

# Technical Report for Time-efficient Model Checking with Magnetic Disk

Tonglaga Bao, Michael Jones

Department of Computer Science  
Brigham Young University  
Provo, Utah, USA

Model checking with magnetic disk can significantly increase the space available for storing visited states. This technical report presents a new algorithm that uses magnetic disk instead of main memory to store state space. It also shows the experimental results of running several models on published algorithms that use magnetic disk and compares the results with the results of new algorithm.

From profiling published algorithms for model checking with magnetic disk, we find that delayed duplicated detection, but not file I/O, is the most time consuming part of these algorithms. Delayed duplicate detection is an extra processing step added to search algorithms that use magnetic disk to store visited states. The delayed duplicate detection step compares recently generated, and possibly new, states with a set of visited states to determine if the recently generated states are duplicates or not.

We propose a new algorithm for explicit model checking with magnetic disk that requires more file IO time but reduces duplicate detection time and reduces total execution time. Section 1 contains the new algorithm. Section 2 shows the experimental results from published and new algorithms. We also compare these experimental results with results of using only internal memory when there is enough memory available.

## 1 A New Algorithm Based on Hash Table Partitioning

The new algorithm uses a memory hash table, an array of queues in memory, an array of queues on disk and an array of disk files. Each of the queues and disk files correspond to one partition in the partitioned hash table. Disk files are swapped into and out of the memory hash table. Duplicate detection occurs only with the current queue and file in memory. States bound for other disk files are stored in the array of queues and processed later.

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. Continue generating states until either memory is full or the queue is empty. Then move the states in memory to disk. Finally, choose the queue with the largest number of states again, load the corresponding disk file into memory and repeat the process.

```

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

```

**Fig. 1.** PART: An algorithm for model checking with magnetic disk that uses a partitioned hash table.

Figure 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 both unexplored and explored states, disk queues also store both unexplored and explored states after 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 queue and memory queue gets cleared. (lines 4-11 ). Here,  $q_i^m$  indicates the memory queue belongs to partition  $i$  and  $q_i^d$  indicates the disk queue belongs to partition  $i$ . The function then selects the queue,  $i$ , with the most states as the active queue and calls the Select function. (lines 12-13).

The Select function loads the disk file that corresponds to the active queue into memory (line 19). Next it dequeues states from the active queue to generate every successor of the states in the queue (line 20). 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 21-22). 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 (line 26). After both the memory queue and the disk queue are empty, the table of expanded states are stored back to disk (line 28). The algorithm then chooses the next longest queue (line 29), loads the corresponding table and continues the exploration. If all the queues are empty, the algorithm terminates (line 30).

The Explore function checks to see if the successors of the states in the active queue belong to the current queue. 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 36). 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 and clear the memory queue. (lines 37-39).

## 2 Experimental Results

Stern and Dill published the first explicit state enumeration algorithm for model checking with magnetic disk. We refer this as MONO algorithm. Della Penna and others published a second algorithm for model checking with magnetic disk that is a modification of the MONO algorithm. we refer this as LOCAL algorithm. We call the new algorithm as PART algorithm.

### 2.1 Double Hash with Hash Compaction

Table 1 and Table 2 contain the results of profiling experiments from published algorithms. File Read/Write rows give the time spent on reading states from disk and writing states to disk. CheckTable row gives the time spent on doing delayed duplicate detection. Time for Insert row gives the time spent on inserting states into memory hash table. StateGeneration row gives the time spent on

Function	atomix	mcslock1	newlist6	dense	atomix2
File Read	12.4	49.3	14.7	144.9	26.0
File Write	0.1	0.3	0.1	2.1	0.1
CheckTable	19300.8	16750.9	5202.0	14505.4	31911.8
Time for Insert	80.5	320.9	107.3	584.9	115.2
StateGeneration	273.4	2683.4	1036.9	344.5	377.2
Total Time	19654.7	19755.2	6346.2	15434.8	32404.2

**Table 1.** MONO algorithm, All times in seconds

Function	atomix	mcslock1	newlist6	dense	atomix2
File Read	637.7	899.2	279.1	427.5	1604.8
File Write	64.9	215.8	49.3	357.4	101.7
CheckTable	3214.7	4611.8	1339.2	2213.4	7745.86
Time for Insert	163.7	279.2	67.5	499.2	257.1
StateGeneration	1159.0	4639.0	1601.9	541.5	1329.5
Total Time	5240.0	10645.0	3337.0	4039.0	11039.0

**Table 2.** LOCAL algorithm, All times in seconds

generating states and their successors. Total Time row gives the total time spent on verification. Obviously, the **CheckTable** time is the most time consuming part of both algorithms.

## 2.2 Chained Hash without Hash Compaction

This section reports test results using a chained hashtable without hash compaction. We report the time spent on the most time consuming parts of each algorithm except the time spent on generating states. The memory allocated for disk algorithms are 5% of the memory allocated for RAM algorithm.

Table 3 shows the results for RAM only Murphi (given just enough memory to complete the verification). The Total States row shows the total number of states generated for each model. The Insert States row shows the time spent on inserting states into memory. The Total Time row reports total time spent on verification.

Table 4 and Table 5 presents the result from MONO and LOCAL disk based algorithm. 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 **CheckTable** reports the time spent on delayed duplicate detection. The Insert States row shows the time spent on inserting states into memory.

Table 6 shows the result from PART algorithm. The Computation Time row reports the computation time spent on calculating each state to gets its corresponding partition number. The EnDequeue row reports time spent on enqueueing

and dequeuing states from queues. The other rows have same meaning as Table 4 and 5. Note that it gets faster insertion speed than RAM algorithm. It outperforms the other two disk algorithms since the time saving it gets from eliminating CheckTable time is bigger than the new overhead introduced to it.

Function	atomix	mcslock1	newlist6	dense	atomix2
Total States	2,966,400	12,782,802	3,619,561	35,831,808	4,112,160
Insert States	36.75	118.43	42.18	257.1	54.02
Total Time	282.4	2553.2	1011.4	344.92	386.7

**Table 3.** RAM without Hash Compaction, All Times in Seconds

Function	atomix	mcslock1	newlist6	dense	atomix2
File Read	18.0	86.5	40.2	143.4	28.3
File Write	0.2	1.6	0.7	2.0	0.4
CheckTable	394.8	1517.0	579.5	3719.8	633.4
Insert States	36.96	110.7	43.13	247.5	53.0
Total Time	705.8	4097.1	1588.8	4038.9	1027.4

**Table 4.** MONO without Hash Compaction, All Times in Seconds

Function	atomix	mcslock1	newlist6	dense	atomix2
File Read	427.5	696.0	142.0	492.2	522.4
File Write	46.1	106.4	27.6	198.5	47.9
CheckTable	1691.6	2300.9	693.2	2055.6	2103.5
Insert States	86.1	147.0	51.9	215.0	113.2
Total Time	1471.1	4812.4	1954.8	1722.0	2365.0

**Table 5.** LOCAL without Hash Compaction, All Times in Seconds

### 2.3 Chained Hash with Hash Compaction

This section reports the results of chained hash with hash compaction, We report the total verification time for each algorithm when the allocated memory becomes smaller. Note that the New algorithm gets more and more speed up over the other two disk based algorithms while the allocated memory becomes

Function	atomix	mcslock1	newlist6	dense	atomix2
File Read	0.9	827.4	25.1	960.5	2.47
File Write	2.4	51.8	102.3	37.9	2.12
Insert States	34.1	107.1	40.1	231.7	52.6
Computation	19.0	50.2	14.8	146.3	27.7
EnDequeue	63.3	180.8	49.9	565.7	158.3
Total Time	316.0	3238.8	1185.0	1217.0	499.1

**Table 6.** PART without Hash Compaction, All Times in Seconds

smaller. Table 7, Table 8, Table 9 and Table 10 report time spent on each algorithm when MONO, LOCAL and PART uses 100%, 50%, 30%, 10% and 5% of memory allocated by RAM algorithm.

Function	atomix	mcslock1	newlist6	dense	atomix2
Total Time	281.2	2570.0	1031.6	322.86	379.8

**Table 7.** RAM with Hash Compaction, All Times in Seconds

Memory	atomix	mcslock1	newlist6	dense	atomix2
1.0	298.7	2722.0	1072.5	608.0	434.4
0.5	323.9	2792.7	1098.6	619.8	452.5
0.3	341.9	2923.8	1127.3	643.2	492.0
0.1	419.6	3342.1	1207.7	957.06	661.8
0.05	488.4	3798.8	1270.8	1678.9	872.8

**Table 8.** MONO with Hash Compaction, All Times in Seconds

## 2.4 Big Models

The model that can not be verified by RAM algorithm is interesting to disk based algorithm. 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 MONO, LOCAL and PART disk based algorithm using SPIN's s-hash-jenkins hash function. Hash compaction is used and 3% of required memory is allocated to each algorithm.

Table 11 shows the results. Models column indicates the name of the models. States column indicates the number of states generated. MONO, LOCAL and

Memory	atomix	mcslock1	newlist6	dense	atomix2
1.0	899.3	4390.5	1821.9	1640.0	1460.0
0.5	1051.2	4970.3	1887.7	2468.0	1660.0
0.3	1067.1	5114.8	1834.1	2530.0	1786.0
0.1	1741.8	5616.6	1856.4	2625.0	3343.0
0.05	2599.3	6616.0	2070.4	2649.0	6081.00

**Table 9.** LOCAL with Hash Compaction, All Times in Seconds

Memory	atomix	mcslock1	newlist6	dense	atomix2
1.0	299.0	2940.7	1126.3	522.2	461.7
0.5	297.4	2774.0	1105.6	627.06	480.6
0.3	295.1	2739.7	1098.1	486.8	474.6
0.1	330.6	2706.8	1104.8	1040.4	494.4
0.05	308.6	2713.1	1043.0	2611.1	545.8

**Table 10.** PART with Hash Compaction, All Times in Seconds

Models	States	MONO	LOCAL	PART
6-peterson	382,513,749	31890.0	83207.8	16218.7
mcslock2	666,254,155	188,129.5	228,271.0	69,726.9

**Table 11.** Verification Times in Seconds

PART report total verification time spent on MONO, LOCAL and PART algorithm respectively. The new algorithm is 2.0 and 2.7 times faster than the MONO algorithm and 5.13 and 3.27 times faster than the LOCAL algorithm.