Brigham Young University
BYU ScholarsArchive

Theses and Dissertations

2004-10-21

Disk Based Model Checking

Tonglaga Bao
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Bao, Tonglaga, "Disk Based Model Checking" (2004). *Theses and Dissertations*. 191.
https://scholarsarchive.byu.edu/etd/191

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

DISK BASED MODEL CHECKING

by

Tonglaga Bao

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

August 2004

Copyright © 2004 Tonglaga Bao

All Rights Reserved

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Tonglaga Bao

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Mike Jones, Chair
Date	Eric Mercer
Date	Scott Woodfield

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Tonglaga Bao in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Mike Jones Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

DISK BASED MODEL CHECKING

Tonglaga Bao

Department of Computer Science

Master of Science

Disk based model checking does not receive much attention in the model checking field becasue of its costly time overhead. In this thesis, we present a new disk based algorithm that can get close to or faster verification speed than a RAM based algorithm that has enough memory to complete its verification. This algorithm also outperforms Stern and Dill's original disk based algorithm. The algorithm partitions the state space to several files, and swaps files into and out of memory during verification. Compared with the RAM only algorithm, the new algoritm reduces hash table insertion time by reducing the cost and growth of the hash load. Compared with Stern's disk based algorithm, the new disk based algorithm significantly reduces disk vs memory comparsion but increases disk read/write time. The size of the model the new algorithm can verify is bound to the available disk size instead of the available RAM size.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mike Jones for funding my education and patiently guiding me throughout my research. I would like to thank Dr Eric Mercer and Dr Scott Woodfield for their time and efforts put on my thesis. I thank all members of the Verification and Validation lab for generously offering me suggestions and help.

I also would like to thank my parents for their love, support and encouragement throughout my education.

TABLE OF CONTENTS

Section Page

1 Introduction **1**
1.1 Background 3
1.2 Thesis Statement 6

2 Related Work **9**
2.1 Stern and Dill's Disk Based Algorithm 9
2.2 Transition Locality 10
2.3 Parallel Model Checking Algorithm, Hash Function and Hash Table . . . 11

3 The Algorithm **13**
3.1 Description 13
3.2 Analysis 16
3.2.1 Comparison with RAM Based Algorithm 16
3.2.2 Comparison with Stern and Dill's Disk Based Algorithm 19

4 Experimental Results **23**
4.1 Using the Original Hash Function 24
4.2 Using SPIN's Hash Function 27
4.3 Big Models 29

5 Conclusions and Future Work **31**
5.1 Conclusion 31
5.2 Future Work 32

LIST OF TABLES

Table Page

2.1 Performance Numbers for Stern and Dill's Disk Based Algorithm in Seconds 10

4.1 RAM Based Murphi, All Times in Seconds, Memory in MB 25
4.2 Stern's Disk Based Murphi, All Times in Seconds, Memory in MB 25
4.3 New Disk Based Murphi, All Times in Seconds, Memory in MB 25
4.4 RAM Based Murphi, All Times in Seconds, All Memory in MB 27
4.5 Stern's Disk Based Murphi, All Times in Seconds, All Memory in MB . . 27
4.6 New Disk Based Murphi, All Times in Seconds, All Memory in MB . . . 27
4.7 All Times in Seconds, All Memory in MB 30

LIST OF FIGURES

Figure — Page

1.1 RAM Based Murphi 3
1.2 Explicit State Enumeration Using RAM 4
1.3 Disk Based Murphi 6
1.4 Explicit State Enumeration Using Magnetic Disks 7
1.5 Parallel Disk Based Murphi 8

2.1 RAM Based Murphi 9

3.1 Parallel Disk Based Algorithm 14

4.1 Speedup of New Algorithm over RAM Algorithm 25
4.2 Speedup of New Algorithm over Stern's Algorithm 26
4.3 Slowdown of New Algorithm over RAM Algorithm 28
4.4 Speedup of New Algorithm over Stern's Algorithm 28

Chapter 1

Introduction

Given a transition system and a property, model checking is a technique that determines if the transition system satisfies the given property. It has been used successfully in industrial distributed communication protocols and circuit designs to verify their correctness or find bugs. Model checking has some advantages compared to traditional verification approaches like simulation and testing.

Simulation and testing are popular ways to find bugs. However, no matter how many simulations are made, there is always a possibility that some of the cases will be missed. This might cause some bugs to go undiscovered. Thus, not finding an error can not guarantee the correctness of the model.

Since model checking deals with finite state machines, it is possible to do exhaustive search to ensure the correctness of the model. Furthermore, it is automatic, and it is able to produce a counterexample to show the source of the error. Model checking is being used by companies like Intel, IBM, AMD and HP.

The application of model checking is restricted because of the state space explosion problem. State space explosion occurs in systems built from many components that interact with each other or in systems with data structures that can have many different values. The state space grows exponentially in these cases. There has been

much research done to address the state space explosion problem. This research has significantly increased the size of the state space that can be verified.

There are typically two ways to reduce state space explosion. One is to compress the size of state space. Examples include symmetry reduction, symbolic representation of the state space, bit state hashing (supertrace), hash compaction, partial order reduction etc. The other way is to increase the storage size of state space by utilizing disk and distributed resources. This thesis presents an algorithm that uses a disk instead of RAM to store the state space.

Disk is much cheaper than RAM memory. This makes it tempting to store the state space on disk instead of in main memory. However, as we expect, the time penalty is a new bottleneck for disk based model checking. The cost of storing and recovering states from disk is so costly that one must carefully design the algorithm to make this approach practical. The algorithm presented in this paper uses an idea from parallel model checking to efficiently use the disk. We implemented our algorithm within the Murphi model checker. Experimental results show its verification speed is close to or faster than the murphi RAM based algorithm. It also outperforms the Stern and Dill's disk based algorithm. Note that the size of the model the new algorithm can verify is bound by the disk size instead of being bound by the RAM size.

The remainder of this chapter introduces background information about model checking, which is needed to understand this thesis work. It also presents a brief introduction of the new algorithm and the thesis statement. Chapter 2 analyzes related work that has been done in the disk based model checking field. Chapter 3 gives the pseudo code of the new algorithm and analyzes its performance. Chapter 4 presents the experimental results of the new algorithm. Chapter 5 gives conclusions and enumerates several questions and possible future work.

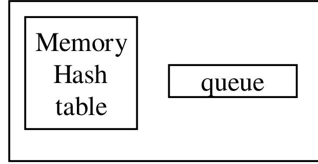

Figure 1.1: RAM Based Murphi

1.1 Background

Model checking is a method for formally verifying finite-state concurrent systems. Model checking goes through the following processes. First of all, the system is translated into a model that can be accepted by the model checker. Then, the desired property is expressed through temporal logic. Finally, the model checker traverses the state space of the model to see if the model satisfies the given property.

State space exploration is the heart of the verification process. State space exploration can be performed using two different methods: explicit and symbolic. Explicit state model checking stores actual states while symbolic model checking stores sets of states symbolically. Explicit state model checking is examined in this thesis.

In explicit state model checking, the model checker enumerates all the reachable states from the start states. Either breadth first or depth first search is used in the state enumeration process. Figure 1.1 illustrates the RAM based explicit state model checking architecture. The outer box represents the RAM. The algorithm uses one queue/stack and one hash table to store the state space. The queue/stack stores unexplored states. The hash table stores explored states and eliminates duplicate states. There are two problems that occur in this algorithm as the number of explored states grows: first, memory becomes full due to state explosion; second, hash insertion slows down due to collisions. Figure 1.2 shows the algorithm.

There are several approaches to reduce state space explosion. Partial order reduction, symmetry reduction, hash compaction and bit state hashing (supertrace) are

```
1   var        // global variables
2     M: hash table;        // main memory table
3     Q: FIFO queues;        // state queue

4   Search()        // main routine
5   begin
6     M := ∅; Q := ∅;        // initialization
7     for each startstate s0 do        // startstate generation
8       Insert(s0);
9     end
10    while Q ≠ ∅ do
11      s := dequeue(Q);
12      for all s' ∈ successors(s) do
13        Insert(s');
14      end
15    end
16  end

17  Insert(s: state)        // insert state s in main memory table
18  begin
19    if s ∉ M then begin
20      insert s in M;
21      insert s in Q;
22    end
23  end
```

Figure 1.2: Explicit State Enumeration Using RAM

used to manage the size of the state space. Partial order reduction constructs a reduced state space according to the commutativity of concurrently executed transitions [1]. Symmetry reduction reduces the size of the state space by exploiting permuataion groups that preserve both the state labeling and the transition relations [2]. Hash compaction and bit state hashing (supertrace) compress the size of each state at the expense of a certain probability that some states of the system are omitted during verification [3][4].

Distributed model checking and disk based model checking enlarge storage space. Distributed model checking utilizes distributed resources, like a network of workstations, to increase the amount of available RAM. Disk based model checking uses a disk instead of main memory to store the state space [5].

Stern and Dill proposed an algorithm that uses a disk instead of main memory to store the state space. Figure 1.3 illustrates their disk based architecture. There is one queue, one memory hash table and one file on disk. It stores the unexplored states in the queue and explored states in memory. The difference is that at every level of the breadth-first search tree, it stores the new states in memory to disk and clears the memory hash table. As a result, all explored states are eventually stored to the disk. However, before storing the states to disk, the algorithm compares every state on disk with the states in memory to identify duplicate states. This is a very time consuming process. Fugure 1.4 shows the algorithm. A detailed analysis of this algorithm is presented in Chapter 2.

Our new disk based algorithm significantly reduces the disk vs memory comparsion. Figure 1.5 illustrates the architecture for the proposed new algorithm. It has a memory hash table, an array of queues in memory, an array of disk queues and an array of disk files. The algorithm swaps a disk file into and out of the memory hash table so that a fast hash function is used to eliminate duplicate states. The new algorithm reduces hash collison penalties associated with the RAM based algo-

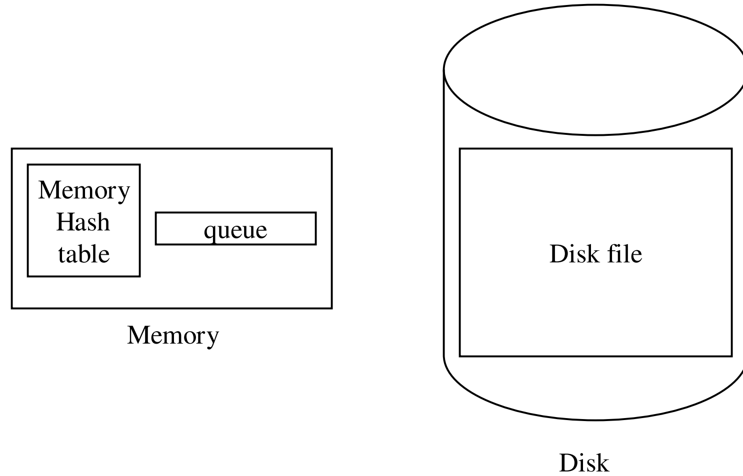

Figure 1.3: Disk Based Murphi

rithm. Two hash functions partition the state space. The first hash function decides which partition the current state belongs to. The second hash function decides which hash entry the current state is hashed into. Using two cooperating hash functions to partition the state space helps reduce the performance penalty associated with the hash load. Detailed analysis of this algorithm and pseudo code of it can be found in Chapter 3.

1.2 Thesis Statement

The new disk based algorithm with two cooperating hash functions and multiple hash tables gets close to or faster verification speed than the traditional RAM based algorithm with one hash function and one hash table due to a decreased hash load in the disk based algorithm. The new disk based algorithm improves running time over the original Stern and Dill's disk based algorithm by reducing the costly disk and memory table synchronization process. The size of the model this new algoirthm can verify is bound to disk size rather than RAM size.

```
1   var         // global variables
2     M: hash table;        // main memory table
3     D: file;              // disk table
4     Q: FIFO queues;       // state queue
5   Search()        // main routine
6   begin
7     M := ∅; D := ∅; Q := ∅;        // initialization
8     for each startstate s0 do       // startstate generation
9       Insert(s0);
10    end
11    do                              //search loop
12      while Q ≠ ∅ do
13        s := dequeue(Q);
14        for all s' ∈ successors(s) do
15          Insert(s');
16        end
17      end
18      CheckTable();
19    while Q ≠ ∅;
20  end

21  Insert(s: state)        // insert state s in main memory table
22  begin
23    if s ∉ M then begin
24      insert s in M;
25      if full(M) then
26        CheckTable();
27    end
28  end

29  CheckTable()        // do old/new check for main memory table
30  begin
31    for all s ∈ D do        // remove old states from main memory table
32      if s ∈ M then
33        M := M - {s};
34    end
35    for all s ∈ M do        //handle remaining (new) states
36      insert s in Q;
37      append s to D;
38      M := M - {s};
39    end
40  end
```

Figure 1.4: Explicit State Enumeration Using Magnetic Disks

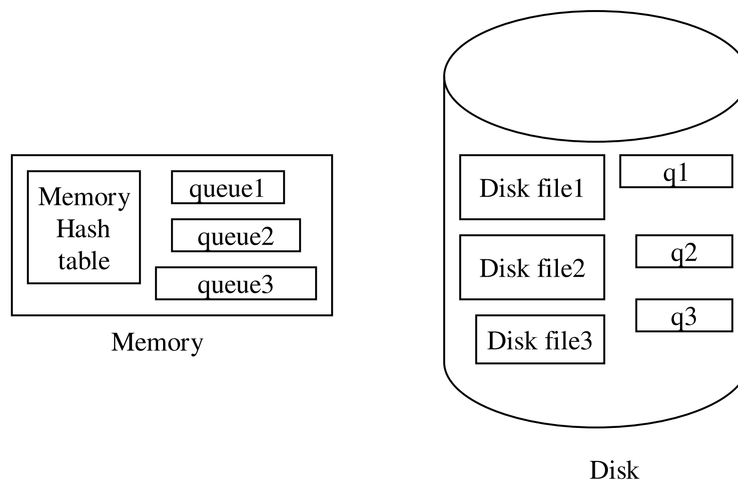

Figure 1.5: Parallel Disk Based Murphi

Chapter 2

Related Work

There have been two algorithms implemented for disk based model checking. The first by Stern and Dill [6], and the other by Penna, Intrigilla, Tronci and Zili [7]. The following sections analyze these two papers in depth. This section also presents the hash functions and hash tables used by the new algorithm.

2.1 Stern and Dill's Disk Based Algorithm

Stern and Dill's disk based algorithm works as follows: Do a breadth first search of the state space while storing the generated states in memory. After each level of the breadth first search, move the states in memory to disk. If memory becomes full before one level of the breadth first search is completed, move the states in memory to disk and continue the breadth first search. Figure 2.1 shows data structure of this algorithm.

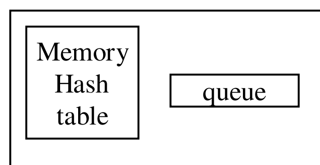

Figure 2.1: RAM Based Murphi

Function	atomix	mcslock1	dp12	newcache6	newlist6
File Read	6.54	51.93	3.15	25.07	23.91
File Write	0.81	5.00	0.50	4.77	2.43
Check Table	568.18	1841.39	134.06	123.97	316.98
Total Time	1235.70	4726.04	373.19	2619.89	1392.53

Table 2.1: Performance Numbers for Stern and Dill's Disk Based Algorithm in Seconds

---

To move states from memory to disk, a comparison between the states currently on the disk with the states in memory is done to delete previously visited states from memory. The algorithm avoids costly random access to disk by sequentially reading all the states from disk and comparing each of them with the states in memory. Every disk write access appends the newly generated states at the end of file.

We have implemented the Stern and Dill's algorithm in Murphi, which is a model checker developed at Stanford University. Table 2.1 illustrates the experimental results on several problems. The File Read row contains total time spent reading to disk, the File Write row gives total time spent writing to disk, the Check Table row shows the time spent comparing states on disk with states in memory and the Total Time row reports total wall clock time elapsed. Table 2.1 shows that File Read/Write overhead is small but the Check Table time is big compared with the total verificaion time. In fact, Disk read/write operations only account for 1% of the total running time whereas the check table operation accounts for 30% of the total running time.

2.2 Transition Locality

The Penna et al develops an algorithm based on transititon locality. Most systems have transition locality, which means a transition made from one state often leads to another state that was recently explored or leads to a new state. This algorithm takes advantage of transition locality to decrease disk access time.

The algorithm does the following: Partition the disk file into several blocks, and select some blocks from disk to use in the comparison with states in memory instead

of using whole disk file. The blocks nearer to the tail, where recently found states are stored, of the file have higher probability of being selected because of transition locality. Due to the breadth first search, the states written at the tail of the file are recently visited states and have a higher probability of being in the memory. The drawback of this algorithm is that it is unable to delete some visited states in memory because it does not check the entire file to identify every visited state; Thus, the algorithm repeatedly explores old states and generates duplicate states.

To mitigate reexpanding states, the algorithm occasionally selects new blocks from disk. It uses a small portion of the blocks that are not selected to do the comparison; and it keeps a statistic that tracks the percentage of the deleted states from selected or nonselected blocks. It adjusts the block selection according to the statistics and the duration of the run time.

The authors claim that time reduction in this algorithm is due to savings in the file I/O. However, file I/O is too small to account for their savings. We claim the time reduction is due to the reduction in the disk table size that needs to be compared against the memory hash table. The comparison cost between the disk table and the memory hash table is described in more detail later.

The main contribution of the Penna et al algorithm is the speed up obtained by only comparing a portion of the states on disk with the states in memory. However, even with high transtion locality, it is still costly to only take part of the disk space into consideration. A single duplicate state missed in the disk check can cause the algorithm to produce many more states than necessary. Fixing the problem is another time consuming process since the algorithm does not know exactly which disk block to select in order to get rid of the duplicate states.

2.3 Parallel Model Checking Algorithm, Hash Function and Hash Table

The new algorithm is a sequential version of the parallel model checking algorithm in [8][9][10]. The parallel algorithm keeps a queue and a hash table on every

workstation participating in the state exploration. For every newly generated state, a partition function calculates which workstation the state belongs to and sends the state to the workstation's queue using network communication. In contrast, the proposed algorithm keeps all the queues in main memory and keeps their corresponding hash tables on disk as files.

There are two hash functions used in state exploration. The first hash function decides which partition the current state belongs to. The second hash function decides which hash entry the current state belongs to. Hash functions used in our experiments include the original Murphi hash function and the SPIN's s-hash-jenkins hash function.

Murphi originally used an open hash table, which produces a lot of collisions. A chained hash table gives better results for RAM based, Dill's disk based and new disk based algorithms. All the experiments done in this thesis is based on chained hash table.

Chapter 3

The Algorithm

The analysis of Stern and Dill's disk based algorithm in Chapter 2 indicates disk I/O takes around 1% of total verification time, while comparing states in disk and memory takes more than 30% of the total verification time in most of the models. Compared with Stern and Dill's disk based algorithm, the new disk based algorithm reduces disk vs memory comparsion but increases disk I/O time. Compared with the RAM only algorithm, the new algoritm reduces hash table insertion time by reducing the cost and growth of hash load. This chapter presents the pseudo code for and an explanation of the new algorithm. It compares the new algorithm with the traditional RAM based and Stern and Dill's disk based algorithms.

3.1 Description

The new algorithm uses a memory hash table, an array of queues in memory, an array of disk queues and an array of disk files. The algorithm swaps disk files into and out of memory so that a hash function can eliminate duplicate states.

The new algorithm works as follows: Apply a hash function to the start states to partition them into queues. Generate successors for states in the queue with the largest number of states. Keep all the states corresponding to this queue in a hash table in memory. Put other generated states into the other appropriate queues. Con-

1 **Search**()
2 **for** every startstate s_0 **do**
3 $i :=$ Partition(s_0);
4 insert s_0 into q_i^m;
5 **if** Full(q_i^m) **then**
6 store q_i^m in q_i^d;
7 $q_i^m := \emptyset$;
8 **end**
9 **end**
10 $i := max_{i \in n}(|q_i^m + q_i^d|)$;
11 Select(i);
12
13 **Select**(i: int)
14 **while** $i \geq 0$
15 **do**
16 **while** $q_i^m \neq \emptyset$ **do**
17 load $D[i]$ into M;
18 $s = dequeue(q_i^m)$;
19 **if** s is not in M **then**
20 insert s in M;
21 Explore (i, s);
22 **end**
23 **end**
24 **if** $q_i^d \neq \emptyset$ **then** load q_i^d to q_i^m;
25 **while** $q_i^m \neq \emptyset$; // end of do-while loop
26 store $D[i]$.
27 $i := max_{i \in n}(|q_i^m + q_i^d|)$;
28 **if** $|q_i^m + q_i^d| = 0$ **then** $i = -1$;
29 **end**
30
31 **Explore**(i: int, s: state)
32 **for** all $s' \in$ successors(s) **do**
33 i':= Partition(s');
34 **if** $i' = i$ and s' is not in M **then** insert s' in q_i^m; **else** insert s' in $q_{i'}^m$;
35 **if** Full($q_{i'}^m$) **then**
36 store $q_{i'}^m$ in $q_{i'}^d$;
37 $q_i^m := \emptyset$;
38 **end**
39 **end**

Figure 3.1: Parallel Disk Based Algorithm

tinue generating states until either memory is full or the queue is empty. Then move the states in memory to disk in a manner similar to the Stern and Dill's algorithm; finally, choose the queue with the largest number of states again, load the corresponding disk file into memory and repeat the process.

Figure 3.1 contains the pseudocode of the new algorithm. There is a partition function that maps every state to a unique memory queue. There are the same number of disk files and memory queues. Memory queues store unexplored and explored states, disk queues store unexplored and explored states when the memory queues are full, and disk files store explored states.

The Search function generates start states and stores the start states in their corresponding queues. If a memory queue becomes full, then that queue is written to disk (lines 1-9). q_i^m is the queue in memory belonging to partition i and q_i^d is the queue in disk belonging to partition i. The function then selects the queue, i , with the most states as the active queue and calls the Select funtion (lines 10-11).

The Select function loads the disk file that corresponds to the active queue into memory (line 17). It dequeues states from the active queue to generate every successor of the states in the queue (line 18). The Select function then stores the dequeued states into the memory hash table if they are not present in the current table in memory (lines 19-20). This allows expanded states to be stored in the hash table in memory. When the active queue becomes empty, the corresponding disk queue is loaded into memory (lines 24). After both the memory queue and the disk queue are empty, the table of expanded states are stored back to disk (lines 26). The algorithm then chooses the next longest queue, loads the corresponding table and continues the exploration (lines 27). If all the queues are empty, the algorithm terminates (line 28).

The Explore funtion checks to see if the successors of the states in the active queue belong to the current queue (lines 32-33). If they do, and they are not present in the current table in memory, then the function adds the states into the current

active queue. If they do not belong to the current queue, then it stores them to their corresponding queues (line 34). This allows duplicate and expanded states to be stored in the work queue. If any of the queues are full, then it stores them to the corresponding disk queue (lines 35-37).

3.2 Analysis

An ideal model checking algorithm completes verification using the smallest amount of memory and time possible. This section compares the new algorithm with the RAM based algorithm and Stern and Dill's disk based algorithm in terms of total verification time and total memory needed to complete the verification.

3.2.1 Comparison with RAM Based Algorithm

The verification time is composed of different elements depending on which verification algorithm is used. For RAM based algorithm

$$RamTotalTime = RamInsertionTime + ConstantTime$$

RamInsertionTime is the time spent on inserting newly generated states into memory. ConstantTime is the time spent on the constant part of verificaiton such as finding the enabled transitions, generating next states, and doing symmetry reduction etc.

For the new algorithm

$$\begin{aligned} NewTotalTime &= NewInsertionTime + ConstantTime + \\ &\quad NewIOTime + OtherOverheads \end{aligned}$$

NewInsertionTime is the time spent on inserting states into memory. ConstantTime is the time spent on constant part of the verification. NewIOTime is the time spent reading the states from disk to memory and writing the states from memory to disk. OtherOverheads includes the time overhead related to computing each state's corresponding partition and the time overhead related to storing duplicate states in the queues.

NewInsertionTime is typically faster than RAMInsertionTime. For the RAM based algorithm, all the states generated will be kept in memory. Many collisions are possible as the hash table fulles, especially when the hash function does not give a good distribution.

For new algorithm, the number and cost of collisions are allevivated by two means: first, the state space is partitioned into several files which reduces the hash load because more space is available. secondly, two cooperating hash functions are used to separate the states that belong to same chain of the chained hash table. The first hash function decides which partition the current state belongs to and the second hash function decides which hash entry in the current partition the state hashes to. The first hash fucntion seperates states that belong to the same hash table entry when a second hash function is used.

NewIOTime needs to be considered for the new algorithm. Swapping occurs when the current queue is empty. In general, reading and writing to disk occurs more frequently in the beginning of the verification and at the end of the verification - when there are fewer states stored in each queue. Frequent swapping in the beginning of verification can be ignored since the file sizes are small in the beginning and reading/writing takes only small amount of time. Swapping is less frequent in the middle of verificaiton since there are many states in each queue, and new states are quickly generated and added into the queues. Thus the main time consuming part of disk i/o overhead is at the end of the verification.

Suppose the state size for some transition system is S bytes, there are P partitions on the disk and assume partition p will be swapped I_p times during the whole verification. Assume there are Q_i^p states in partition p on the ith swap, then the total number of states read from disk is

$$Q = \sum_{p=1}^{P} \sum_{i=1}^{I_p} Q_i^p$$

Assume the read transfer rate is R bytes/second and the write transfer rate is W btyes/second, then the disk file read overhead is $Q * S/R$ seconds and the disk file write overhead is $Q * S/W$ seconds. Thus,

$$NewIOTime = QS(\frac{1}{R} + \frac{1}{W})$$

The OtherOverheads include two kinds of overheads incured by the use of new algorithm. The first is computation time overhead, which occurs because the partition needs to be calculated for every newly generated state. This overhead depends on the total number of states that are explored. Suppose T states are generated and every computation takes c seconds, then the computation overhead is $T * c$.

The second is the overhead related to store duplicate states in the queues. For the RAM based algorithm, newly generated states are checked against the hash table to make sure that they have not been explored before and then inserted into the queue. For the new algorithm, duplicate states may be stored in the queues because only one hash table is active at any given time. This overhead depends on total number of enabled transitions that lead to duplicate states. Suppose total number of enabled transitions that lead to duplciate states is e, and storing a state in a queue takes q seconds, then this overhead will be eq seconds. Thus

$$OtherOverhead = T * c + eq$$

There is also some overhead for writing and reading states to and from disk queues. However, since each queue is only read or written once, this overhead is negligible. Thus,

$$\begin{aligned} NewTotalTime &= NewInsertionTime + ConstantTime + \\ &\quad Q * S/R + Q * S/W + T * c + eq \end{aligned}$$

The new algorithm persforms better than RAM based algorithm when

$$RamInsertionTime - NewInsertionTime > NewIOTime + OtherOverhead$$

For a given model, S, R, W, T, c, e, q are all constant and depend on the machine being used. As a result, Q, $RamInsertionTime$ and $NewInsertionTime$ are variables that decides which algorithm performs better. This equation can be satisfied when the difference between $RamInsertionTime$ and $NewInsertionTime$ is big and Q is small. These values depend on how the partition functions reduce hash insertion time and distribute states to different partitions.

The new disk based algorithm can get several orders of magnitude in memory savings with a similar verification speed as the RAM based algorithm. In the optimal case, the largest model that can be verified by the disk based algorithm has size

$$NumOfPartition * MemoryHashSize$$

Usually, part of the disk is used to store the disk queues, so not all the disk space can be dedicated to store state files.

3.2.2 Comparison with Stern and Dill's Disk Based Algorithm

The new disk based algorithm can get a speed up compared with Stern and Dill's algorithm. Stern's agorithm needs

$$\begin{aligned} DiskTotalTime &= DiskInsertionTime + ConstantTime + \\ &\quad DiskIOTime + DiskComparsionTime \end{aligned}$$

ConstantTime is constant time spent on verification. This number is used before in the new and RAM algorithm. DiskInsertionTime is the time spent on inserting newly generated states into memory. This value is similar to NewInsertionTime but smaller than RamInsertionTime. Collisions are reduced in this case since the verification resumes with an empty hash table every time after every disk read/write occurs. DiskIOTime is the disk i/o time spent in Stern's algorithm and DiskComparisonTime is the time spent on comparing states in disk with states in memory to get rid of the duplicate states. DiskIOTime and DiskComparisonTime require more detailed analysis.

For Stern and Dill's algorithm, assume K_i is the number of states on disk when the file is read the ith time and assume that the disk file is read t times during the entire search process. The total number of states read from disk is $K = \sum_{i=1}^{t} K_i$. Assume the read transfer rate is R bytes/second, and each state contains S bytes, then the read overhead is $K*S/R$ seconds. Assume the write transfer rate is W btyes/second, and the total number of states written to disk is T, then the write overhead is $T*S/W$ seconds. Assume the memory table is M bytes, then the average comparisons each state makes is $M/2S$ and the avarage overhead of comparision is $K*M/2S$ comparisons. Assume the time spent on each comparison of states in disk with states in memory is C seconds, then the DiskIOTime is

$$\frac{KS}{R} + \frac{TS}{W}$$

and DiskComparisonTime is

$$\frac{CKM}{2S}$$

As a result, the total time spent on Stern's algorithm is

$$TotalTime = DiskInsertionTime + ConstantTime + \frac{KS}{R} + \frac{TS}{W} + \frac{CKM}{2S}$$

In most cases DiskInsertionTime was bigger than NewInsertionTime and smaller than RamInsertionTime. It is bigger than NewInsertionTime because the number and cost of collisons grow for Stern and Dill's disk based algorithm as the table in memory becomes fuller. However, the DiskInsertionTime is still smaller than RamInsertionTime since the verification starts with an empty hash table after each file read/write access. Thus, we only need to make sure that

$$\frac{QS}{R} + \frac{QS}{W} + T * c + eq \leq \frac{KS}{R} + \frac{TS}{W} + \frac{CKM}{2S}$$

to get a speed up. Which is

$$\frac{(Q-K)S}{R} + \frac{(Q-T)S}{W} + T * c + eq \leq \frac{CKM}{2S}$$

The values of R, W, C, M, q depend on what machine is being used. For a given model, S, T, c, e are constant. Only K and Q vary when different algorithms are used. K and Q grow quickly as the size of model grows since K and Q depend on the total number of states. Thus, the dominating factors in this equation are Q and K. As the model grows bigger, K increases faster than Q, since the growth of K is reflected on one file, while the growth of Q is amortized into several smaller files. Thus, mathematical analysis predicts the new algorithm will acheive more speed up as the model grows bigger. The memory saving factor gained by both algorithms is similar.

Chapter 4

Experimental Results

This chapter presents the experimental results of running the RAM based, Stern and Dill's disk based and new disk based algorithms on several models using both the original hash function of Murphi and SPIN's s-hash-jenkins hash function. We report the time spent on the most time consuming parts of each algorithm except the constant time spent during generating states. The models are atomix, mcslock1, dp, newcache6, newlist6 and dense.

We give results for two kinds of models: those that can be verified in less than 2GB of RAM and those that can not. We use smaller models to test all algorithms and use bigger models to test Stern and Dill's disk based algorithm and new disk based algorithm since a unix process on a 32 bit machine only can addresses 2GB of memory. For smaller models, we use two different hash functions to demonstrate the importance of good hash performance.

Our implementation in Murphi uses a chained hash table. A double hash table, as used in prior work in explicit state model checking [6][7] causes all 3 algorithms to run more slowly and has the most significant impact on the orginial RAM and disk algorithms.

4.1 Using the Original Hash Function

This section reports test results using Murphi's original hash function. This hash funciton is a global hash funtion. It gets the hash index by adding all the bits of a state together.

Table 4.1 shows the result from RAM based Murphi, which has enough memory to complete the verification. The Total States row shows the total number of states generated for each model. The Memory Used row reports the memory allocated for the hash table and memory queue. The Insert States row shows the time spent on inserting states into memory. The Total Time row reports total time spent on verification. The Insert States time occupies more than 30% of the total time in most of these models.

Table 4.2 presents the result from Stern's disk based algoirthm. The Memory Used row presents the size of memory allocated for the memory hash table and memory queue. The File Read row shows total time spent reading the file from disk. The File Write row shows the total time spent writing the file to the disk. The Check Table reports the time spent comparing states on disk with states in memory. The Insert States row shows the time spent on inserting states into memory. The Check Table time takes more than 30% of total time in most of the models.

Table 4.3 shows the result from new disk based algorithm. The Computation Time row reports the computation time spent on calculating each state to gets its corresponding parition number. The EnDequeue row reports time spent on enqueuing and dequeuing states from queues. The other rows have same meaning as Table 4.2. The constant time spent on generating states is not reported since it is same for all three algorithms.

Figure 4.1 illustrates the speedup of the new algorithm over the RAM algorithm. Atomix, mcslock1 and dp12 get speedup. Newcache6 and newlist6 are slightly slower. Comparing table 4.1 and 4.3, we can see the new algorithm gets time savings for all

Function	atomix	mcslock1	dp12	newcache6	newlist6
Total States	2,966,400	12,782,802	13,811,712	1,342,089	3,619,561
Memory Used	157.00	930.00	74.00	1000.00	430.00
Insert States	701.35	1837.22	388.31	50.67	195.48
Total Time	1024.00	4772.86	432.55	2486.50	1246.98

Table 4.1: RAM Based Murphi, All Times in Seconds, Memory in MB

Function	atomix	mcslock1	dp12	newcache6	newlist6
Memory Used	80.00	320.00	32.00	200.00	82.00
File Read	6.54	51.93	3.15	25.07	23.91
File Write	0.81	5.00	0.50	4.77	2.43
Check Table	568.18	1841.39	134.06	123.97	316.98
Insert States	354.41	255.01	197.31	33.81	54.00
Total Time	1235.70	4726.04	373.19	2619.89	1392.53

Table 4.2: Stern's Disk Based Murphi, All Times in Seconds, Memory in MB

Function	atomix	mcslock1	dp12	newcache6	newlist6
Memory Used	82.00	290.00	32.00	200.00	82.00
File Read	5.43	34.48	1.78	15.61	13.95
File Write	16.50	325.62	5.57	106.34	103.74
Insert States	293.33	617.15	243.12	37.75	97.52
Computation	14.29	98.02	6.93	47.06	27.01
EnDequeue	40.78	214.41	17.50	73.76	64.49
Total Time	662.34	3756.81	310.03	2714.44	1290.17

Table 4.3: New Disk Based Murphi, All Times in Seconds, Memory in MB

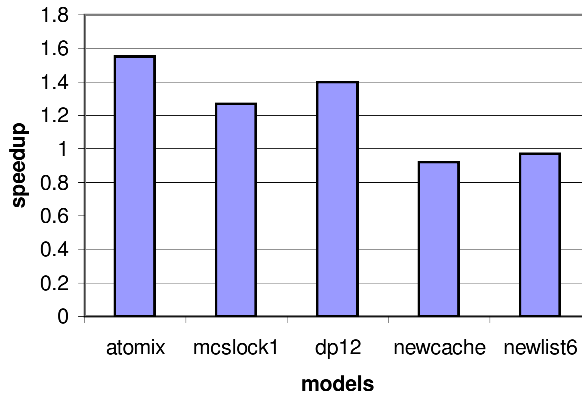

Figure 4.1: Speedup of New Algorithm over RAM Algorithm

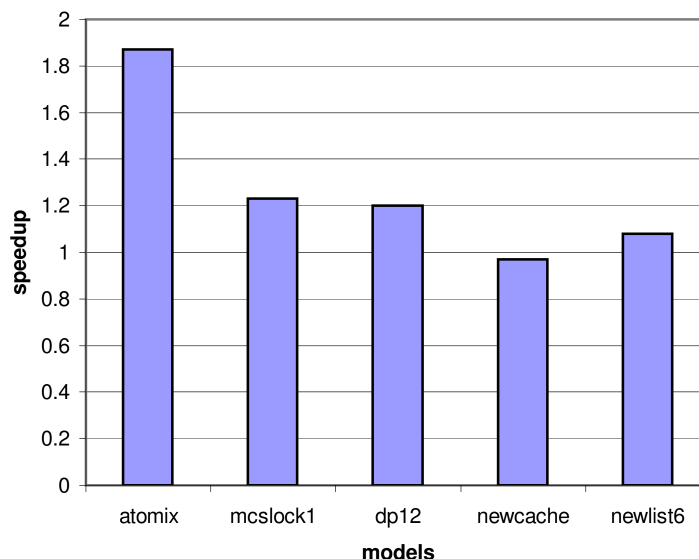

Figure 4.2: Speedup of New Algorithm over Stern's Algorithm

---

models in Insert States row. As described earlier by equation, the speedup obtained by the new algorithm depends on whether the time saving gained by hash insertion is bigger than the overhead introduced by the new algorithm. On average, the speedup of the new algorithm is 1.22. The memory used by the RAM algorithm is slightly bigger, like 10 - 20 MB, than the exact amount of memory it needs to complete the verification to ease up collisions. The memory used for the new algorithm is 20% - 50% of the memory used for RAM algorithm.

Figure 4.2 illustrates the speedup of the new algorithm over Stern's disk based algorithm. The new algorithm gets a speed up for all models except newcache6. As described earlier by equation, whether the new algorithm can get a speedup in this case depends on whether the Check Table time is bigger than the overhead introduced by the new algorithm. For example, in newcahce6, the file read/write and computation time overhead of the new algorithm outran the Check Table time overhead of Stern's algorithm. Thus, the new algorithm does not get a speed up in

Function	atomix	mcslock1	dp14	newcache6	newlist6
Memory Used	157.00	930.00	1900.00	1000.00	430.00
Insert States	32.41	142.93	248.16	74.63	52.53
Total Time	367.63	2831.28	717.40	2504.42	1092.25

Table 4.4: RAM Based Murphi, All Times in Seconds, All Memory in MB

Function	atomix	mcslock1	dp14	newcache6	newlist6
Memory Used	80.00	320.00	290.00	200.00	82.00
File Read	6.24	50.05	21.77	25.09	23.95
File Write	0.81	4.91	4.31	4.74	2.50
Check Table	60.95	841.86	459.44	188.62	290.67
Insert States	33.16	104.40	251.86	72.11	52.68
Total Time	423.27	3780.75	1109.74	2976.28	1397.59

Table 4.5: Stern's Disk Based Murphi, All Times in Seconds, All Memory in MB

this model. On average, the speed up of the new algorithm is 1.27 over Stern and Dill's algorithm. The memory used for both algorithms is the same.

4.2 Using SPIN's Hash Function

In this section, we report test results using a SPIN's s-hash-jenkins hash function to demonstrate the importance of good hash performance. Table 4.4 indicates the results from the RAM algorithm. Table 4.5 indicates the results from Stern and Dill's algorithm. Table 4.6 indicates the results from the New algorithm.

Figure 4.3 shows the slowdown of the new algorithm over RAM based algorithm. From Table 4.3 and 4.5, we can see that the new algorithm slows down compared with

Function	atomix	mcslock1	dp14	newcache6	newlist6
Memory Used	82.00	290.00	200.00	200.00	82.00
File Read	8.01	36.58	13.21	17.09	16.94
File Write	19.19	318.57	85.98	115.31	98.08
Insert States	28.01	94.19	193.95	61.45	41.57
Computation	15.15	99.98	169.13	32.75	28.80
EnDequeue	50.10	190.00	371.00	77.12	54.79
Total Time	393.45	3349.95	872.79	2739.50	1227.56

Table 4.6: New Disk Based Murphi, All Times in Seconds, All Memory in MB

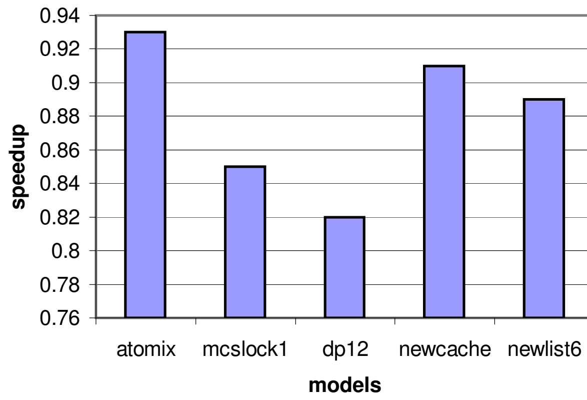

Figure 4.3: Slowdown of New Algorithm over RAM Algorithm

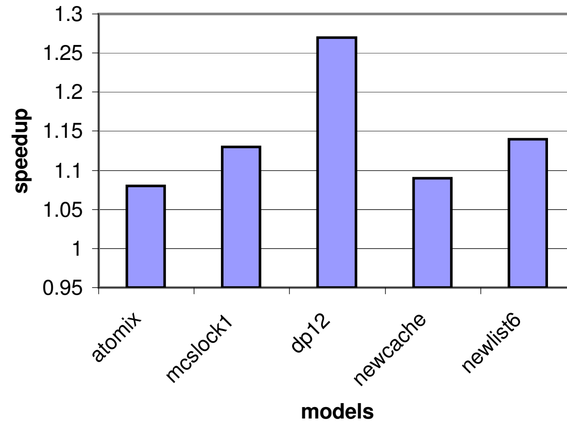

Figure 4.4: Speedup of New Algorithm over Stern's Algorithm

the RAM algorithm for all models. The SPIN's s-hash-jenkins hash function reduces hash collisions. Thus, the time savings gained by reducing hash insertion time is less than the overhead in the new algorithm. For bigger models, we believe the hash insertion time will increase in spite of the hash function, so that the new algorithm will show better results. However, since the memory size that can be allocated is restricted to 2GB for 32 bit machines, we are not able to do tests on bigger models. The avarage slow down here is 1.12.

Figure 4.4 shows the speedup of the new algorithm over Stern and Dill's algorithm. The average speed up here is 1.12. The new hash function decreases the Check Table time significanlty when compared with the results from the previous section, since the new hash function gives a faster disk versus memory comparison speed. Again, we believe the advantage of the new algorithm will be shown more obviously when the model gets bigger. In the next section, we present results from two bigger models which require more than 2GB of memory.

4.3 Big Models

Mathematical analysis in Chapter 3 predicts the new algorithm to have more speed up over Stern's algorithm when the size of model increases. In this section, we report results from two models that require more than 2GB of main memory to complete. We tested these two models on Stern's and the new disk based algorithm using SPIN's s-hash-jenkins hash function.

Table 4.7 reports test results. The Models column shows the model that is used. The States column presents the number of states that are generated. The Memory column reports the size of memory needed for the hash table and the queue. The RAM column presents the size of memory allocated for the hash table and the queue. The Stern Time column shows the time spent on Stern and Dill algorithm. The New Time shows the time spent on new algorithm. The Dense model gets a 1.92 speed up and newlist6 gets a 1.12 speed up over the Stern and Dill's algorithm.

Models	States	Memory	RAM	Stern Alg	New Alg
dense	134,217,728	2560.00	800.00	8708.44	4519.16
newlist6	80,109,363	9000.00	1300.00	54113.17	47593.36

Table 4.7: All Times in Seconds, All Memory in MB

Chapter 5

Conclusions and Future Work

This chapter gives conclusions from the above analysis and experimental results, and presents limitations of the new algorithm and suggests possible future work.

5.1 Conclusion

This thesis presents a new disk based model checking algorithm that uses a common parallel model checking algorithm serially. It partitions the state space into different files, then swaps each file into and out of memory to store the expanded states. Compared with Stern and Dill's disk based algorithm, the new disk based algorithm reduces the disk versus memory comparsion but increases disk i/o time. Compared with the RAM only algorithm, the new algoritm reduces hash table insertion time by reducing the cost and growth of hash load. Experimental results show the new algoirthm can get close to or faster verification speed than the RAM based algorithm. It also outperforms Stern and Dill's disk based algorithm by more than 10% in our benchmarks. This number increases as the size of the model increases. The size of the model that can be verified by the new algorithm is bound to the available disk size instead of being bound to the available RAM size. This work provides an effective reduction to the state space explosion problem with a low cost. It fully utilizes a single computer's resources to complete large model checking problems.

5.2 Future Work

This work does not address the following issues:

- The individual file size must not exceed the size of memory throughout the verification. If the file size exceeds the memory size, the algorithm crashes.
- This algorithm can not eliminate duplicate states when it stores states in the memory queue or disk queue. Duplicate states take extra disk space.

Exploring the relationship among the number of partitions, the memory queue size allocated to each partiton, the disk file size, the characteristics of the model, and the partitioning hash functions would be an interesting future work for this algorithm. In this thesis, we choose random values for each of the above parameters. With careful analysis, it is possible to find a formula that can find optimal values for the above paramters that results in a faster verification speed.

Exploring the relationship between the two partitioning functions needs to be examined more closely. Well coordinated partitioning functions are able to effectively reduce the hash load of the memory hash table hence speed up the insertion time. Incorporating hash compaction on this algorithm will further reduce the time spent on file I/O.

A shared memory multiprocessor architecture [11] can be applied to the new algorithm to speed up computation time to calculate the corresponding partition of each state and time overhead related to store duplicates states in queues.

LIST OF REFERENCES

[1] D. Peled, "Combining partial order reductions with on-the-fly model-checking," in *Computer Aided Verification*, Stanford, California, 1994, pp. 377–390.

[2] C. N. Ip and D. L. Dill, "Better verification through symmetry," in *Computer Hardware Description Languages and their Applications*, Ottawa, Canada, 1993, pp. 87–100.

[3] U. Stern and D. L. Dill, "Improved probabilistic verification by hash compaction," in *Correct Hardware Design and Verification Methods*, Stanford, California, 1995, pp. 206–224.

[4] W. Knottenbelt, M. Mestern, P. G. Harrison, and P. S. Kritzinger, "Probability, parallelism and the state space exploration problem," in *Computer Performance Evaluation: Modelling Techniques and Tools*, Palma, Spain, 1998, pp. 165–179.

[5] A. Bell, "Disk-based and distributed generation and analysis of large stochastic models," Schloss Dagstuhl, 2002.

[6] U. Stern and D. L. Dill, "Using magnatic disk instead of main memory in the murphi verifier," in *Computer Aided Verification*, Vancouver, Canada, 1998, pp. 172–183.

[7] G. D. Penna, B. Intrigila, E. Tronci, and M. V. Zilli, “Exploiting transition locality in the disk based murphi verifier,” in *Formal Methods in Computer-Aided Design*, Portland, Oregon, 2002, pp. 202–219.

[8] U. Stern and D. L. Dill, “Parallelizing the murphi verifier,” in *Computer Aided Verification*, Haifa, Israel, 1997, pp. 256–267.

[9] R. Kumar and E. Mercer, “Load balancing parallel explicit state model checking,” in *Parallel and Distributed Methods in Verification*, London, U.K., 2004, pp. 21–36.

[10] G. Behrmann, “A performance study of distributed timed automata reachability analysis,” in *Parallel and Distributed Methods in Verification*, Brno, Czech Republic, 2002, pp. 7–23.

[11] D. D. Deavours and W. H. Sanders, “An efficient disk-based tool for solving large markov models,” *Performance Evaluation*, vol. 33, no. 1, pp. 67–84, 1998.