Remembrance of Things Past: Locality and Memory in BDDs [*][†]

Srilatha Manne[§] Dirk Grunwald[‡] Fabio Somenzi[§]

[§] University of Colorado
Dept. of Electrical and Computer Engineering
Boulder, CO 80309

[‡] University of Colorado
Department of Computer Science
Boulder, CO 80309

Abstract

Binary Decision Diagrams (BDDs) are efficient at manipulating large sets in a compact manner. BDDs, however, are inefficient at utilizing the memory hierarchy of the computer. Recent work addresses this problem by manipulating the BDDs in breath-first manner (BFS). BFS processing is quite successful at reducing the number of page faults when the BDDs do not fit in the available physical memory. When paging does not take place, it is much less clear which paradigm leads to the better performance. In this paper, we perform a detailed analysis of BFS and DFS packages using simulation and direct performance monitoring of the memory hierarchy. We show that there is very little difference in TLB and cache miss rates for DFS and BFS paradigms. We also show that differences in execution time between carefully tuned BFS and DFS implementations are primarily a function of the lossless computed table used in BFS implementations, and not a function of memory locality. Furthermore, we present implementation changes to the the CUDD package that can improve execution times by as much as 26% when the problem fits in main memory, and a factor of six when paging is involved.

1 Introduction

Binary Decision Diagrams (BDDs) [4] have found many uses in CAD applications from logic optimization to combinational and sequential verification. BDDs provide an efficient way to manipulate large boolean functions through recombination of shared subexpressions. Although BDDs are a compact memory representation, their access patterns are inefficient. BDDs are graphs, where the next element to be accessed is referenced by a pointer. Most architectures on the other hand, have a memory hierarchy designed under the assumption of linear temporal and spatial locality in memory references [6]. Since BDD packages spend considerable time executing loads and stores, improving this aspect can significantly improve performance.

Improving the efficiency of memory access can take place at many levels of the hardware memory hierarchy: cache, translation look-aside buffer (TLB), or main memory. Ochi [8], Ashar [2], and Sanghavi [10] have addressed the problem of page locality by implementing BDD packages using Breadth First Search (BFS). With a BFS implementation, the nodes associated with each variable are accessed all together. Therefore, by using one or more pages of memory for each variable, page faults are reduced. The usual depth first search (DFS) implementation accesses nodes across all pages in a fairly random fashion. Therefore, there is no locality of access at the page level and the page faults are usually many more if indeed the application exceeds main memory.

The performance of any BDD package is dependent on three primary factors: Algorithm, implementation, and application. In the context of this paper, *algorithm* refers to DFS vs. BFS; *implementation* refers to how the code is tuned to best use the memory hierarchy; *application* refers to the context in which the BDD package is used such as computing output BDDs or performing reachability analysis.

In this paper, we present an in-depth analysis of the performance of the memory hierarchy for both BFS and DFS implementations. In particular, we:

- Present runtime results with and without paging.
- Report results obtained by running extensive hardware and software analysis programs.
- Analyze the performance of each package in terms of algorithm, implementation, and application.
- Show that the TLB miss rate for the CUDD package [12] is similar to the CAL package and significantly better then that of the CMU package [7].
- Show that the page-level miss ratio of the CAL package is marginally better than that of the CUDD package, and that both are significantly better than that of the CMU package.
- Demonstrate that the majority of the benefits of the CAL package is not due to data locality, but is primarily a function of its lossless computed table.
- Demonstrate that reordering is suboptimal in the CAL package due to *algorithmic* and *implementation* difficulties.
- Show that further improvements in memory locality in DFS packages can provide up to a 26% and 6 X improvement with and without paging, respectively.

Section 2 reviews the memory design of the DEC Alpha. Section 3 qualitatively compares the BFS and DFS algorithms. We present results in Section 4 and improvements to DFS algorithms in Section 5.

[*]This work is supported in part by NSF grant MIP-94-22268 and SRC contract 95-DJ-560, and in part by NSF grant No. CCR-9404669, ARPA contract ARMY DABT63-94-C-0029 and a software and hardware grant from Digital Equipment Corporation.

2 Memory Hierarchy

Modern computers typically use a *memory hierarchy* [11] to balance the desire for a large memory with the cost of that memory. In this paper, we used an AlphaServer 2100-4/275 with a DECchip 21064A processor to compare the performance of the BDD packages. This system had a small 8 KByte first-level data cache with 32-byte cache lines, a 4 MByte second-level cache and a 512 MByte main memory.

The *translation look-aside buffer* (TLB) is one other important part of the memory hierarchy often ignored when designing data structures. The TLB provides a cache for the data structure that maps a processes virtual address space to the system physical address space. In most systems, the TLB is accessed in parallel with the cache, and a memory reference must hit in both the cache and the TLB, because the TLB is also used to control access to memory for virtual memory paging. The TLB is itself a cache and suffers from misses called TLB misses. On the 2100-4/275 we used, the TLB has 32 data entries with a miss penalty of $\approx 45 - 50$ cycles. Hardware TLBs can map from 32-128 entries, with sizes of 32-64 being typical. A hardware TLB is usually designed as a fully associative cache, and increasing its size beyond 64 provides limited benefits to most programs and may compromise the processor cycle time.

3 Decision Diagrams

BDDs naturally lend themselves to depth-first manipulation. Let $\circ$ be any binary boolean operator. Then, from Boole's expansion theorem, $f \circ g = x(f_x \circ g_x) + x'(f_{x'} \circ g_{x'})$, where h_v denotes the cofactor of h with respect to v. Choosing x as the variable appearing in f or g that is first in the variable order leads to a simple recursive algorithm for $\circ$.

A naive implementation of the recursive algorithm just outlined takes time exponential in the number of variables; to avoid that, a *computed table* is used to store the results of recent computations. The computed table is usually implemented as a direct-mapped or two-way set-associative cache. The mapping is performed by a hashing function. Of critical importance is the size of the computed table. A large computed table decreases the chance of useful data being overwritten. But a large table also increases memory usage and the overhead for operations such as garbage collection. Therefore, the size of the computed table must be carefully balanced against its associated cost.

BFS manipulation of BDDs proceeds in two phases: *Apply* and *Reduce*. In the former, the operands are traversed from top to bottom creating as many request lists as there are variables. In the latter, the request lists are scanned starting from the bottom, thereby generating the result. Redundant nodes are created during the *Apply* phase and removed during the *Reduce* phase. The number of redundant nodes is usually not very large.

A BFS BDD package uses a specialized memory manager that assigns sets of memory pages to different variables. This, combined with a breadth-first processing of the BDDs, is intended to reduce the number of page faults through an increased locality of accesses. Unlike DFS, BFS processing does not specifically require a computed table. The request lists created during the *Apply* phase serve as a lossless computed table, which may consume a large amount of memory, but guarantees that an operation will not be repeated during a particular execution of the *Apply* and *Reduce* phases. However, there is no sharing of results between different invocations of *Apply* and *Reduce*. *Superscalarity* and *pipelining* provide for inter-operation sharing of computed results, albeit at a cost of memory overhead. In Section 4 we address the question of whether the DFS algorithm with a large computed table will perform as well as the BFS algorithm.

From the point of view of reordering, there are two major difficulties for the BFS implementation. On the one hand, swapping two adjacent variables destroys the separation between the memory pages dedicated to different variables. Such separation must be restored. The second difficulty is that the index of a node may change as a result of swapping. The CAL package stores the index of the node within the parent nodes while DFS implementations store the index in the node itself. Therefore, the CAL package requires that one must update the index of the node in its parent nodes after the node is swapped. This makes reordering of variables a two step operation.

4 Quantitative Comparison of DFS and BFS

We used the ATOM [13] software instrumentation system to instrument the BDD packages and measure the cache and virtual memory characteristics of those packages. We constructed a simulator for a variety of first-level caches, ranging from 8 KByte to 256 KByte direct-mapped caches. We also simulated the paging behavior of the different BDD packages based on an 8 KByte page size. This measurement records the miss rate when a certain number of 8 KByte memory pages are available and an LRU replacement strategy is used. We can use this measurement to compare the influence of TLB misses on the Alpha, because the TLB miss rate will be the page miss rate for 32 entries.

Although the software instrumentation produces accurate results for the references in the program, it does not model the operating system or effects from multiprogrammed systems. While these effects can be significant [1], the memory simulations provide standard metrics that can be used to compare the different algorithms. We used the performance monitoring hardware (accessed via the IPROBE program) of the DECchip 21064A to gain a better understanding of the performance implications for TLB misses. We measured the amount of time spent in *PALcode*, a special processor state used to implement a variety of primitives for the operating system, to measure the percent of the total execution time spent handling TLB misses. The DECchip 21064A has a software-managed TLB, and most of the time spent in PALcode is servicing TLB misses. The software cache and TLB measurements are precise and repeatable between runs. The hardware performance monitors are stochastic, and we report the mean value of 1 to 50 runs of the program, depending on the size of the job.

We measured three BDD packages: CMU, CAL, and CUDD. To enable reasonable comparisons to the results in [10], the experimental setup and routines reported in [10] were reproduced for the CUDD package. We kept all experimental parameters the same between the various packages

NAME	Time	Mem(MB)	Cache Sizes		
			8K	64K	256K
c1908	1.52	15	8.2	3.9	2.5
c1355	1.50	17	8.4	4.4	2.8
c499	1.11	14	8.2	4.1	2.6
s1423	2.34	23	8.2	4.0	2.8
c880	16.57	61	8.8	6.7	5.6
s4863	13.83	54	8.3	5.1	3.8
s6669	13.08	81	8.3	4.7	3.4
c3540	53.24	126	7.8	5.4	4.5

Table 1: Execution time, memory usage in MBytes, and percent cache miss rates for CAL package.

regardless of the implications this has on the performance of CUDD . For example, CUDD supports 32-bit pointers, but we chose to use 64-bit pointers since this is what CAL and CMU supported. Our goal was to minimize the variables in the experimental setup so that we could perform a fair comparison. The analysis was run mostly on a variety of ISCAS and MCNC benchmark circuits. Since the instrumentation routines involve a high overhead in run times, we were only able to work with small circuits. As we show, the data are well behaved, and the analysis holds for larger circuits, also.

The experimental data in this paper are presented in two forms: tabular and graphical. The bar graphs show the minimum, maximum, and average of the ratios of CMU or CUDD with respect to CAL for the examples covered in this section. Since we report the averages of the ratios, every example has the same weight, and small test cases have as much influence on the average as large ones.

4.1 Measured Performance

Table 1 shows the execution time and memory used by the CAL package on the Alpha. Execution time is in seconds, and memory is in MBytes. All examples shown fit into main memory, and did not require any paging. Figure 1(a) shows the relative performance of CUDD and CMU packages with respect to CAL. It is clear that CAL has the best execution times, CMU has the smallest memory usage, and CUDD is in the middle for both metrics, and that the *implementation* of the DFS package effects its performance. The CMU package was completed in 1993, and memory performance was not a priority when developing the package. The CUDD package is newer and some effort has gone into optimizing it for performance.

The data above also bring to question the relationship between memory usage and execution speed. Specifically, the trade-off between the size of the computed table and the execution time. We generated the output BDDs for C3540 with varying computed table sizes to determine the effect of increased memory usage on execution time. The results are shown in Figure 1(b), along with the data points for the CAL and CMU runs. The graph shows that the CUDD outperforms both CAL and CMU for comparable memory usage. For large examples such as C3540, S6669, C880, and S4863, CUDD is either better or within 3% of the execution times produced by CAL. These results suggest that the lossless computed table in CAL is responsible for the performance benefits of the CAL package.

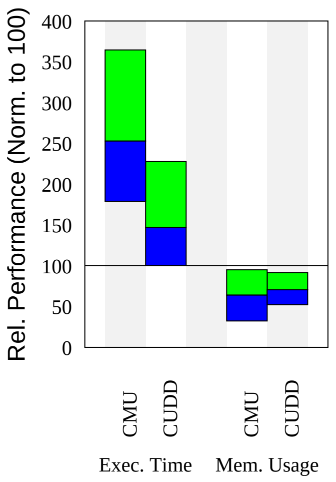
Circuit	CMU		CUDD		CAL	
	A	I	A	I	A	I
c1908	3.5	19.0	2.0	11.9	1.1	10.8
c1355	2.8	15.5	1.6	9.1	0.8	8.8
c499	3.0	17.6	2.1	10.5	0.5	6.9
s1423	3.1	15.0	1.7	9.2	0.6	8.2
c880	4.5	17.5	2.6	13.5	4.4	20.7
s4863	4.4	19.3	2.8	14.4	3.0	18.0
s6669	3.9	18.2	2.4	13.4	2.9	14.3
c3540	—	20.7	—	16.3	—	23.5

Table 2: Simulated percent TLB miss rate (A) and measured TLB cost (I) for 32 entry TLB.

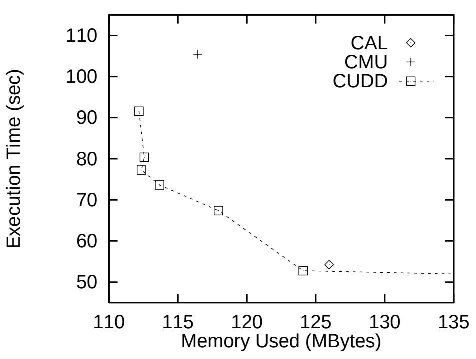
4.2 The Effect of Memory Reference

Table 1 also shows the simulated cache miss rate for CAL for a variety of sizes. There are many possible cache configurations, and our simulations attempt to give an indication of the performance for a range of cache sizes. The Alpha in particular has a 8KByte, direct mapped L1 cache. Normally, a cache miss rate of 5–10% is considered acceptable for a small, direct-mapped cache, and most of the miss rates shown are within that range. Figure 1(c) shows the relative performance of CMU and CUDD packages with respect to CAL. The CMU package has a consistently higher miss rate than either CAL or CUDD. CUDD has a slightly higher miss rate than CAL for the 8KByte cache, and similar miss rates for larges caches. Again, this is an issue of *implementation*, where the CMU package might not have been optimized for the cache. For example, individual BDD nodes which span multiple cache lines contribute to a higher miss rate. To summarize, although different *implementations* of the DFS package have varying cache miss rates, both DFS and BFS *algorithms* have similar and reasonable L1 cache miss rates.

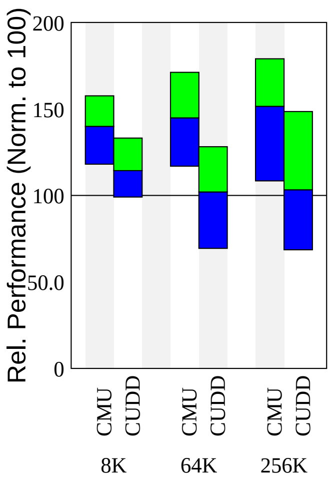
Table 2 describes the simulated and measured effects of TLB misses for a 32-entry TLB, which is the size of the data TLB on the Alpha 21064. The columns labeled **A** refer to the miss rate as simulated by ATOM. The columns labeled **I** refer to IPROBE measurements of the percent of execution time spent handling TLB misses. We were unable to complete the simulation experiments for c3540 due to a lack of time. The data from the hardware performance monitors and memory simulation disagree slightly. This is expected, because the hardware performance monitor captures the interactions between processes and the operating system whereas the simulations do not. For the smaller examples (c1908, c1355, c499, s1423), CAL is better than either DFS package both in terms of simulated miss rate and percent execution time. For the larger examples, though, CUDD outperforms both CAL and CMU. To gain a better understanding of the situation, we simulated the page miss ratio as a function of the number of pages. The data for C880 are presented in Figure 2. Figure 2 can be used to determine both TLB and main memory miss rate. For example, the miss rate for 32 pages corresponds to the TLB miss rate for the Alpha. The miss rate for 2000 pages would indicate the *paging rate* for a machine with 16 MBytes of available physical memory. The



(a) Average ratio of runtimes and memory usage.



(b) Run time as a function of memory usage for c3540.



(c) Average ratio of cache miss rates.

Figure 1: Measured and simulated results.

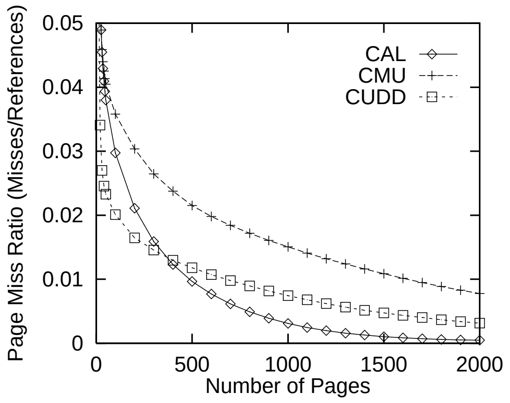


Figure 2: Miss ratios as a function of number of pages.

figure clearly shows the advantage of the CAL package when dealing with main memory miss rates, but CUDD has a much better miss rate than CAL for a 32 entry TLB.

Figure 3 shows further data about the three packages. The first set of bars shows that CAL and CUDD execute similar numbers of instructions. However, the next two sets show that CAL performs more memory references, while CUDD has more function calls. While it is clear that CUDD has more function calls because many of its functions are implemented recursively, the reasons why CAL accesses memory more often are still being investigated. These data suggest that iterative implementation of key recursive functions in CUDD may narrow the performance gap even when CUDD uses a small computed table. The final set of bars in Figure 3 shows the ratio of percent execution time spent handling TLB misses. Although CMU doesn't perform as well as CAL, CUDD is, on the average, slightly better than CAL.

So far, we have shown that CUDD is only slightly worse CAL when it comes to TLB miss rate, but that CAL is superior when it comes to handling page misses. Given that a page miss may incur a large penalty in the range of 2,800,000

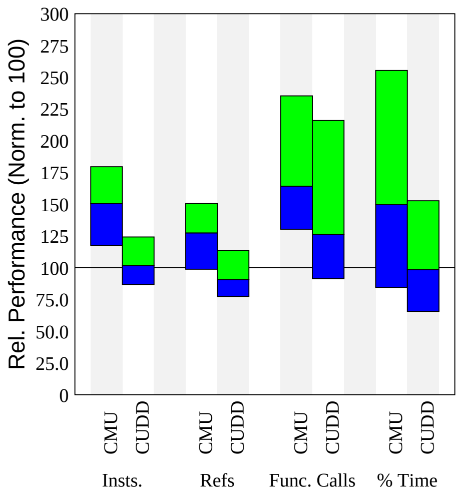


Figure 3: Performance metrics.

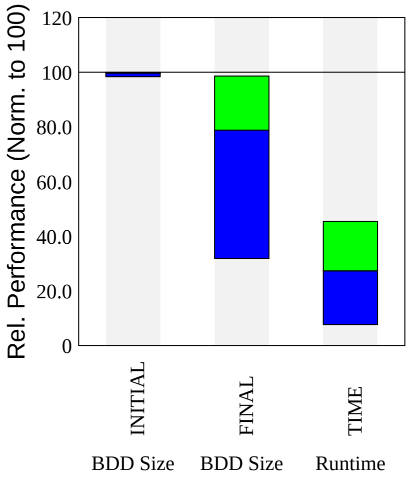


Figure 4: Average ratio of reordering parameters.

	CPU Time		Elap. Time		Page Misses	
NAME	Cu	CAL	Cu	CAL	Cu	CAL
c3540	—	164	>10 h	2281	—	114.1
c880	62	49	6120	381	331	19.9
s4863	38	44	85	60	2.5	0.9
s6669	48	43	304	72	3.7	1.4

Table 3: Exec. time (in sec) and page faults ($\times 10^3$) for CAL and CUDD package when process exceeds main memory.

cycles, a page miss rate of even 0.003% means an average delay per memory access of 84 cycles due to paging. To determine the real cost of paging, we ran the largest examples on a DECstation 3000-300LX using a DECchip 21064, which only had 64 MBytes of memory. Table 3 shows the elapsed and CPU time in seconds along with thousands of page faults for each example. The better memory locality of CAL can exceed a factor of 20 reduction in run time with respect to CUDD when paging occurs.

4.3 Dynamic Reordering

Dynamic reordering [5, 9] automatically reorders the BDD to reduce its size. Although the process of reordering is expensive, it has proven to be invaluable for many applications.

From the point of view of reordering, there are two major difficulties for the BFS implementation. On the one hand, swapping two adjacent variables may destroy the separation between the memory pages dedicated to different variables. Such separation must be either preserved at some memory and CPU time cost or restored. The second difficulty is that the index of a node may change as a result of swapping. The CAL package stores the index of the node within the parent nodes while DFS implementations store the index in the node itself. Therefore, the CAL package requires that one must update the index of the node in its parent nodes after the node is swapped. This makes reordering of variables a two step operation.

We computed the output BDDs for the previous test cases and then ran dynamic reordering to reduce the size of the resulting BDDs. Figure 4 shows the relative performance of CUDD with respect to CAL. CUDD outperforms CAL in both BDD size reduction, and execution time. Some of the performance difference is due to the *algorithmic* complexity of implementing reordering in the CAL package. But a sizeable portion of the performance for CUDD comes from a good *implementation* of reordering heuristics. The difference in performance is significant, but a better *implementation* of reordering in CAL will reduce the performance gap. (We could not incorporate a newer release of CAL that has faster reordering algorithms in our experimental setup in time for inclusion in the Proceedings.)

5 Improvements to DFS

The analysis and results presented in the last section show that there is room for improvement in the DFS implementation of the BDD package in the area of memory management. As previous results show, TLB miss rate in the CUDD package is comparable to that of the CAL package for small TLB sizes. But as the size of the TLB increases or when the program pages out, the CAL package outperforms both DFS implementations. Our next goal was to reduce the amount of TLB and page misses in the CUDD package with small *implementation* changes.

	No Sorting		With Sorting	
NAME	Time	% TLB	Time	% TLB
elevator	16.5	13.6	16.2	12.1
clma	165.8	17.6	130.7	7.0
sbc	158.4	18.5	157.2	15.6
key	1833.8	21.0	1851.9	15.5
simple	8144.0	27.3	6084.2	4.8
rcnum	24414.4	27.4	22269.2	25.6

Table 4: VIS execution time (in sec) and percent time spent handling TLB misses with and without sorting.

The CUDD package contains its own garbage collection mechanism for allocating and freeing BDD nodes. When a node is freed, which only happens during garbage collection, it is placed on top of a *free list*. If a new node is needed, the package initially tries to procure the node from the free list. If the free list is empty, a new block of BDD nodes is allocated from memory and attached to the free list. In the implementation of the CUDD package used for the experiments, nodes were inserted in the free list in the order in which they were collected, which is essentially random. Therefore, the next time a BDD structure was created, the nodes used in the structure were also in a random order.

We modified the garbage collection procedure to sort the free list by pages. Complete sorting of the free list is expensive, and may not be necessary, since cache locality was shown to be in an acceptable range. Table 4 shows the results for running reachability analysis and model checking in VIS [3] with a variety of examples. All examples fit in main memory. Some of the results don't show much improvement in execution time, but others show an improvement of over 26%. This can be explained as follows: One incurs a cost for sorting the free list in the hope that further, repeated use of the sorted BDD reduces the overall cost of the operation. If the sorted BDD is not accessed frequently, then the benefit is minimal.

To verify our hypothesis that sorting of the free list benefits memory locality, we repeated the simulation and hardware measurements on the VIS [3] package. The simulation experiments are computationally expensive, so we were only able to complete a small example. Figure 5 shows the simulation results for miss rate as a function of number of pages. Even for a small example, it is obvious that the page miss rate visibly drops when the free list is sorted. Table 4 also shows IPROBE measurements of the percent of total execution time spent handling TLB misses. For the two test cases CLMA and SIMPLE_BDLC, the effect of TLB misses on execution time is 2 to 6 times larger for the implementation without sorting. For examples such as KEY and SBC, the difference is just enough to compensate for the cost of sorting.

Finally, one last test is to see the effect of sorting on programs which exceed main memory size. We again used a DECStation 3000-300LX 64 MBytes of main memory. Due

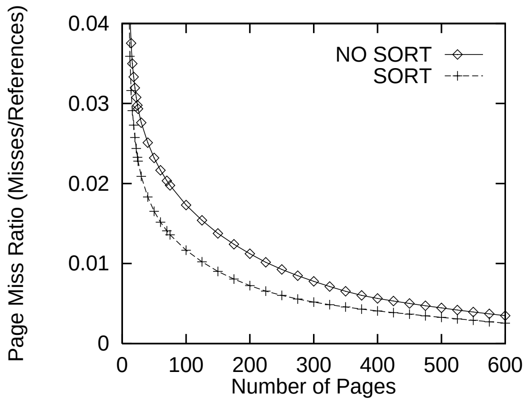


Figure 5: Page miss rate vs. number of pages for ELEVATOR.

	CPU Time		Elap. Time		Page Faults	
NAME	NS	S	NS	S	NS	S
bigkey	365	317	1705	573	41	9.8
clma	651	516	1997	1710	61	59
gcd16	315	317	1369	995	33	22
mult32b	>447	378	>12 h	7108	>2797	377

Table 5: Exec time (in sec) and pages faults ($\times 10^3$) for VIS when program exceeds main memory.

to time constraints, we were only able to run small test cases. Table 5 shows the results for 4 examples. The most dramatic example is MULT32B which shows at least a sixfold difference in run times and a sevenfold difference in page faults between executions with and without sorting of the free list.

To summarize, we have shown that a different *implementation* of the DFS algorithm can produce improvements of over 26% when the problem fits in main memory, and up to a factor of six when the program is memory limited. Again, the *implementation* is closely related to the *application* in which the program is run, since only *applications* which allocate and release large numbers of BDD nodes and which reuse existing structures show a significant benefit from the sorting of the free list. Fortunately, construction and destruction of BDD structures is common practice in most *applications*, and is even more prevalent when dynamic reordering is used.

6 Conclusions

In this paper we have applied detailed quantitative analysis to determine the factors that influence the performance of different BDD paradigms. This is the first step in understanding and improving the performance of the packages. When possible, we have differentiated the benefits stemming from *algorithms*, *implementations*, and *applications*. We have shown that carefully tuned implementations of DFS and BFS algorithms have similar cache and TLB statistics. The performance differences that we have observed can be ascribed to memory/CPU time trade-offs and possibly to different instruction mixes. The DFS approach is preferable when variable reordering is a major concern; the BFS approach is preferable when the size of both programs exceeds main memory by a limited factor (beyond which not even BFS processing helps). However, it may be the case that a DFS package may avoid paging altogether by using less memory.

Based on our observations and analysis, we have shown that changes in the *implementation* of the DFS algorithm can substantially improve performance. In the future, we hope to explore memory locality issues during dynamic reordering, which is an important part of any BDD package, and the impact of instruction mixes.

Acknowledgments

We want to thank Rajeev Ranjan and Jagesh Sanghavi for the CAL package and for answering our many questions, and Jean-Christophe Madre for many fruitful discussions. We would also like to thank Artur Klauser for his graphics package.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. Cache performance of operating system and multiprocessing workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, November 1988.

[2] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 622–627, San Jose, CA, November 1994.

[3] R. K. Brayton et al. VIS: A system for verification and synthesis. Technical Report UCB/ERL M95/104, Electronics Research Lab, Univ. of California, December 1995.

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

[5] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, Amsterdam, February 1991.

[6] J. H. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[7] D. E. Long. Robdd package, 1993.

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

[9] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, Santa Clara, CA, November 1993.

[10] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. Sangiovanni-Vincentelli. High performance BDD package based on exploiting memory hierarchy. In *Proceedings of the Design Automation Conference*, Las Vegas, NV, June 1996. To appear.

[11] A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[12] F. Somenzi. *CUDD: CU Decision Diagram Package*. ftp://vlsi.colorado.edu/pub/.

[13] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*. ACM, 1994.