Time-Efficient Model Checking with Magnetic Disk

Tonglaga Bao and Michael Jones

Dept. of Computer Science, Brigham Young U., Provo, Utah, USA
{tonga, jones}@cs.byu.edu

Abstract. Explicit model checking with magnetic disk is prohibitively slow if file input/output (IO) is not carefully managed. We give an empirical analysis of the two published algorithms for model checking with magnetic disk and show that both algorithms minimize file IO time but are dominated by delayed duplicate detection time (which is required to avoid regenerating parts of the transition graph). We present and analyze a more *time*-efficient algorithm for model checking with magnetic disk that requires more file IO time, but less delayed duplicate detection time and less total execution time. The new algorithm is a variant of parallel partitioned hash table algorithms and uses a time-efficient chained hash table implementation.

Model checking with magnetic disk can significantly increase the space available for storing visited states. In explicit model checking, visited states are stored to avoid generating duplicate states and to aid in termination detection. In this paper, we analyze the performance of the two published model checking algorithms for use with magnetic disk and find that, while file IO is an overhead in algorithms that use disk, delayed duplicate detection is the single largest overhead. We propose a new algorithm for explicit model checking with magnetic disk that requires more file IO time but reduces duplicate detection time and total execution time. The new algorithm solves large model checking problems in less time than other disk-based algorithms and solves small problems in 15% to 27% of the time required by the RAM-only algorithm.

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 states with a set of visited states. The purpose of this comparison is to determine if the recently generated states are duplicates of visited states or not. The set of already visited states is called the *visited candidate set* and the set of new states is called the *duplicate candidate set*. Each state in the duplicate candidate set may have a different visited candidate set. During delayed duplicate detection, each state in the duplicate candidate set is compared with the states in its visited candidate set. The cost of delayed duplicate detection is a multiple of the product of the size of the visited candidate set and the average size of the delayed candidate sets. Reducing the size of either candidate set reduces the cost of delayed duplicate detection.

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 526–540, 2005.

Korf [5] and Zhou [13] give a more thorough discussion of the role of delayed and immediate duplicate detection in search using magnetic disk in the context of artificial intelligence. Zhou's algorithm eliminates the delayed duplicate detection step for search problems that satisfy a strict locality requirement. The locality requirement is that all successors for a group of states to fall within a subset of the entire state space. The resulting search algorithm is as fast as RAM-only search for some problems and actually faster than the RAM-only algorithm for other problems (due to cache effects). Unfortunately, transition graphs often encountered in model checking do not satisfy the strict locality requirement. However, a similar focus on delayed duplicate detection in explicit model checking can yield similar positive results.

Published algorithms for explicit model checking with magnetic disk have focused on minimizing file IO time. File IO time is the time spent reading states from and writing states to magnetic disk. While these algorithms have successfully minimized file IO time, delayed duplicate detection time for these algorithms is actually much greater than file IO time. We propose two methods for reducing delayed duplicate detection time: a partitioned hash table search algorithm and a chained hash table data structure. Both methods require more space and less time than algorithms and data structures in the published methods for model checking with magnetic disk. However, such a trade off is well suited for model checking with large, but relatively slow, magnetic disks.

Stern and Dill published the first results for an explicit state enumeration algorithm for model checking with magnetic disk [11]. The Stern and Dill algorithm is inspired by an Roscoe's earlier algorithm for model checking CSP using magnetic disk [10]. The Stern and Dill algorithm, which we will refer to as the MONO algorithm, stores the visited candidate set in a large monolithic hash table on disk and keeps a smaller table of duplicate candidates in RAM. This algorithm was originally implemented using an open-address hash table for the delayed candidate set. In this implementation of this algorithm, the entire table on disk is the visited candidate set for each duplicate candidate state in RAM. The Roscoe algorithm also stores visited states on disk and delays duplicate detection, but the algorithm performs duplicate detection by sorting then merging duplicate and visited candidate sets (rather than performing a pairwise comparison as in MONO). Stern and Dill point out [11] that the sort and merge step is, perhaps, unduly complicated because it requires sorting a list that will not fit in RAM. The cost of delayed duplicate detection in the MONO algorithm grows quickly with the size of the visited candidate set because detecting duplicates requires searching the entire visited candidate set for each duplicate candidate.

Della Penna et al. published an algorithm for model checking with magnetic disk that is a modification of the MONO algorithm [7]. This algorithm, which we will refer to as the LOCAL algorithm, exploits transition locality by reading only the most recently written states from disk during delayed duplicate detection. The motivation for the design of this algorithm is to improve efficiency by reducing file IO. The algorithm is indeed faster than the MONO algorithm. Our empirical analysis of both algorithms shows that exploiting locality in this

manner actually increases file IO time while greatly decreasing delayed duplicate detection time. Delayed duplicate detection time in the LOCAL algorithm is decreased because the visited candidate set is reduced to just the most recently written states rather than all visited states (as in the MONO algorithm). File IO is increased because some duplicates are not detected and their successors, which are also duplicates, are repeatedly transferred to and from disk. The LOCAL algorithm occasionally uses all visited states in duplicate detection to reduce the number and impact of missed duplicates.

In this paper, we reduce delayed duplicate detection time in a new model checking algorithm, called the PART algorithm, by reducing the visited candidate set with a partitioned hash table and reducing the duplicate candidate set with a chained hash table. While use of a chained hash table is perhaps the least interesting aspect of the new algorithm, it is responsible for the majority of the performance improvement. We reimplement the MONO and LOCAL algorithms using a chained hash table and compare the performance of both the MONO and LOCAL algorithms using either implementation. This allows us to focus on the algorithms rather than hash table implementations. As expected, both algorithms are faster when implemented with a chained hash table. However, the MONO algorithm becomes faster than the LOCAL algorithm when both are implemented with a chained hash table.

The second feature of the PART algorithm is the use of a partitioned hash table. The partitioned hash table is composed of n individual tables, each of which store a fraction of the visited states. A secondary hash function is used to divide states into partitions. The partitioned hash table improves delayed duplicate detection by reducing the visited candidate set. The partitioned hash table does increase file IO requirements because states must be read from and written to disk more frequently than the other algorithms.

Unlike switching to a chained hash table, switching to a partitioned hash table requires changing the state exploration algorithm. The most significant impact is that the usual double depth first search (DFS) used in linear temporal logic (LTL) model checking will miss property violations if naively implemented on a partitioned hash table. The state generation algorithm used in the PART algorithm is based on partitioned hash table algorithms used in parallel model checking and is not designed to detect violations of LTL properties. LTL model checking algorithms for partitioned hash tables require a serialization step to properly order the second DFS. This is costly for parallel algorithms, but should have little impact on the PART algorithm which is an inherently serial algorithm. The investigation of LTL model checking with magnetic disk is left for future work.

In the next section, we profile the MONO and LOCAL algorithms to show that delayed duplicate detection, not file IO, is the dominant overhead in both algorithms. Section 2 presents and discusses the PART algorithm. Section 3 contains an analytical comparison of the PART algorithm to the RAM-only, MONO and LOCAL algorithms. We give experimental results in section 4 and offer conclusions and future work in section 5.

1 Profiling Existing Uses of Disk

The original algorithm for model checking with magnetic disk was proposed by Stern and Dill [12]. Stern's algorithm, MONO, strives to minimize file IO time. Simply storing visited states on disk and performing immediate duplicate detection requires a disk access for every state generated. This creates a series of small disk accesses with poor locality which defeats caching and buffering techniques used to minimize average latency. The MONO algorithm minimizes file IO time by delaying duplicate state detection and writing each visited state to disk exactly once. The MONO algorithm maintains two sets of visited states[1] and a queue of states. One set stores the signatures of expanded states on magnetic disk and the other stores the signatures of expanded sets in RAM. The search is conducted breadth-first.

Delayed duplicate detection occurs when either a level of breadth-first search is finished or the table in memory becomes full. Delayed duplicate detection sequentially checks whether or not every state in the table on disk is contained in the table in RAM. If a state in the table in RAM is also in the table on magnetic disk, then that state in table is deleted from RAM. After the duplicate states are deleted, the remaining newly visited states are appended to the visited states on disk. The search then continues until either the disk becomes full or the queue becomes empty.

The more recent LOCAL algorithm proposed by Della Penna et al. improves on the execution time of Stern's algorithm by exploiting transition locality [7]. The LOCAL algorithm divides the table of states on disk into blocks. Rather than loading and comparing every state in the disk table during delayed duplicate detection, the LOCAL algorithm loads and compares states in only the most recently stored blocks. To avoid extensive duplication of previous work, older blocks are occasionally loaded for comparison.

Table 1. Total verification time, in seconds, for five verification problems. Both algorithms are implemented with an open-address hash table

Algorithm	atomix	mcslock1	newlist6	dense	atomix2
MONO	19654.7	19755.2	6346.2	15434.8	32404.2
LOCAL	5240.0	10645.0	3337.0	4039.0	11039.0

We have reimplemented and profiled the MONO and LOCAL algorithms in Hopper [4]. Both algorithms were profiled on a collection of five model checking problems. We measured wall clock time for disk access, performing delayed duplicate detection, inserting states into the table in RAM, generating successor states and the total from start to finish. Wall clock time, which is the amount of time that passes for "a clock on the wall" during program execution, is reported

[1] They are actually hash signatures created using hash compaction, but the difference is not relevant.

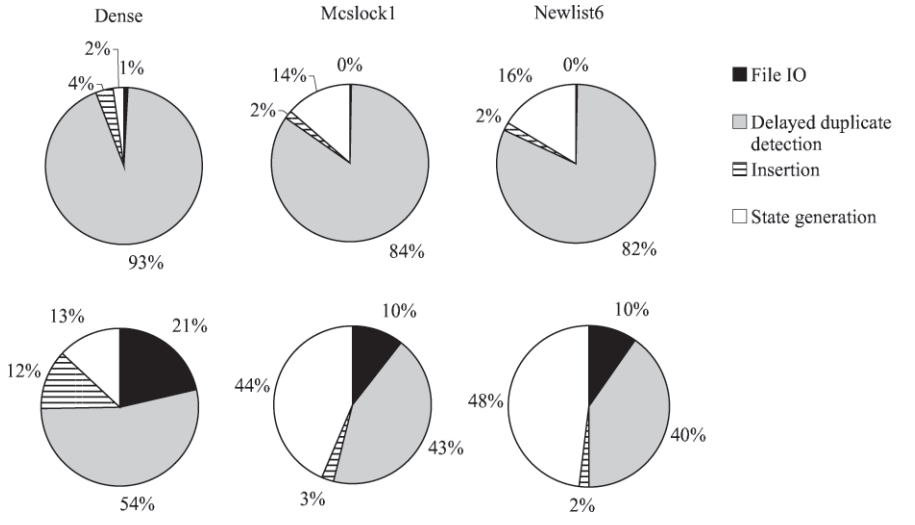

Fig. 1. Distribution of time between file IO, delayed duplicate detection, inserting into the RAM hash table and state generation in the MONO and LOCAL algorithm given 5% of the total RAM needed to complete each problem. Results for the MONO algorithm are on the top row and results for the LOCAL algorithm are on the bottom row

because it includes the time that a process is idle while waiting for file IO. All experiments were completed on an otherwise idle server with an Intel Pentium 4, 2.4 Ghz processor, 2 GB of RAM and a 42 GB SCSI hard drive.

Table 1 gives the total verification time for five problems and figure 1 shows the distribution of time between file IO, delayed duplicate detection, insertion and state generation for three of the five problems. The remaining two are similar, but ommited for brevity. See [1] for the full results of these and all other experiments in this paper. The LOCAL algorithm completes all problems in one-half to one-fourth of the time as the MONO algorithm. These results are comparable to the results in the original paper describing the LOCAL algorithm. In our profiling, the LOCAL algorithm is faster due to a reduction in the delayed duplicate detection time, not reduced file IO time.

2 A New Algorithm Based on Hash Table Partitioning

The analysis of MONO and LOCAL disk-based algorithm in the last section suggest that the performance of model checking algorithms that use magnetic disk can be improved by reducing state generation and delayed duplicate detection time, rather than file IO or even hash table insertion time. In this paper, we focus on improving delayed duplicate detection. State generation is a significant factor, but generic model checker implementation details, like how the formal description of the transition system is translated into an executable format, play

a critical role in state generation time and are not generally applicable to different implementations of executable translations of formal protocol descriptions. We propose the PART algorithm to reduce delayed duplicate detection time in model checking with magnetic disk by

1. Using partitioned hash tables to reduce the cost of delayed duplicate detection. Partitioned hash tables have been used in distributed model checking [11] for some time. Rather than store each partition on a separate computing node, we store all but one of the partitions on disk and the remaining partition in RAM. Partitioning the hash table reduces the size of the set of states which must be searched during delayed duplicate detection.
2. Using a chained hash table rather than an open-address hash table. Previously, the hash table was implemented using an open-address table with double hashing, presumably to avoid the overhead of storing pointers and swapping them between disk and memory. Instead, we use a chained hash table which further reduces the size of the candidate set for delayed duplicate detection by requiring a state to be in a particular bucket or chain rather than allowing a state to be stored anywhere in the table. Finally, we use separate hash functions to partition and store states. This is a method, common to other parallel search problem domains, which further reduces penalties associated with increasing hash loads.

The pseudocode for the PART algorithm is given in figure 2. The algorithm uses a Partition function to map every state to a unique memory queue. There is the same number of disk files and memory queues. Memory queues store both unexplored and explored states; disk queues store states that overflow from the memory queues. The disk files store the explored candidate sets.

The search begins when 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 is cleared (lines 4-6). In the pseudocode, q_i^m indicates the memory queue belongs to partition i and q_i^d indicates the disk queue belongs to partition i. The Search function then selects the queue, i , with the most states as the active queue and calls the Select function (lines 7-8).

The Select function loads the disk file that corresponds to the active queue into memory (line 12). Next, the function dequeues states from the active queue. If the newly dequeued states are not already in the table of visited states, then the states are stored in memory and every successor is generated (line 13-14). When the active queue in memory becomes empty, the corresponding disk queue, if it exists, is loaded into memory (line 15). After both the memory queue and the disk queue are empty, the table of expanded states is stored back to disk (line 17). The algorithm then chooses the next longest queue (line 18), loads the corresponding table and continues exploration. If all the queues are empty, the algorithm terminates (line 19).

The Explore function checks to see if the successors of the states in the active queue belong to the active queue. If they do, and they are not present in the current table in memory, then they are added to the current active queue. If they

```
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 := Partition(s_0); insert s_0 into q_i^m;
6      if Full(q_i^m) then store q_i^m in q_i^d; q_i^m := ∅;
7    i := max_{i∈n}(|q_i^m + q_i^d|);
8    Select(i);
9  Select(i: int)
10   while i ≥ 0 do
11       while q_i^m ≠ ∅ do
12         load D[i] into M;
13         s = dequeue(q_i^m);
14         if s is not in M then insert s in M; Explore (i, s);
15       if q_i^d ≠ ∅ then load q_i^d to q_i^m;
16     while q_i^m ≠ ∅;
17     store M in D[i].
18     i := max_{i∈n}(|q_i^m + q_i^d|);
19     if |q_i^m + q_i^d| = 0 then i = −1;
20 Explore(i: int, s: state)
21   for all s' ∈ successors(s) do
22     i':= Partition(s');
23     if i' = i and s' is not in M then insert s' in q_i^m; else insert s' in q_i'^m;
24     if Full(q_i'^m) then store q_i'^m in q_i'^d; q_i'^m := ∅;
```

Fig. 2. The PART algorithm for model checking with magnetic disk

do not belong to the current queue, then they are stored to their corresponding queues (line 23). This allows duplicate and expanded states to be stored in the work queues. If any of the queues are full, then they are written to disk (line-24).

3 Comparative Analysis

First we compare the computation time of new algorithm with the usual RAM-only model checking algorithm. Then we compare the new algorithm with the MONO and LOCAL algorithms for model checking with magnetic disk. The analysis clarifies sources and costs of overheads associated with the PART algorithm.

3.1 Comparison with the RAM-Only Algorithm

In the results presented later, the new algorithm for model checking with magnetic disk requires up to 27% more computational time than the RAM-only algorithm–for large models, when only 5% of RAM memory required by the RAM-only algorithm is used.

Verification time in the RAM-only algorithm is composed of the following parts:

$$Total_{RAM} = Insertion_{RAM} + StateGeneration$$

Where $Insertion_{RAM}$ is the time spent on inserting newly generated states into memory and *StateGeneration* is the time spent on the generation of successor states.

For the PART algorithm verification time consists of:

$$Total_{\text{PART}} = Insertion_{\text{PART}} + StateGeneration + IO_{\text{PART}} + Computation + EnqueueDequeue$$

Where $Insertion_{\text{PART}}$ is the time spent on inserting states into memory, IO_{PART} is the time spent reading the states from disk to memory and writing the states from memory to disk, *Computation* is the time spent on computing each state's corresponding partition and *EnqueueDequeue* is the time overhead related to storing states in the queues.

For the PART algorithm, $Insertion_{\text{PART}}$ indicates delayed duplicate detection because the algorithm detects duplicates by attempting to insert them into the table of visited states. This value is similar to $Insertion_{RAM}$ since the number of reachable states, including duplicates, generated by both algorithms are the same. The new overhead introduced by PART algorithm is then:

$$IO_{\text{PART}} + Computation + EnqueueDequeue$$

The comparative performance of the PART algorithm can be improved by decreasing any of these values. The performance of the PART algorithm improves as the following conditions are met.

A small state vector, v, is used to minimize file IO time. Hash compaction reduces the size of the state vector which reduces the memory requirements for verification and also reduces the file IO overhead, IO_{PART} (the results in section 4 are generated using hash compaction).

The transition graph is partitioned so that states and their successors are often in the same partition. This reduces the number of duplicate states that are stored in the queues and hence reduces *EnqueueDequeue* time. We do not address the problem of partitioning in this paper, but the partitioning scheme proposed by Rangarajan and others for parallel partitioned model checking [8] can be applied here.

The algorithms are applied to transition graphs with low average in-degree because this reduces the *EnqueueDequeue* time associated with partitioning states and storing several copies of duplicate states for delayed duplication detection. This is a reasonable assumption in certain situations. More specifically, models of machine code tend to have low average in-degree while models of scheduling algorithms tend to have high average in-degree. Machine code models have low in-degree because very few instructions are the targets of multiple different branches. Scheduling algorithms tend to have higher in-degree because there are typically many different computational paths that lead to entry into a process–espeically if non-determinism is used to obtain a control abstraction.

Pelánek's study of structural properties of state spaces concludes that industrial problems tend to have low average in-degree [6]. This suggests that algorithms that perform well on models with low average in-degree can be expected

to perform well on most industrial models. In the study, Pelánek takes industrial models to be the non-trivial models distributed with the SPIN, Murphi, CADP and μCRL model checkers. The *clustering coefficient* used in Pelánek's study measures how many edges in a k-vertex neighborhood remain in that neighborhood. High clustering coefficients correspond to high in-degrees.[2] Pelánek concludes that the sampled industrial models have low clustering coefficients while "toy" examples tend to have high clustering coefficients. The dense problem in section 4 is a toy problem with a high clustering coefficient of nearly 7 and the PART algorithm performs poorly on this problem compared to the RAM-only algorithm.

3.2 Comparison with the Mono and Local Algorithms

The following analysis reveals why PART is usually faster. MONO must read each state from disk every time the memory table becomes full.When RAM is a small fraction of the total space needed, the memory table becomes full more often and thus MONO performs more reads.

The total time required by the MONO algorithm is

$$Total_{\text{MONO}} = Insertion_{\text{MONO}} + StateGeneration + IO_{\text{MONO}} + DDD_{\text{MONO}}$$

where $StateGeneration$ measures time spent generating states. $Insertion_{\text{MONO}}$ measures the time spent on inserting newly generated states into memory. IO_{MONO} is the disk IO time and DDD_{MONO} is delayed duplicate detection time spent on comparing states in disk with states in memory. The value of IO_{MONO} is less than IO_{PART}, but DDD_{MONO} is significantly greater than $Insertion_{\text{PART}}$.

The comparison time for the MONO algorithm depends on the implementation of the hash table used in memory. The MONO algorithm was originally implemented with an open address hash table in memory. The double hash implementation of an open address table is slower on average than a chained hash table because it requires more comparison time than a chained hash table. An open address hash table stores states directly in the table. A chained hash table is essentially a table of pointers to chains of states. The double hash implementation of an open address hash table uses a sequence of combinations of two hash values to probe the table for a given state. While the performance trade-offs between chained and open-address implementations are well-understood (see [2] for a thourough discussion), they are of interest in this work because their use in the MONO algorithm amplifies the differences. We make the relevant differences more precise with the following equations.

Assume there are K_i states on disk when the passed state file is read the ith time and assume that the disk file is read a total of t times during the entire verification process. The total number of states read from disk is $K = \sum_{i=1}^{t} K_i$.

[2] However, clustering coefficients include only edges from "local" nodes in the neighborhood; edges from outside the neighborhood are not counted and will only increase the in-degree.

Assume the table of states stored in memory contains M bytes. For the open-address implementation, MONO-open, the average comparisons required to do duplicate detection for a state on disk is $M/(2|v|)$ because the state may be anywhere in the table. Duplicate detection occurs for every state read from disk and K states are read from disk so:

$$DDD_{\text{MONO}-open} = \frac{CKM}{2|v|}$$

where C is the time required to compute the next hash value and compare two states. Note that K is often quite large and the product KM in $DDD_{\text{MONO}-open}$ is significant.

However, if the MONO algorithm is implemented with a chained hash table then DDD_{MONO} is reduced because duplicate candidates are confined to a particular chain rather than the entire hash table. This means that the K term is multiplied by the average chain length, $\bar{C}$, instead of the entire table size:

$$DDD_{\text{MONO}-Chained} = \frac{CK\bar{C}}{2|v|}.$$

The product of $K\bar{C}$ is much smaller than KM as K becomes large.

Although an open-address hash table requires less memory (one fewer pointer per state stored) than a chained hash table, the space savings is offset by the increase in delayed duplicate detection overhead. This design trade-off is contradictory given that the fundamental trade-off in model checking with magnetic disk is to *increase* space at the expense of *increasing* computation time.

Returning to the comparison of the PART and MONO algorithms, we will assume that the MONO algorithm is implemented with a chained hash table. To make the PART algorithm faster than the MONO algorithm, we only need to make sure that the sum of IO_{PART}, *Computation*, and *EnqueueDequeue* is less than or equal to the sum of IO_{MONO} and DDD_{MONO}. We ignore hash table insertion time because it is negligible. This is often the case because DDD_{MONO} is often much greater than any other overhead in the PART algorithm.

Similar analysis applies to the LOCAL algorithm; the only difference is that each K_i value is smaller because the algorithm considers only part of the disk table. However, the sum of the K_i terms may be bigger due to duplicate states.

When the RAM table is smaller, the PART algorithm becomes more efficient when compared with the MONO and LOCAL algorithms. For the MONO and LOCAL algorithms, the RAM table is full more often as RAM decreases and more invocations of the delayed duplicate detection process are required. Each of the additional delayed duplicate detection checks require traversing the set of all visited candidate states. For the PART algorithm, the number of partitions increases as RAM decreases and there are more swaps between disk file and memory. Since file IO time is significantly smaller than delayed duplicate detection time, the PART algorithm is comparatively more efficient with less RAM.

4 Experimental Results

This section presents the experimental results obtained by running the RAM-only, MONO, LOCAL and PART algorithms on several verification problems. 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 the smaller models to test all algorithms and use the bigger models to test only the algorithms that use magnetic disk. The RAM-only algorithm can not complete the larger models because 32-bit UNIX processes can address at most 2GB of memory. The verification problems used are atomix, mcslock1, newlist6, dense, atomix2, 6-peterson and mcslock2. Each of the models can be obtained at [3].

Most of the following results are reported using a chained rather than an open-address hash table. Executing Stern's MONO algorithm and Della Penna's LOCAL algorithm with chained, rather than open-address, tables increases the space requirements for both, but reduces their execution time.

4.1 Small Models

In this section, we report results for problems that require less than 2 GB of memory to store visited states. The pie charts in figure 3 demonstrate the reduction in delayed duplicate detection time when using a chained hash table in the MONO and PART algorithms. Table 2 gives results for five models using all algorithms, including the RAM-only algorithm. Figure 4 gives the slowdown of each algorithm for model checking with magnetic disk relative to the RAM-only algorithm as the amount of allowed RAM varies on the same five models.

Figure 3 gives the distribution of time in various parts of the algorithm for MONO and PART implemented with chained hash tables. Compared with the pie graph in figure 1, the MONO algorithm using a chained hash table significantly decreases delayed duplicate detection time to 30% and 17% on the Mcslock1 and Newlist6 problems. Duplicate detection time continues to dominate in the Dense problem. File IO and insertion take between 1% and 4% and the percentage of state generation is increased to 65% and 77% for to Mcslock1 and Newlist6 problems due to the decrease in delayed duplicate detection time.

For the PART algorithm, delayed duplicate detection time is reflected in insertion time, which is comparable to insertion time for the other algorithms. The largest overhead introduced by the new algorithm is the enqueue/dequeue overhead which ranges from 6% to 53%. Duplicate states are detected before enqueueing them into the queues for the other three algorithms. However, states are all enqueued before doing duplicate detection in the PART algorithm. Since the number of duplicate states is usually much more than the number of unique states (particularly for the Dense problem), enqueue/dequeue takes a significant amount of time. The state generation section of the PART algorithm takes a greater percentage of total time, between 21% and 88%. This indicates an overall speed increase of PART algorithm since the time required for state generation is similar for both algorithms. The partitioning category measures the extra time

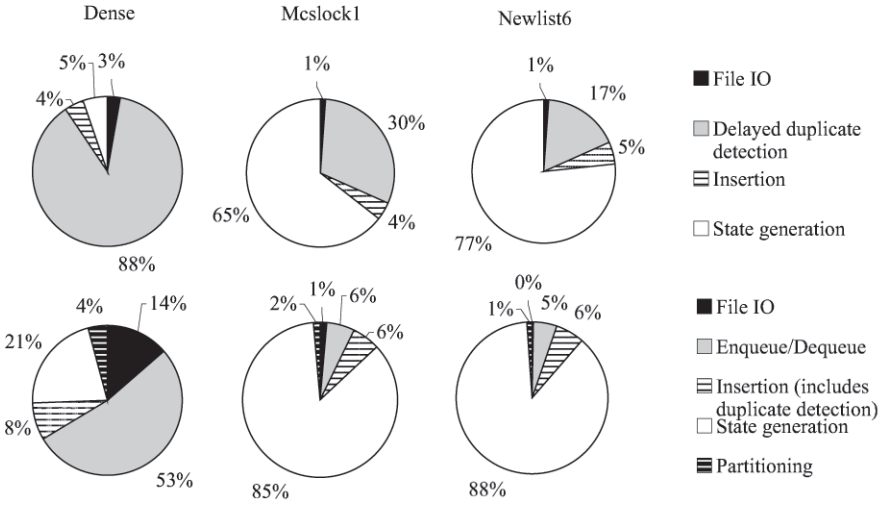

Fig. 3. Distribution of time between various parts of the MONO and PART algorithm given 5% of the total RAM needed to complete each problem. The MONO algorithm is on the top row and the PART algorithm is on the bottom row. Both algorithms are implemented with a chained hash table

required to partition states into hash tables. This new overhead accounts for only 1% to 4% of the total execution time.

Table 2 shows total verification time for all algorithms on five models. Included are the three models shown in figures 1 and 3. All algorithms except the RAM-only algorithm are allowed 5% of the total RAM necessary to complete the verification. The table shows a decrease in execution time for the PART algorithm compared to each of the the other disk based algorithms. Interestingly, the MONO algorithm benefits from chained hashing more than the LOCAL algorithm.

Figure 4 shows the average slowdown of the MONO, LOCAL and PART algorithms compared with the RAM-only algorithm. The average slowdown is

Table 2. Total verification times, in seconds, for five verification problems for several algorithm and hash table implementation combinations. The algorithms that use magnetic disk are given 5% of the total required RAM

Algorithm (hash table)	atomix	mcslock1	newlist6	dense	atomix2
RAM-only (chained)	281.2	2570.0	1031.6	322.9	379.8
PART(chained)	308.6	2713.1	1043.0	2611.1	545.8
MONO(chained)	488.4	3798.8	1270.8	5444.5	872.8
LOCAL(chained)	2599.3	6616.0	2070.4	3336.0	6081.0
LOCAL(open-address)	5240.0	10645.0	3337.0	4039.0	11039.0
MONO(open-address)	19654.7	19755.2	6346.2	15434.8	32404.2

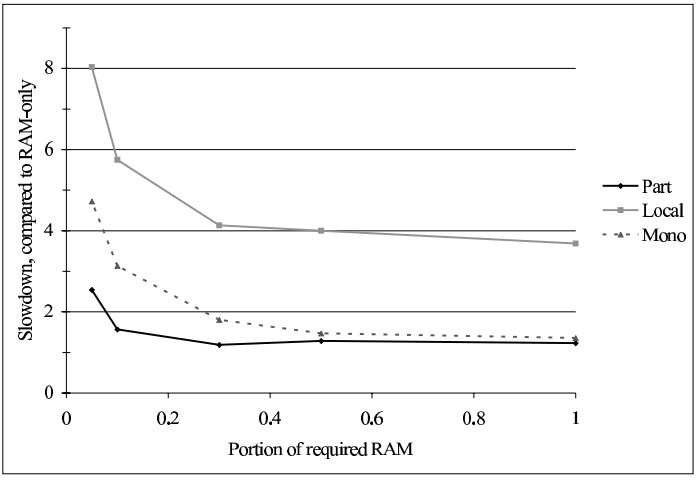

Fig. 4. Average slowdown, relative to the RAM-only algorithm, for all three algorithms for different portions of required RAM and averaged over five problems. Each algorithm is implemented with a chained hash table. A slowdown of x indicates that an algorithm is x times slower than the RAM-only algorithm

computed by dividing the running time of the algorithm by the running time of the RAM-only algorithm, and taking the average over the five problems in table 2. Given the same amount of memory as the RAM-only algorithm, the MONO and PART algorithms have similar slowdowns but LOCAL is the slowest. As the amount of available memory decreases (moving to the left on the x-axis in the figure), each algorithm experiences greater slowdown. The PART algorithm experiences the least slowdown and has a slowdown of 2.8 when given 1/20th of the memory used by the RAM-only algorithm.

4.2 Big Models

Results for verification problems that can not be verified by the RAM-only algorithm are particularly interesting for disk based algorithms. In this section, we report results for two problems that require 5 GB and 8 GB of space to store visited states. The peak storage space during verification grows to 31 GB due to large queues of states waiting to be expanded. We verified these two models using the MONO, LOCAL and PART algorithms. Hash compaction is used and 3% of space required to store visited states is allocated to each algorithm in RAM.

Table 3 shows the results. The PART algorithm is 2.0 and 2.7 times faster than the MONO algorithm and 5.1 and 3.3 times faster than the LOCAL algorithm. Because the LOCAL algorithm may miss duplicates and regenerate portions of the state space, the LOCAL algorithm actually stores 999 M visited states for the mcslock2 model.

Table 3. Number of reachable states and total verification time, in seconds, for two large models. Each algorithm is allowed approximately 3% of the RAM required to store all reachable states

Models	States	MONO	LOCAL	PART
6-peterson	382,513,749	31,890.0	83,207.8	16,218.7
mcslock2	666,254,155	188,129.5	228,271.0	69,726.9

The ldash6 model is even larger than any of the "big" models in table 3. None of the algorithms are able to generate all of the states in this model using 42 GB of disk. The incomplete results reveal some drawbacks of the PART algorithm. The MONO algorithm requires 14992 seconds to generate 6 million states of ldash6. The PART algorithm requires 22788 seconds to generate the same number of states. Although it in not clear which algorithm would finish first,[3] it is clear that MONO performs more efficiently in the first 6 million states. This is due to the 1020 byte state vector, which is larger the state vectors of all other models reported in this paper, used in the ldash6 model. Since PART algorithm allows duplicate states to be stored in queues and queues have to store the actual state, instead of just the hash compacted state, the disk read/write overhead for disk queues becomes worse in proportion with the state vector size.

5 Conclusion

Reducing delayed duplicate detection time, even at the expense of increasing file IO time, reduces the total execution time of model checking algorithms that use magnetic disk. Delayed duplicate detection time can be reduced by partitioning the table of visited states and by storing visited states in a chained, rather than an open-address, hash table. The resulting model checking algorithm is faster than other published algorithms for model checking with magnetic disk, even when the other algorithms are reimplemented with the faster chained hash table. The performance of the new algorithm degrades more slowly than that of other algorithms as the amount of available RAM decreases.

In the new partitioned hash table algorithm, duplicate states are stored in queues for delayed duplicate detection. While this incurs the same amount of duplicate detection work as required by a RAM-only algorithm, enqueueing and dequeueing duplicate states becomes the dominant overhead. Storing more than one partition in memory at a time may reduce the enqueue and dequeue overhead by allowing immediate duplicate detection between partitions. Ideally, one could store exactly the partitions that would allow the immediate detection of all duplicates, as in [13]. However, an approximation based on prior locality patterns may provide some improvement.

[3] Because the state generation rates vary with hash and disk loads. The state generation rate of the new algorithm also depends on how often partitions are sswapped.

In this paper, we focus on explicit model checking with magnetic disk. Ranjan et al. give an efficient algorithm for symbolic model checking with magnetic disk [9]. This algorithm constructs binary decision diagrams using an iterative breadth-first technique that localizes memory accesses. The key problem in symbolic model checking with magnetic disk is determining child-node variable indices rather than delayed duplicate detection, as addressed here.

References

1. T. Bao. Emperical comparison of algorithms for model checking with magnetic disk. Technical Report VV-0402, Dept. of Computer Science, Brigham Young U., 2004.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 11. McGraw-Hill, 1999.
3. Explicit model checking benchmarks. Available at vv.cs.byu.edu/emc, 2004.
4. M. D. Jones and E. Mercer. Explicit state model checking with Hopper. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, pages 146–150, Barcelona, Spain, March 2004. Springer.
5. R. E. Korf. Delayed duplicate detection: Extended abstract. In Georg Gottlob and Toby Walsh, editors, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03), Acapulco, Mexico*. Morgan Kaufmann, 2003.
6. R. Pelanek. Typical structural properties of state spaces. In *International SPIN Workshop on Software Model Checking (SPIN'04)*, number 2989 in LNCS, pages 5–23, Barcelona, Spain, March 2004. Springer.
7. G. Della Penna, B. Intrigila, E. Tronci, and M. Venturini Zilli. Exploiting transition locality in the disk based Murϕ verifier. In M. Aagaard and J. O'Leary, editors, *Formal Methods in Computer Aided Design*, volume 2517 of *LNCS*, pages 202–219. Springer-Verlag, 2002.
8. M. Rangarajan, S. Dajani-Brown, K. Schloegel, and D. Cofer. Analysis of distributed spin applied to industrial-scale models. In S. Graf and L. Mounier, editors, *11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume LNCS 2989 of *Lecture Notes in Computer Science*, 2004.
9. R. K. Ranjan, J. V. Sanghavi, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package by exploiting memory hierarchy. In *33rd Annual Conference on Design Automation (DAC'96)*, pages 635–641. ACM Press, June 1996.
10. A. W. Roscoe. Model-checking CSP. In *A Classical Mind: Essays in Honor of C. A. R. Hoare*, pages 353 – 378. Prentic-Hall, 1994.
11. Ulrich Stern and David L. Dill. Parallelizing the Murϕ verifier. In Orna Grumburg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267, Haifa, Israel, June 1997. Springer-Verlag.
12. Ulrich Stern and David L. Dill. Using magnetic disk instead of main memory in the Murϕ verifier. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.
13. R. Zhou and E. Hansen. Structured duplicate detection in external-memory graph search. In D. L. McGuinness and G. Ferguson, editors, *Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 677–682, San Jose, California, July 2004. AAAI Press / MIT Press.