A Performance Study
# of BDD-Based Model Checking

Bwolen Yang[1], Randal E. Bryant[1], David R. O'Hallaron[1],
Armin Biere[1], Olivier Coudert[2], Geert Janssen[3],
Rajeev K. Ranjan[2], and Fabio Somenzi[4]

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] Synopsys Inc., Mountain View CA 94043, USA
[3] Eindhoven University of Technology, 5600 MB Eindhoven, Netherlands
[4] University of Colorado, Boulder CO 90309, USA

Abstract. We present a study of the computational aspects of model checking based on binary decision diagrams (BDDs). By using a trace-based evaluation framework, we are able to generate realistic benchmarks and perform this evaluation collaboratively across several different BDD packages. This collaboration has resulted in significant performance improvements and in the discovery of several interesting characteristics of model checking computations. One of the main conclusions of this work is that the BDD computations in model checking and in building BDDs for the outputs of combinational circuits have fundamentally different performance characteristics. The systematic evaluation has also uncovered several open issues that suggest new research directions. We hope that the evaluation methodology used in this study will help lay the foundation for future evaluation of BDD-based algorithms.

1 Introduction

The binary decision diagram (BDD) has been shown to be a powerful tool in formal verification. Since Bryant's original publication of BDD algorithms [7], there has been a great deal of research in the area [8, 9]. One of the most powerful applications of BDDs has been to symbolic model checking, used to formally verify digital circuits and other finite state systems. Characterizations and comparisons of new BDD-based algorithms have historically been based on two sets of benchmark circuits: ISCAS85 [6] and ISCAS89 [5]. There has been little work on characterizing the computational aspects of BDD-based model checking.

There are two qualitative differences between building BDD representations for combinational circuits versus model checking. The first difference is that for combinational circuits, the *output BDDs* (BDD representations for the circuit outputs) are built and then are only used for constant-time equivalence checking. In contrast, a model checker first builds the BDD representations for the system transition relation, and then performs a series of fixed point computations analyzing the state space of the system. In doing so, it is solving PSPACE-complete problems. Another difference is that BDD construction algorithms for combinational circuit operations have polynomial complexity [7], while the key operations in model checking are NP-hard [16]. These differences indicate that results based on combinational circuit benchmarks may not accurately characterize BDD computations in model checking.

This paper introduces a new methodology for the systematic evaluation of BDD computations, and then applies this methodology to gain a better understanding of the computational aspects of model checking. The evaluation is a collaborative effort among many BDD package designers. As results of this evaluation, we have significantly improved model checking performance, and have identified some open problems and new research directions.

The evaluation methodology is based on a trace-driven framework where execution traces are recorded from verification tools and then replayed on several BDD packages. In this study, the benchmark consists of 16 execution traces from the Symbolic Model Verifier (SMV) [16]. For comparison with combinational circuits, we also studied 4 circuit traces derived from the ISCAS85 benchmark. The other part of our evaluation methodology is a set of platform independent metrics. Throughout this study, we have identified useful metrics to measure work, space, and memory locality.

This systematic and collaborative evaluation methodology has led to better understanding of the effects of cache size and garbage collection frequency, and has also resulted in significant performance improvement for model checking computations. Systematic evaluation also uncovered vast differences in the computational characteristics of model checking and combinational circuits. These differences include the effects of the cache size, the garbage collection frequency, the complement edge representation [1], and the memory locality of the breadth-first BDD packages. For the difficult issue of dynamic variable reordering, we introduce some methodologies for studying the effects of variable reordering algorithms and initial variable orders.

It is important to note that the results in this study are obtained based on a very small sample of all possible BDD-based model checking computations. Thus, in the subsequent sections, most of the results are presented as hypotheses along with their supporting evidence. These results are not conclusive. Instead, they raise a number of interesting issues and suggest new research directions.

The rest of this paper is organized as follows. We first present a brief overview of BDDs and relevant BDD algorithms (Sec. 2) and then describe the experimental setup for the study (Sec. 3). This is followed by three sections of experimental results. First, we report the findings without dynamic variable reordering (Sec. 4). Then, we present the results on dynamic variable reordering algorithms and the effects of initial variable orders (Sec. 5). Third, we present results that may be generally helpful in studying or improving BDD packages (Sec. 6). After these result sections, we discuss some unresolved issues (Sec. 7) and then wrap up with related work (Sec. 8) and concluding remarks (Sec. 9).

2 Overview

This section gives a brief overview of BDDs and pertinent BDD algorithms. Detailed descriptions can be found in [7] and [16].

2.1 BDD Basics

A BDD is a directed acyclic graph (DAG) representation of a Boolean function where equivalent Boolean sub-expressions are uniquely represented. Due to this uniqueness property, a BDD can be exponentially more compact than its corresponding truth table representation. One criterion for guaranteeing the uniqueness of the BDD representation is that all the BDDs constructed must follow the same variable order. The choice of this variable order can have a significant impact on the size of the BDD graph.

BDD construction is a memoization-based dynamic programming algorithm. Due to the large number of distinct subproblems, a cache, known as the *computed cache*, is used instead of a memoization table. Given a Boolean operation, the construction of its BDD representation consists of two main phases. In the top-down *expansion phase*, the Boolean operation is recursively decomposed into subproblems based on the Shannon decomposition. In the bottom-up *reduction phase*, the result of each subproblem is put into the canonical form. The uniqueness of the result's representation is enforced by hash tables known as *unique tables*. The new subproblems are generally recursively solved in a depth-first order as in Bryant's original BDD publication [7]. Recently, there has been some work that tries to exploit memory locality by using a breadth-first order [2, 18, 19, 21, 26].

Before moving on, we first define some terminology. We will refer to the Boolean operations issued by a user of a BDD package as the *top-level operations* to distinguish them from *sub-operations* (subproblems) generated internally by the Shannon expansion process. A BDD node is *reachable* if it is in some BDDs that external users have references to. As external users free references to BDDs, some BDD nodes may no longer be reachable. We will refer to these nodes as *unreachable* BDD nodes. Note that unreachable BDD nodes can still be referenced within a BDD package by either the unique tables or the computed cache. Some of these unreachable BDD nodes may become reachable again if they end up being the results for new subproblems. When a reachable BDD node becomes unreachable, we say a *death* has occurred. Similarly, when an unreachable BDD node becomes reachable again, we say a *rebirth* has occurred. We define the *death rate* as the number of deaths over the number of subproblems (*time*) and define the *rebirth rate* as the fraction of the unreachable nodes that become reachable again, i.e., the number of rebirths over the number of deaths.

2.2 Common Implementation Features

Modern BDD packages typically share the following common implementation features based on [4, 22]. The BDD construction is based on depth-first traversal.

The unique tables are hash tables with the hash collisions resolved by chaining. A separate unique table is associated with each variable to facilitate the dynamic variable reordering process. The computed cache is a hash-based direct mapped (1-way associative) cache. BDD nodes support *complement edges* where, for each edge, an extra bit is used to indicate whether or not the target function should be inverted. Garbage collection of unreachable BDD nodes is based on reference counting and the reclaimed unreachable nodes are maintained in a *free-list* for later reuse. Garbage collection is invoked when the percentage of the unreachable BDD nodes exceeds a preset threshold.

As the variable order can have significant impact on the size of a BDD graph, dynamic variable reordering is an essential part of all modern BDD packages. The dynamic variable reordering algorithms are generally based on *sifting* or *window permutation* algorithms [22]. Typically, when a variable reordering algorithm is invoked, all top-level operations that are currently being processed are aborted. When the variable reordering algorithm terminates, these aborted operations are restarted from the beginning.

2.3 Model Checking and Relational Product

There are two popular BDD-based algorithms for computing state transitions: one is based on applying the *relational product* operator (also known as *AndExists* or *and-smooth*) on the transition relations and the state sets [10]; the other is based on applying the *constrain* operator to Boolean functional vectors [11, 12].

The benchmarks in this study are based on SMV, which uses the relational product operation. This operation computes "$\exists \boldsymbol{v}.f \wedge g$" and is used to compute the set of states by the forward or the backward state transitions. It has been proven to be NP-hard [16]. Figure 1 shows a typical BDD algorithm for computing the relational product operation. This algorithm is structurally very similar to the BDD-based algorithm for the *AND* Boolean operation. The main difference (lines 5–11) is that when the top variable (τ) needs to be quantified, a new BDD operation ($OR(r_0,\, r_1)$) is generated. Due to this additional recursion, the worst case complexity of this algorithm is exponential in the graph size of the input arguments.

3 Setup

3.1 Benchmark

The benchmark used in this study is a set of execution traces gathered from the Symbolic Model Verifier (SMV) [16] from Carnegie Mellon University. The traces were gathered by recording BDD function calls made during the execution of SMV. To facilitate the porting process for different packages, we only recorded a set of the key Boolean operations and discarded all word-level operations. The coverage of this selected set of BDD operations is greater than 95% of the total SMV execution time for all but one case (*abp11*) which spends 21% of CPU time in the word-level functions constructing the transition relation.

```
RP($\boldsymbol{v}$, $f$, $g$)
  /* compute relational product: $\exists \boldsymbol{v}.f \wedge g$ */
  if (terminal case) return result
  if the result of (RP, $\boldsymbol{v}$, $f$, $g$) is cached, return the result
  let $\tau$ be the top variable of $f$ and $g$
  $r_0 \leftarrow$ RP($\boldsymbol{v}$, $f|_{\tau\leftarrow 0}$, $g|_{\tau\leftarrow 0}$)  /* Shannon expansion on 0-cofactors */
  if ($\tau \in \boldsymbol{v}$)  /* existential quantification on $\tau \equiv$ OR($r_0$, RP($\boldsymbol{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)) */
      if ($r_0$ == true)  /* OR(true, RP($\boldsymbol{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)) $\equiv$ true */
          $r \leftarrow$ true.
      else
          $r_1 \leftarrow$ RP($\boldsymbol{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)  /* Shannon expansion on 1-cofactors */
          $r \leftarrow$ OR($r_0$, $r_1$)
  else
      $r_1 \leftarrow$ RP($\boldsymbol{v}$, $f|_{\tau\leftarrow 1}$, $g|_{\tau\leftarrow 1}$)  /* Shannon expansion on 1-cofactors */
      $r \leftarrow$ reduced, unique BDD node for ($\tau$, $r_0$, $r_1$)
  cache the result of this operation
  return $r$
```

Fig. 1. A typical relational product algorithm.

A side effect of recording only a subset of BDD operations is that the construction process of some BDDs is skipped, and these BDDs might be needed later by some of the selected operations. Thus in the trace file, these BDDs need to be reconstructed before their first reference. This reconstruction is performed bottom-up using the *If-Then-Else* operation. This process is based on the property that each BDD node (v_i, $child_0$, $child_1$) essentially represents the Boolean function "If v_i then $child_1$ else $child_0$".

For this study, we have selected 16 SMV models to generate the traces. The following is a brief description of these models along with their sources.

abp11: alternating bit protocol.
Source: Armin Biere, Universität Karlsruhe.

dartes: communication protocol of an Ada program.
dpd75: dining philosophers protocol.
ftp3: file transfer protocol.
furnace17: remote furnace program.
key10: keyboard/screen interaction protocol in a window manager.
mmgt20: distributed memory manager protocol.
over12: automated highway system overtake protocol.
Source: James Corbett, University of Hawaii.

dme2-16: distributed mutual exclusion protocol.
Source: SMV distribution, Carnegie Mellon University.

futurebus: futurebus cache coherence protocol.
Source: Somesh Jha, Carnegie Mellon University.

motor-stuck: batch-reactor system model.
valves-gates: batch-reactor system model.
Source: Adam Turk, Carnegie Mellon University.

phone-async: asynchronous model of a simple telephone system.
phone-sync-CW: synchronous model of a telephone system with call waiting.
Source: Malte Plath and Mark Ryan, University of Birmingham, Great Britain.

tcas: traffic alert and collision system for airplanes.
Source: William Chan, University of Washington.

tomasulo: a buggy model of the Tomasulo algorithm for instruction scheduling in superscalar processors.
Source: Yunshan Zhu, Carnegie Mellon University.

As we studied and improved on the model checking computations during the course of the study, we compared their performance with the BDD construction of combinational circuit outputs. For this comparison, we used the ISCAS85 benchmark circuits as the representative circuits. We chose these benchmarks because they are perhaps the most popular benchmarks used for BDD performance evaluations. The ISCAS85 circuits were converted into the same format as the model checking traces. The variable orders used were generated by the *order-dfs* in SIS [24]. We excluded cases that were either too small ($<$ 5 CPU seconds) or too large ($>$ 1 GBytes of memory requirement). Based on this criteria, we were left with two circuits — C2670 and C3540. To obtain more circuits, we derived 13-bit and 14-bit integer multipliers, based on the C6288, which we refer to as C6288-13 and C6288-14. For the multipliers, the variable order is $a_{n-1} \prec a_{n-2} \prec ... \prec a_0 \prec b_{n-1} \prec b_{n-2} \prec ... \prec b_0$, where $A = \sum_{i=0}^{n-1} 2^i a_i$ and $B = \sum_{i=0}^{n-1} 2^i b_i$ are the two n-bit input operands to the multiplier.

Figure 2 quantifies the sizes of the traces we used in the study. The statistic "*# of BDD Vars*" is the number of BDD variables used. The statistic "*Min. # of Ops*" is the minimum number of sub-operations (or subproblems) needed for the computation. This statistic characterizes the minimum amount of work for each trace. It was gathered using a BDD package with a complete cache and no garbage collection. Thus, this statistic represents the minimum number of sub-operations needed for a typical BDD package. Due to insufficient memory, there are 4 cases (*futurebus, phone-sync-CW, tcas, tomasulo*) for which we were not able to collect this statistic. For these cases, the results shown are the minimum across all the packages used in the study. These results are marked with the "<" symbol. The third statistic, "*Peak # of Live BDDs*", represents the peak number of reachable BDD nodes during the execution. It provides a lower bound on the memory required to execute the corresponding trace. Note that neither "*Min. # of Ops*" nor "*Peak # of Live BDDs*" reflects the effects of the dynamic variable reordering process.

Trace	# of BDD Vars	Min. # of Ops	Peak # of Live BDDs
		($\times 10^6$)	($\times 10^3$)
abp11	122	116	53
dartes	198	6	468
dme2-16	586	106	905
dpd75	600	41	1719
ftp3	100	132	763
furnace17	184	30	2109
futurebus	348	< 10270	4473
key10	140	91	626
mmgt20	264	35	1113
motors-stuck	172	29	325
over12	174	58	3008
phone-async	86	329	1446
phone-sync-CW	88	< 3803	22829
tcas	292	< 1323	19921
tomasulo	212	< 1497	26944
valves-gates	172	44	433
c2670	233	15	4363
c3540	50	57	7775
c6288-13	26	60	3378
c6288-14	28	178	9662

Fig. 2. Sizes of the benchmark traces. "# of BDD Vars" is the number of BDD variables. "*Min. # of Ops*" is the minimum number of sub-operations which characterizes work. "*Peak # of Live BDDs*" is the maximum number of reachable BDD nodes, which characterizes the minimum memory requirement.

3.2 BDD Packages

The following is a list of the BDD packages used in the study. For each BDD package, we note how it differs from the common implementation described in Sec. 2.2. Although many of these BDD packages contain a wide variety of useful features, only those pertinent to the study are described in this section.

ABCD (Author: Armin Biere)

ABCD [3] is an experimental BDD package based on the classical depth-first traversal. Interesting features include mark-and-sweep based garbage collection, the integration of BDD nodes with the BDD unique table by using open addressing, and index-based (instead of pointer-based) references to

BDD nodes. These techniques reduce the BDD node size by half (2 machine words instead of 4). In addition, to avoid clustering in open addressing, ABCD uses a quadratic probe sequence for the hashing collision resolution.

CAL (Authors: Rajeev Ranjan and Jagesh Sanghavi)

CAL [20] is a publicly available BDD package based on breadth-first traversal to exploit memory locality. The garbage collection algorithm is based on reference-counting with memory compaction. To increase locality of reference, each BDD node contains the indices of its cofactor nodes. To keep the node size to 4 machine words, bit tagging is used to store and retrieve the value of the reference count of a node. For this study, the relational product operation is based on the depth-first traversal with the quantification step (line 7 in Fig. 1) computed using the breadth-first traversal.

CUDD (Author: Fabio Somenzi)

CUDD [25] is a publicly available BDD package based on depth-first traversal. In CUDD, the reference counts of the nodes are kept up-to-date throughout the computation. To counter the impact on performance of these updates when many nodes are freed and reclaimed, CUDD enqueues the requests for updates and performs them only if they are still valid when they are extracted from the queue. The growth of the tables in CUDD is determined by a reward policy. For instance, the cache grows if the hit rate is high. CUDD partially sorts the free list during garbage collection to improve memory locality. Another distinguishing feature is that CUDD contains a suite of heuristics for dynamic variable reordering.

EHV (Author: Geert Janssen)

EHV [14] is a publicly available BDD package based on depth-first traversal. The main differences from the common implementation are additional support for inverted inputs [17] and provisions for user data to be attached to a BDD node. The latter feature allows intermediate results to be stored in the BDD nodes, which in turn, removes the need to use separate computed caches for some special BDD operations. This feature incurs a memory overhead of 2 extra machine words per BDD node.

PBF (Authors: Bwolen Yang and Yirng-An Chen)

PBF [26] is an experimental BDD package based on partial breadth-first traversal. The partial breadth-first traversal along with per-variable memory managers and the memory-compacting mark-and-sweep garbage collector are used to exploit memory locality. The partial breadth-first traversal also bounds the breadth-first expansion to avoid the potential excessive memory overhead of a full breadth-first expansion.

TiGeR (Authors: Olivier Coudert, Jean C. Madre and Herve Touati)

TiGeR [13] is a commercial BDD package based on the depth-first approach. Interesting features include the segmentation of the computed caches and the garbage collection algorithm. In TiGeR, each operation type has its own cache. This allows the caches to be tuned independently. For this study, the caches for the non-polynomial operations such as relational product are set to be about four times as sparse as the caches for the polynomial operations. TiGeR's garbage collection algorithm is different from typical garbage collection algorithms in two ways: the free-list is sorted to maintain memory locality, and the memory compaction is performed when memory resources become critical.

3.3 Evaluation Process

The performance study was carried out in two phases. The first phase studied performance issues in BDD construction without variable reordering. The second phase focused on the dynamic variable reordering computation. The evaluation process was iterative, with the study evolving dynamically as new issues were raised and new insights gained. Based on the results from each iteration, we collaboratively tried to identify the performance issues and possible improvements. Each BDD package designer then incorporated and validated the suggested improvements. During this iterative process, we also tried to hypothesize the characteristics of the computation and design new experiments to test these hypotheses.

4 Phase 1 Results: No Variable Reordering

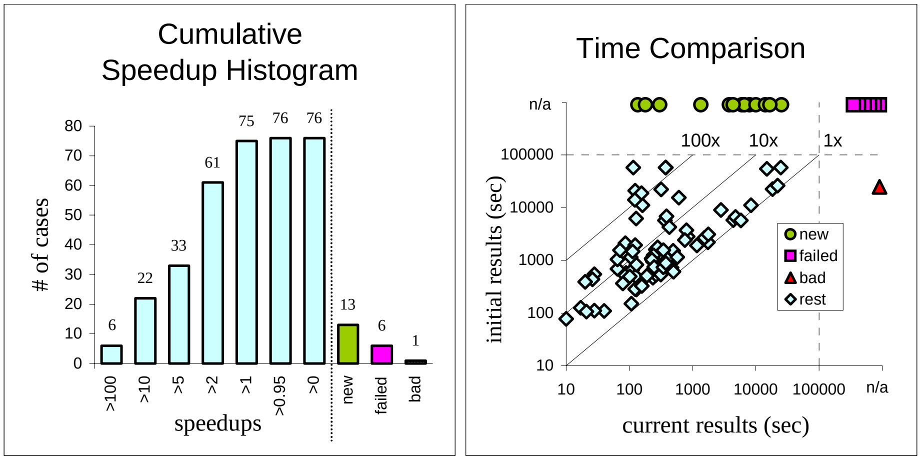
Figure 3 presents the overall performance improvements for Phase 1 with dynamic variable reordering disabled. There are 6 packages and 16 model checking traces, for a total of 96 cases. Figure 3(a) categorizes the results for these cases based on speedups. Note that the speedups are plotted in a cumulative fashion; i.e., the $> x$ column represents the total number of cases with speedups greater than x. Figure 3(b) presents a comparison between the initial timing results (when we first started the study) and the current timing results (after the authors made changes to their packages based on insights gained from previous iterations). The *n/a* results represent cases where results could not be obtained.

Initially, 19 cases did not complete because of implementation bugs or memory limits. Currently, 13 of these 19 cases now complete (the *new* cases in the figures). The other 6 cases still do not complete within the the resource limit of 8 hours and 900 MBytes (the *failed* cases in the figures). There is one case (the *bad* case in the charts) that initially completed, but now does not complete within the memory limit.

Figure 3(a) shows that significant speedups have been obtained for many cases. Most notably, 22 cases have speedups greater than an order of magnitude (the > 10 column), and 6 out of these 22 cases actually achieve speedups greater than two orders of magnitude (the > 100 column)!

Figure 3(b) shows that significant speedups have been obtained mostly from the small to medium traces, although some of the larger traces have achieved speedups greater than 3. Another interesting point is that the *new* cases (those that initially failed but are now doable) range across small to large traces.

Overall, for the 76 cases where the comparison could be made, the total CPU time was reduced from 554,949 seconds to 127,786 seconds — a speedup of 4.34. Another interesting overall statistic is that initially none of the 6 BDD packages could complete all 16 traces, but currently 3 BDD packages can complete all of them.



(a)

(b)

Fig. 3. Overall results. The *new* cases represent the number of cases that failed initially and are now doable. The *failed* cases represent those that currently still exceed the limits of 8 CPU hours and 900 MBytes. The *bad* shows the case that finished initially, but cannot complete currently. The *rest* are the remaining cases. (a) Results shown as histograms. For the 76 cases where both the initial and the current results are available, the speedup results are shown in a cumulative fashion; i.e., the $> x$ column represents the total number of cases with speedups greater than x. (b) Time comparison (in seconds) between the initial and the current results. *n/a* represents results that are not available due to resource limits.

The remainder of this section presents results on a series of experiments that characterize the computational aspects of the BDD traces. We first present results on two aspects with significant performance impact — computed cache size and garbage collection frequency. Then we present results on the effects of the complement edge representation. Finally, we give results on memory locality issues for the breadth-first based traversal.

4.1 Computed Cache Size

We have found that dramatic performance improvements are possible by using a larger computed cache. To study the impact of the computed cache, we performed some experiments and arrived at the following two hypotheses.

Hypothesis 1 *Model checking computations have a large number of repeated subproblems across the top-level operations. On the other hand, combinational circuit computations generally have far fewer such repeated subproblems.*

Experiment: Measure the minimum number of subproblems needed by using a complete cache (denoted CC-NO-GC). Compare this with the same setup but with the cache flushed between top-level operations (denoted CC-GC). For both cases, BDD-node garbage collection is disabled.

Result: Figure 4 shows the results of this experiment. Note that the results for the four largest model checking traces are not available due to insufficient memory.

These results show that for model checking traces, there are indeed many subproblems repeated across the top-level operations. For 8 traces, the ratio of the number of operations in CC-GC over the number of operations in CC-NO-GC is greater than 10. In contrast, this ratio is less than 2 for building output BDDs for the ISCAS85 circuits. For model checking computations, since subproblems can be repeated further apart in time, a larger cache is crucial.

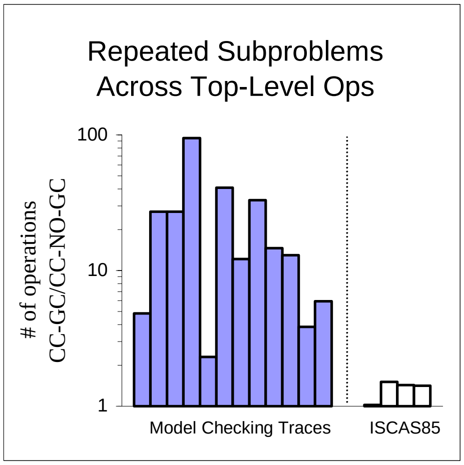


Fig. 4. Performance measurement on the frequency of repeated subproblems across the top-level operations. CC-GC denotes the case in which the cache is flushed between the top-level operations. CC-NO-GC denotes the case in which the cache is never flushed. In both cases, a complete cache is maintained within a top-level operation and BDD-node garbage collection is disabled. For four model checking traces, the results are not available (and are not shown) due to insufficient memory.

Hypothesis 2 *The computed cache is more important for model checking than for combinational circuits.*

Experiment: Vary the cache size as a percentage of the number of BDD nodes and collect the statistics on the number of subproblems generated to measure the effect of the cache size. In this experiment, the cache sizes vary from 10%

to 80% of the number of BDD nodes. The cache replacement policy used is FIFO (first-in-first-out).

Results: Figure 5 plots the results of this experiment. Each curve represents the result for a trace with varying cache sizes. The "*# of Ops*" statistic is normalized over the minimum number of operations necessary (i.e., the CC-NO-GC results). Note that for the four largest model checking traces, the results are not available due to insufficient memory.

These results clearly show that the cache size can have much more significant effects on the model checking computations than on building BDDs for the ISCAS85 circuit outputs.

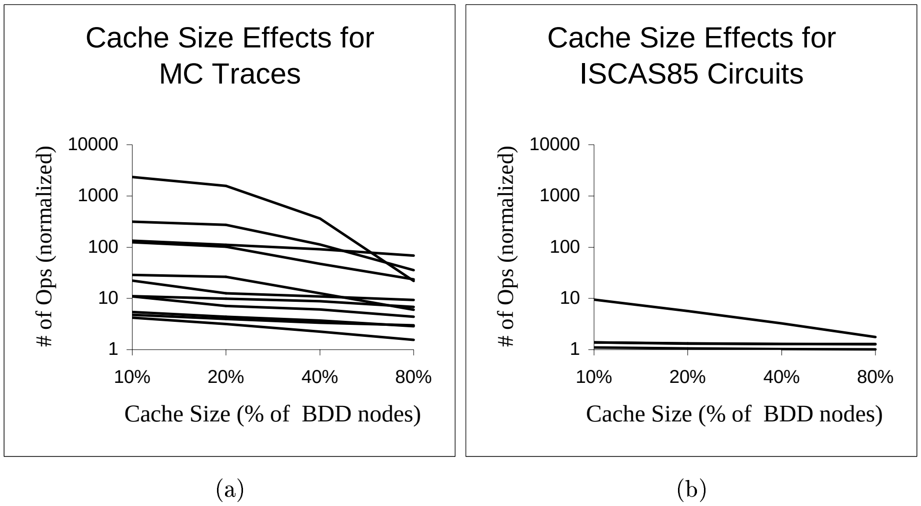


Fig. 5. Effects of cache size on overall performance for (a) the model checking traces and (b) the ISCAS85 circuits. The cache size is set to be a percentage of the number of BDD nodes. The number of operations (subproblems) is normalized to the minimum number of subproblems necessary (i.e., the CC-NO-GC results).

4.2 Garbage Collection Frequency

The other source of significant performance improvement is the reduction of the garbage collection frequency. We have found that for the model checking traces, the rate at which reachable BDD nodes become unreachable (death rate) and the rate at which unreachable BDD nodes become reachable (rebirth rate) can be quite high. This leads to the following conclusions:

- Garbage collection should occur less frequently.

– Garbage collection should not be triggered solely based on the percentage of the unreachable nodes.
– For reference-counting based garbage collection algorithms, maintaining accurate reference counts all the time may incur non-negligible overhead.

Hypothesis 3 *Model checking computations can have very high death and rebirth rates, whereas combinational circuit computations have very low death and rebirth rates.*

Experiment: Measure the death and rebirth rates for the model checking traces and the ISCAS85 circuits.

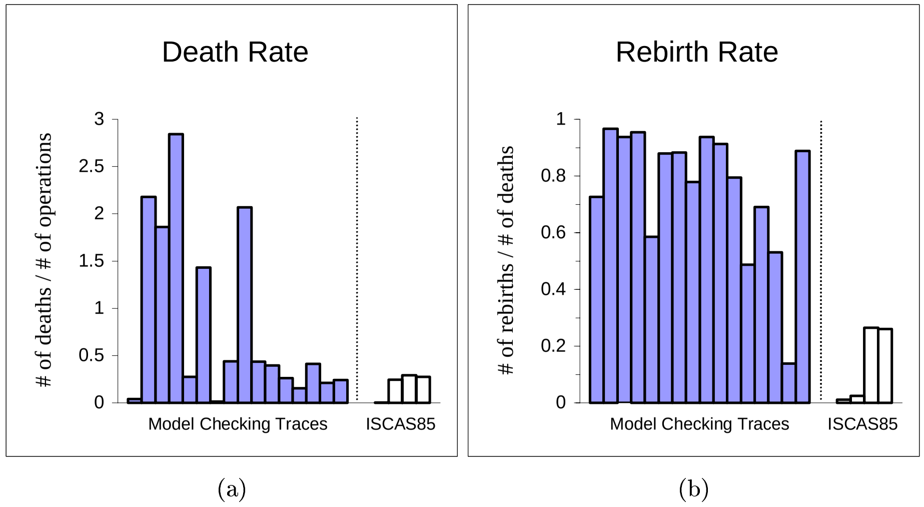
Results: Figure 6(a) plots the ratio of the total number of deaths over the total number of sub-operations. The number of sub-operations is used to represent *time*. This chart shows that the death rates for the model checking traces can vary considerably. In 5 cases, the number of deaths is higher than the number of sub-operations (i.e., death rate is greater than 1). In contrast, the death rates of the ISCAS85 circuits are all less than 0.3.

That the death rates exceed 1 is quite unexpected. To explain the significance of this result, we digress briefly to describe the process of BDD nodes becoming unreachable (death) and then becoming reachable again (rebirth). When a BDD node become unreachable, its children can also become unreachable if this BDD node is its children's only reference. Thus, it is possible that when a BDD node become unreachable, a large number of its descendants also become unreachable. Similarly, if an unreachable BDD node becomes reachable again, a large number of its unreachable descendants can also become reachable. Other than rebirth, the only way the number of reachable nodes can increase is when a sub-operation creates a new BDD node as its result. As each sub-operation can produce at most one new BDD node, a death rate of greater than 1 can only occur when the corresponding rebirth rate is also very high. In general, high death rate coupled with high rebirth rate indicates that many nodes are toggling between being reachable and being unreachable. Thus, for reference-counting based garbage collection algorithms, maintaining accurate reference count all the time may incur significant overhead. This problem can be addressed by using a bounded-size queue to delay the reference-count updates until the queue overflows.

Figure 6(b) plots the ratio of the total number of rebirths over the total number of deaths. Since garbage collection is enabled in these runs and does reclaim unreachable nodes, the rebirth rates shown may be lower than without garbage collection. This figure shows that the rebirth rates for the model checking traces are generally very high — 8 out of 16 cases have rebirth rates greater than 80%. In comparison, the rebirth rate for the ISCAS85 circuits are all less than 30%.

The high rebirth rates indicate that garbage collection for the model checking traces should be delayed as long as possible. There are two reasons for this: first, since a large number of unreachable nodes do become reachable again, garbage collection will not be very effective in reducing the memory usage. Second, the high rebirth rate may result in repeated subproblems involving the currently unreachable nodes. By garbage collecting these unreachable nodes, their corresponding computed cache entries must also be cleared. Thus, garbage collection may greatly increase the number of recomputations of identical subproblems.

The high rebirth rates and the potentially high death rates also suggest that the garbage collection algorithm should not be triggered based solely on the percentage of the dead nodes, as with the classical BDD packages.



(a)

(b)

Fig. 6. (a) Rate at which BDD nodes become unreachable (death). (b) Rate at which unreachable BDD nodes become reachable again (rebirth).

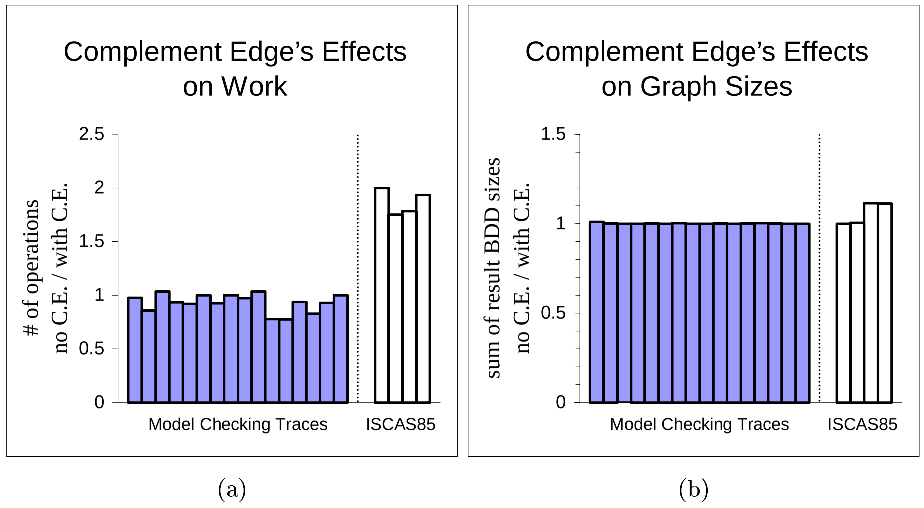
4.3 Effects of the Complement Edge

The complement edge representation [1] has been found to be somewhat useful in reducing both the space and time required to build the output BDDs for the ISCAS85 circuits [17]. In the following experiments, we study the effects of the complement edge on the model checking traces and compare it with the results for the ISCAS85 circuits.

Hypothesis 4 *The complement edge representation can significantly reduce the amount of work for combinational circuit computations, but not for model checking computations. However, in general, it has little impact on memory usage.*

Experiment: Measure and compare the number of subproblems (amount of work) and the resulting graph sizes (memory usage) generated from two BDD packages — one with and the other without the complement-edge feature. For the graph size measurements, sum the resulting BDD graph sizes of all top-level operations. Note that since two packages are used, minor differences in the number of operations can occur due to different garbage collection and caching algorithms.

Results: Figure 7(a) shows that the complement edges have no significant effect for model checking traces. In contrast, for the ISCAS85 circuits, the ratio of the no-complement-edge results over with-complement-edge results ranges from 1.75 to 2.00. Figure 7(b) shows that the complement edges have no significant effect on the BDD graph sizes in any of the benchmark traces.



(a)

(b)

Fig. 7. Effects of the complement edge representation on (a) number of the operations and (b) graph sizes.

4.4 Memory Locality for Breadth-First BDD Construction

In recent years, a number of researchers have proposed breadth-first BDD construction to exploit memory locality [2, 18, 19, 21, 26]. The basic idea is that for each expansion phase, all sub-operations of the same variable are processed together. Similarly, for each reduction phase, all BDD nodes of the same variable are produced together. Note that even though this *levelized* access pattern is slightly different from the traditional notion of breadth-first traversal, we will continue to refer to this pattern as breadth-first to be consistent with previous work. Based on this structured access pattern, we can exploit memory locality by using per-variable memory managers and per-variable breadth-first queues to cluster the nodes of the same variable together. This clustering is beneficial only if many nodes are processed for each breadth-first queue during each expansion and reduction phase.

The breadth-first approach does have some performance drawbacks (at least in the two packages we studied). The breadth-first expansion usually has higher memory overhead. In terms of running time, one drawback is in the implementation of the cache. In the breadth-first approach, the sub-operations are explicitly represented as *operator nodes* and the uniqueness of these nodes is ensured by using a hash table with chaining for collision resolution. Accesses to this hash table are inherently slower than accesses to the direct mapped (1-way associative) computed cache used in the depth-first approaches. Furthermore, handling of the computed and yet-to-be-computed operator nodes adds even more overhead. Depending on the implementation strategy, this overhead could be in the form of an explicit cache garbage collection phase or transferring of a computed result from an operator node's hash table to a computed cache. Maintenance of the breadth-first queues is another source of overhead. This overhead can be higher for operations such as relational products because of the possible additional recursion (e.g., line 7 in Fig. 1). Given that each sub-operation requires only a couple hundred cycles on modern machines, these overheads can have a non-negligible impact on the overall performance.

In this study, we have found no evidence that the breadth-first based packages are better than the depth-first based packages when the computation fits in main memory. Our conjecture is that since the relational product algorithm (Fig. 1) can have exponential complexity, the graph sizes of the BDD arguments do not have to be very large to incur a long running time. As a result, the number of nodes processed each time can be very small. The following experiment tests this conjecture.

Hypothesis 5 *For our test cases, few nodes are processed each time a breadth-first queue is visited. For the same amount of total work, combinational circuit computations have much better* "breadth-first" *locality than model checking computations.*

Experiment: Measure the number of sub-operations processed each time a breadth-first queue is visited. Then compute the maximum, mean, and standard deviation of the results. Note that these calculations do not include the cases where the queues are empty since they have no impact on the memory locality issue.

Result: Figure 8 shows the statistics for this experiment. The top part of the table shows the results for the model checking traces. The bottom part shows the results for the ISCAS85 circuits. We have also included the *"Average / Total # of Ops"* column to show the results for the average number of sub-operations processed per pass, normalized against the total amount of work performed.

The results show that on average, 10 out of 16 model checking traces processed less than 300 sub-operations (less than one 8-KByte memory page) in each pass. Overall, the average number of sub-operations in a breadth-first queue is at most 4685, which is less than 16 memory pages (128 KBytes). This number is quite small given that hundreds of MBytes of total memory are used. This shows that for these traces, the breadth-first approaches are not very effective in clustering accesses.

Trace	# of Ops Processed per Queue Visit			Average / Total # of Ops
	Average	Max.	Std. Dev.	($\times 10^{-6}$)
abp11	228	41108	86.43	1.86
dartes	27	969	12.53	3.56
dme2-16	34	8122	17.22	0.31
dpd75	15	186	4.75	0.32
ftp3	1562	149792	63.11	8.80
furnace17	75	131071	42.40	2.38
futurebus	2176	207797	76.50	0.23
key10	155	31594	48.23	1.70
mmgt20	66	4741	21.67	1.73
motors-stuck	11	41712	50.14	0.39
over12	282	28582	55.60	3.32
phone-async	1497	175532	87.95	3.53
phone-sync-CW	1176	186937	80.83	0.19
tcas	1566	228907	69.86	1.16
tomasulo	2719	182582	71.20	1.95
valves-gates	25	51039	70.41	0.55
c2670	3816	147488	71.18	204.65
c3540	1971	219090	45.49	34.87
c6288-13	4594	229902	24.92	69.52
c6288-14	4685	237494	42.29	23.59

Fig. 8. Statistics for memory locality in the breadth-first approach.

Another interesting result is that the maximum number of nodes in the queues is quite large and is generally more than 100 standard deviations away from the average. This result suggests that some depth-first and breadth-first hybrid (perhaps as an extension to what is done in the CAL package) may obtain further performance improvements.

The result for *"Average / Total # of Ops"* clearly shows that for the same amount of work, the ISCAS85 computations have much better locality for the breadth-first approaches. Thus, for a comparable level of "*breadth-first*" locality, model checking applications might need to be much larger than the combinational circuit applications.

We have also studied the effects of the breadth-first approach's memory locality when the computations do not fit in the main memory. This experiment was performed by varying the size of the physical memory. The results show that the breadth-first based packages are significantly better only for the three largest cases (largest in terms of memory usage). The results are not very conclusive because as an artifact of this BDD study, the participating BDD packages tend to use a lot more memory than they did before the study began, and furthermore, since these BDD packages generally do not adjust memory usage based on the actual physical memory sizes and page fault rates, the results are heavily influenced by excessive memory usage. Thus, they do not accurately reflect the effects of the memory locality of the breadth-first approach.

5 Phase 2 Results: Dynamic Variable Reordering

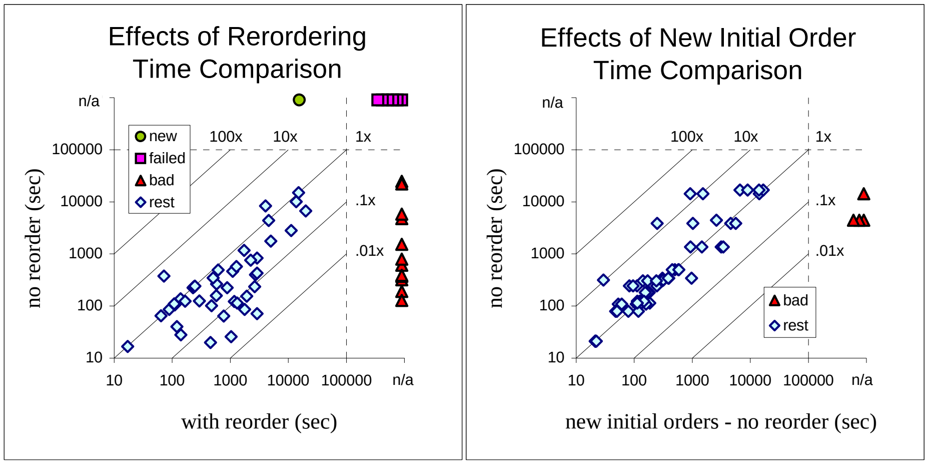
Dynamic variable reordering is inherently difficult for many reasons. First, there is a tradeoff between time spent in variable reordering and the total elapsed time. Second, small changes in the triggering and termination criteria may have significant impact in both the space and time requirements. Another difficulty is that because the space of possible variable orders is so huge and variable reordering algorithms tend to be very expensive, many machines are required to perform a comprehensive study. Due to these inherent difficulties and lack of resources, we were only able to obtain very preliminary results and have performed only one round of evaluation.

For this phase, only the CAL, CUDD, EHV, and TiGeR BDD packages were used, since the ABCD and PBF packages have no support for dynamic variable reordering. There are 4 packages and 16 traces, for a total of 64 cases. Figure 9 presents the timing results for these 64 cases. In this figure, the cases that did not complete within the resource limits are marked with *n/a*. The speedup lines ranging from 0.01x to 100x are included to help classify the performance results.

Figure 9(a) compares running time with and without dynamic variable reordering. With dynamic variable reordering enabled, 19 cases do not finish within the resource limits. Six of these 19 cases also cannot finish without variable reordering (the *failed* cases in Fig. 9(a)). Thirteen of these 19 cases are doable without dynamic variable reordering enabled (the *bad* cases in Fig. 9(a)). There is one case that does not finish without dynamic variable reordering, but finishes with dynamic variable reordering enabled (the *new* in Fig. 9(a)). The remaining 45 cases are marked as the *rest* in Fig. 9(a). These results show that given reasonably good initial orders (e.g., those provided by the original authors of these SMV models), dynamic variable reordering generally slows down the computation. This slowdown may be partially caused by the cache flushing in the dynamic variable reordering phase; i.e., given the importance of the computed cache, cache flushing can increase the number of repeated subproblems.

To evaluate the quality of the orders produced, we used the final orders produced by the dynamic variable reordering algorithms as new initial orders and reran the traces without dynamic variable reordering. Then we compared these results with the results obtained using the original initial order and also without dynamic variable reordering. This comparison is one good way of evaluating the quality of the variable reordering algorithms since in practice, good initial variable orders are often obtained by iteratively feeding back the resulting variable orders from the previous variable reordering runs.

Figure 9(b) plots the results for this experiment. The y-axis represents the cases using the original initial variable orders. The x-axis represents the cases where the final variable orders produced by the dynamic variable reordering algorithms are used as the initial variable orders. In this figure, the cases that finished using the original initial orders but failed using the new initial orders are marked as the *bad* and the remaining cases are marked as the *rest*. The results show that improvements can still be made from the original variable orders. A few cases even achieved a speedup of over 10.



(a)

(b)

Fig. 9. Overall results for variable reordering. The *failed* cases represent those that always exceed the resource limits. The *bad* cases represent those that are originally doable but failed with the new setup. The *rest* represent the remaining cases. (a) Timing comparison between with and without dynamic variable reordering. (b) Timing comparison between original initial variable orders and new initial variable orders. The new initial variable orders are obtained from the final variable orders produced by the dynamic variable reordering algorithms. For results in (b), dynamic variable reordering is disabled.

The remainder of this section presents results of a limited set of experiments for characterizing dynamic variable reordering. We first present the results on two heuristics for dynamic variable reordering. Then we present results on sensitivity of dynamic variable reordering to the initial variable orders. For these experiments, only the CUDD package is used. Note that the results in this section are very limited in scope and are far from being conclusive. Our intent is to suggest new research directions for dynamic variable reordering.

5.1 Present and Next State Variable Grouping

We set up an experiment to study the effects of *variable grouping*, where the grouped variables are always kept adjacent to each other.

Hypothesis 6 *Pairwise grouping of present state variables with their corresponding next state variables is generally beneficial for dynamic variable reordering.*

Experiment: Measure the effects of this grouping on the number of subproblems (work), maximum number of live BDD nodes (space), and number of nodes swapped with their children during dynamic variable reordering (reorder cost).

Results: Figure 10 plots the effects of grouping on work (Fig. 10(a)), space (Fig. 10(b)), and reorder cost (Fig. 10(c)). Note that the results for two traces are not available. One trace (*tomasulo*) exceeded the memory limit, while the other (*abp11*) is too small to trigger variable reordering.

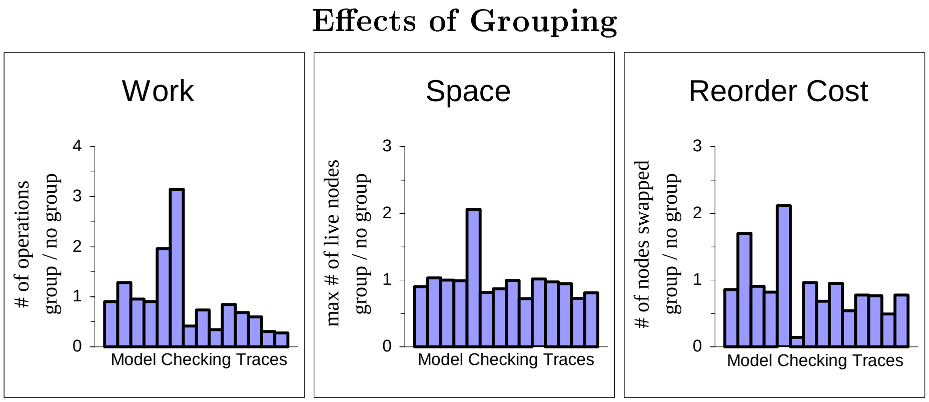
These results show that pairwise grouping of the present and the next state variables is a good heuristic in general. However, there are a couple of exceptions. A better solution might be to use the grouping initially and relax the grouping criteria somewhat as the reordering process progresses.

5.2 Reordering the Transition Relations

Since the BDDs for the transition relations are used repeatedly in model checking computations, we set up an experiment to study the effects of reordering the BDDs for the transition relations.

Hypothesis 7 *Finding a good variable order for the transition relation is an effective heuristic for improving overall performance.*

Experiment: Reorder variables once, immediately after the BDDs for the transition relations are built, and measure the effect on the number of subproblems (work), maximum number of live BDD nodes (space), and number of nodes swapped with their children during dynamic variable reordering (reorder cost).



(a) (b) (c)

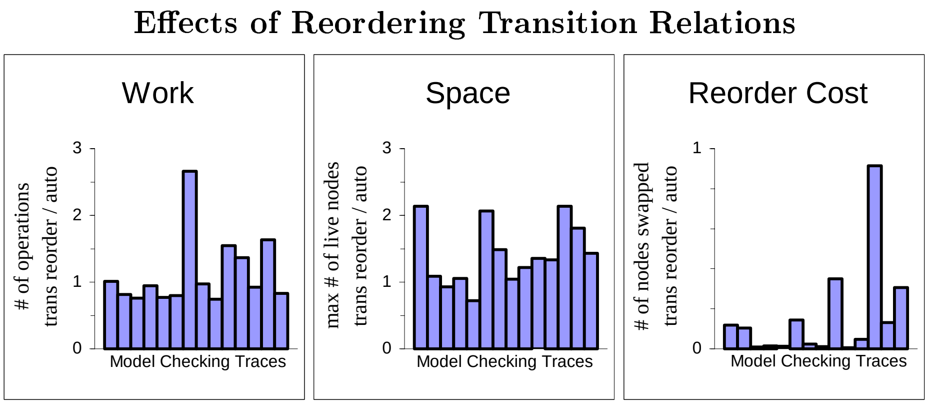
Fig. 10. Effects of pairwise grouping of the current and next state variables on (a) the number of subproblems, (b) the number of maximum live BDD nodes, and (c) the amount of work in performing dynamic variable reordering.

Results: Figure 11 plots the results of this experiment on work (Fig. 11(a)), space (Fig. 11(b)), and reorder cost (Fig. 11(c)). The results are normalized against the results from automatic dynamic variable reordering for comparison purposes. Note that the results for two traces are not available. With automatic dynamic variable reordering, one trace (*tomasulo*) exceeded the memory limit, while the other (*abp11*) is too small to trigger variable reordering.

The results show that reordering once, immediately after the construction of transition relations' BDDs generally works well in reducing the number of subproblems (Fig. 11(a)). This heuristic's effects on the maximum number of live BDD nodes is mixed (Fig. 11(b)). Figure 11(c) shows that this heuristic's reordering cost is generally much lower than automatic dynamic variable reordering. Overall, the the number of variable reordering for automatic dynamic variable reordering is 5.75 times the variable reordering frequency using this heuristic. These results are not strong enough to support our hypothesis as cache flushing may be the main factor for the effects on the number of subproblems. However, it does provide an indication that the automatic dynamic variable reordering algorithm may be invoking the variable reordering process too frequently.

5.3 Effects of Initial Variable Orders

In this section, we study the effects of initial variable orders on BDD construction with and without dynamic variable reordering. We generate a suite of initial variable orders by perturbing a set of good initial orders. In the following, we



(a) (b) (c)

Fig. 11. Effects of variable reordering the transition relations on (a) the number of subproblems, (b) the number of maximum live BDD nodes, and (c) the amount of work in performing variable reordering. For comparison purposes, all results are normalized against the results for automatic dynamic variable reordering.

describe this experimental setup in detail and then present some hypotheses along with supporting evidence.

Experimental Setup

The first step is the selection of good initial variable orders — one for each model checking trace. The quality of an initial variable order is evaluated by the running time using this order without dynamic variable reordering.

Once the best initial variable order is selected, we perturb it based on two perturbation parameters: the probability (p), which is the probability that a variable will be moved, and the distance (d), which controls how far a variable may move. The perturbation algorithm used is shown in Figure 12. Initially, each variable is assigned a weight corresponding to its variable order (line 1). If this variable is chosen (with the probability of p) to be perturbed (by the distance parameter d), then we change its weight by δw, where δw is chosen randomly from the range $[-d, d]$ (lines 3-5). At the end, the perturbed variable order is determined by sorting the variables based on their final weights (line 6). This algorithm has the property that on average, p fraction of the BDD variables are perturbed and each variable's final variable order is at most $2d$ away from its initial order. Another property is that the perturbation pair ($p = 1, d = \infty$) essentially produces a completely random variable order.

Since randomness is involved in the perturbation algorithm, to gain better statistical significance, we generate multiple initial variable orders for each pair of perturbation parameters (p, d). For each trace, if we study n_p different perturbation probabilities, n_d different perturbation distances, and k initial orders for each perturbation pair, we will generate a total of kn_pn_d different initial variable

```
perturb_order($v[n]$, $p$, $d$)
    /* perturb the variable order with probability $p$ and distance $d$.
       $v[\,]$ is an array of $n$ variables sorted based on decreasing
       variable order precedence. */
    for ($0 \leq i < n$) $w[i] \leftarrow i$ /* initialize weight */
    for ($0 \leq i < n$) /* for each variable, with probability $p$, perturb its weight. */
        With probability $p$ do
            $\delta w \leftarrow$ randomly choose an integer from $[-d, d]$
            $w[i] \leftarrow w[i] + \delta w$
    sort variables in array $v[\,]$ based on increasing weight $w[\,]$
    return $v[\,]$
```

Fig. 12. Variable-order perturbation algorithm.

orders. For each initial variable order, we compare the results with and without dynamic variable reordering enabled. Thus, for each trace, there will be $2kn_pn_d$ runs. Due to lack of time and machine resources, we were only able to complete this experiment for one very small trace — *abp11*.

The perturbed initial variable orders were generated from the best initial variable ordering we found for *abp11*. Using this order, the *abp11* trace can be executed (with dynamic variable reordering disabled) using 12.69 seconds of CPU time and 127 MBytes of memory on a 248 MHz UltraSparc II. This initial order and its results are used as the base case for this experiment. Using this base case, we set the time limit of each run to 1624.32 seconds (128 times the base case) and 500 MBytes of memory.

For the perturbation parameters, we let p range from 0.1 to 1.0 with an increment of 0.1. Since *abp11* has 122 BDD variables, we let d range from 10 to 100 with an increment of 10 and added the case for $d = \infty$. These choices result in 110 perturbations pairs (with $n_p = 10$ and $n_d = 11$). For each perturbation pair, we generate 10 initial variable orders ($k = 10$). Thus, there are a total of 1100 initial variable orders and 2200 runs.

Results for *abp11*

Hypothesis 8 *Dynamic variable reordering improves the performance of model checking computations.*

Supporting Results: Figure 13 plots the number of cases that did not complete within various time limits for runs with and without dynamic variable reordering. For these runs, the memory limit is fixed at 500 MBytes. The time limits in this plot are normalized to the base case of 12.69 seconds and are plotted in log scale.

The results clearly show that given enough time, the cases with dynamic variable reordering perform better. Overall, with a time limit of 128 times the base case, only 10.1% of cases with dynamic variable reordering exceeded the resource limits. In comparison, 67.6% of cases without dynamic variable reordering failed to complete.

Note that for the time limit of 2 times the base case (the > 2x case in the chart), the results with dynamic variable reordering is worse. This reflects the fact that dynamic variable reordering can be expensive. As the time limit increases, the number of unfinished cases for with dynamic variable reordering drops more quickly until at about 32 times the base case. After this point, the number of unfinished cases for both with and without dynamic variable reordering appear to be decreasing at about the same rate.

Another interesting result is that none of the cases takes less time to complete than the base case of 12.69 seconds (i.e., the > 1x results are both 1100). This result indicates that the initial variable order of our base case is indeed a very good variable order.

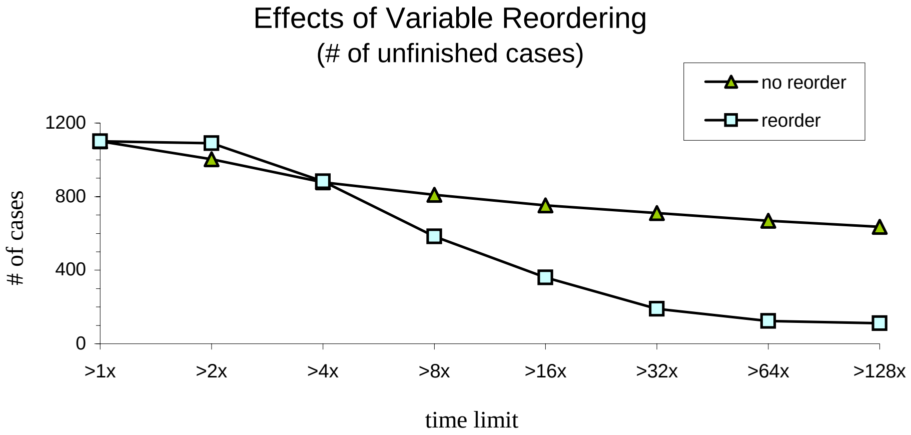


Fig. 13. Effects of variable reordering on *abp11*. This chart plots the number of unfinished cases for various time limits. The time limits are normalized to the base case of 12.69 seconds. The memory limit is set at 500 MBytes.

To better understand the impact of the perturbations on running time, we analyzed the distribution of these results (in Fig. 13) across the perturbation space and formed the following hypothesis.

Hypothesis 9 *The dynamic variable reordering algorithm performs "unnecessary" work when it is already dealing with reasonably good variable orders. Overall, given enough time, dynamic variable reordering is effective in recovering from poor initial variable order.*

Supporting Results: Figure 14(a) shows the results with a time limit of 4 times the base case of 12.69 seconds. These plots show that when there are small perturbations ($p = 0.1$ or $d = 10$), we are better off without dynamic variable reordering. However, for higher levels of perturbations, the cases with dynamic variable reordering usually does a little better.

Figures 14(b) and 14(c) show the results with time limits of 32 and 128 times, respectively, the base case. Note that since 128 times is the maximum time limit we studied, Fig. 14(c) also represents the distribution of the cases that did not complete at all for this study. These results clearly show that given enough time, the cases with dynamic variable reordering perform much better.

Hypothesis 10 *The quality of initial variable order affects the space and time requirements, with or without dynamic variable reordering.*

Supporting Results: Figure 15 classifies the unfinished cases into memory-out (Fig. 15(a)) or timed-out (Fig. 15(b)). For clarity, we repeated the plots for the total number of unfinished cases (memory-out plus timed-out results) in Fig. 15(c). It is important to note that because the BDD packages used in this study still do not adapt *very well* upon exceeding memory limits, memory-out cases should be interpreted as indications of high memory pressure instead of that these cases inherently do not fit within the memory limit.

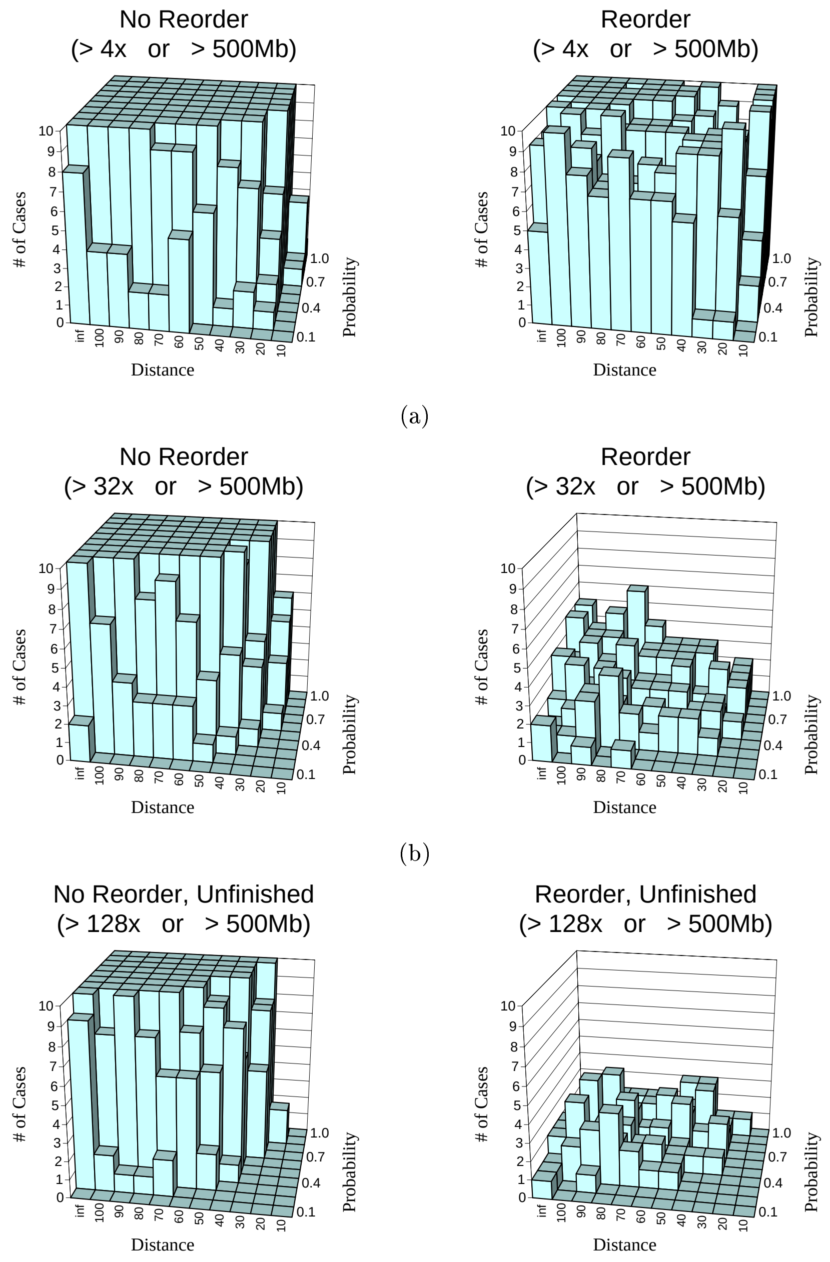
The results show that levels of perturbation directly influence the time and memory requirement. With a very high level of perturbation, most of the unfinished cases are due to exceeding the memory limit of 500 MBytes (the upper-left triangular regions in Fig. 15(a)). For a moderate level of perturbation, most of the unfinished cases are due to the time limit (the diagonal bands from the lower-left to the upper-right in Fig. 15(b)).

Note that the results in Fig. 15 are not very monotonic; i.e., the results are not necessarily worse with a larger degree of perturbation. This leads to the next hypothesis.

Hypothesis 11 *The effects of the dynamic variable reordering algorithm and the initial variable orders are very chaotic.*

Supporting Results: Fig. 16 plots the standard deviation of running time normalized against average running time. For the cases that cannot complete within the resource limits, they are included as if they use exactly the time limit. Note that as an artifact of this calculation, when all 10 variants of a perturbation pair exceed the resource limits, the standard deviation is 0. In particular, without variable reordering, none of the cases can be completed in the highly perturbed region (upper-left triangular region in Fig 15(c)) and thus these results are all shown as 0 in the chart.

The results show that the standard deviations are generally greater than the average time (i.e., with the normalized result of > 1). This finding partially

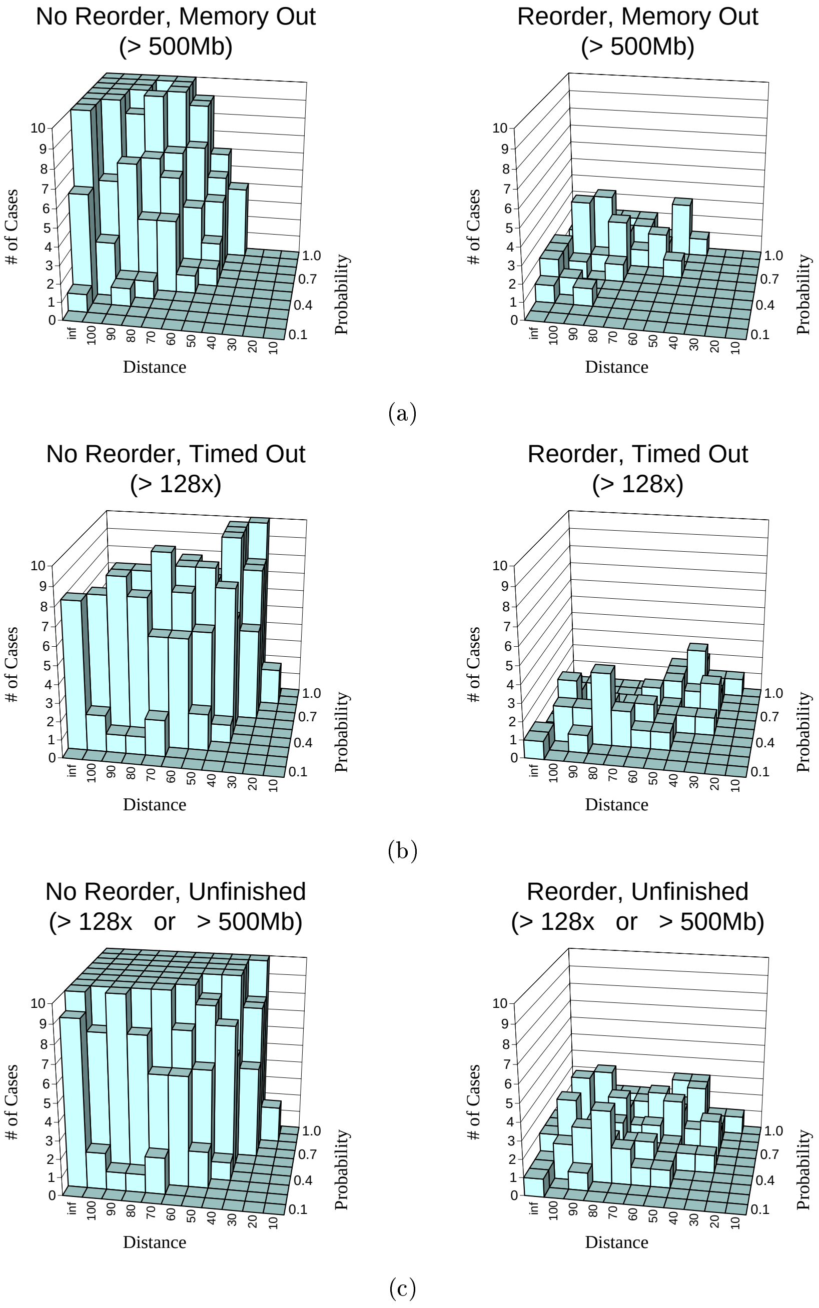


(a)

(b)

(c)

Fig. 14. Histograms on the number of cases that cannot be finished within the specified resource limits. For all cases, the memory limit is set at 500 MBytes. The time limit varies from (a) 4 times, (b) 32 times, to (c) 128 times the base case of 12.69 seconds.



(a)

(b)

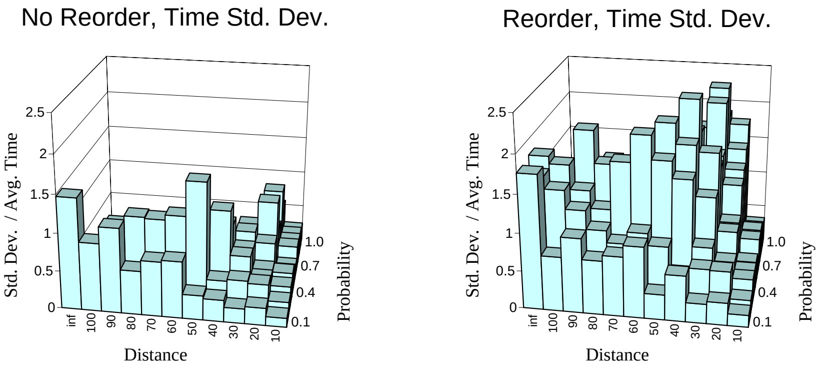
(c)

Fig. 15. Breakdown on the cases that cannot be finished. (a) memory-out cases, (b) timed-out cases, (c) total number of unfinished cases.

confirms our hypothesis. It also indicates that 10 initial variable orders per perturbation pair (p, d) is probably too small for some perturbation pairs.

The results also show that with very low level of perturbation (lower-right triangular region), the normalized standard deviation is generally smaller. This gives an indication that higher perturbation level may result in more unpredictable performance behavior.

Furthermore, the normalized standard deviation for without dynamic variable reordering is generally smaller than the same statistic for with dynamic variable reordering. This result provides an indication that dynamic variable reordering may also have very unpredictable effects.



(a)

(b)

Fig. 16. Standard deviation of the running time for *abp11* with perturbed initial variable orders (a) without dynamic variable reordering, and (b) with dynamic variable reordering. Each results are normalized to the average running time.

6 General Results

This section presents results which may be generally helpful in studying or improving BDD packages.

Hash Function

Hashing is a vital part of BDD construction since both the uniqueness of BDD nodes and the cache accesses are based on hashing. Currently, we have not found any theoretically good hash functions for handling multiple hash keys. In this study, we have empirically found that the hash function used by the TiGeR BDD package worked well in distributing the nodes. This hash function is of the form

$$H(k_1, k_2) = ((k_1 p_1 + k_2) p_2)/2^{w-n}$$

where k's are the hash keys, p's are sufficiently large primes, w is the number of bits in an integer, and 2^n is the size of the hash table. Note that division by 2^{w-n} is used to extract the n most significant bits and is implemented by right shifting $(w - n)$ bits.

The basic idea is to distribute and combine the bits in the hash keys to the higher order bits by using integer multiplications, and then to extract the result from the high order bits. The power-of-2 hash table size is used to avoid the more expensive *modulus* operation. Some small speedups have been observed using this hash function. One pitfall is that for backward compatibility reasons, some compilers might generate a function call to compute integer multiplication, which can cause significant performance degradation (up to a factor of 2). In these cases, architecture-specific compiler flags can be used to ensure the integer-multiplier hardware is used instead.

Caching Strategy

Given the importance of cache, a natural question is: *Can we cache more intelligently?* One heuristic, used in CUDD, is that the cache is accessed only if at least one of the arguments has a reference count greater than 1. This technique is based on the fact that if all arguments have reference counts of 1, then this subproblem is not likely to be repeated within the current top-level operation. In fact, if a complete cache is used, this subproblem will not be repeated within the same top-level operation. Using this technique, CUDD is able to reduce the number of cache lookups by up to half, with a total time reduction of up to 40%.

Relational Product Algorithm

The relational product algorithm in Fig. 1 can be further improved. The new optimizations are based on the following derivations. Let r_0 be the result of the 0-cofactors (line 4 in Fig. 1), $\boldsymbol{v}$ be the set of variables to be quantified, and h be any Boolean function, then

$$r_0 \vee (\exists \boldsymbol{v}.r_0 \wedge h) = r_0 \vee (r_0 \wedge \exists \boldsymbol{v}.h) = r_0$$

and

$$r_0 \vee (\exists \boldsymbol{v}.(\neg\, r_0) \wedge h) = r_0 \vee ((\neg\, r_0) \wedge \exists \boldsymbol{v}.h) = r_0 \vee \exists \boldsymbol{v}.h$$

The validity comes from the fact that r_0 does not depend on the variables in $\boldsymbol{v}$. Based on these equations, we can add the following optimizations (between line 7 and line 8 in Fig. 1) to the relational product algorithm:

```
7.1    else if $(r_0 == f|_{\tau \leftarrow 1})$ or $(r_0 == g|_{\tau \leftarrow 1})$
7.2        $r \leftarrow r_0$
7.3    else if $(r_0 == \neg f|_{\tau \leftarrow 1})$
7.4        $r \leftarrow r_0 \vee (\exists \boldsymbol{v}.g|_{\tau \leftarrow 1})$
7.5    else if $(r_0 == \neg g|_{\tau \leftarrow 1})$
7.6        $r \leftarrow r_0 \vee (\exists \boldsymbol{v}.f|_{\tau \leftarrow 1})$
```

In general, these optimizations only slightly reduces the number of subproblems, with the exception of the *futurebus* trace, where the number of subproblems is reduced by over 20%.

BDD Package Comparisons

In comparing BDD packages, one fairness question is often raised: *Is it fair to compare the performance of a bare-bones experimental BDD package with a more complete public domain BDD package?* This question arises particularly when one package supports dynamic variable reordering, while the other does not. This is an issue because supporting dynamic variable reordering requires additional data structures and indirection overheads to the computation for BDD construction. To partially answer this question, we studied a package with and without its support for variable reordering in place. Our preliminary results show that the additional overhead to support dynamic variable reordering has no measurable performance impact. This may be due to the fact that BDD computation is so memory intensive, a couple additional non-memory intensive operations can be scheduled either by the hardware or the compiler without any measurable performance penalty.

Cache Hit Rate

The computed cache hit rate is not a reliable measure of overall performance. In fact, it can be shown that when the cache hit rate is less than 49%, a cache miss can actually result in a higher hit rate. This is because a cache miss generates more subproblems and these subproblems' results could have already been computed and are still in cache.

Platform Independent Metrics

Throughout this study, we have found several useful machine-independent metrics for characterizing the BDD computations. These metrics are:

- the *number of subproblems* as a measure for work,
- the *maximum number of live nodes* as a measure for the lower bound on memory requirement,
- the *number of subproblems processed for each breadth-first queue visit* to reflect the possibility of exploiting memory locality using the breadth-first traversal, and
- the *number of nodes swapped with their children during dynamic variable reordering* as a measure of the amount of work performed in dynamic variable reordering.

7 Issues and Open Questions

Cache Size Management

In this study, we have found that the size of the compute cache can have a significant impact on model checking computations. Given that BDD computations are very memory intensive, there is an inherent conflict between using a larger cache for better performance and using a smaller cache to conserve memory usage. For BDD packages that maintain multiple compute caches, there are additional conflicts as these caches will compete with each other for the memory resources. As the problem sizes get larger, finding a good dynamic cache management algorithm will become more and more important for building an efficient BDD package.

Garbage Collection Triggering Algorithm

Another dynamic memory management issue is the frequency of garbage collection. The results in Fig. 6(b) clearly suggest that delaying garbage collection can be very beneficial. Again, this is a space and time tradeoff issue. One possibility is to invoke garbage collection when the percentage of unreachable nodes is high and the rebirth rate is low. Note that for BDD packages that do not maintain reference counts, the rebirth rate statistic is not readily available and thus a different strategy is needed.

Resource Awareness

Given the importance of space and time tradeoff, a commercial strength BDD package not only needs to know when to gobble up the memory to reduce computation time, it should also be able to free up space under resource contention. This contention could come from different parts of the same tool chain or from a completely different job. One way to deal with this issue is for BDD packages to become more aware of the environment, in particular, the available physical memory, various memory limits, and the page fault rate. This information is readily available to the users of modern operating systems. Several of the BDD packages used in this study already have some limited form of resource awareness. However, this problem is still not well understood and probably cannot be easily studied using the trace-driven framework.

Cross Top-Level Sharing

For the model checking traces, why are there so many subproblems repeated across the top-level operations? We have two conjectures. First, there is quite a bit of symmetry in some of these SMV models. These inherent symmetries are *somehow* captured by the BDD representation. If so, it might be more effective to use higher level algorithms to exploit the symmetries in the models. The other conjecture is that the same BDDs for the transition relations are used repeatedly throughout model checking in the fixed-point computations. This repeated use of the same set of BDDs increases the likelihood of the same subproblems being repeated across top-level operations. At this point, we do not know how to validate these conjectures. To better understand this property, one starting point would be to identify how far apart are these cross top-level repeated subproblems; i.e., is it within one state transition, within one fixed-point computation, within one temporal logic operator, or across different temporal logic operators?

Breadth-First's Memory Locality

In this study, we have found no evidence that breadth-first based techniques have any advantage when the computation fits in the main memory. An interesting question would be: *As the BDD graph sizes get much larger, is there going to be a crossover point where the breadth-first packages will be significantly better?* If so, another issue would be finding a good depth-first and breadth-first hybrid to get the best of both worlds.

Inconsistent Cross Platform Results

Inconsistency in timing results across machines is yet another unresolved issue in this study. More specifically, for some BDD packages, the CPU-time results on a UltraSparc II machine are up to twice as long as the corresponding results on a PentiumPro, while for other BDD packages, the differences are not so significant. Similar inconsistencies are also observed in the Sentovich study [23]. A related performance discrepancy is that for the depth-first based packages, the garbage collection cost for UltraSparc II is generally twice as high as that of PentiumPro. However, for the breadth-first based packages, the garbage collection performances between these two machines are much closer. In particular, for one breadth-first based package, the ratio is very close to 1. This discrepancy may be a reflection of the memory locality of these BDD packages. To test this conjecture, we have performed a set of simple tests using synthetic workloads. Unfortunately, the results did not confirm this hypothesis. However, the results of this test do indicate that our PentiumPro machine appears to have a better memory hierarchy than our UltraSparc II machine. A better understanding of this issue can probably shed some light on how to improve memory locality for BDD computations.

Pointer- vs. Index-Based References

Another issue is that within the next ten years, machines with memory sizes greater than 4 GBytes are going to become common. Thus the size of a pointer (i.e., memory address) will increase from 32 to 64 bits. Since most BDD packages today are pointer-based, the memory usage will double on 64-bit machines. One way to reduce this extra memory overhead is to use integer indices instead of pointers to reference BDDs as in the case of the ABCD package. One possible drawback of an index-based technique is that an extra level of indirection is introduced for each reference. However, since ABCD's results are generally among the best in this study, this provides a positive indication that the index-based approach may be a feasible solution to this impending memory overhead problem.

Computed Cache Flushing in Dynamic Variable Reordering

In Sec. 5, we showed that dynamic variable reordering can generally slow down the entire computation when given a reasonably good initial variable order. Since the computed cache is typically flushed when dynamic variable reordering takes place, it would be interesting to study what percentage of the slowdown is caused by an increase in the amount of work (number of subproblems) due to cache flushing. If this percentage is high, then another interesting issue would be in finding a good way to incorporate the cache performance as a parameter for controlling dynamic variable reordering frequency.

8 Related Work

In [23], Sentovich presented a BDD study comparing the performance of several BDD packages. Her study covered building output BDDs for combinational circuits, computing reachability of sequential circuits, and variable reordering.

In [15], Manne *et al.* performed a BDD study examining the memory locality issues for several BDD packages. This work compares the hardware cache miss rates, TLB miss rates, and page fault rates in building the output BDDs for combinational circuits.

In contrast to the Sentovich study, our study focuses in characterizing the BDD computations instead of doing a performance comparison of BDD packages. In contrast to the Manne study, our work uses platform independent metrics for performance evaluation instead of hardware specific metrics. Both types of metrics are equally valid and complementary. Our study also differs from these two prior studies in that our performance evaluation is based on the execution of a model checker instead of benchmark circuits.

9 Summary and Conclusions

By applying a new evaluation methodology, we have not only achieved significant performance improvements, we have also identified many interesting characteristics of model checking computations. For example, we have confirmed that model checking and combinational circuit computations have fundamentally different performance characteristics. These differences include the effects of the cache size, the garbage collection frequency, the complement edge representation, and the memory locality for the breadth-first BDD packages. For dynamic variable reordering, we have introduced some new methodologies for studying the effects of variable reordering algorithms and initial variable orders. From these experiments, we have uncovered a number of open problems and future research directions.

As this study is very limited in scope, especially for the dynamic variable reordering phase, further validations of the hypotheses are necessary. It would be especially interesting to repeat the same experiments on execution traces from other BDD-based tools.

The results obtained in this study clearly demonstrate the usefulness of systematic performance characterization and validate our evaluation methodology. We hope that the trace-drive framework and the machine-independent metrics will help lay the foundation for future benchmark collection and performance-characterization methodology.

Acknowledgement

We thank Yirng-An Chen for providing the ISCAS85 circuits in the trace format and providing the software which forms the core of our evaluation tool.

We thank Claudson F. Bornstein and Henry R. Rowley for numerous discussions on experimental setups and data presentation. We are grateful to Edmund M. Clarke for providing additional computing resources. This research is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, in part by DARPA under contract number DABT63-96-C-0071, in part by Cadence Design Systems, and in part by SRC contract 96-DJ-560. This work utilized Silicon Graphics Origin 2000 machines from the National Center for Supercomputing Applications at Urbana-Champaign.

References

[1] Akers, S. B. Functional testing with binary decision diagrams. In *Proceedings of Eighth Annual International Conference on Fault-Tolerant Computing* (June 1978), pp. 75–82.

[2] Ashar, R., and Cheong, M. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1994), pp. 622–627.

[3] Biere, A. ABCD: an experimental BDD library, 1998. `http://iseran.ira.uka.de/~armin/abcd/`.

[4] Brace, K., Rudell, R., and Bryant, R. E. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 40–45.

[5] Brglez, F., Bryan, D., and Kozmiski, K. Combinational profiles of sequential benchmark circuits. In *1989 International Symposium on Circuits And Systems* (May 1989), pp. 1924–1934.

[6] Brglez, F., and Fujiwara, H. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *1985 International Symposium on Circuits And Systems* (June 1985). Partially described in F. Brglez, P. Pownall, R. Hum. Accelerated ATPG and Fault Grading via Testability Analysis. In *1985 International Symposium on circuits and Systems*, pages 695-698, June 1985.

[7] Bryant, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers C-35*, 8 (August 1986), 677–691.

[8] Bryant, R. E. Symbolic Boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys 24*, 3 (September 1992), 293–318.

[9] Bryant, R. E. Binary decision diagrams and beyond: Enabling technologies for formal verification. In *Proceedings of the International Conference on Computer-Aided Design* (November 1995), pp. 236–243.

[10] Burch, J. R., Clarke, E. M., Long, D. E., McMillan, K. L., and Dill, D. L. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 13*, 4 (April 1994), 401–424.

[11] Coudert, O., Berthet, C., and Madre, J. C. Verification of sequential machines using Boolean functional vectors. In *Proceedings of the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design* (November 1989), pp. 179–196.

[12] Coudert, O., and Madre, J. C. A unified framework for the formal verification of circuits. In *Proceedings of the International Conference on Computer-Aided Design* (Feb 1990), pp. 126–129.

[13] Coudert, O., Madre, J. C., and Touati, H. *TiGeR Version 1.0 User Guide*. Digital Paris Research Lab, December 1993.

[14] Janssen, G. *The Eindhoven BDD Package*. University of Eindhoven. Anonymous FTP address: `ftp://ftp.ics.ele.tue.nl/pub/users/geert/bdd.tar.gz`.

[15] Manne, S., Grunwald, D., and Somenzi, F. Remembrance of things past: Locality and memory in BDDs. In *Proceedings of the 34th ACM/IEEE Design Automation Conference* (June 1997), pp. 196–201.

[16] McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] Minato, S., Ishiura, N., and Jajima, S. Shared binary decision diagram with attributed edges for efficient Boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Design Automation Conference* (June 1990), pp. 52–57.

[18] Ochi, H., Ishiura, N., and Yajima, S. Breadth-first manipulation of SBDD of Boolean functions for vector processing. In *Proceedings of the 28th ACM/IEEE Design Automation Conference* (June 1991), pp. 413–416.

[19] Ochi, H., Yasuoka, K., and Yajima, S. Breadth-first manipulation of very large binary-decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1993), pp. 48–55.

[20] Ranjan, R. K., and Sanghavi, J. CAL-2.0: Breadth-first manipulation based BDD library. Public software. University of California, Berkeley, CA, June 1997. `http://www-cad.eecs.berkeley.edu/Research/cal_bdd/`.

[21] Ranjan, R. K., Sanghavi, J. V., Brayton, R. K., and Sangiovanni-Vincentelli, A. High performance BDD package based on exploiting memory hierarchy. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996), pp. 635–640.

[22] Rudell, R. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design* (November 1993), pp. 139–144.

[23] Sentovich, E. M. A brief study of BDD package performance. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 389–403.

[24] Sentovich, E. M., Singh, K. J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P. R., Brayton, R. K., and Sangiovanni-Vincentelli., A. L. SIS: A system for sequential circuit synthesis. Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, University of California, May 1992.

[25] Somenzi, F. CUDD: CU decision diagram package. Public software. University of Colorado, Boulder, CO, April 1997. `http://vlsi.colorado.edu/~fabio/`.

[26] Yang, B., Chen, Y.-A., Bryant, R. E., and O'Hallaron, D. R. Space- and time-efficient BDD construction via working set control. In *1998 Proceedings of Asia and South Pacific Design Automation Conference* (Feb 1998), pp. 423–432.