High Performance BDD Package By Exploiting Memory Hierarchy

Jagesh V. Sanghavi* Rajeev K. Ranjan† Robert K. Brayton Alberto Sangiovanni-Vincentelli
Email: {sanghavi,rajeev,brayton,alberto}@eecs.berkeley.edu
Department of Electrical Engineering and Computer Sciences
University of California at Berkeley, CA 94720

Abstract

The success of binary decision diagram (BDD) based algorithms for verification depend on the availability of a high performance package to manipulate very large BDDs. State-of-the-art BDD packages, based on the conventional depth-first technique, limit the size of the BDDs due to a disorderly memory access patterns that results in unacceptably high elapsed time when the BDD size exceeds the main memory capacity. We present a high performance BDD package that enables manipulation of very large BDDs by using an iterative breadth-first technique directed towards localizing the memory accesses to exploit the memory system hierarchy. The new memory-oriented performance features of this package are 1) an architecture independent customized memory management scheme, 2) the ability to issue multiple independent BDD operations (superscalarity), and 3) the ability to perform multiple BDD operations even when the operands of some BDD operations are the result of some other operations yet to be completed (pipelining). A comprehensive set of BDD manipulation algorithms are implemented using the above techniques. Unlike the breadth-first algorithms presented in the literature, the new package is faster than the state-of-the-art BDD package by a factor of upto 1.5, even for the BDD sizes that fit within the main memory. For BDD sizes that do not fit within the main memory, a performance improvement of up to a factor of 100 can be achieved.

1 Introduction

The manipulation of very large binary decision diagrams (BDDs) [1] (for BDD related terminology, please refer to [2]) is the key to success for BDD-based algorithms for simulation, synthesis, and verification of integrated circuits and systems [3]. Conventional BDD algorithms are based on a recursive formulation that leads to a depth-first traversal of the directed acyclic graphs representing the operand BDDs. A typical recursive depth-first BDD algorithm is shown in Figure 1. The depth-first traversal visits the nodes of the operand BDDs on a path-by-path basis. The large in-degree of a typical BDD node makes it is impossible to assign contiguous memory locations for the BDD nodes along a path. Therefore, the recursive depth-first traversal leads to an extremely disorderly memory access pattern.

In a typical computer system, the memory is organized hierarchically with smaller, faster, and more expensive (per byte) memory closer to the processor [4]. A simplified memory hierarchy consists of processor registers, several levels of on- and off-chip caches (SRAM), main memory (DRAM), and a hard disk. When the BDD size exceeds the capacity of a given level in the memory system, the disorderly pattern of the depth-first algorithms translates to a severe performance penalty. When the BDD size exceeds the cache size, a slowdown by a factor of 2-10 may be observed due to a high cache miss rate. When the BDD size measured in the number of memory pages exceeds the number of translation look-aside buffer (TLB) entries, a further slowdown may be observed. However, the most dramatic degradation in performance is observed when the BDD size exceeds the main memory size; the depth-first algorithms thrash the virtual memory leading to unacceptably high elapsed time even though the amount of CPU time spent doing useful work is low. Therefore, the depth-first algorithms place a severe limit on the size of the BDD that can be effectively manipulated on a given computer system.

```
df_op(op,F,G)
    if (terminal case(op, F, G)) return result;
    else if (computed table has entry(op, F, G)) return result;
    else
        let x be the top variable of F, G;
        T = df_op (op, F_x, G_x);
        E = df_op (op, F_x', G_x');
        if (T equals E) return T;
        result = find or add in the unique table (x, T, E);
        insert in the computed table ((op, F, G) , result);
    endif
return result;
```

Figure 1: Depth-First BDD Manipulation Algorithm

To a first approximation, the performance of BDD manipulation algorithms is dominated by the performance and capacity of each level in the memory system hierarchy. Hence, the design of high performance BDD algorithms require a careful consideration of memory related issues. We present a new BDD package that enables manipulation of very large BDDs by directing the iterative breadth-first technique towards localizing the memory accesses to exploit the memory system hierarchy. Our novel contributions are:

- data structures and memory management techniques to preserve the locality of reference,
- a technique to perform multiple, independent BDD operations simultaneously (*superscalarity*), and
- a technique to perform multiple BDD operations even when the operands of some BDD operations are the result of some other BDD operations yet to be completed (*pipelining*).

Superscalarity and pipelining are targeted towards deriving higher performance from the memory system hierarchy by exploiting the locality of reference in the memory access pattern across several BDD operations.

The rest of the paper is organized as follows. In Section 2, we discuss the breadth-first manipulation algorithm and describe the related work. In Section 3, we present the concepts of performing BDD operations in a superscalar and pipelined manner. Section 4

---

* Supported by Micro Grant
† Supported by Motorola and SRC Grants

33rd Design Automation Conference ®

briefly describes how superscalarity and pipelining are exploited to obtain efficient algorithms for common BDD operations. Experimental results are presented in Section 5 and conclusions in Section 6.

2 Breadth-First Technique for BDD Manipulation

Originally proposed by Ochi *et al.* [5], the iterative breadth-first technique for BDD manipulation attempts to fix the disorderly memory access behavior of the recursive depth-first technique. Unlike the depth-first algorithm that traverses the operand BDDs on a path-by-path basis, the iterative breadth-first algorithm traverses the operand BDDs on a level-by-level basis, where each level corresponds to the index of a BDD variable. The BDD nodes corresponding to a level are allocated from the same memory segment so that temporal locality in accessing the BDD nodes for a specific level translates into spatial locality.

The basic iterative breadth-first technique consists of two phases: a top-down (from root node to leaves) APPLY phase followed by a bottom-up REDUCE phase. The algorithm for a two operand boolean operation is shown in Figures 2, 3, and 4. During the APPLY phase,

```
bf_op(op, F, G)
    if terminal case (op, F, G) return result;
    min_index = minimum variable index of (F, G)
    create a REQUEST (F, G) and insert in REQUEST QUEUE[min_index];
    /* Top down APPLY phase */
    for (index = min_index; index ≤ num_vars; index++) bf_apply(op, index);
    /* Bottom up REDUCE phase */
    for (index = num_vars; index ≥ min_index; index--) bf_reduce(index);
    return REQUEST or the node to which it is forwarded;
```

Figure 2: Breadth-First BDD manipulation algorithm

```
bf_apply(op, index)
    x is variable with index "index";
    /* process each request queue */
    while (REQUEST QUEUE[index] not empty)
        REQUEST (F, G) = unprocessed request from REQUEST QUEUE[index];
        /* process REQUEST by determining its THEN and ELSE */
        if (NOT terminal case ((op, F_x, G_x), result))
            next_index = minimum variable index of (F_x, G_x)
            result = find or add (F_x, G_x) in REQUEST QUEUE[next_index]
        REQUEST → THEN = result;
        if(NOT terminal case ((op, F_x', G_x'), result))
            next_index = minimum variable index of (F_x', G_x')
            result = find or add (F_x', G_x') in REQUEST QUEUE[next_index]
        REQUEST → ELSE = result;
```

Figure 3: Breadth-First BDD manipulation algorithm - APPLY

the outstanding REQUESTS are processed on a level-by-level basis. The processing of a REQUEST R = (op, F, G), in general, results in issuing two new REQUESTs which represent the THEN and the ELSE cofactors of the result (F op G). Since certain isomorphism checks cannot be performed, the result BDD obtained at the end of APPLY phase has redundant nodes. The REDUCE phase traverses the result BDD from the leaves to the root on a level-by-level basis eliminating the redundant nodes.

The APPLY phase of the algorithm needs to determine the indices of cofactor nodes in order to appropriately determine the index of the new REQUESTS (see underlined in Figure 3). In order to preserve the locality of references, it is important to determine the variable index of a BDD node without actually fetching it from memory. In particular, the routine *bf_apply* called with index i should access nodes only at index i. Ochi *et al.* [5] use Quasi-Reduced BDDs (QRBDDs) to solve this problem. Essentially, pad nodes are introduced along each path of the BDD so that consecutive nodes along a path differ in their indices by exactly one. This solution also

```
bf_reduce(index)
    x is variable with index "index";
    /* process each request queue */
    while (REQUEST QUEUE[index] not empty)
        /* process each request */
        REQUEST (F, G) = unprocessed REQUEST from REQUEST QUEUE[index];
        if (REQUEST→THEN is forwarded to T)
            REQUEST→ THEN = T;
        if (REQUEST→ELSE is forwarded to E)
            REQUEST→ ELSE = E;
        if (REQUEST→THEN equals REQUEST→ELSE)
            forward REQUEST to REQUEST → THEN ;
        else if (BDD node with (REQUEST→ THEN ,
            REQUEST → ELSE ) found in UNIQUE TABLE[index])
            forward REQUEST to that BDD node;
        else
            insert REQUEST to the UNIQUE TABLE[index] with key
                (REQUEST THEN , REQUEST ELSE )
```

Figure 4: Breadth-First BDD manipulation algorithm - REDUCE

localizes memory accesses to check for duplicate requests during the APPLY phase and redundant nodes during the REDUCE phase. However, it is observed that the QRBDD is several times larger than the corresponding BDD [6], which makes this approach impractical for manipulating very large BDDs. Ashar *et al.* [6] use a BLOCK-INDEX table to determine the variable index from a BDD pointer by performing an associative lookup. Since this solution employs BDDs (as opposed to QRBDDs), an attempt is made to preserve the locality of reference during the check for duplicate requests and check for redundant nodes by sorted accesses to nodes based on their variable indices. The limitation of this approach is that it has a significant overhead (about a factor of 2.65) as compared to a depth-first based algorithm for manipulating BDDs which fit within the main memory [6].

```
struct Bdd {
  int bddIndex;                  /* 2 Bytes */
  struct BddNode *bddNode;       /* 4 Bytes */
}

struct BddNode {
  struct BddNode *next;          /* 4 Bytes */
  struct Bdd thenBdd;            /* 6 Bytes */
  struct Bdd elseBdd;            /* 6 Bytes */
}
```

Figure 5: BDD and BDD Node Data Structure

Our approach to handling variable index determination problem differs from the works of Ashar *et al.* and Ochi *et al.* in the following aspects:

1. A new BDD node data structure is introduced to determine the variable index while preserving the locality of accesses (see Figure 5. We represent a BDD using {variable index, BDD node pointer} tuple. Therefore a BDD node contains pointers to THEN and ELSE cofactors as well as their variable indices. Hence we do not need to fetch the cofactors to determine their indices. Unlike the conventional BDD data structure that stores its variable index in the BDD node, the new BDD data structure stores the variable indices of its THEN and ELSE BDD nodes. The new BDD node data structure is very compact: on a 32-bit architecture it only requires 16 bytes, which is the same as the memory required to represent the conventional BDD node structure.

2. Optimized processing of REQUEST QUEUES for each level by eliminating the sorted processing of REQUESTs during APPLY

and REDUCE phases as proposed by Ashar *et al.* Empirically, we have observed that this change does not affect the performance of our algorithm for manipulating very large BDDs.

3. Use of a customized memory manager to allocate BDD nodes which are quad-word aligned. The quad-word alignment improves the cache performance by mapping a BDD node to a single cache line. The customized memory allocator aligns the BDD nodes to quad-word boundaries so that a total of 12 bits (last four bits of THEN, ELSE, and NEXT pointers) can be used to tag important data such as complement flags, marking flags, and the reference count. The tag bits are assigned so as to minimize the amount of computational overheads.

Since we eliminate the overheads associated with the previous breadth-first approaches, the new algorithms are faster than corresponding recursive algorithms on many examples for which the BDDs fit in the main memory.

3 Superscalarity and Pipelining

In this section, we propose two new concepts – superscalarity and pipelining – to optimize the memory performance of the iterative breadth-first BDD algorithms by exploiting locality of reference that exists among multiple BDD operations. The concepts of superscalarity and pipelining have their roots in the field of computer architecture in which superscalarity refers to the ability to issue multiple, independent instructions and pipelining refers to the ability to issue a new instruction even before completion of previously issued instructions. We shall see how these concepts can be applied in the context of the breadth-first BDD algorithms to exploit the memory system hierarchy.

To improve the performance of the basic breadth-first algorithm, we take a closer look at its memory access pattern assuming that the variable index can be determined without destroying the locality of references. During the APPLY phase for a specific index, the following types of memory accesses take place:

1. Accesses to UNIQUE TABLE BDD nodes for that index using BDD pointers to obtain their THEN and ELSE cofactors.
2. Accesses to each of the REQUEST for that index.
3. Associative lookups in appropriate REQUEST QUEUEs to check for duplicate REQUESTs.

During the REDUCE phase for a specific level, the following types of memory accesses take place:

1. Accesses to THEN and ELSE BDD nodes to check for redundancy.
2. Associative lookups in the UNIQUE TABLE for that index to determine if another node with the same attributes already exists.

For a very large BDD that exceeds the main memory capacity, the number of page faults is dominated by the memory accesses to a large UNIQUE TABLE. The reason for this is that the UNIQUE TABLE pages are accessed on a level-by-level basis during the APPLY and the REDUCE phase and the UNIQUE TABLE nodes within each level are accessed randomly. *Superscalarity* and *pipelining* attempt to amortize the cost of page faults for accessing UNIQUE TABLE pages for a specific level among several BDD operations.

3.1 Superscalarity

The concept of superscalarity in the context of breadth-first BDD algorithms refers to the ability to issue multiple, independent BDD operations simultaneously. Two BDD operations are said to be *independent* if their operands are reduced ordered BDDs, i.e. the nodes for the operand BDDs are in the UNIQUE TABLE. Performing multiple, independent BDD operations concurrently during the same APPLY and REDUCE phase amortizes the cost of page faults for accessing the UNIQUE TABLE entries. By issuing several independent operations simultaneously, the number of REQUEST nodes in the REQUEST QUEUE increases. However, the number of page faults for accessing UNIQUE TABLE nodes for a specific index does not increase proportionately; it increases at a lesser rate. Empirically, we observe a significant performance enhancement in the BDD algorithms by exploiting superscalarity.

Another major advantage of superscalarity is complete inter-operation caching of intermediate BDD results. A breadth-first algorithm for a single BDD operation provides complete caching of intermediate results during the operation by virtue of the REQUEST QUEUE. However, it is not possible to have inter-operation caching in the breadth-first algorithm without expending additional memory resources to store the cached results and additional computing resource to manage the complex caching scheme since the contents of a REQUEST node are destroyed after it is processed in the APPLY phase and correct result is unavailable until the REQUEST is processed in the REDUCE phase. Superscalarity provides complete inter-operation caching for the set of independent BDD operations that are issued simultaneously, thereby enhancing the performance of the breadth-first algorithm even further.

3.2 Pipelining

The concept of pipelining in the context of breadth-first BDD algorithms refers to the ability to issue multiple, dependent BDD operations simultaneously. A BDD operation op_1 is said to be *dependent* on another BDD operation op_2 if the result BDD of op_2 is an operand of op_1. The pipelining algorithm issues several dependent operations simultaneously using unprocessed requests to represent operands for the dependent operations. The result BDDs for these requests are obtained by a single APPLY and REDUCE phase that amortizes the cost of page faults for accessing the UNIQUE TABLE entries.

We make the following three observations that allow us to process a set of dependent requests in a single APPLY and REDUCE phase. The first observation is related to the nature of the breadth-first iterative algorithm. There is a one-to-one correspondence between requests processed and the nodes in the unreduced BDD created during the APPLY phase. In fact, a processed request with THEN and ELSE pointers pointing to newly issued requests corresponds to the BDD node in the unreduced BDD obtained at the end of the APPLY phase. The second observation is related to manipulating unreduced BDDs. If $b_i, i = 1, \ldots, n$ are unreduced BDDs, then result BDD b obtained by performing a boolean operation with operands b_i is an unreduced BDD. The important point here is that operand BDDs for a boolean operation can be unreduced. The third observation is related to processing of requests in the APPLY phase of the breadth-first algorithm. In general, while processing a REQUEST, two new requests are issued. Each of the new request corresponds to cofactors of the operands that constitute the request. This implies that we need THEN and ELSE pointers only for the operand with the minimum index. From these observations, we state the following theorem without proof.

Theorem 1 Correctness of pipelining: *Given* REQUEST R_1 = *(*op_1, F_1, G_1*) and* REQUEST R_2 = *(*op_2, F_2, G_2*) and* REQUEST R = *(*op, R_1, R_2*), the breadth-first algorithm with modified* APPLY *and* REDUCE *phases, which process the* REQUESTS *in level-by-level order while maintaining the partial order implied by the dependence of* REQUESTS *for that index, correctly computes the reduced BDDs corresponding to the* REQUESTS R_1, R_2*, and* R*.*

The concept of pipelining improves the performance of the breadth-first algorithm by amortizing the cost of page faults across dependent BDD operations. However, pipelining results in operations on unreduced BDDs. Hence, there is an increase in the size of the working memory required and corresponding increase in the amount of computation. If the increase in the working memory is large, the number of page faults will increase.

Pipelining will improve the amount of caching, since the intermediate BDDs in the dependent operation can use the cached result of all operations on which the current REQUEST depends directly or indirectly. However, it introduces the penalty of performing associative lookups in several REQUEST QUEUEs, which offsets the potential gain due to improved caching.

3.3 Application

Superscalarity and pipelining find their applications whenever a set of dependent and/or independent BDD operations needs to be performed. In this section we describe how these techniques are used in creating output BDDs of a circuit.

In many logic synthesis and verification applications we need to compute the BDDs for the outputs of a circuit. Given a network representing a circuit, we try to compute the function of the outputs in terms of the primary inputs. This requires computing the function of nodes of network starting from the primary inputs to primary outputs. Pipelining and superscalarity can be employed to compute the output BDDs in several ways. Our algorithm is as follows:

1. Decompose the given network into two input NAND nodes.
2. Levelize the nodes of the new network.
3. Create the BDDs for nodes belonging to a particular level concurrently (using superscalarity), or
4. Create the BDDs for nodes belonging to two or more levels using pipelining.

The motivation behind decomposing the network into NAND nodes is to obtain as much superscalarity as possible. In Section 5 we provide experimental results indicating the effect of superscalarity and pipelining on creating the output BDDs.

4 Optimized BDD Algorithms

We have incorporated iterative breadth-first technique, superscalarity, and pipelining into a comprehensive set of high performance BDD algorithms for boolean operations such as AND, OR, XOR, NAND, NOR, XNOR, ITE, COFACTOR, RESTRICTION, COMPOSITION, SUBSTITUTION, EXISTENTIAL QUANTIFICATION, UNIVERSAL QUANTIFICATION, RELATIONAL PRODUCT, and VARIABLE SWAPPING. Each of these new algorithms raises specific issues which must be addressed to obtain a high performance BDD package. In Section 5.4, we demonstrate the performance of our algorithms. In this section we describe one of these algorithms.

Existential Quantification: EXISTENTIAL QUANTIFICATION of a function f with respect to a variable x is given by $\exists_x f = f_x + f_{x'}$. EXISTENTIAL QUANTIFICATION of a function f with respect to a set of variables $\mathcal{X} = \{x_1, x_2, \dots x_n\}$, is given as, $\exists_{\mathcal{X}} f = \exists_{x_n}(\exists_{x_{n-1}} \cdots (\exists_{x_1} f))$. The conventional depth-first algorithm for EXISTENTIAL QUANTIFICATION is given in Figure 6. One salient feature of the algorithm for QUANTIFICATION is that only one cofactor needs to be processed in some cases. As seen in Figure 6, if the top variable of the function is quantified, then we need not process the negative cofactor if the result of the positive cofactor is the constant 1 (as underlined in the figure). This feature is distinct from most of the other BDD operation algorithms. However, this optimization requires traversing the BDD on a path-by-path basis. A straight forward breadth-first implementation that processes both the cofactors in the APPLY phase will incur the overhead of unnecessary computations. Hence, it is imperative to adopt a strategy which benefits not only from the regular memory access due to the level-by-level manipulation of the BDDs, but also minimizes the overhead of unnecessary computations.

```
df_exist(F)
    if (terminal case(F)) return result;
    if (computed table has entry(F)) return result;
    let x be the top variable of F;
    T = df_exist(F_x);
    if (x is to be quantified and T == 1) return 1;
    E = df_exist(F_x');
    if (x is to be quantified) return df_or(T, E);
    else result = find or add in the unique table (x, T, E);
    return result;
```

Figure 6: Depth-First Algorithm for Existential Quantification

We propose a new mixed breadth- and depth-first approach to overcome this problem. In this approach, we process the REQUESTs differently depending upon whether the corresponding variable index is quantified or not. For REQUESTs belonging to quantified variables, we process both the cofactors. However, for REQUESTs belonging to remaining variables, we do a path-by-path traversal, i.e., we process just one of the cofactors. In the REDUCE phase, we process the REQUESTs belonging to the indices which are not quantified in the same manner as given in *bf_reduce* (Figure 4). However for variable indices which are to be quantified, we process the REQUESTs differently. If the result of the positive cofactor is the tautology then that node is forwarded to the constant One. Otherwise, we proceed to find the result of the other cofactor and take the "OR" of the results of the two cofactors. We employ superscalarity in finding the "OR" of the cofactor results for nodes belonging to a particular level.

5 Experimental Results

We integrated our package with the synthesis tool SIS [7]. In addition to using standard ISCAS and MCNC benchmark examples for the set of experiments, we use a series of sub-networks of the MCNC benchmark C6288 in order to systematically analyze the performance of our algorithms as BDD size increases. These artificially created examples have the property that the number of BDD nodes needed to represent the BDDs corresponding to the outputs are roughly multiples of one million. This enabled us to illustrate the gradual change in various performance metrics with the change in example size. These examples are denoted as "C6288_iM.blif", implying that the total number of BDD nodes in the manager after computing the BDDs for the outputs of C6288_iM.blif is i millions.

For each of the benchmark examples, we create the BDDs for the outputs of the circuit using "dfs-ordering" in SIS to order the variables. We use these output BDDs as argument BDDs in our experiments. For instance, to compare the performance of the AND operation, we iteratively select random pairs from the output BDDs and compute the AND of the pair. Similarly, to compare the performance of QUANTIFICATION operation, we select one of the output BDDs randomly and also randomly select a set of variables to be quantified. Functions and the variables selected are the same for both the packages.

We made "black box" comparison with best reported BFS algorithm [6] on Sun Sparc2 workstation with 40MB main memory. Our approach is faster by a factor of 4.4 (geometric mean) for creating output BDDs for C6288 subcircuits with one to seven million nodes. The performance improvement is mainly due to new implementation technique, superscalarity, and pipelining. Unfortunately, it was impossible to eliminate the difference in variable ordering, which may have some effect on performance, since we did not have access to their source code.

The following experiments were performed on a DEC5400 with 128KB processor cache, 64MB main memory and 1GB of disk storage.

5.1 Creating Output Bdds For Circuits

In Table 1 we present the performance comparison for creating output BDDs for large examples. We observe that when the number

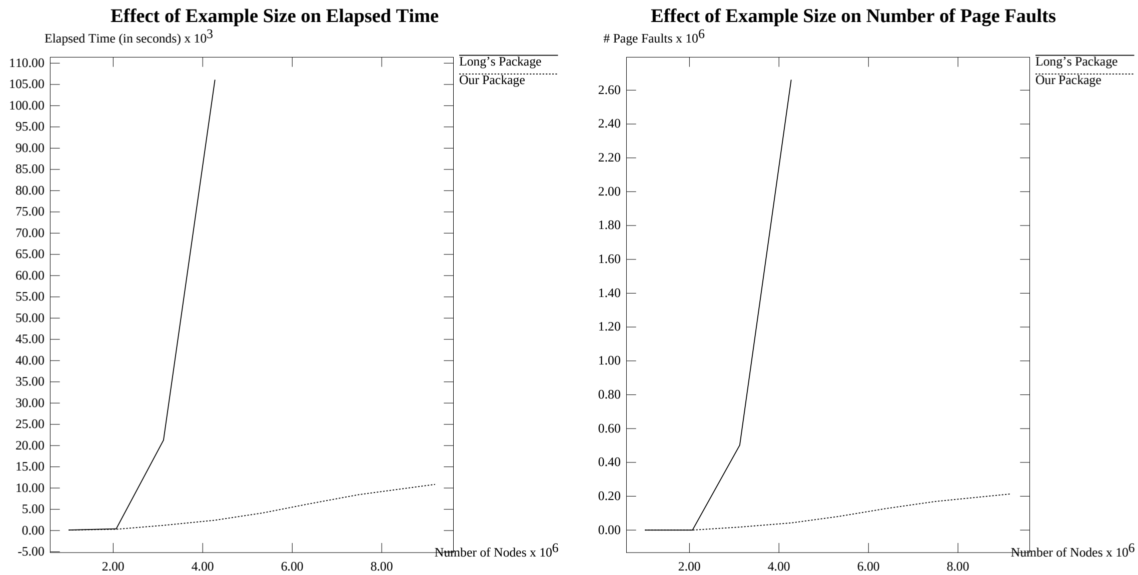

Figure 7: Variation of Elapse Time and Number of Page Faults with Example Size

Example	# Nodes	CPU Time		Elapsed Time		# Page Faults	
		A	B	A	B	A	B
C6288_1M	1,001,855	112	98	127	110	0	0
C6288_2M	2,066,878	273	215	403	306	0	0
C6288_3M	3,123,327	491	347	21281	1218	502059	17800
C6288_4M	4,273,510	820	490	106110	2433	2661738	42509
C6288_5M	5,337,005	**t.o.**	631	–	4140	–	80621
C6288_6M	6,381,496	–	804	–	6295	–	126977
C6288_7M	7,489,064	–	981	–	8454	–	168794
C6288_9M	9,193,222	–	1147	–	10864	–	213976

Table 1: Performance comparison for creating output BDDs: Long's BDD package (A) vs. our package (B)

t.o:Process killed after 21.5 hours of elapsed time.

of BDD nodes becomes too large to fit in the main memory, the number of page faults and the elapsed time increase drastically for Long's package [8]. In Figure 7, we show the number of page faults and the elapsed time as a function of example size. We observe that for Long's package, an increase in the BDD size beyond the main memory size results in a sharp increase in the number of page faults and hence excessive elapsed time. This is in contrast to the page fault behavior of our package which increases linearly with an increase in the example size.

Table 2 gives the performance of our package on building very large BDDs for some benchmark examples. We observe that we have been able to build BDDs with more than 23 million nodes in less than nine hours.

Example	# Nodes	Elapsed Time	# Page Faults
C3540	2.76×10^6	25 mins	26603
C2670	10.40×10^6	4 hrs 4 mins 58 secs	357005
C6288_12M	12.80×10^6	6 hrs 39 mins 54 secs	719697
s38417	23.15×10^6	8 hrs 49 mins 26 secs	868442

Table 2: Performance metrics for creating output BDDs for some very large examples

5.2 Performance Enhancement Due to Superscalarity

We demonstrate the power of superscalarity on three different applications: creating output BDDs, ARRAY AND, and QUANTIFICATION.

Example	CPU		Elapsed		# Page Faults	
	w/o SS	w SS	w/o SS	w SS	w/o SS	w SS
C6288_4M	485.82	436.94	2214	2452	53261	63272
C6288_5M	630.37	602.50	5333	4648	172205	137004
C6288_6M	815.91	724.99	18354	7840	712868	252323
C6288_7M	956.89	831.30	24747	10267	970192	328086

Table 3: Performance improvement using superscalarity for creating output BDDs

We described in the section 3.3 how superscalarity can be exploited for creating output BDDs. In Table 3, we show the performance improvement achieved by employing superscalarity. We observe that in all the cases employing superscalarity results in better performance. Also in all the cases except C6288_4M, we observe that the number of page faults decreases with the use of superscalarity and we achieve a better performance by a factor of more than 2.

In Table 4, we present the results on how superscalarity affects the performance of ARRAY AND and QUANTIFICATION. In ARRAY AND we are given an array of operand BDD pairs and we need to compute the AND of each of the operand pair. These operands were randomly chosen from the set of output BDDs of the circuit. For the quantification operation, we perform OR operations one at a time during its reduce phase, to illustrate the effect of superscalarity.

Example	BDD Operation							
	Array And				Quantify			
	CPU		Elapsed		CPU		Elapsed	
	W	X	W	X	Y	Z	Y	Z
C1355	134.03	119.41	137	123	13.40	12.30	13	12
C6288_1M	411.35	403.90	847	740	5.02	4.27	5	5
C6288_2M	290.55	283.41	595	581	18.17	16.16	18	17
s1423	35.39	18.02	37	18	20.63	19.13	31	30
C6288_3M	682.57	655.21	21005	7178	12.60	11.59	13	12
minmax10	810.32	679.88	19304	1619	66.0	55.8	91	81

Table 4: Performance improvement using superscalarity for ARRAY AND and QUANTIFICATION BDD operations.

W: ARRAY AND performed iteratively.
X: ARRAY AND performed in superscalar manner.
Y: In REDUCE phase of QUANTIFICATION, OR operations performed one be one.
Z: In REDUCE phase of QUANTIFICATION, OR operations performed in superscalar manner.

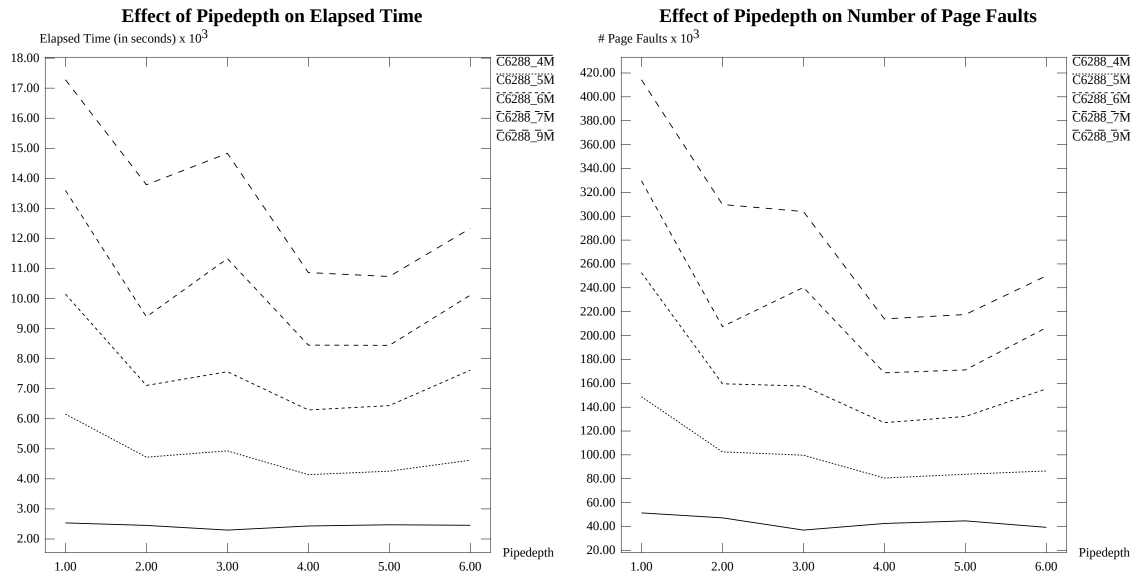

Figure 8: Variation of Elapse Time and Number of Page Faults with Pipedepth in Creating Output BDDs

For small circuits, superscalarity improves the performance due to inter-operation caching. For large circuits, superscalarity improves the performance due to increased locality to the memory access, resulting in less page faults. We observe from Table 4 that for examples which can fit in the main memory, superscalarity helps with improved CPU time. For large examples we observe a performance improvement of upto a factor of 10.

5.3 Performance Enhancement Due to Pipelining

We demonstrate the effect of pipelining on the performance of creating BDD for outputs. We have described in Section 3, how pipelining technique can be exploited. Figure 8, depicts the effect of pipedepth on the elapsed time and the number of page faults for a series of C6288 sub-examples. The pipedepth refers to number of levels of dependencies. As pipedepth is increased, we see a decrease in the number of page faults (hence the decrease in elapsed time). However, the memory overhead increases with increase in pipedepth since we are working with unreduced BDDs. Hence, after a certain value of pipedepth, the decrease in page faults due to pipelining is offset by the increase in page faults due to memory overhead. We observe that a pipedepth of four is optimum in most cases.

5.4 Performance Comparison For Various BDD Operations

One of our objectives was to provide a comprehensive set of algorithms for all BDD operations. We compare the performance of some of our algorithms with those of Long's package for small and medium sized examples.

All examples considered in small size category have less than 7000 BDD nodes. This implies that with a processor cache size of 128KB, it is possible that all the nodes can reside in the cache if node addresses are properly aligned. Since our node data structure is quad word aligned, the node address does not overlap across cache lines. Hence we can expect a significant cache hit rate during BDD manipulations. Long's package, however, does not provide the word alignment and hence it is likely that the BDD node addresses could overlap across cache lines. We ran experiments to compare the performances of various BDD operations. We observed a performance ratio of 1.92 across all small sized examples and four BDD operations.

In Table 5 we provide the performance comparison between packages for medium size examples. Since the number of nodes are of the order of tens of thousands to hundreds of thousands, the cache effect seen for the small size examples is not dominant in this case. However, in most of the cases, we observe a performance improvement over Long's package. Overall performance ratio over all medium sized examples and across four BDD operations given in the tables, is about 1.5. The most significant is the relative performance on SUBSTITUTE. We observe that on many examples, Long's package could not finish the SUBSTITUTION in 10,000 CPU seconds whereas our package took just about 1000 CPU seconds to complete. This substantiates the significant performance enhancement using superscalarity as mentioned in Section 4. We notice that for QUANTIFICATION operation our package consistently performs worse than Long's package by up to a factor of 0.6 due to inevitable management overheads of mixed depth-first breadth-first technique.

6 Conclusions and Future Work

We have presented new techniques targeting the memory access problem for manipulating very large BDDs. These include 1) an architecture independent customized memory management and new BDD data structures, 2) performing multiple BDD operations concurrently (superscalarity), and 3) performing a BDD operation even when the operand(s) are yet to be computed (pipelining). A complete package consisting of the whole suite of BDD operations based on these techniques has been built. We demonstrate the performance of our package by 1) comparing with state-of-the-art BDD package [8], and 2) performing a comprehensive set of experiments to substantiate the capability of our approach. We show that our package provides competitive performance on small examples and a performance ratio of more than 100 on large examples.

We are in the process of extending the breadth-first manipulation technique to exploit the memory and computing resources of a network of workstations efficiently.

Acknowledgement

Authors would like to thank Thomas Shiple for his critical review of the draft.

References

[1] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulation," *IEEE Trans. Computers*, vol. C-35, pp. 677–691, Aug. 1986.

[2] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," in *Proc. of the Design Automation Conf.*, pp. 40–45, June 1990.

[1] SwapVars(f, x, y) is a function obtained from the function f by replacing variable x by y and vice-versa.

Example	BDD Operation											
	bdd_and						bdd_substitute					
	CPU			Elapsed			CPU			Elapsed		
	A	B	A/B	A	B	A/B	A	B	A/B	A	B	A/B
C1355	16.68	14.45	1.15	17	15	1.13	**t.o.**	1005.32	> 10	–	14710	> 20
C1908	7.80	7.05	1.11	8	7	1.14	**t.o.**	62.65	> 150	–	67	> 150
C432	18.23	17.77	1.03	19	18	1.06	3144.85	258.89	12.15	4746	476	9.97
C499	15.53	14.64	1.06	16	15	1.07	**t.o.**	1002.21	> 10	–	14749	> 20
C5315	12.01	6.52	1.84	16	7	2.29	0.91	0.16	5.69	1	1	1
C880	9.03	7.29	1.24	10	8	1.25	1.05	1.18	0.89	1	1.0	1
s1423	2.25	2.48	0.91	2	3	0.67	292.68	88.73	3.3	350	97	3.6
Example	**bdd_exist**						**bdd_swapvars**					
C1355	45.99	63.05	0.72	47	65	0.72	30.42	23.78	1.28	31	25	1.24
C1908	13.62	21.30	0.64	14	22	0.64	9.34	8.22	1.14	9	8	1.12
C432	59.08	99.80	0.59	60	102	0.59	33.94	29.67	1.14	34	31	1.10
C499	40.70	59.28	0.69	41	60	0.68	29.15	23.81	1.22	30	24	1.25
C5315	32.97	35.65	0.92	34	37	0.92	4.12	3.52	1.17	4	4	1
C880	17.28	21.98	0.79	18	22	0.82	10.65	8.62	1.24	11	9	1.22
s1423	16.57	28.31	0.59	17	29	0.59	1.63	2.34	0.70	2	3	0.67

Table 5: Performance comparison on medium size examples for “And”, “Substitute”, “Existential Quantification , and “SwapVars” operations: Long’s BDD package (A) vs. our package (B).

t.o.: Time out after 10,000 CPU seconds.

[3] R. Bryant, “A methodology for hardware verification based on logic simulation,” *Journal of the Association for Computing Machinery*, vol. 38, pp. 299–328, Apr. 1991.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 1990.

[5] H. Ochi, K. Yasuoka, and S. Yajima, “Breadth-First Manipulation of Very Large Binary-Decision Diagrams,” in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 48–55, Nov. 1993.

[6] P. Ashar and M. Cheong, “Efficient Breadth-First Manipulation of Binary Decision Diagrams,” in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 622–627, Nov. 1994.

[7] 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,” Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.

[8] D. E. Long, “ROBDD Package,” (Carnegie Mellon University, Pittsburgh), Nov. 1993.