A Brief Study of BDD Package Performance

Ellen M. Sentovich

Ecole Nationale Supérieure des Mines de Paris
Centre de Mathématiques Appliquées
B.P. 207, F
06904 Sophia-Antipolis Cedex, FRANCE
ellen@cma.cma.fr

Abstract. The use of BDD techniques has become essential in a variety of CAD problems, and particularly in employing formal methods for design and verification. There are now many BDD packages available with many different features. Given a particular application, it is difficult to choose which package is best-suited for solving the problem, despite extensive published results on the packages. This paper contains a study of a few state-of-the art BDD packages. Results comparing the packages are reported on a variety of examples, for a variety of applications, and on several computing platforms. Great care has been taken to make fair and meaningful comparisons. The study is offered to compare current available BDD technology, and to encourage BDD package authors to use this experimental foundation and to make their results and code available to the research community.

1 Introduction

It has been 10 years since Bryant's publication [3] of BDD (*binary decision diagram*) algorithms launched intense activity in BDD theory and package development. There are now many flavors of BDDs, a large body of published work both theoretical and practical, and a number of BDD packages in use. It is difficult to judge their relative merit based on the results published in the papers. Though there are theoretical results on size and space bounds for the algorithms for various definitions of BDDs, these cannot predict the performance for a given application. The practical results published in the result tables for well-known benchmarks are difficult to assess as well. Experiments are run on a variety of platforms and under a variety of computing conditions. Results are reported on different sets of metrics, usually designed to highlight the features of the new package. Although care has been taken to clearly report, for example, the number of BDD nodes, the size of each node, the size of auxiliary hash tables, etc., even these concrete numbers must be computed and examined carefully.[1] For the BDD package user presented with an application and a set

[1] Size has been a particularly troublesome measure for BDD packages, and is nearly impossible to estimate based on the BDD. Packages can allocate memory for several nodes at once even though they are not yet needed, garbage collection can be invoked at any time, memory can be reclaimed in different ways, etc.

of BDD packages, it is difficult to choose the most appropriate package.

The CAD community, and particularly in the formal methods area, is using BDDs ubiquitously – in formal verification, logic synthesis, test generation, and simulation. Further, the research community heavily uses available BDD packages for experimentation with new algorithms. The issue of selecting an appropriate package is critical and will remain so. Often heard in the conference corridors is "and which BDD package did you use?" followed moments later by "but I hear N's is X-times faster." There is no consensus or basis for meaningful comparison. Furthermore, the BDD research community states the new and useful features of their work but not its weaknesses.

This paper contains a brief study and comparison of several state-of-the art, publicly available BDD packages. The goal is to highlight useful features of these packages for several applications, rather than to run a race and declare a winner. The study is oriented toward the BDD user, and as such the packages are applied as is, without tuning parameters per application or per platform. The experiments are performed in a controlled computing environment, with both standard benchmark circuits and new benchmarks generated from higher-level languages. The goals of the study are to provide data on current BDD package technology, to highlight directions for further research, and to encourage further development and reporting to be done under the same, publicly available, experimentation environment.

The remainder of the paper is organized as follows. A brief review of BDDs is given in Section 2. Relevant general BDD package features and brief descriptions of the packages studied here are given in Section 3. The experimental environment is described in Section 4, and the results in Section 5. Conclusions and discussions of other relevant packages to be studied in the future are given in Section 6.

2 Background: BDDs

A description of the original canonical BDD and associated algorithms is found in [3]; the technology is reviewed and applications described in [4]; an update on the most important recent developments is in [5].

Briefly: a BDD is a DAG representation of a function, with each nonterminal node labeled by a variable, each terminal node labeled with a value, and the two children of a node each representing the function with that variable set to one of its two values (0/1). The representation is made canonical by restricting the ordering of the variables and forcing sharing of common subgraphs. This results in a *reduced, ordered binary decision diagram*, or *ROBDD*, herein called a BDD. A canonical BDD admits polynomial time algorithms for many Boolean functions (e.g., verifying the equivalence of two functions); for this reason, BDDs and efficient packages for manipulating them have become essential tools in CAD applications.

There are now many types of BDDs (see the "alphabet soup" list in [5]), which differ in ordering restrictions, functional decomposition, use of edge values, use of multiple terminal values, and reduction rules. The focus here is on the basic *ROBDD*, with *complement edges* and implemented with an *efficient memory management scheme* including caching of intermediate results and storage of unique entries for each variable [2]. These are the essential elements for any state-of-the-art package and are now considered standard. They are the basis for many public and private BDD packages, which are widely used in CAD applications for representation of Boolean circuits and related functions. The packages selected for the study are implemented in this way.

3 BDD Packages

The set of BDD packages has been chosen based on the following requirements.

- They are publically available and free, so the results are immediately useful to the entire community. These packages are excellent and widely used, so this does not restrict the scope to a weak subset of existing packages.
- The packages employ some of the latest algorithms in variable ordering, traversal techniques, memory management, and special applications.
- The source code is available. Thus, features of the packages can be explored and tuned by the general public.

The BDD packages under study are CAL, CMU, CUDD; a description of each follows. A number of other packages were considered to broaden the study, but could not be included at this date for a variety of reasons. They are listed in Section 6.

BDD packages can be distinguished by the following characteristics:

Variable ordering: Since BDD size is dependent on variable ordering, selecting a good ordering is essential. Good heuristics for circuits have been developed [9, 14], but these break down for some examples. It is difficult to determine a good ordering a priori. *Dynamic ordering techniques* (DVO) [21] change the ordering while the BDD is being built to control its size; in particular, *sifting* exploits efficient swapping of adjacent variables to search for better orderings. Many heuristics for employing dynamic ordering have been reported (e.g., [19]).

Memory management: The now-standard memory management scheme is described in [2]. Recent research demonstrates improvements based on localizing memory allocation and access [18, 20]. A report on cache performance of the BDD packages studied here is contained in [15].

Traversal algorithms: The original algorithms traverse the BDD in depth-first order. Breadth-first traversal has been proposed [18, 20] to localize memory access and thus improve performance. In a related development, the addition of temporary new variables, sifting and quantifying them while building BDDs, has been proposed [10] to improve BDD size and run-time during creation.

Specialized algorithms: Some packages have been optimized for particular applications. For example, building partial BDDs on partitions and using these to reorder or obtain a final result without building the complete BDDs has been employed [17, 8, 16]. In another work, the DD data structure and algorithms are optimized to reduce cache misses in MTBDDs [11].

3.1 CAL : UC Berkeley

The CAL package is the latest BDD package from UC Berkeley [20]. Its main goal is to improve performance by using breadth-first traversal and by storing all nodes of a single variable contiguously in memory. This scheme exploits memory hierarchy by reducing accesses *across* the hierarchy (e.g., from cache to main memory). It thus preserves locality of reference in an architecture-independent way. This is particularly important when the size of the BDD exceeds main memory. It also uses pipelining and superscalarity to increase the performance by issuing multiple operations simultaneously. The implementation of the BFS algorithm, which involves a first pass down the BDD to issue computation requests (APPLY) and a second pass up the BDD to process them (REDUCE), implies a complete computed table, or cache of intermediate results. Thus there is never a cache-miss on an intermediate result and this can improve performance. Only limited sifting is available, and hence all experiments were run without it. The same is true for the reachable states computation.

3.2 CMU : David Long

There are many BDD packages from Carnegie Mellon University. The one called CMU here was written by David Long [12]. It was integrated in the SIS logic synthesis system [22] and has been widely distributed stand-alone and through SIS for many years. It has an efficient memory management scheme and contains support for dynamic variable ordering via sifting and window permutations. We use the last publicly available version, released in November 1993.[2]

3.3 CUDD : UC Boulder

The most recent release of the CUDD package (Colorado University Decision Diagram), 1.1.1, became available in June 1996. Its primary distinguishing feature is extensive heuristics for dynamic variable ordering. In particular, support is provided for random variable exchange, sifting, group sifting, window permutations (all variable-order permutations among a group of adjacent variables in the BDD are tested), identification and linking of symmetric variables, simulated annealing, and genetic algorithm-based reordering. It has a very efficient memory management scheme, and finely tuned heuristics for automatically adjusting cache sizes and controlling dynamic variable ordering.

[2] The latest version is said to be significantly upgraded: performance ×2 compared to the 1993 version [13]. It was developed at AT&T and is not available publicly.

4 Experimental Environment

We performed these experiments because we rely heavily on BDD-based algorithms to synthesize and verify large industrial designs. As a result, the experimental focus here is on *computations in real applications* rather than specific operations of BDD packages. While some packages could be engineered to perform better for a given application, we prefer to run them "as-is" – in a real synthesis environment and on real examples to test their current performance.

4.1 Computing Environment

Experiments were carried out on three different machines, each named below and listed with its characteristics.

Alpha:	DEC Alpha 2100, 512MB RAM, running OSF/1 v3.0
Sparc20:	SS20, 128MB RAM, running SunOS 4.1.3
Linux:	DEC Pentium 120 MHz, 32MB DRAM, running Linux/GNU

For consistency and comparison across experiments, most of the results are reported solely for the Sparc20. The general trends noted were observed on all machines. Results on other machines are included occasionally for breadth or to demonstrate a significant difference for some experiments.

4.2 Implementation

The packages were all integrated and run within SIS. The SIS program provides a common interface between a network representing a circuit and the BDD package. Given a particular ordering, the BDDs for the network can built by calling functions in the interface. This ensures that *for a given network and a given ordering, precisely the same external calls to the BDD packages are made.* This is extremely important for package comparison as small differences in the starting point or order of processing can have a large effect on the results. Furthermore, with all packages implemented in the same environment, the time and memory usage statistics can be measured at precisely the same points with precisely the same overhead.

There are some inefficiencies in the SIS implementation that will result in performance degradation roughly equal in all packages. First, when building the BDD for a network node, the BDDs for all intermediate network nodes are saved. This implies memory overhead when building output BDDs, but also effectively implements a lossless cache of computed results on the intermediate nodes – which may slightly improve efficiency. Second, SIS makes several calls to basic BDD operations for each sum-of-products function at a node, rather than making a single call to a more complex function when possible.

For the CAL package, the implementation follows precisely that recommended by its authors, and hence is slightly different to take advantage of the specialized memory access and superscalarity (without this, a comparison is meaningless). As part of this implementation, the initial network is decomposed into AND and invert gates, and again the BDDs are saved at the intermediate nodes. The initial ordering is determined based on the network before decomposition to guarantee the same starting point for all packages.

4.3 Examples

Two sets of examples have been selected: standard benchmark circuits (ISCAS85), and large industrial examples generated from the high-level language Esterel [1] and from the POLIS hardware/software codesign system [7]. These two categories represent a mix of circuits: combinational and sequential, those notoriously difficult to verify, those with very large BDDs compared to the number of latches, those that are highly redundant, and those that are optimized.

The characteristics of the Esterel-based circuits are given in Table 1. The final column gives the number of iterations required to compute the reachable states reported in the previous column. Each of the Esterel circuits was derived from a real design specified in Esterel and synthesized using the Esterel compiler. In the case of the *p1* and *dash* examples, the synthesis was done with POLIS. *tcintnc* is a version of *tcint* without the counters. *p1opt* is *p1* optimized with the standard script in SIS. *dashs* is *dash* with trivially redundant logic removed via the *sweep* command in SIS.

Circuit	pi/po	latches	SOP lit	Rstates	iter
att	47/255	178	3934	226938	14
bm	42/168	112	2192	850609	20
noyau	41/98	179	3743	2524161	11
prosa	46/227	73	1621	7361	27
renault	23/179	66	1838	257	12
sequenceur	55/98	121	1784	?	?
snecma	23/5	70	1689	10241	28
tcint	19/20	96	1232	2826	26
tcintnc	19/20	90	1084	310	10
train	59/81	866	25651	?	?
p1	49/12	63	6886	17620993	123
p1opt	29/8	59	825	17620993	123
dash	49/17	443	33673	?	?
dashs	49/17	356	12133	?	?

Table 1. Characteristics of Esterel-based Circuits

The Esterel examples are control-oriented, and furthermore the Esterel compiler extracts only the control when generating the BLIF files. However, some of the above BLIF specifications were generated using the Esterel compiler in conjunction with a library of arithmetic elements in BLIF format.[3] In partic-

[3] Although some examples contain arithmetic parts, we do not focus on arithmetic circuits as these are known to be better candidates for other types of BDDs, e.g. [6].

ular, *train* contains a number of counters, and the POLIS-generated circuits contain many arithmetic subcircuits (in fact, they are mostly arithmetic as they represent the portion of the design destined to hardware).

4.4 Experiments

Three categories of experiments have been run: basic BDD building, reached state computation, and BDD building and reached state computation interleaved with logic optimization. The packages are compared for different initial variable orderings (but the same ordering for each package), different initial circuit configurations, and with and without dynamic variable ordering. Finally, experiments are interleaved with some optimization steps, demonstrating performance as circuit configuration changes.

Results are measured by CPU time in seconds as reported by *time*, memory usage in megabytes as reported by *ps*, and BDD node count. Care was taken to ensure that the BDD nodes counts were the same for each package, where they were expected to be the same.

5 Results

Due to limited space, quite clearly the results tables for all experiments under all packages and platforms cannot be included. Results are described below, with tables included to illustrate important trends and conclusions.

5.1 Initial Variable Ordering

In the first experiment, all packages were compared under two different initial variable orders: the first based on the input order in the BLIF file, and the second on a topological order generated by *order_ dfs* in SIS ([14]). The BLIF order almost always produced larger BDDs and took more time for all packages, even when dynamic variable ordering was invoked. This illustrates that care must be taken to select a good initial order. A notable exception is C499 on Sparc, where the number of nodes is 31195 for the dfs order, and 1761 for the blif order. On Alpha, dfs yields 45974 and blif 45962. Note that the dfs ordering algorithm produces different results on the Alpha and Sun machines. For this reason, care was taken to always provide the same initial ordering for each package and experiment on a given platform. After the blif experiments, all were performed with the initial ordering given by *order_ dfs*.

5.2 Building Output BDDs

Results for building output BDDs with *order_ dfs* and no DVO are reported on Sparc20 for all packages in Table 2 (ISCAS) and Table 3 (Esterel). *train310* is a version of the *train* example described in Section 5.5. m indicates that the process ran out of memory. Failure to produce a BDD should be taken as failure subject to a particular *platform, memory constraint, time constraint, and initial ordering*, rather than "failure of package X to produce BDDs for benchmark Y". The goal is to make relative comparisons and not produce absolute numbers.

The node columns give the number of BDD nodes in the intermediate result (**int:** with the BDDs for all intermediate nodes) and the final result (**final:** just the output BDDs) for the original circuit. The former gives an indication of the memory overhead in saving the intermediate BDDs, and will track to some extent the memory usage reported by *ps*.[4] The double columns labeled CMU2 and CUDD2 represent runs on the decomposed circuit, as is done in the CAL package, for comparison on the same initial circuit. The node counts are not given for the intermediate BDDs on the decomposed circuit, though they appear in Tables 4 and 5 in the CAL column. Node counts are the same for the final BDDs. The totals are given at the bottom of the tables, though for Table 3 they are obviously dominated by the large examples.

	CAL		CMU		CUDD		Nodes		CMU2		CUDD2	
Circuit	time	mem	time	mem	time	mem	int	final	time	mem	time	mem
C432	4.9	6.5	10.6	4.5	7.2	7.2	146384	31195	10.1	5.4	8.2	7.4
C499	3.8	5.8	4.5	1.9	3.4	2.2	43364	33214	6.4	3.6	5.4	4.6
C880	2.1	3.9	1.3	1.6	1.2	2.0	28316	7926	2.5	2.0	2.2	2.4
C1355	4.7	6.4	6.6	4.3	5.0	4.7	135914	33214	8.0	5.0	6.8	5.2
C1908	4.4	6.1	3.8	2.8	3.4	3.0	42929	12734	5.6	3.5	5.0	3.7
C2670		m		m		m				m		m
C3540		m		m		m				m		m
C5315	12.7	12.1	5.5	5.1	5.0	6.5	49095	21236	14.1	7.9	12.2	8.7
C7552		m		m		m				m		m
Total	32.6	40.8	32.3	20.2	25.2	25.6			46.7	27.4	39.8	32.0

Table 2. Output BDDs on Sparc20

For the ISCAS circuits that completed under this fixed variable ordering, the expense in time and memory is not very high. CAL incurs some overhead in both time and memory for these small examples on the original circuit. This is probably due to its allocation of memory in larger blocks at the outset. It is the fastest package on the decomposed circuits. CMU consistently requires more time but less memory than CUDD.

For the Esterel circuits, CAL requires less time on all the larger examples (for the original and decomposed versions), and more memory on all examples. Again CMU usually takes more time and less memory than CUDD on the original circuits, but note its particularly good memory performance for *p1* with respect to CUDD. This CMU memory advantage is lost on the larger decomposed networks. The relative memory increase for CMU with example size is continued on more intensive applications (e.g., reachable states, Section 5.4).

[4] *ps* also includes the hash tables, which are sized and resized differently for each package.

	CAL		CMU		CUDD		Nodes		CMU2		CUDD2	
Circuit	time	mem	time	mem	time	mem	int	final	time	mem	time	mem
att	25.1	19.9	29.1	14.7	21.4	15.1	417307	161531	38.0	17.7	28.4	17.2
bm	13.2	13.1	21.8	9.5	14.8	11.1	249317	74794	23.4	11.6	18.6	12.0
noyau	23.0	21.4	33.1	14.5	21.5	14.8	445180	171005	38.4	19.2	28.4	17.2
prosa	11.2	13.0	15.7	8.9	11.6	10.0	233894	153006	21.7	11.2	15.9	11.5
renault	5.0	6.3	2.3	2.6	2.0	2.9	18269	8017	5.5	3.8	5.1	4.1
seq		m		m		m				m		m
snecma	66.0	50.1	130.5	41.5	91.2	35.3	1793385	1270801	129.7	55.9	90.2	44.0
tcint	3.3	5.1	.7	1.3	.7	1.6	2613	1446	3.0	2.2	2.9	2.5
tcintnc	2.7	4.8	.5	1.2	.5	1.4	2622	1431	2.5	2.0	2.5	2.3
train310	164.9	91.9	255.1	96.4	160.6	82.3	4106721	1122116	291.9	117.8	175.1	87.1
p1	152.2	72.3	699.8	6.7	176.0	13.4	139025	70268	240.3	48.2	211.2	40.4
p1opt	3.1	4.4	1.0	1.0	.9	1.3	4726	2967	3.0	2.1	3.0	2.2
dash		m		m		m				m		m
dashs		m		m		m				m		m
Total	469.7	302.3	1189.6	198.3	501.2	189.2			797.4	291.7	581.3	240.5

Table 3. Output BDDs on Sparc20

The decomposed network is consistently more costly than the original (disregarding the outlying CPU time for CMU on *p1*). This is undoubtedly due to the storage of all intermediate node BDDs, though we have observed in general that an optimized network is a better starting point than a non-optimized one, even when the BDDs are released immediately.[5]

5.3 Dynamic Variable Ordering

Next, each example was run with dynamic variable ordering. The sifting algorithm reported in [21] is implemented in CMU and CUDD. CMU also supports 3-variable window permutations and CUDD many other reordering techniques; only sifting was employed in these experiments. The results are shown in Tables 4 and 5. The previous CAL results are repeated here for reference (without sifting). The number of nodes reported includes BDDs at the intermediate networknodes, since the package was forced to save these during the computation. Totals are given for the examples that complete for CMU and CUDD.

CMU and CUDD complete all ISCAS examples. CMU completes all but two Esterel examples, and CUDD all but one. CUDD reports much better run times than CMU, and roughly the same memory usage. CAL fails to complete six examples, but otherwise reports the best run times (excepting *p1*).

Sifting dramatically increases runtime, but is essential for completing the large examples. The heuristics for choosing how aggressively to sift must be

[5] This is not always true, just as starting from a good initial ordering and building BDDs with DVO does not necessarily produce a better result than starting with a bad ordering and using DVO. The general trends hold, but care must be taken to avoid particularly costly behavior on outlying examples.

	CAL			CMU			CUDD		
Circuit	time	mem	nodes	time	mem	nodes	time	mem	nodes
C432	4.9	6.5	192671	5.9	1.0	5171	3.8	.9	4674
C499	3.8	5.8	109741	218.1	2.4	46342	161.4	2.1	38643
C880	2.1	3.9	32506	33.5	1.5	16551	15.8	2.0	15623
C1355	4.7	6.4	140038	525.6	5.8	121334	400.9	4.8	121335
C1908	4.4	6.1	58009	53.0	2.5	20762	34.2	3.0	21155
C2670		m		49.7	2.9	16041	55.8	4.8	20903
C3540		m		396.4	9.2	148456	139.8	5.4	126317
C5315	12.7	12.1	71621	13.0	4.2	11622	14.6	4.4	10954
C7552		m		255.9	7.6	50950	83.5	7.8	47289
Total				1551.1	37.1		909.8	35.2	

Table 4. Output BDDs with sifting on Sparc20

	CAL			CMU			CUDD		
Circuit	time	mem	nodes	time	mem	nodes	time	mem	nodes
att	25.1	19.9	494131	1706.0	13.2	112367	76.4	5.9	55648
bm	13.2	13.1	340941	57.8	2.8	18761	78.9	4.7	41395
noyau	23.0	21.4	526481	443.4	6.2	61675	88.0	5.6	34567
prosa	11.2	13.0	355241	128.7	2.9	28032	13.3	2.8	9941
renault	5.0	6.3	28096	14.8	2.2	8764	8.6	2.8	3649
seq		m		t(1hr)			2840.1	12.4	293187
snecma	66.0	50.1	2241334	1528.3	12.5	183705	t(1hr)		
tcint	3.3	5.1	2815	.7	1.3	2613	.7	1.6	2613
tcintnc	2.7	4.8	2770	.6	1.2	2622	.6	1.4	2622
train310	164.9	91.9	4257859	t(1hr)			920.2	11.9	180225
p1	152.2	72.3	1614809	9.7	1.5	4805	11.0	2.2	2196
p1opt	3.1	4.4	15134	1.1	1.0	4726	2.5	1.2	3467
dash		m		254.0	10.6	34338	116.1	14.6	12531
dashs		m		69.5	4.7	32418	76.7	8.6	10089
Total				2686.3	47.6		472.8	51.4	

Table 5. Output BDDs with dfs and sifting on Sparc20

well-engineered. In these tables and in the remainder of the paper, we note the most consistent behavior from the CUDD package, due to a judicious choice in trading off such parameters. Nonetheless, CUDD with sifting fails on the *snecma* example with a timeout after one CPU hour.

As another illustration of the unpredictability of BDD performance, consider the data in Table 6, containing sifting results on the Alpha and Linux platforms. For *dashs*, CUDD is slower than CMU on Sparc and Linux machines, but faster on the Alpha. This is undoubtedly due in large part to a difference in variable orderings computed by *order_dfs* across different platforms which will subsequently trigger sifting differently. Given that the two packages begin with the same ordering nonetheless, the difference in performance is notable.

	Alpha				Linux			
	CMU		CUDD		CMU		CUDD	
Circuit	time	mem	time	mem	time	mem	time	mem
p1	5.7	6.8	6.4	8.3	5.7	3.4	5.8	4.1
p1opt	.6	5.8	1.7	6.5	2.3	2.9	2.4	3.0
dash	180.1	24.3	66.9	29.4	172.7	11.6	50.6	15.6
dashs	47.7	12.4	38.7	20.4	22.9	6.7	39.3	10.4

Table 6. DFS/Sifting Results

The results in Tables 4 and 5 indicate that DVO is more useful for completing large examples than BFS methodology, but that the BFS algorithm is faster when it does complete. This confirms the findings in [15], where it was conjectured that the performance advantages in the CAL package are due more to the lossless computed table and superscalarity than due to the data locality implied by the BFS approach. For a more in-depth study of the memory effects of these three packages, the reader is referred to [20] and [15].

5.4 Application: Reachable States Computation

For the reachable states computation, the BDDs are computed for the next-state functions and the reached state set. The latter has proven to be difficult and to cause memory explosions. This experiment thus reflects a new dimension in stressing the packages.

The CMU and CUDD experiments were done using the reachable states package in SIS, with sifting enabled.

Also included in this table are the runtimes for computing the reachable states in the TiGeR package with sifting enabled. This package is described in Section 6; limited results are included here only to illustrate its efficient reached-states algorithm. The algorithm builds the BDDs for the next state functions *at each iteration* of the reached-states computation, while simplifying each with respect to the current reached state set. Thus, the complete BDDs for the next state variables are never built. For many of our large examples, this is the only practical method for generating the reached state set.

Results on computing the reachable states are given in Table 7 (run on Alpha). TiGeR completes all examples, though not always in the least amount of time.[6] Nonetheless it is effective in consistently determining the reached state set. This algorithm is separate from core BDD technology, but can and should be implemented on top of any BDD package used in an FSM-type verification system. CMU fails on four examples and CUDD on three in one CPU hour. Where they do complete, no real conclusions can be drawn about their relative efficiency, unlike the previous experiments.

			CMU		CUDD		TiGeR
Circuit	Rstates	iter	time	mem	time	mem	time
att	226938	14	t(1hr)		t(1hr)		1398.7
bm	850609	20	628.5	17.1	1115.6	24.0	1883.2
noyau	2524161	11	t(1hr)		t(1hr)		680.3
prosa	7361	27	528.7	16.8	209.5	17.6	195.9
renault	257	12	21.1	10.6	17.1	1.5	6.6
snecma	10241	28	t(1hr)		t(1hr)		59.2
tcint	2826	26	38.8	10.9	173.6	11.1	140.6
tcintnc	310	10	7.9	9.3	1495.6	10.2	19.5
train310	25	2	t(1hr)		966.2	34.8	9.8
p1	17620993	123	108.5	9.8	105.7	13.1	102.4
p1opt	17620993	123	72.1	8.5	36.9	9.8	80.1

Table 7. Reachable States Computation on Alpha

Note that although the CUDD package usually outperformed the CMU in previous experiments, the results are quite different here.

5.5 An Example

The *train* example is our biggest and most difficult example. In order to complete synthesis and verification, we must alternate traditional logic optimization techniques (e.g. SIS) and optimization using reached state sets.

In one phase of the design process for this example, a bug was introduced. As a result, the underlying state graph went from millions of reachable states to just two. This was not an easy change to detect: the buggy design has millions of *equivalent* reachable states. The BDD packages applied to the buggy design ran for hours and produced no results. Only after simple logic optimization brought the design from 866 latches to 310, could TiGeR determine that there were 25 reachables states (in that version of the design). At this point, it was clear there was a bug, but we continued optimization to reduce the circuit as

[6] We found that TiGeR often performed better with sifting disabled; these results are not reported here.

much as possible and test the BDD packages. We used the 25 reachable states computed by TiGeR to compute and further remove redundant latches resulting in a 3-latch design. One final optimization step in SIS brought the design down to its final 1-latch, 2-state version.

The BDD packages were tested on several versions of this circuit. Recall that it is a large complex controller with counters. *train* is the original, correct design. *trains* is the original design after SIS *sweep*, and *traino* is the original after the SIS standard optimization script.

trainb is the original buggy design. *train_310* is *trainb* after *sweep*, with 310 latches. *train_254* is *trainb* after the SIS standard script, with 254 latches. *train_31* is a 31-latch version produced from *train_310* by TiGeR. (TiGeR has the capability of removing latches based on the reached state set.) *train_3* is a 3-latch version produced from *train_31* by our latest latch removal algorithms which are based on the reachable state set.

The results are given in Table 8 for BDD building and reachable states computation with sifting enabled. The examples are listed in increasing estimated difficultly. The experiments were run in this order, stopping when a package could not complete an example, as this uniformly led to failure as well in further testing. The t indicates a timeout after one CPU hour.

	CAL		CMU				CUDD				TiGeR	
	BDDs		BDDs		States		BDDs		States		BDDs	States
Circuit	time	mem	time	mem	time	mem	time	mem	time	mem	time	time
train3	7.8	6.5	3.0	1.5	2.4	1.7	6.9	2.2	3.5	2.5	1.6	1.5
train31	11.6	10.8	36.5	3.9	59.8	5.3	7.0	4.4	7.7	5.2	2.6	3.5
train254	147.4	91.3	t		t		363.9	6.8	2439.1	10.5	1267.1	15.1
train310	175.2	92.0					908.9	12.0	t		1687.0	12.3
trainb		m					2349.7	26.8			t	m
traino							3415.1	13.3			t	t
trains							3008.8	24.1				
train							2931.2	26.5				

Table 8. Train Example on Sparc20

The trends previously observed are repeated in this example: CAL is by far fastest for the examples in which it completes (only BDD building could be run); CMU requires more time but less memory than CMU for the examples in which it completes (albeit only the two smallest); CUDD returns a consistent performance in completing the BDDs for all circuits and returning better run times than TiGeR for BDD building; the TiGeR algorithm returns the best runtimes and the most circuits completed for the reachable states computation.[7]

[7] The BDDs are in TiGeR are usually built after the reachable states computation, and this is much more efficient than direct BDD-building as reported in the table.

6 Conclusions

Some interesting features of BDD packages have been demonstrated, and signal areas for further cooperative development. In particular,

- The CAL package is fast, perhaps due to its lossless computed table (note the high memory usage) and superscalarity features. These features should be further explored for integration in other packages.
- The CUDD package exhibits an excellent tradeoff in memory usage and CPU time consumed. It has well-tuned heuristics for controlling memory allocation and sifting.
- The use of simplified BDDs during the reachable states computation, as that employed in TiGeR, is invaluable and should be incorporated in public domain verification systems.

6.1 Other BDD Packages

A number of additional packages were investigated and may be included in subsequent studies, or otherwise of interest to the reader:

- Ken McMillan's BDD package is part of the SMV verification system, and is rumored to perform significantly better than the Long package.
- A BDD package from the Technical University of Eindhoven is used in the PVS verification system. The code is available but not yet ready for integration in SIS.
- Nils Klarlund and Theis Rauhe have a BDD package that is designed to reduce cache misses. The code has very recently become available and now is ready to be compared to other packages.
- The AMORE [10] package is currently under development at Albert-Ludwigs-University, Germany with plans for public availability soon. It uses a novel idea for creating BDDs: temporary auxiliary variables are introduced at the top of the BDD, sifted to the bottom, and quantified out. The authors report improved efficiency with respect to the traditional algorithms, but the package needs to be compared in a common environment.
- The TiGeR package [8] used for part of this study is sold through DEC. As such, it is publicly available but not free. It was included here solely to illustrate the efficacy of the reachability algorithm, which can be implemented on top of other packages.

7 Acknowledgements

Extensive discussions with Fabio Somenzi and Rajeev Ranjan are gratefully acknowledged. Our industrial partners are acknowledged for providing examples: AT&T, DEC, Renault, Snecma, and especially Yves Auffray of Dassault and Reinhard von Hanxleden of Daimler-Benz. This work was supported in part by NSF grant INT-9505943, by NSF/NATO grant GER-9552795, and by the French GENIE MESR INRIA project.

References

1. G. Berry and G. Gonthier. The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation. *Science of Comp. Prog.*, 19(2):87–152, 1992.
2. K. Brace, R. Rudell, and R. Bryant. Efficient Implementation of a BDD Package. In 27th *DAC*, pages 40–45, June 1990.
3. R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
4. R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
5. R.E. Bryant. Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification. In *ICCAD*, pages 236–243, November 1995.
6. R.E. Bryant and Y.-A. Chen. Verification of Arithmetic Circuits with Binary Moment Diagrams. In *Proceedings of the* 32nd *Design Automation Conference*, pages 535–541, June 1995.
7. M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A Formal Specification Model for Hardware/Software Codesign, June 1993. UC Berkeley Technical REport UCB/ERL M93/48.
8. O. Coudert, J.-C. Madre, and H. Touati, December 1993. TiGeR Version 1.0 User Guide, Digital Paris Research Lab.
9. M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *ICCAD*, pages 2–5, November 1988.
10. A. Hett, R. Drechsler, and B. Becker. MORE: Alternative Implementation of BDD-Packages by Multi-Operand Synthesis. In *Proc of EuroDAC*, 1996. To appear.
11. N. Klarlund and T. Rauhe. BDD algorithms and cache misses, January 1996. Unpublished manuscript.
12. David Long, November 1993. Personal communication.
13. David Long, April 1996. Personal communication.
14. S. Malik, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *ICCAD*, pages 6–9, November 1988.
15. S. Manne, D. C. Grunwald, and F. Somenzi. Remembrance of Things Past: Locality and Memory in BDDs, April 1996. Unpublished manuscript.
16. P.C. McGeer, K.L. McMillan, A. Saldanha, A.L. Sangiovanni-Vincentelli, and P. Scaglia. Fast Discrete Function Evaluation using Decision Diagrams. In *ICCAD*, pages 402–407, November 1995.
17. M.R. Mercer, R. Kapur, and D.E. Ross. Functional Approaches to Generating Orderings for Efficient Symbolic Rep. In 29th *DAC*, pages 624–627, June 1992.
18. H. Ochi, N. Ishiura, and S. Yajima. Breadth-First Manipulation of SBDD of Boolean Functions for Vector Processing. In 28th *DAC*, pages 413–416, June 1991.
19. S. Panda, F. Somenzi, and B.F. Plessier. Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams. In *ICCAD*, pages 628–631, November 1994.
20. R.K. Ranjan, J.V. Sanghavi, R.K. Brayton, and A. Sangiovanni-Vincentelli. High Performance BDD Package Based on Exploiting Memory Hierarchy, June 1996.
21. R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. In *ICCAD*, pages 42–47, November 1993.
22. E.M. Sentovich, K.J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P.R. Stephan, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis, May 1992. UC Berkeley Technical Report UCB/ERL M92/41.