Layered Duplicate Detection in External-Memory Model Checking

Peter Lamborn and Eric A. Hansen

Dept. of Computer Science and Eng.
Mississippi State University
Mississippi State, MS 39762
pcl16@msstate.edu, hansen@cse.msstate.edu

Abstract. This paper presents a disk-based explicit state model checking algorithm that uses an approach called layered duplicate detection. In this approach, states encountered during a breadth-first traversal of the graph of the transition system are stored in memory according to the layer of the graph in which they are first encountered. With this layered organization of memory, transition locality is exploited by checking only the most recent layers for duplicates. In RAM, exploiting transition locality in this way saves time. In external memory, it saves space. In addition, a layered structure allows an easy method of counterexample reconstruction in disk-based model checking. We prove a worst-case linear bound on the redundant work performed by our approach. Experimental results indicate that average case redundant work is much better than the worst-case. The implemented model checker has been used to verify a transition system that required more than 275 GBs of disk storage.

1 Introduction

Explicit state model checking is a methodology for verifying the properties of a system by a systematic search for error states in a state graph that represents the behavior of the system. An "on-the-fly" algorithm for searching the state graph begins with a queue that contains the start state, removes states from the queue in an order that depends on the search strategy (e.g., depth-first, breadth-first, or best-first), and generates the successors of each state and adds them to the queue. Generated states are also stored in a hash table that is used for duplicate detection, that is, for determining whether or not a newly-generated state is a duplicate of a previously-generated state before adding it to the queue. A complete search of the graph is needed to verify a system. Therefore, the memory required is proportional to the number of states in the model. Thus, memory is a bottleneck for model checking.

One solution is to use external memory. When RAM is full, states can be moved to an external memory device. Delayed duplicate detection (DDD) is an approach to external memory graph search that writes duplicates temporarily to disk and eventually eliminates them [1–3] Because disk is accessed most quickly

in a sequential manner, allowing temporary duplicates can save time compared to accessing disk randomly to eliminate the duplicates immediately. But DDD still incurs significant overhead.

When searched in a breadth first manner, most graphs exhibit transition locality [4]. Transition locality means that transitions tend to be in local levels of a breadth first visit [5]. In other words, newly-generated states are more likely to be duplicates of a state in a recent layer than a state in a more distant layer. We introduce *layered duplicate detection* as an approach to exploiting transition locality in eliminating duplicates in external-memory search. By layered duplicate detection, we mean storing the search graph in layers and only saving and checking the most recent layers for duplicates. This approach is useful both in deciding which states to store in a RAM cache, and in deciding which states to store on disk. We show that layered duplicate detection makes it possible to improve the running time of an external-memory model checker, as well as decrease the amount of disk storage it needs. When not all layers of the search graph are stored on disk, we use a divide-and-conquer approach to counterexample reconstruction and we show that this is facilitated by the layered organization of stored states that we adopt.

The paper is organized as follows. Section 2 gives an overview of relevant background and related work. Section 3 presents the new algorithm as an extension of the Murϕ model checker [1]. Section 4 reports experimental results. Section 5 concludes the paper and discusses potential future directions.

2 Background

There are many approaches to addressing the state explosion problem in model checking. Two of the most relevant approaches for this paper are partial storage methods and external memory model checking.

2.1 Cached Model Checking

Cached model checking is a method of model checking where the closed list is not maintained in its entirety. Instead, a partial cache of the closed list is used. Partial caching creates the possibility of searching parts of the state space more than once. To limit the amount of redundant searching, the algorithm must wisely choose what to keep in cache. Correct caching choices reduce the amount of redundant work occurring.

We distinguish three types of partial storage methods: Depth-first search (DFS) approaches, breadth-first search (BFS) without duplicates, and BFS with duplicates. A depth first cached search always has a complete counter example at the time of discovering an error state, but DFS is not guaranteed to find the shallowest error state [6]. BFS has the advantage of always finding the shallowest error, if an error exists. Also BFS takes advantage of the property of transition locality [4]. A disadvantage of Cached BFS is that counter examples may need to be reconstructed.

Depth-first approaches Depth first search explores a branch as far as possible before backtracking. Cached DFS methods allow duplicates, but use various method to limit the number of duplicate states. Geldenhuys et al. [7] were able to bound the number of duplicates by caching all states in certain 'strata' based on distance from the start state. Godefroid et al. [8] preferentially cached newer states, in an attempt to exploit something like transition locality, though it is not clear if transition locality is observed in DFS. Alternatively stateless model checking [9] uses complicated pruning mechanisms to prune duplicate states.

Breadth-first without duplicates There are breadth-first methods that delete states in such a way that no duplicates are allowed. These searches use various properties that exist in certain graphs to determine what states are necessary to cache. Kristensen and Mailund exploit ordering in models where states can be ordered, either with a total ordering [10] or a partial ordering [11]. Parshkevov and Yantchev [12] searched graphs where all states have the same number of parent states, removing states when all parents had been observed. Frontier search exploits the fact that in some graphs duplicates may be guaranteed to only exist in the most recent L layers [3, 13]. Whatever the graph property exploited, states can be removed from the cache without risking duplicate states. These methods are domain dependent because not all graphs exhibit these properties.

Breadth-first with duplicates Breadth-first search with duplicates is are easy to implement in a domain independent manner. These methods allow some duplicates but use various methods to limit the number of duplicates that occur.

Hash Collisions Tronci et al. attempted to maintain a cache of only the most local states in memory [5]. In Tronci's approach the state cache is maintained in memory as a hash table. When two states (s, s') collide in the hash table, the older state (s) is removed and the newer state (s') is remembered. This replacement strategy is attempting to exploit transition locality [4], by keeping the most recent states in the cache. While this method of caching the most local states may work in some cases, mere hash collision of the states does not signify that s is an old state. In fact s may be a quite recent state, thus very local; in this case, deleting s would counter the transition locality property.

Layered Duplicate Detection Zhou and Hansen [14] altered frontier search for directed graphs. The new approach is called layered duplicate detection. In layered duplicate detection they propose keeping one or more previous layers. While the number needed to prevent all duplicates depends on the structure of the graph. They do not attempt to catch all duplicates, because number of layers to eliminate all duplicates cannot be known *a priori*. However, they theoretically bound on duplicates for layered search is $\frac{h}{k}$ where h is the height of the graph and k is the number of layers cached. Zhou and Hansen exploit transition locality by keeping the most recent layers in the RAM cache.

2.2 External Memory Search

Using external memory increases the maximum number of states that can be stored. External memory search increases the memory available to the algorithm from ~ 4 GB to ~ 1.5 TB. This allows for larger state spaces to be completely explored. External memory searches involve a method for eliminating duplicates on disk. Often this is performed by delayed duplicate detection (DDD) [1].

External Memory Bucketing Bao and Jones determined that state comparisons, not disk I/O, took the majority of time in external memory search [15]. In a brute force external memory search, DDD took $O(mn)$ time where m is the number of states on the disk and n is the number of *candidate* states to be checked. One way to improve external memory model checking is to split the state space into several chunks called buckets. All duplicate states will be assigned to the same bucket. Therefore DDD only need to be performed within buckets. This reduces the amount of time for DDD to $O(\Sigma m_i n_i)$, where m_i is the states previously assigned to bucket i and n_i is the *candidate* states in bucket i. In most cases, $O(\Sigma m_i n_i) \ll O(mn)$. Several different methods of bucketing the states have been employed. Including hash functions [2], heuristics [16], transition locality [17–19] and graph structure [20]. All of these algorithms speed the search by reducing the number of state comparisons necessary to perform DDD.

Transition Locality in External Memory Model Checking Some external memory model checkers perform DDD on only the most recent states. These algorithms were attempting to exploit transition locality, However neither algorithm discussed here was fully successful.

Locality in RAM Cache Hammer and Weber [21] implemented an external memory search that takes a cue from the partial storage algorithms. The Hammer algorithm uses a function to guess which states are less likely to be duplicated and swaps them to external memory. They tried several selection functions. The best performing functions used a state's age as part of the criteria, with older states more likely to be sent to disk. Unfortunately, Hammer and Weber's algorithm is not truly BFS due to their special treatment of single successor states. The properties of transition locality have only been shown to exist in BFS.

Locality in DDD Della Penna et al. [17–19] extend Tronci's [5] cached memory model checker for use with external memory. When a child state is generated, it is checked immediately for membership in the RAM cache. When Della Penna's algorithm has a hash collision, the oldest of the two colliding states is written to disk. External memory is partitioned by the order states are written. DDD is usually performed on only the most recent portion of the states in external memory. Occasionally a full DDD was performed to removed duplicates missed in the partial DDD. By checking the *candidate* states against only the most recently written parts of the closed list, Della Penna et al. were attempting to

exploit transition locality in external memory. However, the states are written to disk when a hash collision occurs; hash collisions are not a good measure of locality. Therefore the states checked may not be the most local.

3 Algorithm

The starting point for our implementation is the Murϕ model checker, a domain-independent tool that takes a description of a model as input and uses breadth-first search to verify that the model is correct [22]. If an error is found, it returns an error trace.

3.1 Breadth-first search with layered duplicate detection

In our algorithm we bucket the states by two criteria. First by g-cost, which is a measure of a states distance from the start state. This allows us to take advantage of transition locality. The second bucketing criteria is a hash function that gives an approximately even distribution among buckets. Like bucketing discussed in section 2.2, this second bucketing criteria improves the speed of DDD, with a penalty of increased usage of external memory.

```
1  sequence LDDD(ErrorStates)
2    depth := 0
3    shallow := 0
4    s := start_state
5    Bucket[hash(s)][depth].add(s)
6    Refined[hash(s)][depth].add(s)
7    Q[hash(s)][depth].enqueue(s)
8    while(¬∀_D ∀_B Q[B][D].empty())
9      for(all buckets c)
10       while(¬Q[c][depth].empty())
11         s := Q[c][depth].dequeue()
12         For( all children s′ of s )
13           if(RAM_FULL())
14             for ( all buckets b )
15               delete Bucket[b][shallow]
16             if(depth + 1 > shallow) shallow ++
17           if(∀_{i=shallow}^{depth+1} s′ ∉ Bucket[hash(s′)][i])
18             Bucket[hash(s′)][depth+1].add(s′)
19             candidate[hash(s′)][depth+1].add(s′)
20           if(s′ ∈ ErrorStates) return counter_example_reconstruction(s′)
21     depth++
22     DDD(depth,shallow)
23   return ∅
```

Fig. 1. Pseudo-code for Layered Delayed Duplicate Detection

Figure 1 contains the pseudo-code for Layered Delayed Duplicate Detection. Bucket is the closed list kept in RAM. Refined is the closed list kept in external memory. The queue acts like pointers into the refined list. Candidate is a list of all *candidate* states and is also kept in external memory. The algorithm proceeds as follows, while we still have elements in the queue, line 8, we loop through all the buckets, line 9. We cycle through all states in each queue at the current level, line 10. For each dequeued state, all children states are generated, line 12. If RAM is full we delete all buckets at the shallowest depth currently in RAM, line 15. Shallow states are closest to the start state and are the least likely to be duplicated. When layers are large we sometimes delete the portions of the next layer that have already been generated, line 16. If the generated states are not found in RAM they are added to the proper bucket in RAM and the candidate list, line 18. If a violation of the property is found, the counter example is reconstructed, line 20. At the end of every layer DDD is performed, line 22.

3.2 Delayed Duplicate Detection

```
 1  DDD(depth, shallow)
 2    for ( all buckets b )
 3      l := shallow - L
 4      while ( l < shallow )
 5        offset[l]=0
 6        l++
 7      sort candidate[b][depth]
 8      e := 0
 9      while ( e < candidate[b][depth].size())
10        if(candidate[b][depth].element(e)≠candidate[b][depth].element(e − 1))
11          if(¬ inPreviousLayer(candidate[b][depth].element(e),b,offset)
12            Q[b][depth].enqueue(candidate[b][depth].element(e))
13            Refined[b][depth].add(s)
14        e++
15      delete candidate[b][depth]
```

Fig. 2. Pseudo-code for Delayed Duplicate Detection

The code for performing delayed duplicate detection is in Figure 2. DDD loops through all states of the *candidate* buckets. Each state is checked for redundancy, line 11. States not detected as duplicates are put into the queue, line 12 and the refined list. The code for checking for duplicate states in external memory is in Figure 3.

Performing the DDD is the most time consuming part of a DDD search [15]. DDD takes even more time than disk I/O, which is notoriously slow [15]. Therefore it is in our best interest to perform additional work that will reduce the time spent in DDD. One way to do this is to sort all states stored in external

```
1  boolean inPreviousLayers(s,b,offset)
2    layer := shallow-1
3    while ( layer > shallow - L )
4      while s < Refined[b][layer].element(offset[layer])
5        offset[layer] ++
6      if s == Refined[b][layer].element(offset[layer])
7        return true
8      layer - -;
9    return false
```

Fig. 3. Checking for a state in the L most recent layers in external memory.

memory. If a bucket is sorted, DDD takes $O(n+m)$ time instead of $O(nm)$. The cost of an external sort is $O(n * (1 + \lceil log_{B-1} \lceil \frac{n}{B} \rceil \rceil))$, where n is the number of *candidate* states, m is the number of states in the bucket and B is the number of states cached by the external sort.

Sorting the buckets makes this task easier in three ways. First, all duplicates within a bucket will be stored consecutively once sorted. Thus, a duplicates in the current bucket can be found by merely comparing a state to its neighbors. Therefore, the states are checked against the previous state in the *candidate* bucket, Figure 2 line 10. Detected duplicates are not added to the refined bucket. Another way sorting helps is when eliminating duplicates in external memory. We can compare the *candidate* state against the previous L layers one state at a time, Figure 3 lines 4 and 6. If we reach a state that sorts greater than the *candidate* state we know the *candidate* has no match in that layer and do not have to check any more states in that layer. The third help has to do with how *candidate* states are checked. Because the *candidate*s are sorted the next *candidate* state checked will sort greater than the current *candidate*, thus we can resume checking the layer from the same location, recorded in the offset array, line 5, rather than starting at the beginning of the layer. We know all states in the layer previously checked, sort less than the current *candidate* state, and are therefore not matches. During a DDD that uses sorting, we load a state from external memory in to RAM no more than once. A brute force DDD compares every *candidate* state to every state in external memory. Thus, every state in external memory may be loaded into RAM once for every *candidate* state. Sorting is well worth the minor overhead it costs.

3.3 Counter Example Reconstruction

The pseudo-code for reconstruction of the error trace is in Figure 4. In a standard RAM only search, each state usually includes a pointer back to its parent state. With parent pointers reconstructing an error trace is a matter of tracing back those pointers to the start state. In external memory search some states on the error trace may not be located in RAM. This makes reconstructing the error trace more complicated. Instead of a parent pointer we include two items

```
1  sequence counter_example_reconstruction(error)
2    trace:=∅
3    traceDepth:=depth
4    cur:=error
5    while(traceDepth>shallow)
6      trace:=trace & cur
7      traceDepth - -
8      cur:=Bucket[cur.parentHash][traceDepth].element(cur.parentPosition)
9    while(traceDepth>0)
10     trace:=trace & cur
11     traceDepth - -
12     cur:=Refined[cur.parentHash][traceDepth].element(cur.parentPosition)
13   trace:=trace & cur
14   return trace
```

Fig. 4. Pseudo-code for reconstructing error traces.

of information with each state, We remember the parents hash, to determine which bucket the parent is in, and the parents location in the bucket, line 8. With these two items of information we can find parent states in both RAM and external memory. Since the deepest states will be in RAM we first work from the error state to the shallowest layer in RAM, see line 5. The remaining states in the trace exist only in external memory. The process for finding them is very similar, line 12, but will take longer due to slower I/O speeds.

3.4 Termination

Because the search is breadth-first, if an error is found, the error trace is guaranteed to be the shortest path to the error state. If an error exists, our algorithm will always find it. If no error is found, the search terminates when there are no more states of the graph to explore, i.e. the open list is empty, see Figure 1, line 8. If the search terminates in this way the model is verified. However, when only the most recent layers are checked for duplicates, the search is no longer guaranteed to terminate with an empty open list, even if the graph is finite. It is possible, when no error states exist, for the search to continue forever because of missed duplicate states. Thus, our algorithm, like many partial memory algorithms, is Las Vegas in nature. Answers produced are always correct, but in some cases no answer is produced.

This points to a second way to detect termination and verify a model. If the model is verified for a depth that is equal to or greater than the diameter of the graph, called the *completeness threshold*, verification is complete. Various methods for determining a completeness threshold have been explored in the literature on bounded model checking, and can be applied in our layered approach. This method of detecting termination is similar to that used in bounded model checking, which uses satisfiability testing to verify that a model does not have

any errors up to some depth k [23]. Such a bound would eliminate the Las Vegas nature of the algorithm.

4 Results

We implemented layered duplicate detection in the Murϕ model checker, which uses a RAM cache to immediately eliminate as many duplicates as possible and uses delayed duplicate detection to eventually eliminate duplicates that are written to disk. When our implementation of the model checker finds a error, it uses the the divide-and-conquer technique to return the counterexample.

Our experiments were performed on a 3.0 GHz dual-core Xeon processor with two gigabytes of RAM and one terabyte of disk storage.

	Num	Num	Duplicates			Time		Disk
Model	States	Layers	RAM	DDD	Missed	BFS	Trace	GB
Verified								
newlist	80,109,359	110	346,175,576	116,341,341	288,695	14:35:00	N/A	22
ldash	254,935	64	2,391,696	7	2	3:28:42	N/A	41
directory	1,071,401,426	113	479,609,483	78,501,871	28	1:22:16:29	N/A	279
Error Detected								
adashe	2,156,280	16	3,329,518	169,981	0	4:02:17	7:39	5
arbiter	90,913,223	31	378,993,079	64,426,284	0	6:17:41	1	26

Table 1. Performance results using layered delayed duplicate detection, with a maximum of 50 layers stored on disk. The column *Num states* shows the number of unique states of the model that are visited during the search. The column *Num layers* shows the depth of the search. The column *RAM* records the number of duplicates eliminated in RAM, and the column *DDD* shows the number of duplicates written temporarily to disk and later eliminated by DDD. The column *Missed* shows the number of duplicates missed by our algorithm because not all layers of the search graph are stored on disk. The column *BFS* shows the CPU time for the search, and the column *Trace* shows the CPU time to reconstruct the error trace. Time is reported as Days:Hours:Minutes:Seconds. The column *Disk* shows the amount of external memory required to store the complete state space, reported in gigabytes.

The results reported in Table 1 show the performance of the algorithm in verifying five models. Three models were verified. For the other two models, an error was found and the error trace was returned. For all of the models except *ldash*, the largest layer of the search graph exceeded RAM capacity; this means that RAM cache contained less than one layer at times. A maximum of 50 layers of the search graph were stored on disk at a time to check for duplicates. Table 1 shows the number of unique states of the model. Because the algorithm checks a bounded number of layers for duplicates, and thus could miss some duplicates, the number of unique states of the model was was determined by a post-processing step.

Despite often caching less than a layer in RAM, our results show that more duplicates can be detected in RAM than there are unique states in the model. Fewer duplicates are eliminated in DDD; this is good, since DDD is relatively expensive. The number of duplicates missed because not all layers are stored on disk is typically small, and much less than the number eliminated by DDD. We report the amount of time taken by the algorithm, broken down by how much time was spent on the search and how much time was spent reconstructing the error trace if an error state was discovered. Counter example reconstruction usually takes a small portion of the total time. The counter example reconstruction for *arbiter* is remarkably fast; in this case, the error trace is sorted near the beginning of the refined files. Finally, we report the amount of gigabytes required to completely store the state space. While all these models exceed the RAM we have available, the *directory* model is particularly large, requiring 279 GB to completely store the state space.

4.1 Effective RAM Caching Saves Time and Space

Model	Cache	Time	Delayed Duplicates
ldash	RAM Cache	1:02:42	7
	No Cache	9:03:25	1,536,918
adashe	RAM Cache	4:02:17	172,184
	No Cache	6:12:45	3,459,785
arbiter	RAM Cache	6:17:41	66,980,652
	No Cache	22:15:19	413,937,369

Table 2. Performance with and without a layered RAM cache. The column *Time* shows the overall time to perform the search, and is reported in Hours:Minutes:Seconds. The column *Delayed Duplicates* shows the number of duplicates eliminated during DDD.

Having an effective RAM cache saves both time and space. Table 2 shows that without a RAM cache, there is a large increase in the number of duplicates written temporarily to disk and eliminated during DDD, as well as a corresponding increase in the running time of the model checker. In addition, writing these duplicates to disk increases the amount of disk storage required by the algorithm.

4.2 Layer Sizes

Figure 5 shows the size of each layer of the search graph for the *directory* model. As Pelánek [4] earlier observed, the distribution of layers usually has a near bell curve shape, with most states in the middle layers. For this model, 40% of the states are in layers 62 to 72, out of a total of 113 layers. In fact, layer 66 alone contains almost 5% of the states in the model. Although disk provides much

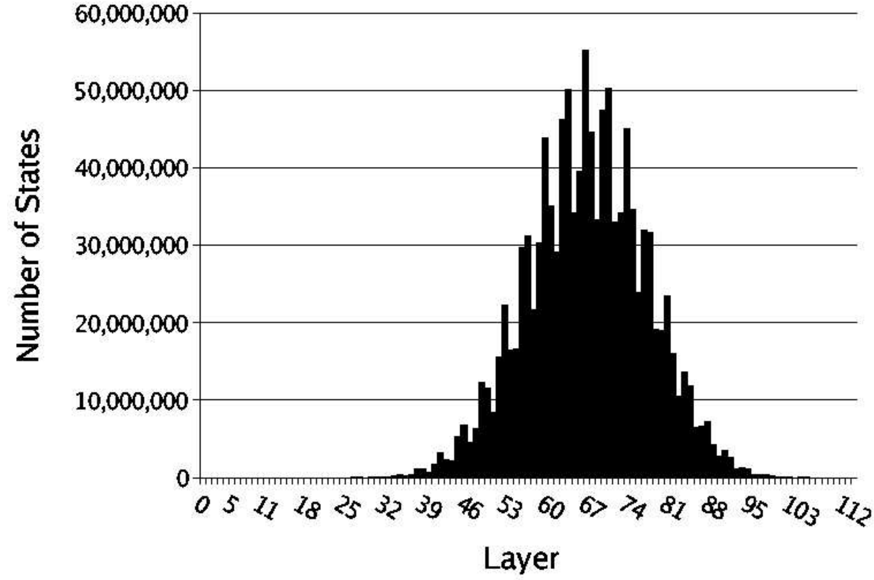

Fig. 5. The number of states at each depth of the *directory* model.

more storage than internal memory, it is also limited. In our experiments, we have not yet reached the limits of our available disk storage, but increasingly large models will test these limits. For example, Korf and Schultze [3] describe a complete breadth-first search of the *Fifteen Puzzle* that requires 1.4 terabytes of disk just to store the largest single layer of the graph. When model checking considers problems that approach this size, it will become important to conserve external memory. One way to economize in the use of external memory is by using our partial DDD algorithm. Since our algorithm only checks a subset of all previous layers during DDD, and only keeps this subset of layers on disk, it can use much less disk storage.

4.3 Transition Locality in Partial Delayed Duplicate Detection

Table 3 shows the result of running our algorithm on four large models and varying the number of layers stored on disk and checked in DDD. If too few layers are checked, large numbers of duplicates are never eliminated, increasing the running time of the algorithm. But it is possible to store more layers than necessary, although this usually doesn't increase the running time of the algorithm much since most duplicates are caught before checking the older layers. For all of these models, the best space-time performance is achieved by storing and checking only a bounded number of layers, but the optimal number of layers is model-dependent and cannot be known *a priori*.

4.4 Model Descriptions

All of our models are based on complex protocols. The following explains the protocols used to create the models.

1 *arbiter*: model of a mutual exclusion algorithm with an arbiter that allocates resources [24]. We used a model with 13 resources.
2 *dash*: model of the Dash Communication Protocol [25]. This protocol has two variant models.
  – *adashe*: abstraction of the dash protocol. This variant has been changed to include error states. *Adashe* was tested on a model that had 1 cluster with memory, 2 clusters without memory, 2 address in the memory cluster, and 2 value types to be saved.
  – *ldash*: concrete model of the dash protocol. We tested a model with 1 cluster with memory, 4 clusters without memory, 1 address per cluster could be locked, with message buffer of size 30.
3 *directory*: model of a directory-based cache protocol [26]. This is a model created by IBM and used for evaluating the protocol for use in servers. We tested a model with 14 clients.
4 *newlist*: protocol for a distributed linked list that covers maintaining coherency in the list and controlling adjustments as the list is sorted. We used a system with 8 distributed nodes.

4.5 Theoretical Results

We next prove some bounds on the number of duplicates that can be generated when not all all layers of the search graph are stored and checked for duplicates. First, we bound the number of duplicates that are never eliminated in the search. We call these redundant states. Then we bound the number of duplicates that will be eliminated in delayed duplicate detection. In our experiments, observed performance is much better than these worst-case bounds.

Bounds on Redundant States Zhou and Hansen [14] give some conditions under which no duplicates will be generated, using layered duplicate detection. In the worst-case, the number of times a state can be regenerated is bounded by the depth of the search (d) and the number of layers checked in DDD (L), and no state can be duplicated more than d/L times.

Theorem 1. *In layered external memory search, the worst-case number of times a state can be regenerated is bounded by* $\frac{d}{L}$

Proof. Let $L \geq 1$ be the number of layers checked by the algorithm. No duplicates can exist in these L layers. New states are checked against those L layers before being inserted into the open list. The least layers a duplicate state s can reappear is at layer $g^*(s)+L$ where $g^*(s)$ is the first layer at which state s is encountered. State s may then appear every L layers for the rest of the search. If the depth of the search is d, the state s can appear a maximum of j times where $g^*(s)+j*L \leq$

d. Thus, state s may recur at most $\frac{d}{L}$ times, which assumes the state appears in layer 0 and every L layer thereafter.

Theorem 2. *In layered breadth-first search, the worst-case number of number of states that can be regenerated is bounded by* $\frac{S*d}{L}$.

Proof. This is a small step from Theorem 1. Since a state s can be duplicated no more than $\frac{d}{L}$ times and there are S states, no more than $\frac{S*d}{L}$ states can be searched. Since searching $\frac{S*d}{L}$ states would mean that every state in the graph is duplicated the maximum times allowed by the bound in Theorem 1.

Note as well if $L = d$ then S states are searched, i.e. each state is searched once. Since $L = d$ in a traditional BFS this result is appropriate.

Theorem 3. *In layered breadth-first search, the worst-case redundant work factor is bounded by* $\frac{d}{L}$.

Proof. The redundant work factor is a metric used by Geldenhuys [7]. The metric is the ratio of reported states to actual states. Actual states are S. Reported states are bounded by Theorem 2 to $\frac{S*d}{L}$. Thus, the redundant work factor is bounded by $\frac{\frac{S*d}{L}}{S}$ which reduces to $\frac{d}{L}$.

Therefore, according to Theorem 3, if $L = d$, the redundant work factor is 1, which is, again, what we would expect from a traditional search. In the worst-case $L = 1$ making rwf $= d$.

Bounds on Delayed Duplicates Temporary duplicates are duplicates that are not eliminated immediately in RAM cache, but are eventually eliminated during DDD. Each temporary duplicate adds to the overhead of our search. We can bound the number of temporary duplicates by similar reasoning as in Theorem 1. We simply switch the number of layers checked in DDD (L) for the number of layers cached in RAM (k).

Theorem 4. *In layered external memory search, the worst-case number of times a state can be temporally duplicated is bounded by* $\frac{d}{k}$

Proof. If k is the number of layers cached by the algorithm. No duplicates will be temporarily missed in these k layers. New states will be checked against those k layers immediately. When a layer is larger than our RAM cache, i.e. $k < 1$, states may be repeated more than once per layer. Still, the least layers a duplicate state s can reappear is at layer $g^*(s) + k$ where $g^*(s)$ is the first layer at which state s is encountered. State s may then appear every k layers for the rest of the search. If the depth of the search is d, the state s can appear a maximum of j times where $g^*(s) + j * k \leq d$. Thus, state s may recur at most $\frac{d}{k}$ times, which assumes the state appears in layer 0 and every k layer thereafter. This bound is applicable whether $k \geq 1$ or $1 > k$.

Theorem 5. *In layered breadth-first search, the worst-case number of number of states that can be temporary duplicates is bounded by* $\frac{S*d}{k}$.

Proof. This is a small step from Theorem 4. Since a state s can be a temporary duplicate no more than $\frac{d}{k}$ times and there are S states, no more than $\frac{S*d}{k}$ states can be seen during the search. Since $\frac{S*d}{k}$ temporary duplicates states would mean that every state in the graph is duplicated the maximum times allowed by the bound in Theorem 4.

5 Conclusion and Future Work

External layered duplicate detection improved on previous work in several ways. First, layered duplicate detection is easy to implement in a domain independent way. Second, the layered framework allows us to exploit transition locality in the RAM cache. The vast majority of duplicates are identified instantly in RAM because the most recent states are cached in RAM. This is even true when only a small portion of the state space is cached. On our largest model, *directory* we eliminated 86% of duplicates in RAM while caching approximately 1% of the state space. Transition locality is clearly a good metric for caching. Third, transition locality can also be exploited in external memory. In external memory we can still rely on most duplicates occurring in the most recent layers. If we only perform DDD on the L most recent layers, we can save large amounts of space in external memory while insignificantly increasing time requirements, in the case of *ldash* we require approximately one quarter of the disk space needed to store the entire state space, with a delay of 11 minutes, out of a search taking three and a half hours. Fourth, we generate several worst-case linear bounds on the amount of redundant work performed by our algorithm. Results are well below the bound. Finally, a layered approach lends itself to a speedy counter example reconstruction. In *adashe* reconstruction takes 3% of the overall time.

Besides testing this approach on larger and more varied models, we hope to eventually extend it in several ways. We pointed out that only the L layers checked in DDD must be retained in our search. But up to this point we do not delete old layers because they are used during counter example reconstruction. We will extend our tool to allow complete search of graphs that do not fit on disk. This will include reconstructing parts of the counter example that have been removed from disk using a divide and conquer method [14].

We would also like a better hashing function. Jabbar and Edelkamp [16] use a hashing function that limits a bucket's successors to a few other buckets. The successor guarantees were used to parallelize the search. Completed buckets could be refined by delayed duplicate detection in parallel with bucket expansions. Their hash is domain dependent, however. We would like a hash that limited the number of successor buckets but works in a domain independent way.

Another valuable addition would be to bound the diameter of the graphs. This bound could be used to prune the search deeper than the bound. Such a bound would eliminate the Las Vegas nature of the algorithm. If error states

exist they would always be found before the bound. If all active states were pruned by the bound, the model would be verified.

A final addition would be a system for dynamically varying the number of layers checked in DDD. Since the ideal number of layers checked differs for every model, it would be desirable to determine the optimal number of layers automatically. We propose a system that would vary the number of layers checked based on the maximal back edge length observed.

References

1. Stern, U., Dill, D.L.: Using magnetic disk instead of main memory in the Murϕ verifier. In Hu, A.J., Vardi, M.Y., eds.: Computer-Aided Verification, CAV '98. Volume 1427 of Lecture Notes in Computer Science., Vancouver, BC, Canada, Springer-Verlag (1998) 172–183
2. Bao, T., Jones, M.: Time-efficient model checking with magnetic disk. In Halbwachs, N., Zuck, L.D., eds.: Proceedings of 11th International Conference Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005. Volume 3440 of Lecture Notes in Computer Science., Springer (2005) 526–540
3. Korf, R., Schultze, P.: Large-scale parallel breadth-first search. In: Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-05). (2005) 1380–1385
4. Pelánek, R.: Typical structural properties of state spaces. [27] 5–22
5. Tronci, E., Penna, G.D., Intrigila, B., Zilli, M.V.: Exploiting transition locality in automatic verification. 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME) LNCS **2144** (2001) 259–274
6. Gastin, P., Moro, P.: Minimal counterexample generation for spin. In Bosnacki, D., Edelkamp, S., eds.: SPIN. Volume 4595 of Lecture Notes in Computer Science., Springer (2007) 24–38
7. Geldenhuys, J.: State caching reconsidered. [27] 23–38
8. Godefroid, P., Holzmann, G.J., Pirottin, D.: State-space caching revisited. Formal Methods in System Design **7**(3) (1995) 227–241
9. Godefroid, P.: Software model checking: The VeriSoft approach. Technical report, Bell Laboratories, Lucent Technologies (2003)
10. Kristensen, L.M., Mailund, T.: A compositional sweep-line state space exploration method. In Peled, D., Vardi, M.Y., eds.: Formal Techniques for Networked and Distributed Systems. Volume 2529 of Lecture Notes in Computer Science., Springer (2002) 327–343
11. Kristensen, L.M., Mailund, T.: A generalised sweep-line method for safety properties. In Eriksson, L.H., Lindsay, P.A., eds.: Proceedings of the International Symposium of Formal Methods Europe. Volume 2391 of Lecture Notes in Computer Science., Springer (2002) 549–567
12. Parashkevov, A.N., Yantchev, J.: Space efficient reachability analysis through use of pseudo-root states. In: Tools and Algorithms for Construction and Analysis of Systems. (1997) 50–64
13. Zhou, R., Hansen, E.A.: Breadth-first heuristic search. In Zilberstein, S., Koehler, J., Koenig, S., eds.: ICAPS, AAAI (2004) 92–100

14. Zhou, R., Hansen, E.: Breadth-first heuristic search. Artificial Intelligence **170** (2006) 385–408
15. Bao, T.: Empirical comparison of algorithms for model checking with magnetic disk. Technical Report VV-0402, Department of Computer Science, Brigham Young University (2004)
16. Jabbar, S., Edelkamp, S.: Parallel external directed model checking with linear I/O. In Emerson, E.A., Namjoshi, K.S., eds.: VMCAI. Volume 3855 of Lecture Notes in Computer Science., Springer (2006) 237–251
17. Penna, G.D., Intrigila, B., Tronci, E., Zilli, M.V.: Exploiting transition locality in the disk based Murϕ verifier. In Aagaard, M., O'Leary, J., eds.: Formal Methods in Computer Aided Design. Volume 2517 of LNCS., Portland, OR, USA, Springer-Verlag (2002) 202–219
18. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Integrating RAM and disk based verification within the Murphi verifier. In: 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2003). Volume 2860 of LNCS., L'Aquila, Italy, Springer-Verlag (2003) 277–282
19. Penna, G.D., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.V.: Exploiting transition locality in automatic verification of finite-state concurrent systems. International Journal on Software Tools for Technology Transfer **6**(4) (2004) 320–341
20. Zhou, R., Hansen, E.: Structured duplicate detection in external-memory graph search. In McGuinness, D.L., Ferguson, G., eds.: Proceedings of the 19th National Conference on Artificial Intelligence (AAAI-04), San Jose, CA, AAAI Press / MIT Press (2004) 683–688
21. Hammer, M., Weber, M.: "To Store or not to Store" Reloaded: Reclaiming memory on demand. In: 11th International Workshop on Formal Methods for Industrial Critical Systems, Springer-Verlag (2006)
22. Dill, D.L.: The Murϕ verification system. In R. Alur, T. Henzinger, eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102 of Lecture Notes in Computer Science., New Brunswick, NJ, Springer Verlag (1996) 390–393
23. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**(1) (2001) 7–34
24. Kumar, R., Mercer, E.G.: Load balancing parallel explicit state model checking. Electr. Notes Theor. Comput. Sci. **128**(3) (2005) 19–34
25. Lenoski, D.: DASH Prototype System. PhD thesis, Stanford University (1992)
26. Emerson, A.E., German, S., Havlicek, J., Venkataramani, A.: Model checking a parameterized directory-based cache protocol (2002)
27. Graf, S., Mounier, L., eds.: Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings. In Graf, S., Mounier, L., eds.: SPIN. Volume 2989 of Lecture Notes in Computer Science., Springer (2004)

Layers	CPU Time	Disk Storage
adashe		
15	4:02:17	5.00
10	3:48:25	4.99
5	3:53:46	4.56
0	4:04:00	2.04
arbiter		
30	6:04:49	26.00
25	6:05:09	25.99
20	6:05:35	25.93
15	6:03:22	25.68
10	6:18:58	25.72
5	7:17:08	25.22
0	6:07:12	8.46
ldash		
46	1:07:15	40.88
26	1:03:51	37.58
10	1:03:34	20.50
8	1:01:43	17.09
6	1:11:05	13.39
5	1:12:46	11.41
4	2:21:06	9.44
2	>8:34:22	>6.23
newlist		
88	19:22:47	22.00
66	19:15:24	21.96
44	19:51:39	21.70

Table 3. Tradeoffs for checking a bounded number of previous layers for duplicates, as the number of layers is varied. The column *Layers* gives the number of previous layers used in DDD. The column *CPU Time* gives the amount of time the algorithm took, reported in Hours:Minutes:Seconds. The column *Disk Storage* gives the maximum amount of external memory required for the algorithm to complete the search, reported in gigabytes.