. This section covers the technique of utilizing hard disks to store state spaces larger than fit in RAM, and the next section covers distributed techniques utilizing networks of workstations or other aggregrates of machines.

2.2.1 Disk-based Murphi

In [33] the authors investigate how to use magnetic disks for storage of the reachable state hash table while keeping the overhead low. Using magnetic disk allows the state table to be of greater size than when using RAM memory alone. The difficult consideration is that the state table is normally randomly accessed, which if on disk would take orders of magnitudes longer.

The key insight in [33] is that disk accesses do not have to be randomly accessed, incurring the seek time penalty with each access. Disk accesses can be grouped and performed using a linear read of the entire disk state queue. The authors also report that hash compaction performs extremely well in reducing runtime for algorithms using disks.

The basic disk-based algorithm shown in Figure 1 of [33] is reproduced in Figure 2.1. For this algorithm, two state tables are used, one contained in memory and one on disk. The in memory table is accessed as normal, while disk accesses happen sequentially in $CheckTable()$. Unlike the usual breadth first search algorithm, successor states are not checked for presence in the disk table or added to the FIFO work queue right away. Instead, for

each level of the breadth first seach, successors of all states in the queue are generated and stored in the memory table. Once the queue becomes empty, new states (those in the memory table and not in the disk table) are added to both the disk table and the work queue. Determining which states are new is done by linearly reading the disk table and removing from the in memory table any previously stored states. In addition to per breadth first search level, $CheckTable()$ is also invoked when the memory table fills up.

The authors of [33] estimate the performance overhead for using this algorithm, and find it to have increasing slowdown as the ratio of states on disk to states in memory (called memory savings factor in [33]) increases. They find that, comparatively, for an instance of the SCI protocol their algorithm performs with 151% runtime overhead over the conventional algorithm using just memory, while the conventional algorithm, if seeking for every access, would have a 4108% runtime overhead. When combined with hash compaction, the authors report runtime overhead around 15% for the disk algorithm.

Although disk-based Murphi allows larger state tables than can fit in RAM, it cannot process states in parallel and is thus much slower than PReach.

2.2.2 Transition Locality

A refinement presented in [10] to the disk-based algorithm uses the property of transition locality within the protocol transition graph to decrease disk accesses. Transition locality is defined in terms of breadth-first search (BFS) levels [36], where *at level* k is defined as $L(0) = I, L(k+1) = \{s' \mid \exists s$ s.t. $s \in L(k)$ and $R(s, s')$ and $s' \notin \bigcup_{i=0}^{i=k} L(i)\}$. A graph exhibits 1-local transition locality if most transitions from a source state lead to a destination state either on the previous level ($L(k-1)$), the same level ($L(k)$), or the next level ($L(k+1)$) as that of the source state ($L(k)$). In [36] the authors find that for a handful of protocols provided with the Murphi distribution, "for most states more than 75% of the transitions are 1-local."

The locality-based disk-based BFS algorithm in [10] is similar to the algorithm in [33] except for the function $CheckTable()$. Whereas in [33] the entire disk table is read to remove states from the memory table and the state queue of unchecked states, in [10] only parts of the disk table are read. By decreasing the amount of time spent reading from disk, the authors are able to verify protocols using less runtime. Assuming states are added to the disk table in BFS level order, the insight of this paper is to look primarily at the *tail* of the disk table, which contains the 1-local transition states.

```
var     // global variables
  M: hash table;      // main memory table
  D: file;            // disk table
  Q: FIFO queue;      // state queue

SEARCH()     // main routine
begin
  M := ∅; D := ∅; Q := ∅;      // initialization
  for each startstate s0 do     // startstate generation
    INSERT(s0);
  end
  do                            // search loop
    while Q ≠ ∅ do
      s := dequeue(Q);
      for all s' ∈ successors(s) do
        INSERT(s');
      end
    end
    CHECKTABLE();
  while Q ≠ ∅;
end

INSERT(s: state)     // insert state s in main memory table
begin
  if s ∉ M then begin
    insert s in M;
    if full(M) then
      CHECKTABLE();
  end
end

CHECKTABLE()     // do old/new check for main memory table
begin
  for all s ∈ D do     // remove old states from main memory table
    if s ∈ M then
      M := M − {s};
  end
  for all s ∈ M do     // handle remaining (new) states
    insert s in Q;
    append s to D;
    M := M − {s};
  end
end
```

Fig. 1. Explicit State Enumeration Using Magnetic Disk

Figure 2.1: Figure 1 from [33].

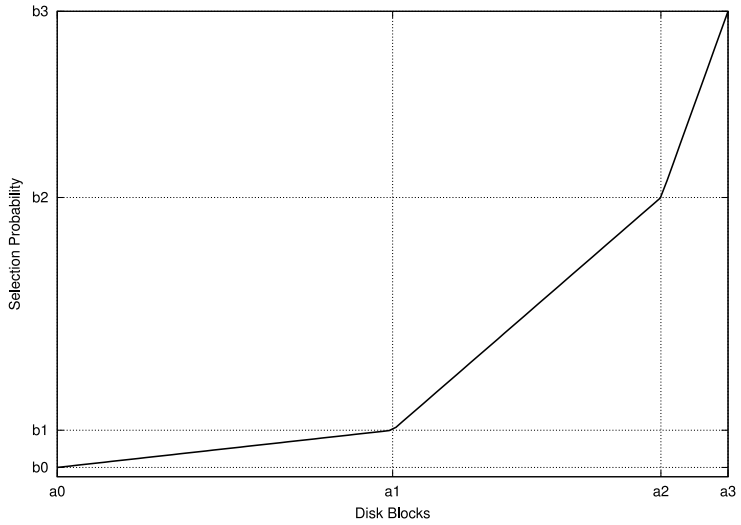

Fig. 6. Probability curve for disk cloud block selection (used by GetDiskCloud())

Figure 2.2: Figure 6 from [10].

To probabilistically improve the chance of finding a non-1-local state (a *far back* state) in the disk table, the authors propose a scheme for dynamically choosing which segments of disk states to read. The probability of selection increases as per Figure 6 of [10], reproduced in Figure 2.2, where variable values are chosen experimentally. Both axes are relative between values 0 and 1, where the relative disk block index is ρ = block index / number of blocks.

The authors of [10] tune for how many states to read from disk based on the disk table size. The disk table is segmented into a fixed number (N) of variable size blocks (of size $\lceil S/N \rceil$ where S is the number of state hash signatures in the disk table). When $\lceil S/N \rceil$ is less than a minimum value B, B is used for block size instead to prevent unnecessary seeks when the disk table is small. Thus while the disk table is small, this algorithm reads the entire table for each BFS level, as in [33]. The program also dynamically tunes for block selection effectiveness by periodically running a calibration $CheckTable()$ that reads the entire disk table while removing old states and that counts how many states would have been in the disk block selection and how many would not have been. Based on the counts, the program decides whether or not the block selection is effective, and either increases or decreases the number of blocks to read at each $CheckTable()$ accordingly.

The authors of [10] report that their algorithm is about 10 times faster

than the algorithm in [33]. For protocols where a tenth of the model size ($M(p)$ = the minimum amount of memory needed to complete state space exploration for protocol p) fits in RAM, the authors find that the verifer is between 1.4 and 5.3 (3 on average) times slower than a RAM-BFS with enough RAM to complete the verification task. The overhead is likely larger than that reported in [33] due to the significantly larger models being verified in [10]. The models in [33] are of size able to store 250K to just over 1M reachable states, whereas in [10] they are of size able to store 2M to over 125M reachable states. This suggests that tuning the disk algorithm is effective at increasing the size of what we can verify with real hardware, but it cannot overcome the increasing slowdown as the *memory savings factor* increases, as noted in [33]. For large models it is still not competitive with PReach for speed.

2.3 Distributed Explicit-State Model Checking

Distributing state generation and storage over multiple machines has several advantages over using a single machine. The greater aggregate memory allows us to check models of greater size than could be held by a single machine. Additional processors allow us to check properties and generate successors of multiple protocol states at a time. Given the potential speedup and scalability advantages of distributing the task, it is not surprising that much work has been done in this direction.

2.3.1 Parallelizing the Murphi Verifier

The seminal paper on parallelizing an explicit-state model checker is by Stern and Dill [35]. The authors took the existing Murphi verifier algorithm and implementation and presented a parallel and distributed version of each. Unlike previous work, which was aimed at reducing the size of the reachability graph or reducing memory footprint, this work aimed to use parallelism to reduce the runtime. This paper laid the foundation for our and other model checkers, and much of the terminology and methods originate from this paper.

Parallel Murphi, the tool presented in [35], utilizes a statically partitioned state table to store reached protocol states. Each instance of Parallel Murphi, either as a thread or a process on either a separate or the same machine, is called a node. Each state s has a single node as its *owner*. The owner is computed statically using a hash function $Owner(s)$. When a node generates a new state, it is sent to the state's owner. At each node, owned

states are stored in the local partition of the state table. In addition, each node utilizes a private work queue. When a state is received, if it has not been entered in the state table then it is added to both the state table and the work queue. States are removed from the work queue when their successor protocol states have been generated and sent to each of their owning nodes.

A received state may either be

1. in neither the state table nor the work queue, indicating it is a new state we have not reached before;

2. in the state table and the work queue, indicating it is counted in the reachability graph, but its successor protocol states have not been generated for exploration yet; or

3. in the state table and not in the work queue, indicating it is counted in the reachability graph, and its successors have been generated and sent to the appropriate nodes.

The basic algorithm presented in Figure 1 of [35] is reproduced in Figure 2.3. The basic algorithm uses active messages to communicate, where both the data and procedure address are sent. The authors of [35] chose active messages for their efficiency when nodes are executing asynchronously. Current message passing libraries provide the desired functionality in an easy-to-use form, and thus our model checker abandons active messages for library methods.

Parallel Murphi's termination conditions are that there are no more messages (containing states) in flight and that there are no more states in the per-node work queues. Parallel Murphi designates one node as the master node. The master node sends out the initial set of states at startup and is responsible for checking for termination conditions. Termination detection in parallel Murphi and in PReach is determined at the master by the equation

$$\sum_{i=0}^{N-1} Sent_i - Received_i + |Q_i| = 0 \tag{2.1}$$

where $Sent_i$ and $Received_i$ are counter variables for each of the $i = 0, ..., N-1$ nodes, where N is the number of nodes, and where $|Q_i|$ is the size of the work queue at node i. The master node queries each node for these values when it has been idle longer than a threshold amount of time. Before

```
var      // global variables, but local to each node
  T: hash table;      // state table
  Q: FIFO queue;      // state queue
  StopSend: boolean;      // for termination detection
  Work, Sent, Received: integer;

Search()      // main routine
begin
  T := ∅; Q := ∅;      // initialization
  StopSend := false; Sent := 0; Received := 0;
  barrier();
  if I am the master then      // master generates startstates
    for each startstate s0 do
      Send(s0);
    end
  do      // search loop
    if Q ≠ ∅ then begin
      s := top(Q);
      for all s' ∈ successors(s) do
        Send(s');
      end
      Q := Q − {s};
    end
    poll();
  while not Terminated();
end

Send(s: state)      // send state s to "random" node h(s)
begin
  sc := canonicalize(s);      // symmetry reduction
  while StopSend do      // wait for StopSend = false
    poll();      // (for termination detection)
  end
  Sent++;
  send active message Receive(sc) to node h(sc);
end

Receive(s: state)      // receive state (active message handler)
begin
  Received++;
  if s ∉ T then begin
    insert s in T;
    insert s in Q;
  end
end
```

Fig. 1. Parallel Explicit State Enumeration

Figure 2.3: Figure 1 from [35].

answering the master node's query, each node sets *StopSend* to disallow sending messages. This allows us a snapshot of the system.[1]

Parallel Murphi writes out a log file at each node of its local state table partition. Each record contains a compressed state value and a pointer to a predecessor in the form of the predecessor's owning node's number and a record position in the predecessor's owning node's log file. This allows error trace generation when a protocol error is found. PReach at Intel [5] can provide some error trace generation, but PReachDB currently does not support it.

The authors of Parallel Murphi identified two areas of potential improvements for a parallel model checker. They identified load balancing as a problem when bandwidth is non-uniformly available. Nodes with limited bandwidth become bottlenecks. In such a case, the randomized load balancing provided by the static hash function provides poor performance. They also identified communication overhead as a variable affecting runtime. They were able to decrease runtime by aggregating messages containing states into batches of 10 when 20 or more states exist in the work queue. These problem areas continue to be of great importance to model checker performance and are further addressed in the literature.

PReach is based off of Parallel Murphi and PReach explores both areas for improvement identified by the authors of Parallel Murphi.

2.3.2 Eddy

The Eddy project's [27, 28] main achievement is the specialization of threads local to a compute node in order to achieve a modular design for a parallel, distributed model checker. Eddy uses shared memory to commuicate between two threads on a single node (implemented in POSIX Theads [17]), and uses message passing (implemented in MPI [15]) to communicate between nodes. The tasks on a compute node are separated into two categories: state generation and inter-node communication. One thread takes on the role of generating successor states and checking safety properties, and another handles bundling states into packages and sending and receiving messages containing states.

Eddy's architecture is structured to take advantage of MPI [15] and PThreads [17] primitives. For example, in termination detection, a token is passed along the nodes in a ring accumulating the $Sent_i$ and $Received_i$ when the node is inactive. If the counters are equal when passed back to the root

[1] PReachDB's termination detection is simplified by the use of acknowledgment messages; see Section 3.7.1.

node, the computation is complete. MPI broadcast is used to terminate all nodes in the case where a state is found not to satisfy the safety property. One way in which using the MPI interface with Eddy is slightly unwieldly is the way Eddy batches states per destination node. Separate buffers, called lines, per destination node are stored on each node, and complicated bookkeeping is required to manage which lines are ready to be sent. Lines which have been sent also must not be overwritten until the send operation has completed, otherwise the sent data could be corrupted.

The version of Eddy implemented to work with Murphi models is called Eddy_Murphi. The authors present experimental performance results for Eddy_Murphi compared with single machine, standard Murphi on models with 10^6 to 10^8 states. They observe near linear speedup as the number of nodes increases, with a note that one of their models does not perform well with their static state partitioning function for certain numbers of nodes.[2] The authors report being unable to reproduce the work in [35] due to changing cluster technology. They report that an MPI port of parallel Murphi [32], performs worse than standard Murphi. They also report being able to verify the FLASH protocol [23] for 5 processors and 2 data values as parameters, a model with more than 3×10^9 states, in approximately 9 hours on Eddy_Murphi with 60 nodes. This model would have been expensive to verify on a single machine at the time, requiring an estimate 15 GB of RAM memory for the hash table on standard Murphi and requiring an estimated compute time of 3 weeks on disk Murphi [30].

2.3.3 PReach

PReach [5], short for parallel reachability, is a new implementation of distributed, parallel Murphi written in Erlang. Erlang is a concurrent, functional language with elegant communication primitives. Implementing PReach in Erlang allows the authors to easily represent and make adjustments to the communication aspects of distributed Murphi. The core algorithm is represented in under 1000 lines of code, and PReach has been used to verify an industrial cache coherence protocol with approximately 30 billion states.

PReach uses existing Murphi C code for "front end parsing of Murphi models, state expansion and initial state generation, symmetry reduction, state hash table look-ups and insertions, invariant and assertion violation detection, and state pretty printing (for error traces)" [5]. The Erlang code handles the complex distributed communication code. This architectural

[2] The authors cite [25], which more directly focuses on load balancing.

division allows the authors to harness existing proven technology and to focus on correctly implementing the communication layer.

The basic algorithm is the same as that in [35]. Three features explored in [5] are batching of states, load balancing, and crediting. I will discuss them in the following paragraphs.

In the distributed algorithm, when states are generated, they are sent to their respective owners as messages in a message passing system. As shown previously in [35] and [27], aggregating messages containing states into batches improves runtime performance by decreasing the percent of time spent as overhead processing messages. In the PReach paper [5], the authors observed a throughput benchmark speedup factor of 10 to 20 batching messages of 100 to 1000 states compared with messages of individual states. Unlike Eddy Murphi [27], which has a fixed size communication buffer of 8 lines per destination, PReach has a single variable-size queue per destination of states to be sent. A fixed size buffer may require the exploring thread to block on insert if the buffer is full, but there is no chance of running out of memory due to the send queue. With the variable size buffer, exploration does not have to block, but the send queue may overrun memory. To handle this, PReach will write the send queue to disk if it is sufficiently large.

The problem of load imbalance in distributed explicit-state model checking has been noted and addressed often in the literature [4, 25, 35]. Load imbalance occurs when the distribution of work is uneven per unit time. The main symptom is a disproportionate work queue size distribution over the set of nodes. Since the runtime of a distributed problem is determined by the slowest worker, techniques for evening out the size of the work queues have been successful at decreasing runtime. Load balancing also has the benefit of decreasing memory usage of the work queue, which is beneficial because model checkers are inherently memory-bound. In [25] the authors present an aggressive balancing scheme for achieving nearly perfectly equal queue lengths. Balancing is done by comparing queue lengths with dimensional neighbors (for [25] nodes are grouped in a hypercube-like structure), and sending an amount of states equal to half of the difference between the queue lengths. In [5] a more relaxed load balancing scheme is implemented with the goal of never having an idle node with an empty work queue. PReach achieves this by including queue size information with messages containing batched states. When a node receives size information from a peer and the peer's queue size is smaller than its own by a factor of 5 (an empirically found factor), the node sends to its peer some states from its work queue. This scheme achieves the desired result and achieves runtimes similar to and sometimes better than that of [25].

Crediting is a method to protect against overwhelming nodes with excessive messages during the distributed computation. Erlang's message passing system guarantees delivery eventually and is implemented with a variable-size message receive buffer. If a node receives large numbers of messages continuously faster than it can service the messages, the node will allocate more memory for the message buffer and eventually start paging. In early versions of PReach this behavior is observed and often leads to the standstill or crash of the node. Crediting is implemented by keeping on each node credit counters for each peer node. When a message is sent to the peer, the appropriate counter is decremented. When an acknowledgment message is received, the appropriate counter in incremented. While the counter is zero no new messages are sent to that peer. This scheme provides an upper bound on the size of the inbound message queue of $n \times C$ where n is the number of nodes and C is the per peer initial credit amount.

PReach is useful as a distributed model checker prototyping platform because it separates the distributed computation and model checking aspects cleanly. PReachDB builds off of an early version of PReach, and focuses on handling node crashes.

2.4 Fault Tolerance in Distributed Computing

For a survey of fault tolerance in distributed computing, we direct the reader to [16].

PReachDB is a distributed system that uses point-to-point message passing for communication. The main tactics employed in PReachDB to handle faults are to persist data to disk and to replicate data across multiple nodes. The Mnesia DBMS is part of the Erlang OTP framework. We use Mnesia to implement our distributed data storage.

2.4.1 CAP and Mnesia

The CAP Theorem [7] describes a space of tradeoffs in designing a distributed system. The rule of thumb is: from consistency (C), high availability (A), and tolerance to network partitions (P), a distributed system can have at most two. In [6] the author clarifies that the apparent tradeoff is less strict if the creators of the system are able to handle operations during a network partition with some finesse.

Mnesia is a CP system. It guarantees consistency and is tolerant of network partitions, but it does not guarantee availability. In particular if

all nodes storing some data are unreachable, then on a read or write of that data Mnesia will fail with an error.

2.5 Summary

The major difference between PReachDB and previous distributed model checker is the use of a database backend to provide persistent storage of key data structures. To the best of our knowledge, there is no previous work of using persistent, database storage in model checkers.

Chapter 3

PReachDB

3.1 Overview

The goal of this project is to implement and demonstrate a fault tolerant, distributed, explicit-state model checker. This chapter describes PReachDB, our extension to PReach that uses the persistent, transactional storage provided by a disk-based database system. Here we address issues required for fault tolerance and correctness. Chapter 4 evaluates the performance of PReachDB, and Chapter 5 examines the feasibility and practicality of this approach. Using disk storage with model checkers for increasing the maximum model size capable of being checked has been studied [33], but not with the aim of handling faults of the distributed computation.

Standard distributed explicit-state model checkers (DEMC) assume that all nodes will remain online and connected to each other and that all messages that are sent will be received. This guarantees that the entire state space will be reached, given enough memory.

In this project, we allow node failures and dropped messages to happen. Our goal is to build a DEMC with tolerance for these kinds of failures. Messages could be dropped or could be sent to an inactive node, and node-local state could be lost if a node crashes. PReachDB provides solutions for these new circumstances, and we explain them in this chapter.

This chapter gives details of the PReachDB tool developed for this project. For details of PReach, upon which the code was based, see Section 2.3.3 and [5]. After Section 3.2, in which we present the PReach DEMC algorithm, this chapter focuses on the differences between PReach and PReachDB. The extra data structures required to persist and replicate data are discussed in depth in 3.3. A sketch of the program is presented in 3.4. The need for message acknowledgments when nodes can fail is presented in 3.5. The implementation of the fault tolerance features using Erlang's Mnesia database system is presented in 3.6. The modified DEMC algorithm and new termination detection method are shown in 3.7.

Algorithm 1 Stern-Dill DEMC as presented in [5]

```
 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 start state 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:             Check(s)
18:             for all s' ∈ Successors(s) do
19:                 s'_c := Canonicalize(s')
20:                 Send s'_c to Owner(s'_c)
21:             end for
22:         end if
23:     end while
24: end procedure
25:
26: procedure GETSTATES( )
27:     while there is an incoming state s do
28:         Receive(s)
29:         if s ∉ T then
30:             Insert(s, T)
31:             Append(s, WQ)
32:         end if
33:     end while
34: end procedure
```

3.2 Distributed Explicit-State Model Checking Algorithm

The PReach algorithm, shown in Algorithm 1, is based on Stern and Dill's distributed, explicit-state model checking (DEMC) algorithm. The Stern and Dill algorithm is described in [35], and the PReach algorithm in [5]. The basic idea is graph search over an explicit-state space, starting from a given set of states and exhaustively computing the reachable space. While visiting each state, properties from the specification are checked against the state. If a violation of the specification occurs in the set of reachable states, then the model does not meet the specification and the computation is terminated. If there are no further reachable states to check, then the computation is terminated and the model satisfies the specification.

3.3 Data Storage

States in PReach are represented as Erlang tuples. PReachDB uses the same representation and a similar set of data structures, but the key data structures are stored in an Mnesia database instead of in memory.

3.3.1 Mnesia

Mnesia [24] is a distributed DBMS for Erlang designed for fault tolerant telecommunications systems. It provides key/value store and lookup, sharding of a database table over a pool of computing nodes, replication of tables or shards, and reconfiguration of the system while on-line. Given these features, it seems an ideal candidate for adding fault tolerance to PReach.

In this section, an Mnesia table refers to a key/value store implemented in Mnesia. For tables keyed on states, we use $type = set$ to enforce unique keys. Mnesia tables may be held in memory, on disk, or both.

3.3.2 Data Structures in PReach

In the basic version of PReach, a state is inserted once into a hash table of all the visited states when a compute node first visits the state. The entry is never updated, and it is looked up several times by compute nodes deciding whether this state should be visited or not. In more complicated versions of PReach, each tuple may be updated several times, such as if in-degree is being tracked.

Each compute node has a work queue containing the states whose predecessors have been explored, but who themselves have not yet been explored. Each queue contains only the states owned by the compute node it is associated with. States are added to the queue when they are received over the Erlang messaging system from other nodes or from the same node. When the compute node is ready to explore a new state, it will pop a state from the queue.

3.3.3 Data Structures in PReachDB

There are five data structures used in PReachDB. Two are held in program memory, one is held on disk, and two are held either in memory or on disk depending on the user's preference. The data structures are

1. a per-node, in-memory work queue;
2. a per-node, in-memory table of states waiting for acknowledgments;
3. a per-node, on-disk Mnesia table containing a single entry, the epoch number of the node;
4. the global work queue, implemented as an Mnesia table; and
5. the global visited state table, implemented as an Mnesia table.

Each of these will be explained in turn.

The major difference between PReachDB's and previous distributed model checkers' data structures is the presence of Mnesia database tables. For the current version of PReachDB, these tables hold data that is accessible from any node when accessed through the Mnesia API. In Section 3.7 we show how these additional structures fit into our fault tolerant DEMC algorithm.

Per-Node Work Queue

The per-node, in-memory work queue in PReachDB is similar to that in PReach. It is implemented as an Erlang list argument to the tail-recursive function that performs the main loop in SEARCH(). Within SEARCH(), the first element of the in-memory queue is removed, verified for satisfying safety properties, and used to generate successor states. When a state that has never been seen before is received in a message, the state is added to the end of the in-memory queue. See Section 3.6.3 for further discussion of the differences between the Per-Node Work Queue and the Global Work Queue and of how they are used when work is migrated between nodes.

Acknowledgment Table

PReach makes the assumption that the nodes will stay connected. As Erlang guarantees message delivery if possible, this is a reasonable assumption. For PReachDB we do not make that assumption, as it is possible that a message will be sent but never received because the receiver crashes. This means that if a message gets dropped, an entire portion of the search space may not get explored. To handle this, we wait for acknowledgment messages from the receivers before removing a state from the global work queue. We keep a table in memory consisting of a triple of {a state, the number of its successors for whom we have not received acknowledgments, the sequence number for this round of acknowledgments}. The counter of unreceived acknowledgments is decremented when acknowledgment messages are received. When the counter reaches 0, we remove the state from the global work queue. This process is further explained in Section 3.5. When all of the successors of a state have been acknowledged and safely stored to the disk work queue, it is safe to remove the predecessor state from the disk work queue, as we will not need to regenerate those graph edges again.

Epoch Number Table

When we allow for a node to rejoin the computation after crashing and restarting, we need to provide the means for a node to detect if the messages it receives are no longer relevant to the data in memory. To solve this problem we introduce epoch numbers. Each node has its own epoch number which tracks how many times that node has started up. These per-node epoch numbers are stored using persistent storage, and each node increments its number on start-up. Nodes include their current epoch numbers in messages sent to other nodes, and acknowledgment messages note the epoch of the message that is being acknowledged. In this way, a node can disregard acknowledgments for messages from earlier epochs. Section 3.5 describes this process in more detail.

Global Work Queue

The states yet to be explored are stored in an Mnesia table, along with states we may have to re-explore if a crash were to occur. The Mnesia work queue stored the states but does not maintain order. The states in the Mnesia work queue are a superset of the states in the in-memory work queues.

If a crash occurs, the in-memory lists are lost, but the Mnesia work queue is preserved. On a restart, when a node starts up and has an empty

in memory queue, it will ask Mnesia to read some states from the Mnesia work queue. Currently PreachDB uses the *mnesia:all_keys/1* function, which returns all of the states, and pseudorandomly choses a sublist of states to add to the in-memory work queue. Which states are explored by which node does not matter much, because all states from the Mnesia work queue will eventually be fetched in this way or otherwise explored from another predecessor link. States are removed from the Mnesia work queue when the number of acknowledgments outstanding is zero.

In an earlier iteration of PReachDB, each node maintained its own disk work queue. This configuration worked under the assumption that all nodes would eventually come back online, and that swapping out a node would not be required.

Global Visited State Table

The global visited state table is an Mnesia table fragmented across all of the compute nodes. The table can be held on disk and/or in memory. Fragment replication count can also be set between 1 and N, the number of compute nodes. A disk copy of each fragment of the table must be kept by at least one node for all the data to persist through a system restart. The replication count must be at least 2 to handle a single node disk failure.

The key of the hash table is the state itself (an Erlang tuple). This is possible because Mnesia is an extended relational DBMS, which can use arbitrary Erlang data structures as keys. A hash of the state could be used as a key if probabilistic search coverage is desired instead. Using the state as the key makes the table much larger than it would be with hash compaction [37]. PReach uses hash compaction.

3.4 Implementation

Like Parallel Murphi [35] and as described in 2.3.1, PReachDB designates one node as the master node. The master has the same responsibilities as in Parallel Murphi but must also create the database tables.

The sketch of the program with regards to Mnesia is as follows.

1. Start up each compute node.
2. Each compute node calls `mnesia:start()`.
3. If this is the first run, have the master node create the tables. Each compute node connects to the Mnesia tables.

4. Each compute node waits for the tables to load.
5. Each node does its work, accessing the tables.
6. Each compute node calls `mnesia:stop()`.

3.4.1 Launch

Node pool launch was implemented in PReach using the Erlang module **slave**. **slave** is a module to start additional Erlang nodes on local or remote hosts. An Erlang node is a single thread of execution in the Erlang runtime environment. When connecting to a remote host through ssh, **slave** does not correctly escape (backslash) quotes in command-line arguments. Mnesia requires as command-line arguments the properly quoted filesystem location where the schema table and data tables will be stored. It is not possible to start Mnesia remotely using the **slave** module. We instead use Perl scripts to call Erlang directly on each machine.

3.4.2 Table Creation and Management

Setting up the tables is a multi-step process that requires coordination between the nodes. We use message passing to block a node's progress until coordination messages are received. The following describes how we set up the tables in roughly chronological order.

Mnesia runs in the same address space as the application. Any Erlang configuration done on a node affects its Mnesia application. Each node calls `mnesia:start()`. The non-master nodes invoke a remote procedure on master to send to itself the list of `db_nodes` registered to Mnesia. The non-master nodes add the retrieved `db_nodes` to its `extra_db_nodes` through the Mnesia configuration. This sets up the node's Mnesia application to connect to the given nodes and share table definitions, including the schema. "The schema is a special table which contains information such as the table names and each table's storage type." [14] Each node then sets the schema to be stored both on disk and in RAM (the default is in RAM only). The schema needs to be set to type `disc_copies` before any disk resident tables are created.

Each non-master node sends a message to master that it has completed the step to set schema to disk. When master has received ready messages from each of the other nodes, it creates the tables.

The epoch number table is created using `mnesia:create_table/2`.[3] This table is created with the `local_content` option set to true, so each node stores its own value in this table. This table is always stored to disk.

We use an open source Erlang library called **Fragmentron** [29] to create our fragmented tables. Both the Global Work Queue and the Global Visited State Table are created in this manner. We call `fragmentron:create_table/2` with the following value for `frag_properties`.

```
[{n_fragments, length(NodePool)},  % N table fragments
 {node_pool, NodePool},            % Use our nodes in node pool
 {n_disc_copies, NDiscCopies}]     % Disc fragment replication
```

This command tells **Fragmentron** to create a table with N fragments, where N is the number of nodes in our node set. The table's node pool is set to our compute nodes. The number of fragment replicas to maintain is set to NDiscCopies. We make no distinction between the "true" fragment and its replica in this section. An appropriate value for NDiscCopies is discussed further in 3.6.1. If RAM only tables are desired, `n_disc_copies` should be replaced with `n_ram_copies`.

Fragmentron is a helper library for fragment replica balancing between the nodes in the node pool. The addition of additional nodes to the node pool or deletion of an existing node is handled gracefully. Fragment replicas are spread out evenly among nodes. If a node deletion causes the number of replicas to drop below `n_disc_copies`, a new replica is created on an existent node in the node pool.

Once the master creates the tables, the master sends a message to the non-master nodes to proceed. All nodes then call `mnesia:wait_for_tables/2` to wait for the tables to load.

3.4.3 SEARCH()

The core logic of SEARCH() remains largely the same as in the original PReach. Successor states are generated from the model using the same successor function, and each successor is in turn sent to a node to be processed. However, many of the data structures present in PReach are replaced with Mnesia tables in PReachDB. These tables are accessed using Mnesia trans-

[3] Erlang is untyped, but functions can be overloaded according to the number of parameters that the function has. Functions in external modules are referenced by the module name, the function name, and the number of arguments with the syntax `moduleName:functionName/argumentCount`.

actions. In Section 5.3, we discuss why this is bad for performance and propose possible next steps for improving data access performance.

For this project we focused more on correctness after system faults rather than on performance. The two most interesting problems we investigated were what considerations we must make if messages containing states to be explored are arbitrarily dropped and how to deal with node crashes. Changes to the implementation of SEARCH() mostly reflect how we dealt with these.

3.5 Message Acknowledgments

When processing a state, a compute node enumerates the successors of the state. Each successor state is sent to a compute node, which explores the new state. In PReach the destination for each state is based on a mapping from the hash of the state to one of the nodes. A node, in a sense, owns a set of states; it is responsible for exploring the set of states mapped to it.

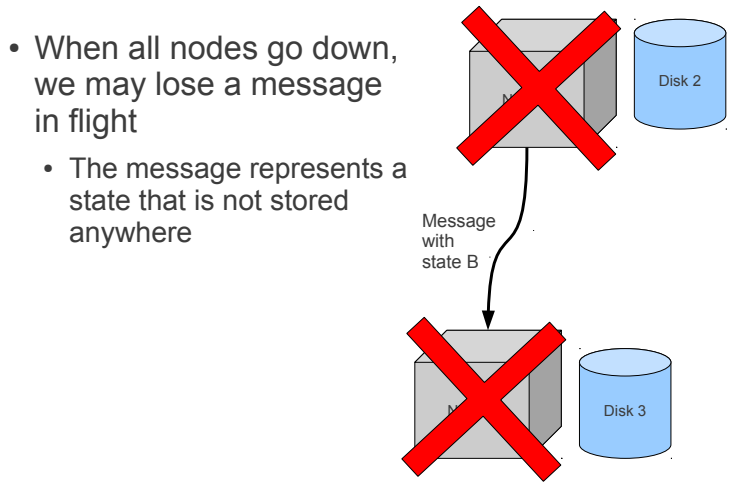

Figure 3.1: A message lost in flight.

If we allow for failures, a state could fail to reach its destination due to the destination node being down or due to a network partitioning. Not exploring the state means the reachable state space was incompletely explored;

this is a failure in correctness. The solution we use in PReachDB is to have a node send an acknowledgment (ACK) in response to receiving a state. A sending node stores the number of successors of the state in the Acknowledgment Table and waits for that many ACK messages in response. When a receiving node receives a state, it logs the state to disk in the Global Visited State Table and sends an ACK to the sending node. When the number of outstanding ACKs for a state is 0, we know that all its successors have been logged to disk.

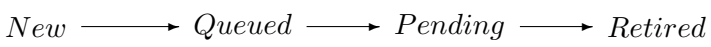

Figure 3.2: Lifecycle stages of a model state in PReachDB.

The lifecycle of a state throughout the computation is as follows. A state is first discovered to be part of the reachable set by being enumerated as a successor of another state in the reachable set. At this point it is "new" as shown in Figure 3.2; it is not stored to memory or to disk and it is not ACKed. The state is sent as an Erlang message to its owner node. The owner stores it in the global work queue and in the visited state table as unexplored. Once stored, the state is "queued." An ACK is sent back to the sending node. After the owner explores the state and sends out the successors of the state, the state is "pending." When ACKs for all of the successors have been received, the node removes the state from the work queue, and the state becomes "retired."

Life stage	Global State Table	Global Queue	ACKs outstanding
New	Not present	Not present	Not yet sent
Queued	Present	Present	Not yet sent
Pending	Present	Present	Yes
Retired	Present	Not present	No

Table 3.1: Presence of data for lifecycle stages.

In the case where not all successors of a state are acknowledged after a certain amount of time, that is the outstanding ACK counter is greater than 0, the situation can be remedied by resending all of the successors and resetting the outstanding ACK counter for that predecessor state to its

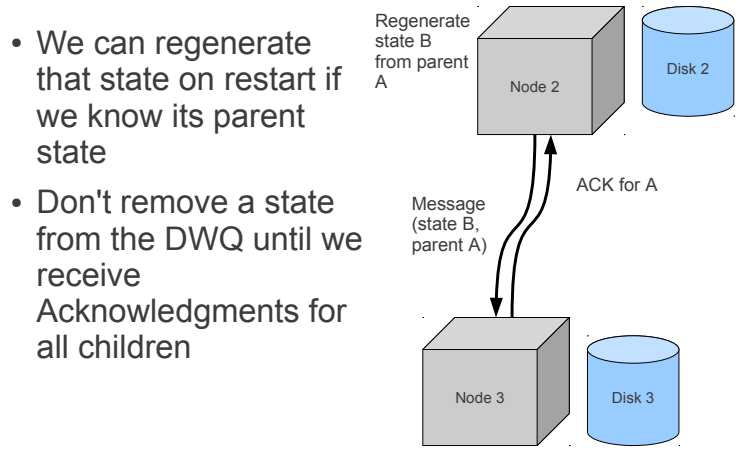

Figure 3.3: Regenerating the lost message.

number of successors. For example, a node receives a state it has already processed whose outstanding ACK counter is greater than 0. PReachDB does not keep track of which successors have not been ACKed, and there may have been messages lost in transit previously. The node resends all successors states. To avoid double counting acknowledgments, the node resets the counter to the number of successors. The nodes that receive the successors send ACKs regardless of whether they have received the states before or not. Furthermore, the states are sent with sequence and epoch numbers that are subsequently included in the acknowledgments as described below in Section 3.5.1. When the outstanding ACK count reaches 0 for the predecessor state, the predecessor state is removed from the Global Work Queue because the node does not ever have to resend that state's successors states.

3.5.1 Sequence Numbers

Resending the successors generates a new problem. It is possible for a message from the original dispatch to be delayed in the system long enough that the original ACK is not received until after the successors states have been resent and the outstanding ACK counter reset. These delayed ACKs could run down the counter to 0, triggering the removal of the predecessor state

from the disk work queue, before all successors states have been logged to disk. Should a successor state, falsely counted off, not be received or logged to disk, we would not know that we had missed it.

In PReachDB we solve this problem by keeping track of an increasing sequence number associated with each dispatch. When successors states are dispatched, they are sent with a sequence number incremented from that of the previous dispatch and with the epoch number of the sending node. The current sequence number is stored in the Acknowledgment Table along with the outstanding ACK count. When an ACK is received, if its sequence number is less than the current or if the epoch number does not match, the ACK is ignored; if its numbers match, the outstanding ACK counter is decremented. On the last decrement, the state is deleted from the Global Work Queue and from the Acknowledgment Table.

The full message and acknowledgment protocol also includes the epoch number of the sender. If the epoch number of an ACK does not match the epoch number the sender is currently on, the ACK is discarded.

Figures 3.4 to 3.9 illustrate the motivation for using sequence numbers in the acknowledgment protocol. Consider a system with three compute nodes with point-to-point communication. Node 1 is processing state A, and generates states B and C from $Successors(A)$. B is sent to Node 2 and C is sent to Node 3. However the messages are slow, and Node 1 begins reprocessing A before any ACKs have been received. Node 1 resends B and C to Nodes 2 and 3 and resets its ACK counter to 2. While the messages are taking a long time to reach Node 2, Node 3 receives state C twice, sending an ACK for each. Node 1 received 2 ACKs for A and decrements its counter twice. Since the counter is 0, A is removed from the Global Work Queue. Meanwhile, Node 2 has not yet received B and then suddenly fails. B is never processed and the system misses some of the reachable state space.

3.6 Failure Handling

A node failure may be caused by anything, but the result is that the PReachDB process running on that node is ended, forcibly or not. In PreachDB we address two types of failures on a node, where either the node can recover by restarting the node and/or the PReachDB process, or where the node is not recoverable and is essentially dead. We consider two failure scenarios.

One scenario is that all N nodes fail at the same time with their data intact. This scenario might happen if the user is running on a scheduled cluster and his reservation has ended. The user must halt computation, but

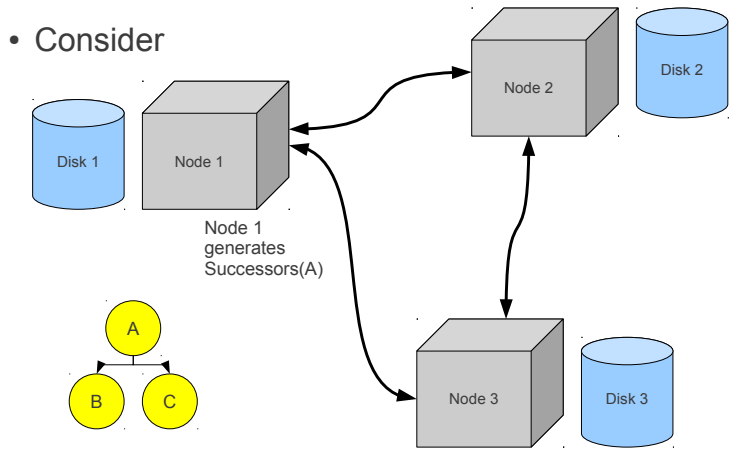

Figure 3.4: Sequence Numbers: Example system configuration.

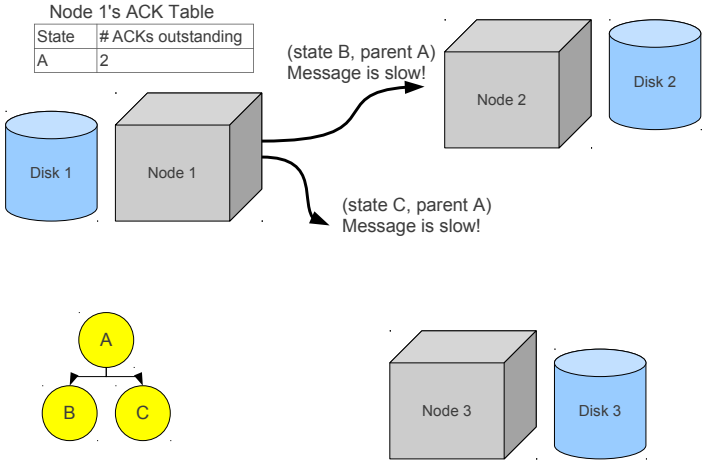

Figure 3.5: Sequence Numbers: Slow messages.

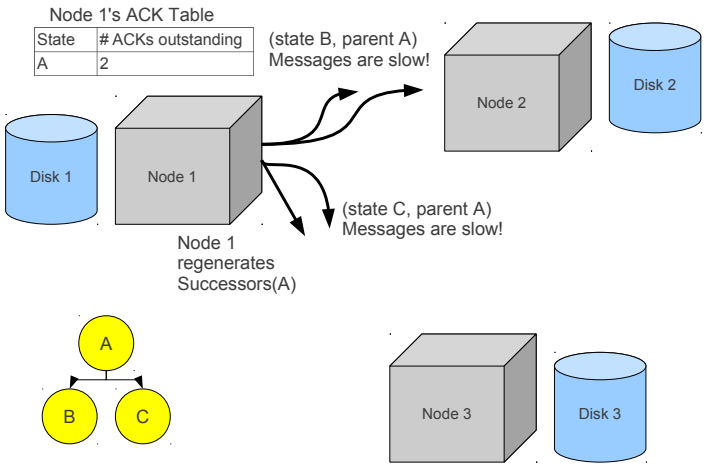

Figure 3.6: Sequence Numbers: Resending the successors.

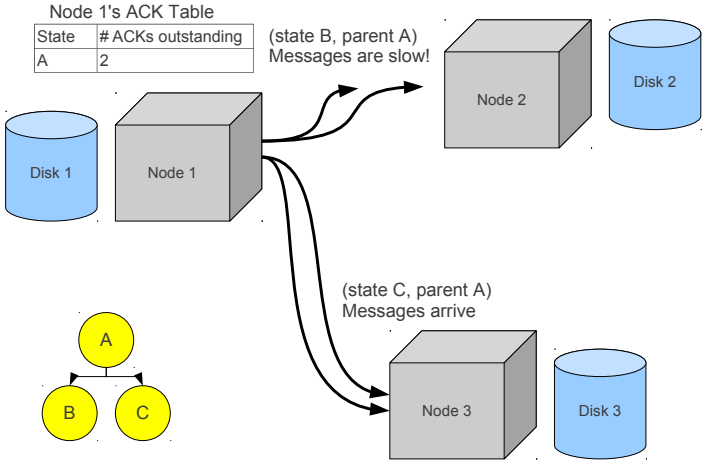

Figure 3.7: Sequence Numbers: C arrives at Node 3 twice.

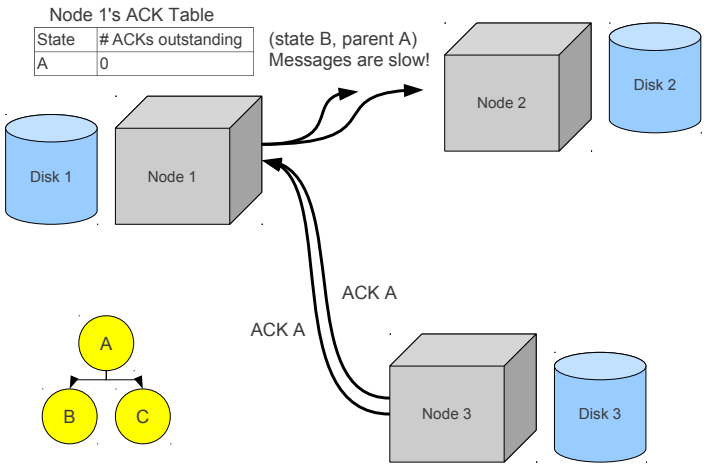

Figure 3.8: Sequence Numbers: Node 3 acknowledges predecessor A twice.

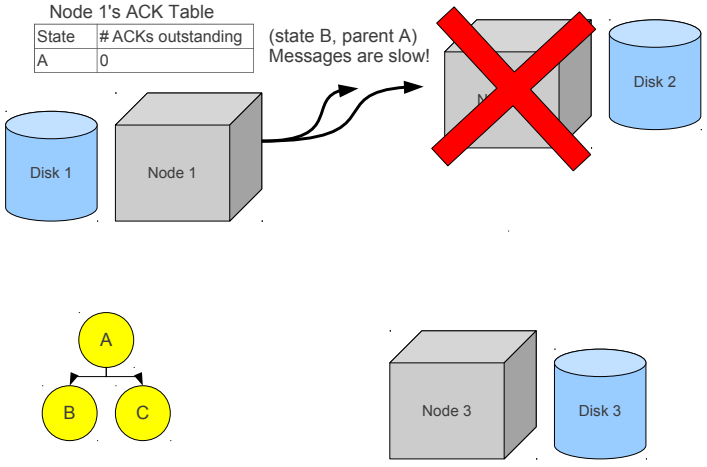

Figure 3.9: Sequence Numbers: Node 2 fails and B is missed.

the progress is not lost.

Another scenario is that a node has a non-recoverable failure. For a non-recoverable failure we look at two options: either continuing the computation with one fewer node and making the other nodes pick up the dead node's work, or inserting a new node into the node pool to replace the dead node. Both options require enough redundancy of data for the remaining nodes to access to the dead node's work. We discuss the implementation of the data redundancy and strategies for dealing with node failure in the rest of this section.

3.6.1 Table Fragmentation and Replication

We use data redundancy across nodes to prevent data loss in the event that a node is disabled. There are many means of disabling a node, such as network disconnect, hardware failure, and loss of power. We will not distinguish between these in the current work, and consider only the case where a single node or where all nodes become completely unresponsive. Handling more general failure scenarios such as multiple nodes failing simultaneously or in sequence is a topic for future work.

One scheme for storing redundant data is create a single master database node through which all database writes and reads are done. The master database is passively replicated by slave database nodes. In the event of a failure of the master, a slave is promoted to master. We did not utilize this scheme for this project as a matter of choice based on the design of PReach and the existing standard libraries for Erlang. This master database would create a bottleneck, as it would have to handle all hash table and work queue transactions. On standard hardware we would quickly saturate the master database's throughput.

Another redundancy scheme is to store all global tables in their entirety on every node. This is expensive in storage and in effort required to coordinate. Decreasing the number of replicas of the global tables, R, decreases the storage space required per state during the computation. We would like this number R to be high enough to provide reasonable reliability without unnecessarily taking up space. If we ignore the data location for a moment and consider only how many replicas we need, we set $R = 3$ [31] for most cases but allow the user to override. For our experimental setup, we use $R = 2$ since we are concerned with only one node failure.

We like keeping the data on the nodes, as was done in PReach. But rather than storing entire global tables on every node, we can utilize Mnesia's table fragmentation feature to partition the global tables into chunks.

Mnesia maps each data item in the data table onto a fragment; for example, the default mapping is based on the hash of the data item. A fragment of a table can be replicated through Mnesia onto any number of nodes in the node pool. If we set the number of fragments per table to N, the number of nodes, then we can have one fragment per node (when there are no replicas) to evenly distribute the data across the node pool, assuming uniform hash distribution. Ideally the fragment residing locally to the node stores the data for the states that are owned by the node; local data accesses improve read performances especially.

When $R = 3$ and the number of fragments is set to N, the number of nodes, a node stores three fragments (per table) locally. Should a node go down permanently, we lose one replica of each fragment that was stored on that node, but two other replicas of those fragments exist on other nodes. We can repair the loss of redundancy by making a copy of each affected fragment onto a new node from the existent replicas. In the common case where $R < N$, the copying of fragments is less work than making a complete copy of the entire table.

Tables created with `fragmentron:create_table/2` will automatically balance the location of table fragments and create new fragment replicas as nodes join or leave the node pool. In Section 3.4.2, we discussed values for `frag_properties`. `n_fragments` is set to N and `n_disc_copies` is set to R.

3.6.2 Hot Spare and Data Migration

Figures 3.10, 3.11, and 3.12 show how a hot spare node can be introduced to a system to replace a dead node. In the figures, Nodes 1, 2, and 3 store a database table. The table has 3 fragments, colored blue, pink, and green, and each node stores two different fragments. Node 3 has failed and its replicas of the pink and green table fragments are lost. There are still active replicas of those fragments on Nodes 1 and 2. We introduce Node 4 to the system to replace Node 3. The system is paused while Node 4 is added to the Mnesia node pool and pink and green fragment replicas are created on Node 4. The system updates its message destinations to include Node 4 and drops Node 3. The system then resumes computation.

We created messages to pause and resume all nodes in the system. When a hot spare is started, it sends these message in its startup sequence.

On receiving a resume message, an existing node updates its fragment owner map. This mapping of fragment to owning node is used for looking up the destination for every generated successor state. The fragment owner mapping is a small optimization when choosing for each state which node to

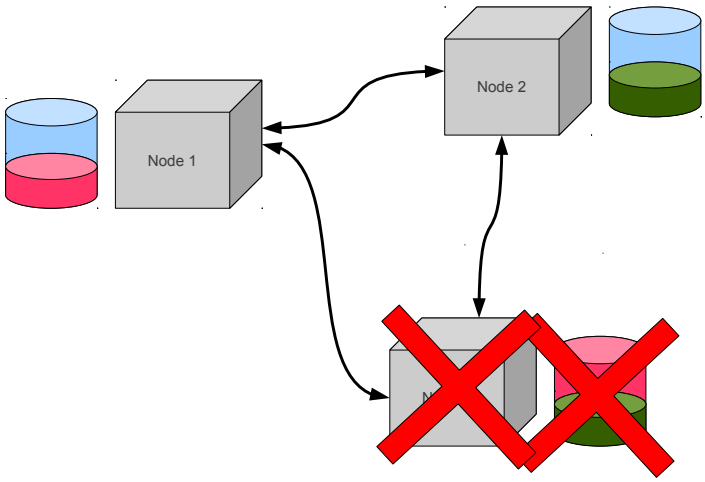

Figure 3.10: Three node replicating three fragments of a table. Node 3 has failed.

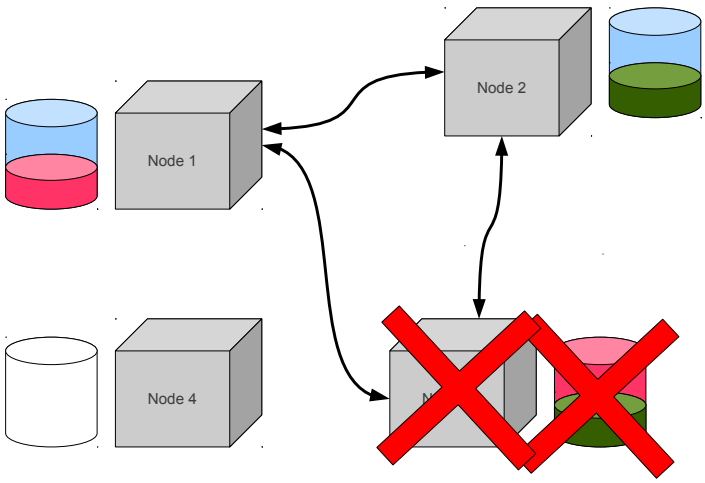

Figure 3.11: Hot spare Node 4 is introduced to replace Node 3.

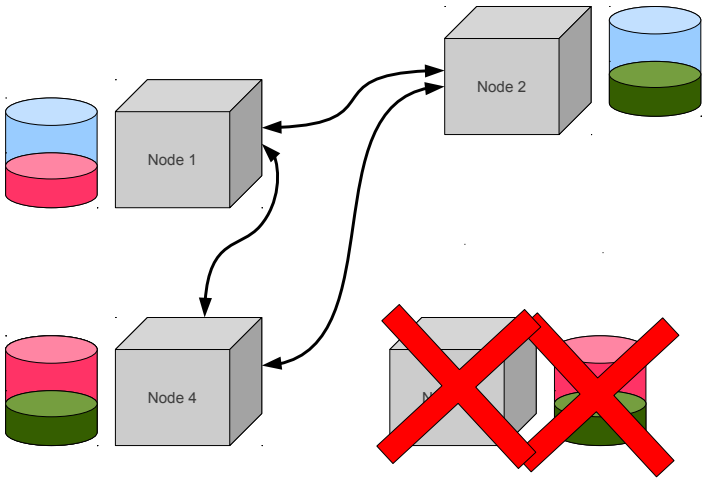

Figure 3.12: Fragments are copied to Node 4 and Node 4 joins the computation.

send it to. We could choose randomly each time, and we could ask Mnesia from which nodes a data item can be written or read and then send it to one of those. But if we know which fragment a data item would be written to, we can use the fragment owner map to send the state to a node which stores that fragment locally. Which node stores which fragment is known after table creation, and we create the mapping then. After the node pool changes, the mapping must be updated.

One caveat is that if there are several choices of nodes for location to read/write, Mnesia will pick a node based on network latency. This means a node with slightly higher network latency would never normally be chosen, even if it is starved for work. Adding a work queue aware load balancer to the system is one direction for future work.

The mapping is created using Algorithm 2.

In selecting an owner, we will choose from among hosts where the fragment is resident and will prefer hosts that have been passed over for selection previously.

Algorithm 2 Fragment owner mapping creation

```
 1: m : map of fragment to host
 2: seenBefore : map of host to counter
 3: hosts_f : list of hosts
 4:
 5: procedure OWNERS( )
 6:     m := {}
 7:     seenBefore := {}
 8:
 9:     for each fragment f do
10:         hosts_f = mnesia : table_info(f, where_to_write)
11:         if hosts_f = [] then
12:             halt()
13:         end if
14:         sort hosts_f on host h by value of seenBefore[h] descending
15:         m[f] = hd(hosts_f)
16:         for each host h in tl(hosts_f) do
17:             insert seenBefore[h] if needed
18:             Increment(seenBefore[h])
19:         end for
20:     end for
21:     return m
22:
23: end procedure
```

3.6.3 Work Migration

In PReach, a node's work queue, represented as an Erlang list, is maintained in memory as the first argument to the recursive function `reach`, which implements the loop body of SEARCH(). PreachDB maintains this same structure. The states yet to be explored are also stored in the Global Work Queue, along with states we may have to re-explore if a crash were to occur. The Global Work Queue stored the states but does not maintain order. The states in the Global Work Queue are a superset of the states in any node's in-memory work queue.

If a crash occurs, the in-memory list is lost, but the Global Work Queue is preserved. On a restart, when a node starts up and has an empty in-memory queue, it will ask Mnesia to read a selection of states from the Global Work Queue. PreachDB grabs a random selection of keys in the Global Work Queue from the `mnesia:all_keys()` function. Which states are returned does not matter, because all states from the Global Work Queue will eventually be fetched or otherwise explored. States are removed from the Global Work Queue when the number of acknowledgments outstanding is zero.

In fact, whenever a node reaches an in-memory work queue size of zero, it will retrieve states from the Global Work Queue in the same fashion. All states from all nodes are inserted into the Global Work Queue, and any node could read and process (check invariants on and enumerate the successors of) any state. When retrieving states that belong to a remote node, there is an inefficiency in that the node doing the exploring does not read and write states that are locally resident on the disk. Each Mnesia transaction will go over the network. However this does allow work stealing, by which a node that is making progress quickly may run its local work queue to zero and then retrieve states owned by a slower node. This can help reduce the bogging down of one particular node. Reading from the Global Work Queue when the in-memory work queue is empty essentially provides the load balancing features of PReach through Mnesia without additional code.

Another tactic implemented in PReachDB to try to alleviate bogging down is a limit on the in-memory queue size. PReachDB nodes will store up to 5,000 states in their work queues before spilling them off. When a state is to be added to a work queue that is full, it is written to the Global Work Queue as normal, but it is not written to the in-memory queue of the node. This data is not lost because it remains in the Global Work Queue. Since the computation does not end until the Global Work Queue is empty, and since a node with an empty in-memory work queue will fetch states from

the Global Work Queue, that spilled off state will eventually be processed.

3.7 Modified Algorithm

We present in Algorithms 3 and 4 the modified DEMC algorithm including the additional message passing used in PReachDB.

The definition of *Owner(s)* differs from that described in 2.3.1. In PReachDB, the owning node for a state can change over the course of the computation due to a node failing and its states being adopted by other nodes. Thus, we cannot use PReach's *static* mapping from states to nodes that was described in Section 2.3.1. Instead, states are *dynamically* mapped to nodes based on the fragment number for the state. See Algorithm 2.

3.7.1 Termination Detection

The termination detection algorithm for PReachDB differs from PReach's in that it uses just the global work queue size to determine if there is work outstanding rather than the message counters and queue size of each node.

Adding a node to the queue increases the queue size. A state is not removed from the global work queue unless each of its successors has either been added to the global work queue or been processed already. A queue size of zero means all start states and all states reachable from the start states have been processed.

$$|Q| = 0 \tag{3.1}$$

where Q is the global work queue.

3.8 Summary

PReachDB builds on PReach to add fault tolerance for node failures. It maintains the same general structure of PReach while changing the data structure accesses to be mostly database accesses. Saving data to disk and replicating the data on multiple nodes allows us to recover the system while it stays online. In addition to database accesses, the main changes are the addition of message acknowledgments, the addition of sequence and epoch numbers to messages, the modification of the termination detection algorithm, and the addition of the procedure to add a new node to the node pool.

Algorithm 3 PReachDB modified DEMC algorithm: per-node process

```
1: pGT : persistent globally accessible set of states seen so far
2: pGWQ : persistent globally accessible set of states yet to process
3: pEpoch : persistent per-node epoch counter
4: WQ : queue of states to process locally
5: ACK : map of states to (seq, acknowledgments outstanding) tuples
6:
7: procedure SEARCH( )
8:     ACK := {}
9:     WQ := []
10:    initialize pEpoch if needed
11:    Increment(pEpoch)
12:    if I am the root then
13:        initialize pGT and pGWQ if needed
14:    end if
15:    wait for Mnesia ready
16:    barrier()
17:    if I am the root then
18:        for each s ∈ initialStates do
19:            Send s to Owner(s)
20:        end for
21:    end if
22:    while ¬Terminated() do
23:        GETSTATES()
24:        if WQ ≠ [] then
25:            s := Pop(WQ)
26:            if s ∉ pGWQ then
27:                seq := 1 + (hd(ACK[s]) or 0)
28:                ACK[s] := (seq, |Successors(s)|)
29:                Check(s)      ▷ verify s satisfies specified safety properties
30:                for all s' ∈ Successors(s) do
31:                    s'_c := Canonicalize(s')
32:                    Send state (s'_c, s, seq, pEpoch, Self()) to Owner(s'_c)
33:                end for
34:            end if
35:        end if
36:    end while
37: end procedure
```

Algorithm 4 PReachDB modified DEMC algorithm, continued

```
1: procedure GetStates( )
2:     while there is an incoming message do
3:         if Receive ack (predecessor, seq, epoch) then
4:             if epoch = pEpoch and hd(ACK[predecessor]) = seq then
5:                 acks := Decrement(tl(ACK[predecessor]))
6:                 if acks <= 0 then
7:                     Remove(predecessor, pGWQ)
8:                     Remove(predecessor, ACK)
9:                 end if
10:            end if
11:        end if
12:        if Receive state (s, predecessor, seq, epoch, sender) then
13:            if s ∉ pT then
14:                Insert(s, pT)
15:                Insert(s, pGWQ)
16:                if |WQ| < spill threshold then
17:                    Append(s, WQ)
18:                end if
19:            end if
20:            Send ack (predecessor, seq, epoch) to sender
21:        end if
22:        if Timeout and pGWQ ≠ [] then
23:            Append(RandomSubset(pGWQ), WQ)
24:        end if
25:    end while
26: end procedure
```

The main benefit of PReachDB is that by using the Mnesia DBMS we get fault recovery with almost no additional code. We merely need to check if the database already exists on system start. Another nice side effect of using a database is that the termination detection algorithm is now even simpler than in PReach.

Chapter 4

Results and Performance

The results of this project are the proof-of-concept demostrations of the following scenarios.

1. Terminate all nodes and restart without significant loss of progress.
2. Terminate one node and have the rest of the nodes run to completion utilizing data replication.
3. Terminate one node, replace it with a hot spare, and have the hot spare and the rest of the nodes run to completion.

We also present a performance comparison between PReachDB and PReach.

4.1 Results

We present here the experimental setup and the results from four test runs demonstrating different features added in PReachDB.

4.1.1 Setup

The machines used for testing were of these two configurations running Linux.

Processor	Memory
Intel® Core™2 Duo CPU E6550 @ 2.33GHz	4GB
Intel® Xeon® CPU 5160 @ 3.00GHz	6GB

The machines were connected over the UBC Computer Science department internal network, and average ping time was less than 0.1 ms. The Mnesia database files were stored locally per machine. The code path was to a shared server filesystem.

Each of the following tests was performed on the model file `n5_peterson_modified.erl`. Fragment replication, where applicable, was $R = 2$.

4.1.2 System-Wide Restart

The purpose of this test was to demonstrate that the Mnesia disk tables retain the data over a PReachDB restart. When all nodes are forced to exit, we can restart the entire system with all progress retained that was committed prior to the crash.

For this test we ran PReachDB with $N = 3$ nodes and with table fragment replication enabled. We let PReachDB run through the initialization sequence and begin processing states from the model. We then ran a shell script similar to the `ptest` start up script. The script connected to each machine via **ssh** and ran `kill -9` on all Erlang processes running under our account. We then ran `ptest` to restart all of the nodes.

We verified that the total number of states in the Mnesia visited state table on completion of the of the program (visited in aggregate over all processing sessions) matched that of our control run using PReach. The program output the number of states visited in the most recent session, and it was less than the number of states contained in the Mnesia visited states table. This shows that PReachDB did not visit all states in the model during the last session. PReachDB retained the progress made in the previous session.

4.1.3 Sequence Number on Acknowledgments

The purpose of this test was to demostrate the ACK counter decrement issue described in Section 3.5.1.

We ran PReachDB (with $N = 3$ nodes and with table fragmentation) uninterrupted with and without the logic for sequence numbers on acknowledgments. We included a debug conditional to print a debug statement if an ACK counter in the ACK table is decremented below 0. Without the sequence number implemented, we observed the debug print, and with the sequence number implemented, we did not.

4.1.4 Single Node Termination

The purpose of this test was to demonstrate that with distributed replication enabled we could complete the model exploration with a single node completely removed from the node pool part way through. With $N = 3$ nodes and $R = 2$ fragment replication, the data on the node that is removed is still held on the other two nodes. The remaining two nodes should provide the necessary data to complete the computation without pausing or

restarting the system, as the remaining nodes combine have the same data as the removed node.

We ran PReachDB on three nodes, letting the initialization complete, and then ran `kill -9` on all Erlang processes on a single machine. We observed the remaining two nodes continue to output debug messages during exploration. The remaining two nodes processed the entire reachable state space and output the same number of unique explored states as the control run without terminating a node.

4.1.5 Hot Spare

The purpose of this test was to demonstrate that when one node dies, computation can continue after a hot spare replaces the dead node. After the hot spare adds itself to the node pool with **fragmentron**, Mnesia copies over the fragments to the new node in the background. The hot spare waits for the Mnesia tables to be ready, updates the other nodes to send states to it, and then calls `reach`.

We ran PReachDB on three nodes and used `kill -9` on one node, similar to the previous test. We then sent a user command to pause the all nodes. The two remaining nodes responded and waited. We ran the script to add a spare node from a different machine, and when it was ready ran the user command to resume all nodes. Although the spare had not participated in the work before, it had access to the same data as the dead node. The three nodes were able to complete and reported the same number of unique explored states as the control.

4.2 Performance

A strong benefit of using Mnesia is how well it provides recovery from system faults. Recovery time for most operations appears to the user as practically instant. Both restarting a node with a disk table and inserting a hot spare node into the Mnesia node pool for fragment replication do not negatively impact the other nodes. Mnesia copies table fragments in the background to the hot spare. This does have the possible effect of overloading the Mnesia system.

Unfortunately, in the normal usage of PReachDB, Mnesia reported frequently that it was overloaded. These messages decreased when replication was reduced to $R = 1$, but once any node became bogged down the system became stuck.

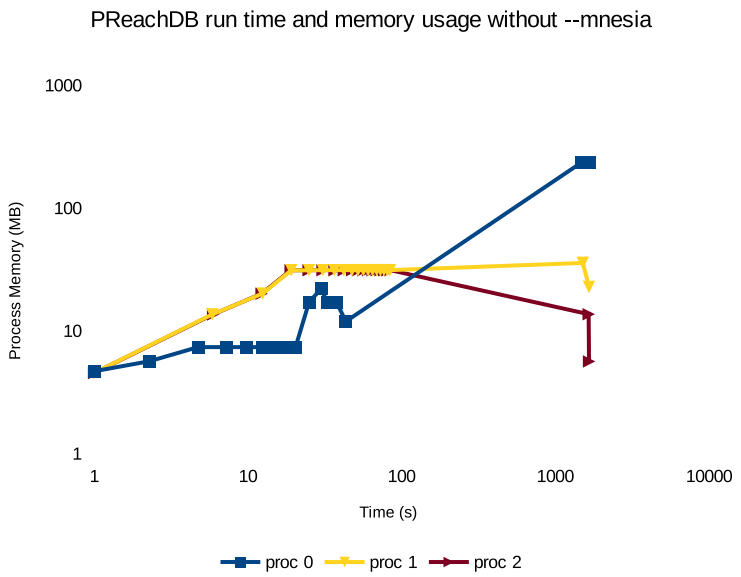

Figure 4.1: Without Mnesia, time on logarithmic scale: There is a long period with very little progress as process 0 catches up.

With Mnesia transactions, writes to remote nodes (required for replication) over the network must complete in full before a node can continue processing. We use transactions for correctness because Mnesia cannot guarantee ACID properties with dirty database writes. We relax this later in this section to test the performance with dirty operations, which return as soon as the operation completes for at least one node in the Mnesia node pool.

4.2.1 Performance without Replication

Running PReachDB without the –mnesia flag causes it to run with the PReach data structures implemented at the time of code forking. Each node runs with a local visited state table and a local state queue. The visited state table is by default a bloom filter based on [1] with fixed capacity $N = 40,000,000$ and error probability $E = 0.000001$. The state queue is maintained as a list of states sent as argument to the recursive function `reach`. The acknowledgment table is not used.

Figure 4.1 shows the time in seconds on a logarithmic scale per 10,000 states visited and the Erlang process memory in MB (log scale) of a typical

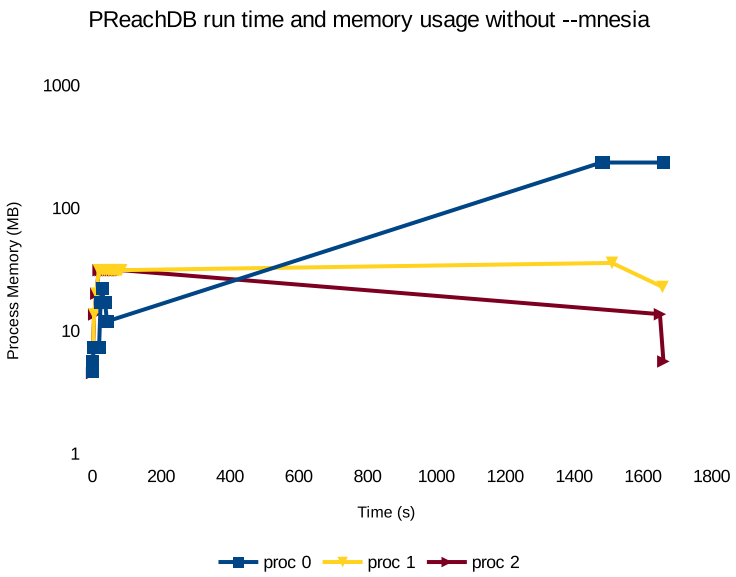

Figure 4.2: Without Mnesia.

run. Figure 4.2 shows the same information with time on a non-logarithmic scale. On the non-logarithmic time scale, the horizontal distance between successive data points is the time to process 10,000 states. Longer distances mean a slower rate. There is a long gap on process 0 between 130,000 and 140,000 states visited, and between those data points, process memory increases greatly. State queue size also jumps by an order of magnitude between those data points, from 2,308 to 34,354 states. The other processes wait for process 0 to catch up, and begin processing new states received from process 0 as soon as they are ready.

The slowdown may an artifact of the older PReach code base, as current PReach does not have this behavior. The load balancing added to PReach addresses the bogging down of a particular node, which in turn helps the whole system continue to make progress.

Figure 4.3 shows the time and process memory of PReachDB with the –mnesia flag enabled. Progress is slow but steady. When –mnesia is enabled, the size of the in-memory state queue is limited to a fixed maximum size of 5,000 states. When the in-memory state queue reaches that length, additional states are written to the global work queue but not to the in-memory queue. This queue length limitation plus work queue stealing prevents any one process from becoming bogged down and holding up the other processes.

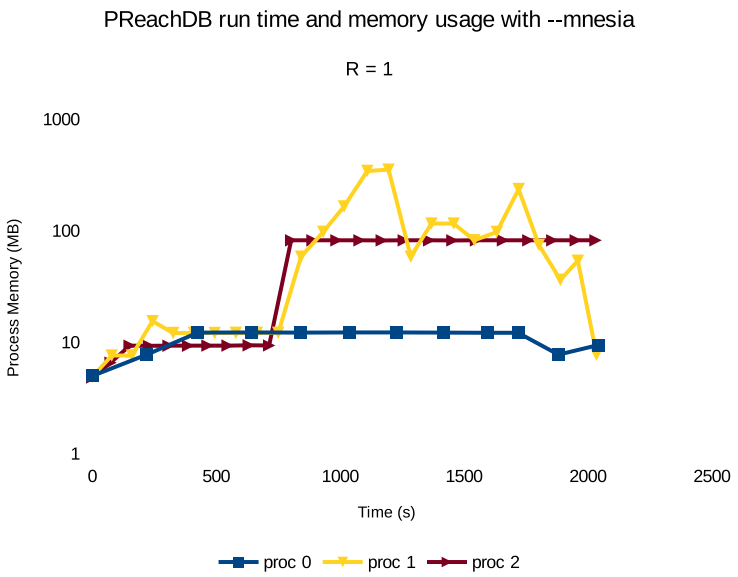

Figure 4.3: With Mnesia: Progress is slow but steady.

4.2.2 In-memory Queue Spilling

One possible reason for PReachDB's slow performance, as suggested by one of the authors of PReach, is Erlang's inefficient list operations when the number of states in the per-node work queue is large. The Erlang documentation warns that improperly implemented list operations will result in a $O(n^2)$ operation due to repeated list copying [12]. To test if performance is impacted by this issue, we implemented queue spilling for the in-memory work queue. This was previously discussed in Section 3.6.3.

Without limiting the in-memory work queue size, PReachDB with –mnesia does not slow down the same as PReach when the in-memory work queue of any node grows large. Figure 4.4 shows that the run time without limiting the queue size is similar to the run time when it is limited. This implies that the queue size limitation has no effect on performance. The list operations may be implemented as suggested in [12]. It is also possible that the Mnesia write/read bottleneck masks a slow down caused by the in-memory work queue size. If this effect is happening, then PReach would demonstrate the slow down while PReachDB would not.

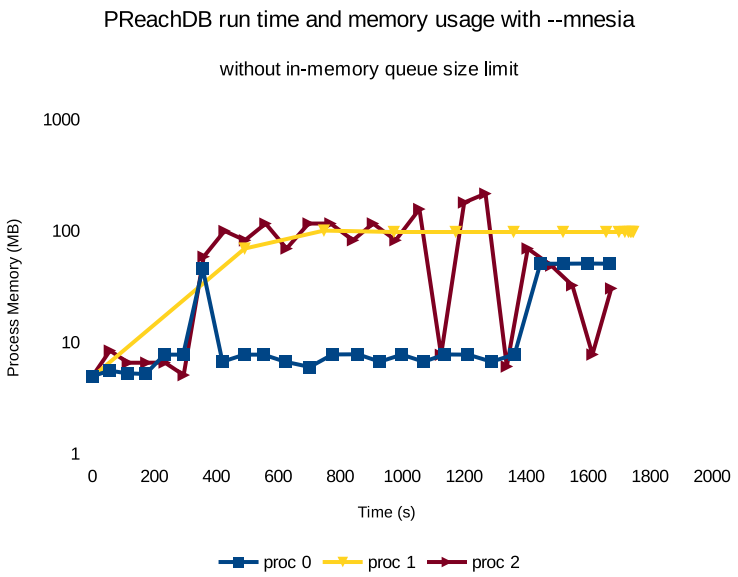

Figure 4.4: With Mnesia and no limit on the in-memory queue size. Progress is similar to with limiting enabled.

4.2.3 Replication Factor

There does not seem to be significant performance impact when running with –mnesia between a replication factor of $R = 1$ and $R = 2$. The experiment we ran with $R = 2$ (Figure 4.5) finished sooner than for $R = 1$ (Figure 4.3).

4.2.4 Work Queue Replication

We tried running PReachDB with Mnesia-backed non-global work queues, where the option `local_content` on the work queue is set to true for each node. This inherently disallows work stealing between nodes.

This exacerbated the bogging down problem. The fast nodes repeatedly sent states they had already sent. Since states are not removed from the work queue until all acknowledgments are received, the fast nodes would revisit unacknowledged states in its work queue before the slow node could send its initial acknowledgment. With no work stealing, the slow nodes never caught up and the reachability computation did not complete. The system got stuck in livelock. By the time the slow node acknowledged a state from the fast node, the fast node had resent that state with a new

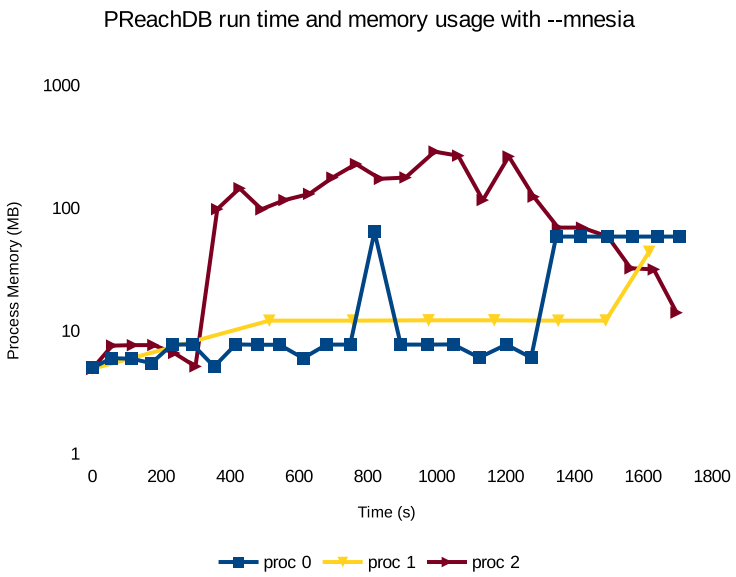

Figure 4.5: $R = 2$

sequence number, and so the fast node discarded the acknowledgment.

4.2.5 Dirty Table Operations

In this section we compare the performance of PReachDB using dirty database table reads and writes to the visited state table and the global work queue versus using synchronous transactions. According to the Mnesia documentation, a dirty operation will not wait for changes to be fully replicated to all nodes, but will instead return as soon as one node completes the operation. If the table fragment is resident on the node, this operation should take less time than waiting for all remote nodes to acknowledge the replication. "This still involves logging, replication and subscriptions, but there is no locking, local transaction storage, or commit protocols involved. Checkpoint retainers and indices are updated, but they will be updated dirty. ... A dirty operation does, however, guarantee a certain level of consistency and it is not possible for the dirty operations to return garbled records. ... However, it must be noted that it is possible for the database to be left in an inconsistent state if dirty operations are used to update it." [13]

The dirty operations we perform are:

1. Read from the visited state table if a state exists and if not insert the

state into the visited state table and the global work queue.

2. Read from the global work queue to determine if a state being processed has already been retired and is no longer in the global work queue.

3. Delete the state from the global work queue when its acknowledgment table counter reaches 0 outstanding ACKs.

The main concern here is that with work stealing, it is possible and probable for two nodes to explore the same state and read and write locally without seeing the dirty operations of the other node. In the case of PReachDB, this should not affect the correctness of the reachability computation, but it may do redundant work. The correctness claim is justified as follows.

One problem with this approach is termination detection. An incorrect read of 0 states left in the global work queue could cause the nodes to attempt early termination. We fix this by doing a synchronous read of the global work queue size if the dirty read returns 0 states left. The synchronous read returns the actual number.

Figure 4.6 shows that the performance is similar to that with synchronous transactions. Synchronous transactions adds a 21% overhead over dirty operations as implemented, with run times of 1435s for dirty operations and 1740s for synchronous transactions.

We also tried dirty operations with Mnesia in-memory only tables, shown in 4.7. This did not significantly affect performance versus saving to disk.

4.3 Summary

We tested PReachDB under a selection of simple fault scenarios. It successfully recovered from temporary system-wide node failure and from permanent single node failures with and without adding a replacement node to the node pool. However, performance of PReachDB is currently much worse than that of PReach; thus it is not yet suitable for practical use.

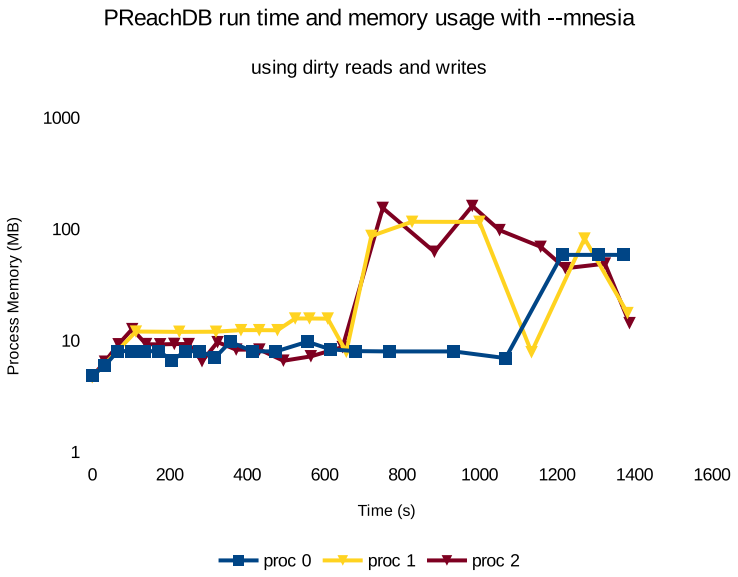

Figure 4.6: With Mnesia and dirty reads and writes to the visited state table and the global work queue.

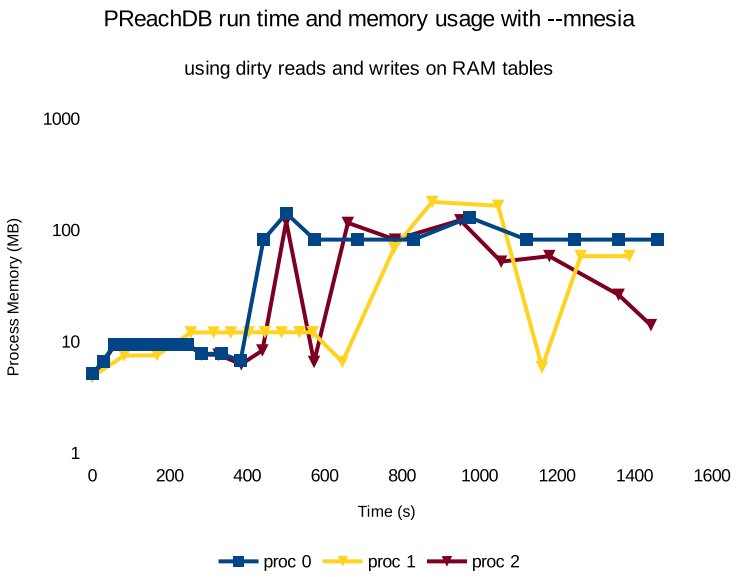

Figure 4.7: With Mnesia and dirty reads and writes to the visited state table and the global work queue. Both tables were held in memory and not on disk by Mnesia.

Chapter 5

Discussion and Future Work

Distributed explicit-state model checking attempts to provide an edge over the state space explosion problem by distributing work across multiple compute nodes. As with other distributed computation, it is brittle in the face of failures. It is a long-running, memory-intensive, communication-intensive distributed computation. Some possible failures are:

1. A node has a system failure.
2. A node runs out of memory.
3. In the case of persisted data, a node has a disk failure.
4. There is a network partition.

Ideally none of these happen while trying to verify a model with a large state space. However, failures do occur in practice and there are measures that can be taken to save the progress of the system. Keeping track of which states are retired and which states are queued or pending in a durable way is required for a solution. To recover from a complete node loss also requires data redundancy.

In PReachDB, we implement redundancy and persistency through Mnesia. Mnesia provides distributed database tables with configuration to enable replication, table fragmentation, and disk storage. We are able to demonstrate recovery from a single node crashing in three ways: by restarting the node, by inserting a hot spare new node, and by letting the remaining nodes finish the computation without a replacement node.

We do not implement explicit checkpointing, but rather use the default transaction management through Mnesia, which does checkpointing within the Mnesia subsystem. One option for performance improvement would be to implement explicit checkpointing while doing fewer Mnesia operations.

We do not address Byzantine failures, which can occur in PReachDB in a few ways. For instance, a node could make an error while verifying an invariant predicate on a reachable state. A node could make an error in

applying $Successors(s)$ and potentially miss a large chunk of the reachable state space. Or in the case of dirty database operations, an error occurring during a dirty write could leave a record in an inconsistent state.

We do however address the problems of how to handle messages being dropped and how to handle messages when a node has been restarted. The use of acknowledgments also simplies the termination detection equation.

As discussed in Section 4.2.1, the performance comparison between PReach and PReachDB is affected by an artifact of the older code base. Current PReach with load balancing would finish at least an order of magnitude faster than presented in 4.2.1. Given that, the overhead of enabling distributed tables in Mnesia is very high even without replication. Limiting the in-memory work queue size did not seem to affect the performance. Using dirty database operations did marginally improve performance, but based on [13] the expected improvement from using dirty operations should have been higher. Likewise the impact of disk versus in-memory Mnesia tables was lower than expected, and suggests that using Mnesia at all may be the problem.

The following sections describe some avenues for improvements and future work.

5.1 Implementation Improvements

The implementation was written assuming that the root node is always reachable. The root node is used as the host for generating the list of active nodes when a hot spare is added. Even with this limitation, PReachDB is more tolerant to node failures than PReach, which will fail when any single node fails.

The implementation also assumes that the number of table fragments equals the number of compute nodes at table creation. When the program is started, the tables are created with a fixed number of fragments, and this number is assumed to stay constant until completion. It would make the system more robust to allow the number of table fragments to grow or shrink over time as the number of compute nodes changes.

5.2 Automated Recovery

Recovering from system failures in PReachDB is a manual process and requires the user to run shell scripts. There are user commands to pause or resume the system, restart a node, or insert a new node into the node pool

as a spare. It would be helpful for usage of the system and for running experiments to automate the recovery process. This would involve writing event handlers to automatically trigger recovery using Erlang and Mnesia events.

5.3 Performance

There are several places where the performance of PReachDB could be improved. There is already work in the literature that could be leveraged in most of these cases. Further study of the Mnesia system and best practices when dealing with it would also be beneficial.

5.3.1 Load Balancing

During normal usage of PReachDB, Mnesia reports that it is overloaded several times. The node from which the warning messages originate also tends to get bogged down and progresses very slowly. Adding a work queue aware load balancer to the system is one direction for future work which has been met with success in other model checkers. [5, 25]

5.3.2 Messages

PReachDB is structured to send at least as many messages as there are reachable state transitions. The cost of communication is particularly important because it is higher for PReachDB than for PReach. PReachDB is designed to run on separate machines due to disk IO, while PReach has no disk IO and can run multiple nodes on one machine with multithreading. Given that [5] found multiple factor speedup by batching messages into groups of 100 or 1000, applying the same method to PReachDB should yield sizable performance improvement.

5.3.3 Memory and Disk Usage

Batching can also be applied to writes to Mnesia. The implementation writes states to the Global Visited State Table and the Global Work Queue as it explores, but it would be possible to decouple exploration from saving by using different threads for the two tasks. This would allow the explorer to run ahead. In the event of a failure any work done by the explorer that was not saved would need to be repeated. Batching writes may fit more

naturally into PReachDB than explicit checkpointing, since we are already doing transactions with single writes.

We could decrease both memory and disk footprint by using shorter keys to the Mnesia tables. The key is currently the full state descriptor Erlang object.

Another method we could try is to use an in-memory Erlang `ets` cache as the first line of defense before reading from Mnesia. If the cache does have the item, then we do not need to do the lookup through Mnesia and we can avoid resending successors states which have recently been sent.

We could potentially try a different approach using Mnesia as a distributed log of operations done on in-memory hash tables. There may be a nice way to use the Murphi in-memory hash table, which is tuned for this problem. Writes to the log can be batched. Any operations not written to the log and lost when a node crashes can be rediscovered by replaying the log and continuing from there.

5.4 Conclusion

PReachDB adds the fault tolerance capabilities of redundancy and persistence to the PReach distributed explicit-state model checker. It does this through use of the Mnesia distributed database system for Erlang. This project provides demonstration that PReachDB can recover from faults both off- and on-line. The overhead of the proof-of-concept implementation explored here is too high to recommend it for practical usage. Many of the possible performance improvements suggested are equally applicable to PReach as to PReachDB. We currently recommend using PReach and investigating performance improvements that could be applied to both model checkers.

Bibliography

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[2] O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, 1974.

[3] Joe Armstrong. Erlang - A survey of the language and its industrial applications. *The Ninth Exhibitions and Symposium on Industrial Applications of Prolog*, 1996.

[4] Gerd Behrmann. A Performance Study of Distributed Timed Automata Reachability Analysis. *Electronic Notes in Theoretical Computer Science*, 68(4):486–502, 2002. PDMC 2002, Parallel and Distributed Model Checking (Satellite Workshop of CONCUR 2002).

[5] Brad Bingham, Jesse Bingham, Flavio M. de Paula, John Erickson, Gaurav Singh, and Mark Reitblatt. Industrial Strength Distributed Explicit State Model Checking. *Parallel and Distributed Methods in Verification, 2010 Ninth International Workshop on, and High Performance Computational Systems Biology, Second International Workshop on*, 0:28–36, 2010.

[6] Eric Brewer. Cap twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012.

[7] Eric A Brewer. Towards robust distributed systems. In *PODC*, page 7, 2000.

[8] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[9] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[10] Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Zilli. Exploiting Transition Locality in the Disk Based Murϕ Verifier. In *Formal Methods in Computer-Aided Design*, volume 2517 of *Lecture Notes in Computer Science*, pages 202–219. Springer Berlin / Heidelberg, 2002.

[11] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol Verification as a Hardware Design Aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors. ICCD '92. Proceedings.*, pages 522–525, October 1992.

[12] Ericsson AB. *Efficiency Guide User's Guide*, 6.1 edition. `http://www.erlang.org/doc/efficiency_guide/users_guide.html`.

[13] Ericsson AB. *Mnesia Reference Manual*, 4.12.1 edition. `http://www.erlang.org/doc/man/mnesia.html`.

[14] Ericsson Utvecklings AB. *MNESIA User's Guide*, 3.9.2 edition. `http://www.erlang.org/documentation/doc-5.0.1/lib/mnesia-3.9.2/doc/html/part_frame.html`.

[15] Message Passing Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[16] Felix C Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[17] The Open Group. Information technology - Portable Operating System Interface (POSIX). *ISO/IEC/IEEE 9945 (First edition 2009-09-15)*, pages c1–3830, Sept 2009.

[18] Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In *Protocol Specification, Testing, and Verification. 7th International Conference*, pages 339–44, 1987.

[19] Gerard J. Holzmann. An Analysis of Bitstate Hashing. *Formal Methods in System Design*, 13:289–307, 1998.

[20] Alan J. Hu, Gary York, and David L. Dill. New techniques for efficient verification with implicitly conjoined BDDs. In *Proceedings of the 31st annual Design Automation Conference*, DAC '94, pages 276–282, New York, NY, USA, 1994. ACM.

[21] C. Norris Ip and David L. Dill. Better Verification Through Symmetry. In *Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, CHDL '93, pages 97–111, 1993.

[22] Valerie Ishida, Brad Bingham, and Flavio M. de Paula. PReachDB. `https://github.com/ishidav/PreachDB`.

[23] J. Kuskin and D. Ofelt, et al. The Stanford FLASH multiprocessor. In *Proc. of SIGARCH 1994*, pages 302–313, 1994.

[24] Håkan Mattsson, Hans Nilsson, and Claes Wikstrom. Mnesia - A Distributed Robust DBMS for Telecommunications Applications. *First International Workshop on Practical Aspects of Declarative Languages*, 1999.

[25] R. Kumar and E. Mercer. Load balancing parallel explicit state model checking. *Proc. of PDMC 2004, volume 128 issue 3 of Electronic Notes in Theoretical Computer Science*, pages 19–34, 2004.

[26] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. *Proc. of SPIN 1999, volume 1680 of LNCS. Springer*, 1999.

[27] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and Distributed Model Checking in Eddy. *SPIN*, 2006.

[28] Igor Melatti, Robert Palmer, Geoffrey Sawaya, Yu Yang, Robert M. Kirby, and Ganesh Gopalakrishnan. Parallel and Distributed Model Checking in Eddy. *STTT*, 2008.

[29] Paul Mineiro. Fragmentron. `http://code.google.com/p/fragmentron/`.

[30] G. Della Penna, B. Intrigila, I. Melatti, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in automatic verification of finite-state concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):320–341, 2004.

[31] S. Ghemawat, et al. The Google file system. *SOSP*, 2003.

[32] H. Sivaraj and G. Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. In *Proc. of PDMC*

2003, volume 89 issue 1 of *Electronic Notes in Theoretical Computer Science*, pages 51–67. Elsevier, 2003.

[33] Ulrich Stern and David Dill. Using magnetic disk instead of main memory in the Murϕ verifier. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer Berlin / Heidelberg, 1998.

[34] Ulrich Stern and David L. Dill. A New Scheme for Memory-Efficient Probabilistic Verification. In *Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification*, pages 333–348, 1996.

[35] Ulrich Stern and David L. Dill. Parallelizing the Murϕ Verifier. In *Computer Aided Verification. 9th International Conference*, pages 256–267. Springer-Verlag, 1997.

[36] Enrico Tronci, Giuseppe Della Penna, Benedetto Intrigila, and Marisa Zilli. Exploiting Transition Locality in Automatic Verification. In *Correct Hardware Design and Verification Methods*, volume 2144 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin / Heidelberg, 2001.

[37] Pierre Wolper and Denis Leroy. Reliable hashing without collision detection. In Costas Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 59–70. Springer Berlin / Heidelberg, 1993.

Code

The PReachDB code is hosted on Github. See [22].

Tabular Data

For each table, time is in seconds and memory is in MB.

Table A1: Data for Figures 4.1, 4.2

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	4.7	0.0	4.6	0.0	4.5
10000	2.3	5.7	5.9	13.6	6.0	13.6
20000	4.7	7.4	12.5	20.1	12.3	20.1
30000	7.2	7.4	19.2	31.2	19.0	31.2
40000	9.8	7.4	25.0	31.2	24.9	31.2
50000	12.5	7.4	30.7	31.2	30.6	31.2
60000	15.1	7.4	36.6	31.2	36.6	31.2
70000	17.7	7.4	42.1	31.2	42.2	31.2
80000	20.4	7.4	47.7	31.2	47.7	31.2
90000	25.1	17.2	52.1	31.2	52.2	31.2
100000	30.1	22.0	56.5	31.2	56.6	31.2
110000	32.8	17.1	60.6	31.2	60.7	31.2
120000	37.6	17.1	64.7	31.2	64.9	31.2
130000	43.5	11.9	69.0	31.2	69.1	31.2
140000	1479.4	235.0	73.0	31.2	73.1	31.2
150000	1481.7	235.0	76.9	31.2	77.1	31.2
160000	1483.4	235.0	80.6	31.2	80.9	31.2
170000	1485.1	235.0	84.2	31.2	84.4	31.2
180000	1486.8	235.0	1510.5	35.8	1649.8	13.7
190000	1658.6	235.0	1657.1	22.8	1659.8	5.6

Table A2: Data for Figure 4.3

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	5.0	0.0	4.9	0.0	4.8
10000	220.0	7.7	78.6	7.6	74.0	6.6
20000	426.2	12.2	160.2	7.6	151.2	9.3
30000	641.0	12.2	243.4	15.4	229.1	9.3
40000	840.5	12.2	326.8	12.1	308.2	9.3
50000	1039.1	12.2	410.0	12.1	387.2	9.3
60000	1227.7	12.2	494.1	12.1	465.6	9.3
70000	1417.0	12.2	579.4	12.1	546.7	9.3
80000	1593.4	12.1	664.3	12.1	631.8	9.4
90000	1718.6	12.2	750.6	12.1	713.9	9.3
100000	1881.3	7.7	842.6	58.5	802.6	81.9
110000	2040.9	9.4	932.1	97.4	888.9	81.9
120000			1016.2	164.9	985.4	81.9
130000			1109.9	343.2	1078.7	82.0
140000			1195.7	352.7	1168.2	81.9
150000			1285.2	58.5	1257.2	82.0
160000			1368.8	116.0	1359.9	81.9
170000			1458.4	116.0	1453.7	81.9
180000			1541.9	82.1	1548.6	81.9
190000			1632.5	97.5	1657.6	82.0
200000			1720.7	236.1	1759.8	81.9
210000			1802.1	74.6	1856.3	82.0
220000			1888.8	36.2	1951.1	81.9
230000			1959.2	53.7	2030.6	81.9
240000			2034.5	7.7		

Table A3: Data for Figure 4.4

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	4.9	0.0	4.9	0.0	4.9
10000	55.9	5.6	491.8	69.1	57.4	8.4
20000	113.0	5.3	747.8	100.2	115.6	6.5
30000	172.4	5.2	973.6	97.4	175.0	6.6
40000	233.2	7.7	1173.1	97.3	234.1	6.6
50000	293.6	7.7	1359.9	97.4	295.4	5.1
60000	356.6	45.8	1519.6	97.4	359.8	58.5
70000	420.2	6.7	1658.5	97.4	427.3	99.4
80000	490.8	7.7	1699.7	97.4	493.2	82.0
90000	552.7	7.7	1719.1	97.4	559.5	115.9
100000	622.4	6.7	1730.7	97.4	627.4	69.2
110000	702.4	6.0	1737.0	97.4	698.7	116.0
120000	775.7	7.8	1741.9	97.4	769.5	116.0
130000	857.1	7.8	1746.4	97.4	842.1	82.1
140000	928.2	6.7			906.5	116.0
150000	999.6	7.7			979.2	82.1
160000	1069.9	6.7			1056.1	156.6
170000	1138.5	7.8			1125.9	7.7
180000	1213.0	7.7			1200.6	179.1
190000	1288.8	6.7			1269.2	215.0
200000	1363.3	7.8			1338.4	6.1
210000	1446.1	50.6			1405.8	69.2
220000	1519.9	50.6			1481.5	48.7
230000	1600.3	50.7			1549.7	32.3
240000	1670.5	50.6			1614.8	7.8
					1675.3	30.3

Table A4: Data for Figure 4.5

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	5.0	0.0	4.9	0.0	4.9
10000	56.1	6.0	514.4	12.1	57.9	7.6
20000	113.6	5.9	757.4	12.1	118.2	7.6
30000	172.3	5.5	976.9	12.2	178.1	7.7
40000	232.9	7.8	1167.8	12.2	238.1	6.6
50000	292.4	7.7	1353.7	12.1	300.1	5.1
60000	354.2	5.1	1491.4	12.1	363.3	97.8
70000	416.7	7.7	1617.9	44.0	428.0	144.7
80000	480.0	7.7			491.9	97.5
90000	544.3	7.7			558.5	116.0
100000	612.6	6.0			630.5	130.2
110000	680.9	7.7			698.0	176.9
120000	749.2	7.7			766.3	227.7
130000	820.2	64.0			836.2	172.7
140000	896.5	7.7			910.2	177.0
150000	974.8	7.7			994.6	287.5
160000	1048.6	7.8			1063.3	266.7
170000	1123.8	6.0			1134.6	116.0
180000	1202.6	7.7			1209.9	262.8
190000	1275.0	6.1			1282.6	123.5
200000	1347.7	58.5			1353.2	69.2
210000	1419.3	58.5			1423.0	69.2
220000	1497.5	58.5			1497.5	58.5
230000	1568.5	58.5			1564.0	32.3
240000	1642.2	58.5			1631.3	31.6
250000	1706.7	58.5			1697.6	14.1

Table A5: Data for Figure 4.6

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	4.8	0.0	4.7	0.0	4.8
10000	32.1	5.9	112.9	12.0	33.0	6.5
20000	66.0	8.1	224.8	12.0	68.0	9.3
30000	100.5	8.1	319.6	12.0	103.4	12.7
40000	135.6	8.0	384.1	12.4	139.2	9.3
50000	170.7	8.1	431.1	12.4	175.4	9.2
60000	205.7	6.7	478.4	12.4	212.0	9.3
70000	241.5	8.0	524.0	15.8	249.3	9.3
80000	278.3	8.0	560.9	15.8	283.5	6.5
90000	316.6	7.0	608.3	15.7	323.6	9.7
100000	356.6	9.7	656.3	7.9	372.0	8.3
110000	414.3	8.0	720.8	86.7	431.7	8.3
120000	474.6	8.0	826.0	116.2	493.7	6.6
130000	555.4	9.8	999.9	116.2	564.5	7.2
140000	614.8	8.4	1134.3	7.9	646.5	8.7
150000	679.1	8.0	1271.1	82.2	750.3	154.8
160000	766.5	8.0	1381.9	17.6	883.5	62.5
170000	934.0	8.0			981.2	161.3
180000	1067.2	7.0			1051.9	97.8
190000	1213.1	58.8			1157.9	69.4
200000	1307.4	58.8			1221.4	44.6
210000	1372.4	58.8			1323.9	49.0
220000					1387.2	14.3

Table A6: Data for Figure 4.7

# states visited	proc 0		proc 1		proc 2	
	time	memory	time	memory	time	memory
0	0.0	5.1	0.0	4.8	0.0	4.9
10000	28.1	6.7	82.6	7.5	28.3	6.5
20000	57.9	9.4	167.8	7.5	58.2	9.3
30000	88.6	9.4	254.8	12.1	88.9	9.3
40000	119.3	9.4	313.6	12.1	119.9	9.3
50000	151.1	9.4	357.8	12.0	151.7	9.3
60000	182.2	9.5	403.7	12.1	182.9	9.3
70000	214.0	9.4	445.8	12.1	214.8	9.3
80000	247.3	9.5	488.8	12.1	246.5	9.3
90000	282.4	7.7	535.1	12.1	282.5	7.6
100000	332.8	7.8	569.1	12.1	326.5	7.6
110000	383.1	6.7	645.9	6.5	384.7	6.3
120000	442.1	82.2	779.7	69.1	439.4	8.3
130000	501.6	141.0	877.8	178.6	501.0	119.5
140000	574.2	82.1	1048.1	164.9	572.8	6.6
150000	684.9	82.1	1161.3	5.9	660.1	115.9
160000	829.7	82.1	1261.0	58.4	781.9	82.1
170000	974.3	129.6	1386.7	58.4	950.6	122.3
180000	1121.3	82.1			1055.8	52.0
190000	1245.3	82.1			1180.2	58.4
200000	1360.1	82.1			1359.1	26.2
210000	1459.0	82.1			1442.5	14.0