Space- and Time-Efficient BDD Construction via Working Set Control

Bwolen Yang* Yirng-An Chen† Randal E. Bryant† David R. O'Hallaron*

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213. USA

Abstract— Binary decision diagrams (BDDs) have been shown to be a powerful tool in formal verification. Efficient BDD construction techniques become more important as the complexity of protocol and circuit designs increases. This paper addresses this issue by introducing three techniques based on working set control. First, we introduce a novel BDD construction algorithm based on partial breadth-first expansion. This approach has the good memory locality of the breadth-first BDD construction while maintaining the low memory overhead of the depth-first approach. Second, we describe how memory management on a per-variable basis can improve spatial locality of BDD construction at all levels, including expansion, reduction, and rehashing. Finally, we introduce a memory compacting garbage collection algorithm to remove unreachable BDD nodes and minimize memory fragmentation. Experimental results show that when the applications fit in physical memory, our approach has speedups of up to 1.6 in comparison to both depth-first (CUDD) and breadth-first (CAL) packages. When the applications do not fit into physical memory, our approach outperforms both CUDD and CAL by up to an order of magnitude. Furthermore, the good memory locality and low memory overhead of this approach has enabled us to be the first to have successfully constructed the entire C6288 multiplication circuit from the ISCAS85 benchmark set using only conventional BDD representations.

I. Introduction

With the increasing complexity of protocol and circuit designs, formal verification has become an important research area. Binary decision diagrams (BDDs) have been shown to be a powerful tool in formal verification [4]. Even though many functions have compact BDD representations, some functions can have very large BDDs. For example, BDD representations for integer multiplication have been shown to be exponential in the number of input bits [5]. To address this issue, there are many BDD related research efforts directed towards reducing the size of the graph with techniques like new compact representations for specific classes of functions (KFDD [9] and *BMD [6]), divide-and-conquer (POBDD [11] and ACV [7]), function abstraction (aBdd [12]), and variable reordering [19]. Despite these efforts, large graphs can still naturally arise for more irregular functions or for incorrect implementations of a specification. Incorrect implementation can break the structure of a function and thus can greatly increase the graph size. For example, the *BMD representation for integer multiplication is linear. However, a mistake in the implementation of integer multiplication logic can cause an exponential explosion of the resulting graph. The ability to handle large graphs efficiently can enable us to represent more irregular functions and to provide counterexamples for incorrect implementations.

Conventional BDD algorithms [2] are based on depth-first traversal of BDD graphs. This approach has small memory overhead, but poor memory locality. To address the issue of constructing large BDDs efficiently, there have been many implementations [14, 15, 1, 10, 18] based on breadth-first traversal. The breadth-first approach, which exploits its graph traversal pattern by using specialized memory layouts, has better memory access locality and thus often has better performance. However, the breadth-first approach can have a large memory overhead, up to quadratic in the size of BDD operands. This extra memory overhead can result in an increased number of page faults and thus poor performance.

To maintain memory access locality with low memory overhead, we introduce a new algorithm based on *partial breadth-first expansion*. This algorithm improves locality of reference by controlling the working set size and thus reducing overhead due to page faults. We describe how memory management on a per-variable basis can improve spatial locality of BDD construction at all levels, including expansion, reduction, and rehashing. Finally, we introduce a breadth-first BDD garbage collection algorithm which performs memory compaction without incurring additional memory overhead. All of these techniques work together to control the working set size and have a significant impact on performance of BDD construction. As these techniques exploit inherent properties of BDD construction, graph reduction techniques (like *BMD, POBDD, and dynamic variable reordering) can be incorporated into our algorithms to further expand the usefulness of these

*Effort sponsored in part by the Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0287, in part by the National Science Foundation under Grant CMS-9318163, and in part by a grant from the Intel Corporation. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

†Supported in part by the Defense Advanced Research Project Agency (DARPA) under contract number DABT63-96-C-0071.

algorithms.

Experimental results show that when the applications fit in physical memory, our approach has speedups of up to 1.6 in comparison to leading depth-first (CUDD) and breadth-first (CAL) packages. When the applications do not fit into physical memory, our algorithm outperforms both CUDD and CAL by up to an order of magnitude. Furthermore, to demonstrate how our techniques can efficiently build very large graphs, we constructed the output BDDs for the C6288 multiplication circuit from the ISCAS85 benchmark. To the best of our knowledge, this has never been done before.

Beyond the sequential world, another advantage of the partial breadth-first algorithm is that it can be parallelized [22]. This approach achieves speedups of up to four on eight processors of a shared memory system.

The rest of this paper is as follows: Section II gives an overview of BDDs and how they are constructed. Section III describes the partial breadth-first algorithm and other techniques for controlling the working set size. Section IV presents performance evaluation of our implementation. Section V demonstrates the usefulness of this implementation by constructing very large BDDs for 16-bit array multipliers. Finally, Section VII summarizes this paper and offers some concluding remarks.

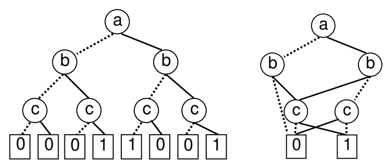
II. BDD Overview

A boolean expression can be represented by a complete binary tree called a *binary decision tree*, which is based on the expression's truth table. Fig.1(a) shows the truth table for a boolean expression and Fig.1(b) shows the corresponding binary decision tree. Each internal vertex is labeled by a variable and has edges directed toward two children: the 0-branch (shown as a dashed line) corresponds to the case where the variable is assigned 0, and the 1-branch (shown as a solid line) corresponds to the case where the variable is assigned 1. Each leaf node is labeled 0 or 1. Each path from the root to a leaf node corresponds to a truth table entry where the value of the leaf node is the value of the function and the path corresponds to the assignment of the boolean variables.

$$f = (b \wedge c) \vee (a \wedge \overline{b} \wedge \overline{c})$$

a	b	c	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

(a)



(b) (c)

Fig. 1. A boolean expression represented with (a) Truth table, (b) Binary decision tree, (c) Binary decision diagram. The dashed-edges are 0-branches and the solid-edges are the 1-branches.

A binary decision diagram (BDD) is a directed acyclic graph (DAG) representation of a binary decision tree where equivalent boolean subexpressions are uniquely represented. Fig.1(c) shows the BDD representation of the binary decision tree in Fig.1(b). Since all subexpressions in a BDD are uniquely represented, a BDD can be exponentially more compact than its corresponding truth table or binary decision tree representations.

One necessary condition for guaranteeing uniqueness of the BDD representation is that all the BDDs constructed must follow the same variable ordering; i.e., for any two variables x and y, if x has higher precedence than y ($x \prec y$), then for any path that contains both x and y, x must appear before y on this path. Note that the BDD size can be very sensitive to the variable ordering where the graph size of one ordering can be exponentially more compact than the graph size of another ordering.

Before describing the basis for BDD construction, we will first introduce some terminology and notation. $f_{x=0}$ and $f_{x=1}$ are *cofactor functions* of the function f with respect to the boolean variable x, where $f_{x=0}$ is equal to f with the value of x set to 0, and $f_{x=1}$ is equal to f with the value of x set to 1. A *reachable subgraph* of a node n is defined to be all the nodes that can be reached from n by traversing 0 or more directed edges. *BDD nodes* are defined to be internal vertices of BDDs. Given a BDD b, the function f represented by b is recursively defined by

$$f = \overline{x} \cdot f_{x=0} + x \cdot f_{x=1} \tag{1}$$

where x is the variable corresponds to b's root node and the cofactor function $f_{x=0}$ is recursively defined by the reachable subgraph of b's 0-branch child. Similarly, $f_{x=1}$ is recursively defined by the reachable subgraph of b's 1-branch child.

A. Basis for BDD Construction

BDD construction is a memoization-based dynamic programming algorithm. Due to the large number of distinct subproblems, instead of a memoization table, a cache known as the *computed cache* is used to record the result of each subproblem. Given a variable ordering and two BDDs f and g, the resulting BDD r of a boolean operation $f\ op\ g$ is constructed based on the Shannon expansion

$$r = f\ op\ g = \overline{\tau} \cdot (f_{\tau=0}\ op\ g_{\tau=0}) + \tau \cdot (f_{\tau=1}\ op\ g_{\tau=1}) \tag{2}$$

where τ is the variable (*top variable*) with the highest precedence among all the variables in f and g, and $f_{\tau=0}$, $f_{\tau=1}$, $g_{\tau=0}$, and $g_{\tau=1}$ are the corresponding cofactor functions of f and g.

In the top-down *expansion phase*, this Shannon expansion process repeats recursively following the given variable ordering for all the boolean variables in f and g. The base case (also called the *terminal case*) of this recursive process is when the operation can be trivially evaluated. For example, the boolean operation $f \wedge f$ is a terminal case because it can be trivially evaluated to f. Similarly, $f \wedge 0$ is also a terminal case. At the end of the expansion phase, there may be unreduced subexpressions like $(\overline{x} \cdot h + x \cdot h)$. Thus, in order to ensure uniqueness, a bottom-up *reduction phase* is necessary to reduce expressions like $(\overline{x} \cdot h + x \cdot h)$ to h. This reduction phase also needs to ensure that each BDD node created is unique.

Fig.2 illustrates the Shannon expansion (Equation 2) for the operation $r = f \; op \; g$. On the left side of this figure, the operation is represented with an *operator node* which refers to BDD representations of f and g as operands. The right side of this figure shows the Shannon expansion of this operation with respect to the variable x. Further expansion of operator nodes can be performed in any order. In particular, the depth-first construction always expands the operator node with the greatest depth. Note that the depth-first algorithm does not explicitly store the operations as operator nodes. Instead, the operation is implicitly stored in the stack as arguments to the recursive calls.

In the breadth-first construction, the Shannon expansion is performed top-down from the variables with the highest to the lowest precedence so that operations with the same top variable are expanded together. The reduction phase is performed bottom-up in reverse order. Thus, all operations with the same top variable are reduced at the same time.

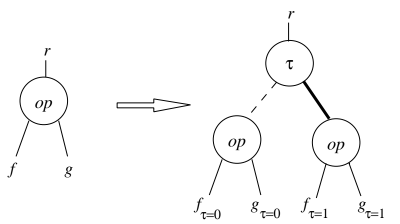


Fig. 2. Shannon Expansion: The dashed edge represent the 0-branch of a variable and the thick solid edge represents the 1-branch

For the rest of this paper, we will refer to boolean operations issued by a user of BDD package as the *top level operations* to distinguish them from operations generated internally by the Shannon expansion process.

B. Memory Overhead and Access Locality

BDD construction is often memory intensive, especially when large graphs are involved. It not only requires a lot of memory, it also requires frequent accesses to many small data structures (the node size is typically 16 bytes on 32-bit machines). The depth-first BDD construction has poor memory behavior because of irregular control flow and memory access patterns. The control flow is irregular because the recursive expansion can terminate at any time when a terminal case is detected or when the operation is cached in the computed cache. The memory access pattern is irregular because a BDD node can be accessed due to expansion on any of its many parents; and, since the BDD is traversed in the depth-first manner, expansions on the parents are scattered in time. The performance impact for the depth-first algorithm's poor memory locality is especially severe for BDDs larger than the physical memory.

Recently, there has been much interest in BDD construction based on breadth-first traversal [14, 15, 1, 10, 18]. In a breadth-first traversal, the expansion phase expands operations one variable at a time with all the operations of the same variable expanded together. Furthermore, during the reduction phase, all the new BDD nodes of the same variable are created together. The breadth-first construction exploits this structured access by clustering nodes (for both BDD and operator nodes) of the same variable together in memory with specialized node managers.

Despite its better memory locality, the breadth-first construction has much larger memory overhead in comparison to the depth-first construction. The number of operations that the depth-first construction keeps tracks of at any given time is the depth of the recursion, which is at most the number of variables. Since the number of variables is typically small, the depth-first construction does not require much memory to store these operations. In contrast, for each top level operation, the breadth-first construction will keep all operations generated by Shannon expansion of this top level operation until the result for this top level operation is constructed. Since the number of operations can be quadratic in the size of the BDD operands, the breadth-first approach can incur a large memory overhead. Thus, on some applications where the depth-first construction fits in physical memory while the breadth-first construction does not, the performance of the breadth-first construction can be significantly worse due to page faults.

III. Our Approach to BDD Construction

Since BDD construction involves a large number of accesses of many small data structures, localizing the memory access pattern to bound the working set size is critical because good memory access locality results in good hardware cache locality and fewer page faults. This section introduces three techniques to control the working set size by limiting memory overhead and by improving both temporal and spatial locality. These are followed by a brief discussion on how these techniques can work together with variable reordering algorithms.

A. Partial Breadth-First Construction

For the pure breadth-first construction (which normally has good memory locality), if the BDD operands do not fit in physical memory, then the pages of operator nodes swapped in during the expansion phase will be swapped out by the time the reduction phase takes place. Furthermore, as described in Section II.B, breadth-first construction can incur a large memory overhead.

To overcome these drawbacks while bounding the memory overhead, we introduce *partial breadth-first expansion* based on *context switch*. Within each evaluation context, the breadth-first expansion is used until a fixed evaluation threshold is reached. Upon reaching this threshold, the current context is pushed onto a *context stack* and a new child context is started. The remaining operations of the parent context are partitioned into smaller groups and the child context evaluates these operations one group at a time. This process repeats each time the current evaluation context reaches its threshold. By keeping the evaluation threshold to be a small fraction of the available physical memory, we can bound the number of BDD nodes and compute cache nodes created and accessed and thus control the working set size. Note that by setting the evaluation threshold to 1, this algorithm degenerates to depth-first construction. Similarly, by setting the evaluation threshold to ∞, this algorithm is identical to pure breadth-first construction.

Fig.3(a) shows an example of a context switch. In this figure, the top triangle denotes the graph of the initial expansion. Upon reaching the evaluation threshold, the remaining unexpanded operations are divided into two partitions (shown as two dashed rectangles) and the new child context is started. This new child context continues to expand on the first partition. After the child context finishes building BDD results for the first partition, it continues to expand on the second partition as shown in Fig.3(b). Note that expansion of these two partitions might share some operations in common. For these common operations, the expansion of the second partition can benefit from the results computed from the expansion of the first partition via the compute cache. However, since the compute cache is not a complete cache, some common operations may need to be recomputed. This figure also depicts how the partial breadth-first construction can reduce memory overhead. The operator nodes created from expanding the first partition do not need to be kept during the expansion of the second partition. In comparison, the pure breadth-first construction (shown in Fig.3(c)) needs to keep all the operator nodes until after the reduction phase.

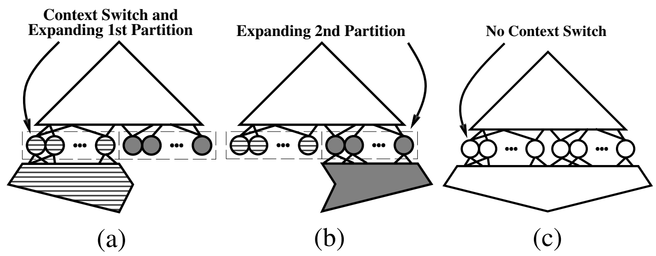


Fig. 3. A Context Switch Example. (a) Upon reaching the evaluation threshold, current unexpanded operations are divided into two partitions (shown as two dashed rectangles) and the new child context continues to expand on the first partition. (b) After the reduction for the first partition, this child context expands on the second partition. (c) Pure breadth-first expansion is shown for comparison.

Other than the memory locality and the memory overhead, the evaluation threshold can also impact the effectiveness of the compute cache. In the pure breadth-first traversal, the expanded operator nodes must be kept until after the reduction phase. This feature effectively resulted in a complete cache within an expansion phase. Similarly for the partial breadth-first approach, expansion within each evaluation context maintains a complete cache. Thus, a larger evaluation threshold results in a larger and more complete cache for the current evaluation context at the cost of higher memory overhead.

The rest of this section formally describes this partial breadth-first algorithm. Fig.4 shows the top level procedure and a helper function for this partial breadth-first construction. For each variable, there is an expansion queue and a reduction queue. An *expansion queue* queues the operations of the same variable to be Shannon expanded during the expansion phase. A *reduction queue* queues the operations of the same variable to be reduced in the reduction phase. The top level procedure pbf-op() builds the result BDD by repeatedly doing the Shannon expansion (line 3) and reduction (line 4) until there are no more operations in the top context (lines 5 to 8) and until there are no more evaluation contexts on the context stack (lines 9 to 11). Procedure preprocess-op() first determines whether or not the operation is a terminal case or is cached (lines 13 to 15). If not, this operation is added to its top variable's expansion queue (lines 17 and 18) to indicate that further Shannon expansion is necessary for this operation. This operation is also inserted into the compute cache (line 19) to avoid expanding redundant operations in the future. This procedure returns either the BDD result (for the terminal case and for the case when the cached result is a BDD) or an operator node. If an operator node is returned, this operator node's field *opNode.result* will contain the result BDD after this operator node is processed in the reduction phase.

```
    pbf-op(op, f, g)
1     opNode ← preprocess-op(op, f, g)
2     if opNode is a BDD node, return opNode.
3     call expansion()
4     call reduction()
5     if top context of the context stack has operations, then
6        take a group of operations from the top context
7        add each operation to its top variable's expansion queue
8        goto line 3 and repeat until top context is empty
9     if context stack is not empty,
10       pop the top context and use it as the current context
11       goto line 3 and repeat until context stack is empty
12    return opNode.result

    preprocess-op(op, f, g)
13    if terminal case, return simplified result
14    if the operation (op, f, g) is in compute cache,
15       return result found in cache
16    opNode ← (op, f, g)
17    τ ← top variable of f and g
18    add opNode to τ's expansion queue
19    insert opNode into the compute cache
20    return opNode
```

Fig. 4. Partial Breadth-First Construction: top level procedure and a helper function

Fig.5 shows the expansion phase. This top-down expansion phase processes operations queued from the variable with the highest to the lowest precedence. Here, all the operations of the same variable are Shannon expanded together (lines 3 to 7). The $branch_0$ and the $branch_1$ fields of an operator node are used to store the results of Shannon expansion, and as described earlier, these results returned by the procedure preprocess-op()

can be either a BDD node or an operator node. In the later case, the procedure preprocess-op() would have queued the new operator nodes to be processed by the expansion phase later. The variable *nOpsProcessed* is used to track the size of the current evaluation context and when it exceeds a constant evaluation threshold *evalThreshold*, the current context is pushed onto the context stack and a new child context is started (lines 9 to 13).

```
    expansion()
1     nOpsProcessed ← 0
2     for each variable x in the current evaluation context
        from the highest to lowest precedence
3       for each node opNode in x's expansion queue
4         (op, f, g) ← opNode
5         opNode.branch_0 ← preprocess-op(op, f_{x=0}, g_{x=0})
6         opNode.branch_1 ← preprocess-op(op, f_{x=1}, g_{x=1})
7         add opNode to variable x's reduce queue
8         nOpsProcessed++
9         if (nOpsProcessed > evalThreshold)
10          partition the remaining operators into small groups.
11          push current context with these operation groups
              onto the context stack
12          start a new evaluation context
13          return
```

Fig. 5. Partial Breadth-First Construction: expansion phase

Fig.6 shows the reduction phase. This bottom-up reduction algorithm is the same as the pure breadth-first construction's reduction phase where Shannon expanded operations are processed together one variable at a time, starting from the variable with the lowest precedence moving upwards to the variables with the highest precedence. The results from the children are obtained in lines 4 to 11. Lines 12 to 19 perform the reduction and ensure the result is unique. The result of a reduction is stored in the *opNode.result* field of an operator node (line 13 and 19).

B. Memory Management

As in breadth-first BDD algorithms, specialized node managers are the key factors in exploiting structured access in the partial breadth-first approach. In our implementation, each variable is associated with a BDD-node manager as in [18]'s breadth-first algorithm. Each variable's BDD-node manager clusters BDD nodes of the same variable by allocating memory in terms of blocks and allocates BDD nodes contiguously within each block. We further extend this clustering concept to using one operator-node manager for each variable. With this design, we not only benefit from good locality of node clustering, we also eliminate the need for having both the expansion and the reduction queues, since we can access all the operator nodes of each variable by simply traversing memory blocks of each operator-node manager.

Furthermore, we associate one compute cache and one unique table per variable. Thus, cache lookup in the expansion phase and the BDD unique table lookup in the reduction

```
    reduction()
1     for each variable x in the current evaluation context
        from the lowest to highest precedence
2       for each node opNode in x's reduce queue
3         (op, f, g) ← opNode
4         if opNode.branch_0 is a BDD,
5           res_0 ← opNode.branch_0
6         else
7           res_0 ← opNode.branch_0.result
8         if opNode.branch_1 is a BDD,
9           res_1 ← opNode.branch_1
10        else
11          res_1 ← opNode.branch_1.result
12        if (res_0 == res_1)
13          opNode.result = res_0
14        else
15          b ← BDD node (x, res_0, res_1)
16          opNode.result ← lookup(unique table, b)
17          if BDD node b does not exist in the unique table,
18            insert b into the unique table
19            opNode.result ← b
```

Fig. 6. Partial Breadth-First Construction: reduction phase

phase will only traverse nodes of the same variable. Since nodes of the same variables are clustered by the node managers, this results in better memory locality. Combined with per-variable node managers, we can perform rehashing for each variable independently by traversing the memory blocks of the corresponding node manager. Again, this rehashing approach has better memory locality than the traditional approach, which traverses the hash table.

C. Garbage Collection

No BDD package is complete without a good garbage collector. External users of a BDD package can free references to exported BDDs and since BDD construction is a memory intensive application, reusing the space of unreachable BDD nodes is important. Most BDD packages use reference counting and maintain a free list of unreferenced nodes. This approach has several drawbacks. Most notably it has poor memory locality because the free-list approach can scatter newly created BDD nodes in memory and thus reversing the clustering effects of specialized node managers.

In our implementation, a mark-and-sweep garbage collector with memory compaction is used. Unlike a copying garbage collector, our garbage collection algorithm performs memory compaction without requiring any additional memory. This compaction algorithm is *stable*; i.e, the nodes' linear ordering is maintained. This property allows nodes which are allocated nearby in time to stay together. This can help access locality because nodes allocated together are likely to be accessed together in the future.

Our garbage collection algorithm consists of two phases, both of which are breadth-first traversal from the variable with highest precedence to the variable with the lowest precedence. The first phase marks and compacts all the reachable nodes and the second phase fixes all the references and rehashes these nodes.

Fig.7 shows the algorithm for the mark-and-compact phase. Line 1 marks all the roots of exported BDDs to indicate that these nodes and their descendants are all the nodes that we need to keep. The top-down breadth-first marking of descendants is performed by traversing BDD nodes in each node manager (lines 2 to 6). In this algorithm, n denotes the marked BDD node that is being processed and new denotes the next target location for compaction. For each marked BDD node n, its children are marked (line 7). Line 8 establishes the new location new for node n by setting n's *forward* field. Lines 9 and 10 copy the relevant information in n to this new target location new. Line 11 advances new to the next node in the node manager mgr as the new target location. Line 12 advances n to the next marked node in this node-manager. This process repeats until we have processed all the marked nodes in this node manager mgr; after which, all the marked nodes are compacted into memory blocks before *new* and thus all the blocks after *new* are marked as free blocks to be freed after the second phase (line 13).

```
    mark-and-compact()
1     mark all the root nodes of exported BDDs we need to keep.
2     for each variable x from the highest to lowest precedence,
3       mgr ← x's BDD-node manager
4       n ← first marked node in manager mgr
5       new ← first node in manager mgr
6       while n is still in node manager mgr,
7         mark children n.left and n.right
8         n.forward ← new
9         new.left ← n.left
10        new.right ← n.right
11        new ← ManagerNextNode(mgr, new)
12        n ← ManagerNextMarkedNode(mgr, n)
13      put memory blocks for all the nodes after new
          into mgr.freeBlocks.
```

Fig. 7. Garbage Collection's Mark and Compact Phase. This phase marks nodes that we want to keep and at the same time compact the memory to avoid memory fragmentation.

Fig.8 shows the second phase of the garbage collection algorithm. Initially, all external references are updated (lines 2 and 3). Then it proceeds in a top-down breadth-first manner to fix each BDD node's children references (lines 7 and 8) and reinsert this node back into the unique table (line 9). After all the references of a BDD-node manager are updated, its associated free blocks are freed (line 10).

For the purpose of explanation, the garbage collection algorithm shown uses an additional field *forward* for each BDD node. In the actual implementation, each BDD node's *hashNext* field, used for chained hashing, is also used as the *forward* field during the garbage collection. This dual use of the same field is only correct if hash insertion of a node does not occur until

```
    fix-and-rehash()
1     clear all unique tables
2     for each root node r of exported BDDs
3       update root nodes of exported BDDs to the forwarded location
4     for each variable x from the highest to lowest precedence,
5       mgr ← x's Bdd-node manager
6       for each node n in manager mgr
7         n.left ← n.left.forward
8         n.right ← n.right.forward
9         insert n into variable x's unique table
10      free all memory blocks in mgr.freeBlocks.
```

Fig. 8. Garbage Collection's Fix and Rehash Phase. This phase updates all the children references and reinserts the BDD nodes into unique tables.

after all the references to this node are fixed. This condition is guaranteed by first fixing external references (lines 2 and 3 in Fig.8) and then performing the top-down breadth-first traversal, which updates all the parents' references before inserting a node into the hash table. Thus, this two phase breadth-first garbage collection algorithm is able to perform memory compaction without requiring any additional memory.

D. Variable Reordering

Dynamic variable reordering is an important part of BDD construction. Even though we have not yet implemented dynamic variable reordering, the following is an outline of potential problems and their solutions.

1. Some variable reordering algorithms require reference counts. Since garbage collection is generally invoked right before variable reordering, we can compute reference counts during the mark-and-compact phase of garbage collection (line 1 and line 7 of Fig.7).

2. Dynamic variable reordering can counteract the clustering effects achieved by the per-variable memory managers [16]. The solutions proposed in [16] should be directly applicable to our approach.

IV. Performance Evaluation

In this section, we present a performance evaluation of our approach. The test cases are the ISCAS85 benchmarks [3], a collection of ten circuits used in industry. The variable ordering we used is generated by *order_dfs* in SIS [20]. To get more test cases, we generate difference size array multiplier circuits based on carry ripple adders [6]. For the rest of this section, we shall refer to this multiplier circuit as MCRA (Multiplier based on Carry Ripple Adders). For n-bit multiplier with two operands $A = \sum_{i=0}^{n-1} 2^i a_i$ and $B = \sum_{i=0}^{n-1} 2^i b_i$, the variable ordering used is $a_{n-1} \prec a_{n-2} \prec \ldots \prec a_0 \prec b_{n-1} \prec b_{n-2} \prec \ldots \prec b_0$. For all the test cases, to minimize memory usage, we freed the intermediate results (those that are neither inputs nor outputs of the circuit) immediately after its the last reference.

In this section, we use two leading BDD packages for comparison. The first package is CAL version 2.0 from UC Berkeley, which implements the breadth-first algorithm described in [18]. The second package is CUDD version 2.1.2 [21] from the University of Colorado at Boulder, which implements the depth-first algorithm for BDD construction. Both are the latest releases as of November, 1997. All packages are compiled with gcc using the optimization flag -O3. In this section, we will refer to our package as PBF.

For both CAL and CUDD, we used all the default settings with the exception of dynamic variable reordering features which we disabled for two reasons. First, we have not implemented dynamic variable reordering yet. Second, turning off the dynamic reordering features removes the performance impact due to different dynamic reordering algorithms. For the CAL package, the results we present are without its superscalarity and pipelining features [18] because of adverse performance impact. These features require decomposing all operations into a single operation type. For the multipliers, such decomposition increases the running time by up to 60% and superscalarity of 10 with automatic pipelining increases the memory usage by 30% with little (< 1%) or no performance improvement. For C2670 and C3540 from ISCAS85 benchmarks, the results are less clear. Thus, for these two circuits, the results using superscalarity of 10 with automatic pipelining will also be included.

A. Evaluation Threshold

In this section, we examine how different evaluation thresholds impact the memory usage and running time of our approach. The system used for this evaluation is an SGI Power Challenge with 1 GBytes of physical memory. This system has 12 processors running IRIX 6.2 with 32-bit address space. Each processor is a 196MHz MIPS R10000. We perform our experiments using one processor under light load conditions where our processes are the only active processes. Timing results reported are measured CPU time.

In this study, the evaluation threshold ranges from 8 KBytes to ∞ where the ∞ case corresponds to the pure breadth-first case. The results from very small cases ($<$ 10 seconds CPU time and $<$ 10 MBytes memory usage) are omitted.

The results in Fig.9 show that in general, the running time varies about 10 to 20%, except for the C2670. For C2670, there is a speedup of 2 for the ∞ case vs. the cases with smaller evaluation thresholds. This is most likely caused by the fact that a larger evaluation threshold results in a more complete cache (as discussed in Section III.A). This is substantiated by the fact that the ∞ case has a total of 23 million Shannon expansions, while the smaller evaluation thresholds cases have over 135 million Shannon expansions.

The results in Fig.9 also show that different evaluation thresholds can have an impact on the memory usage; e.g, for C2670, the ratio between maximum and minimum memory usage is 1.64. In general, this memory usage difference may be the key factor on whether or not an application fits into physical memory and thus can have a significant effect on the running time.

Threshold (KBytes)	CPU Time(seconds) / Memory Usage(MBytes)			
	C2670	C3540	MCRA14	MCRA15
8	277 / 169	254 / 158	979 / 134	3820 / 359
64	264 / 169	252 / 158	968 / 133	3726 / 359
512	268 / 169	241 / 157	884 / 134	3477 / 359
4096	240 / 180	251 / 165	837 / 139	3104 / 365
32768	147 / 213	234 / 198	953 / 175	3343 / 419
∞	102 / 278	229 / 176	964 / 168	3561 / 491

Fig. 9. Effects of Evaluation Threshold. ∞ case corresponds to the case with pure breadth-first.

Note that overall, the evaluation threshold of 4096 KBytes strikes a reasonable balance between memory usage and running time. Since 4906 KBytes is $\frac{1}{256}$ of the physical memory size (1 GBytes), for the rest of the performance evaluation in this paper, we choose the evaluation threshold for our package to be $\frac{1}{256}$ of the physical memory size.

B. Performance Comparison – No Paging

This section compares our approach (PBF) to CAL and CUDD when the test cases fit in physical memory. The system used for evaluation is the same as in the previous section. The memory usage limit is set to 1 GBytes. The evaluation threshold chosen for our package is 4 MBytes which is $\frac{1}{256}$ of physical memory size of 1 GBytes.

Fig. 10 shows the results of this study. The results for smaller cases are shown at the top half of this table. The results for the C6288 and C7552 cases are not available because they both exceeded the memory limit. Note that for CAL, C2670 and C3540 have better performance using CAL's superscalarity and pipelining feature at the cost of 71% to 84% higher memory usage. These results are marked with § in Fig. 10.

The results show that for the larger cases, PBF consistently outperforms both CAL and CUDD, with speedups ranging from 1.10 (MCRA15) to 1.60 (C3540) in comparison to the best of CAL and CUDD. For the smaller cases, PBF is slower. However, since these smaller cases take less than 2 seconds to finish, performance differences among the different approaches are less significant.

As for memory usage, PBF's memory usage tracks very closely with CUDD's depth-first implementation. For small cases ($<$ 10 MBytes), PBF's memory usage is higher due to the memory overhead of per variable data structures. However, for large cases like C3540 and MCRA circuits, PBF's memory usage is actually slightly smaller than CUDD's memory usage. In contrast, CAL's memory usage is up to a factor of 1.6 (MCRA15) in comparison to PBF's memory usage.

C. Performance Comparison – Paging

This section compares our approach (PBF) to CAL and CUDD when the test cases do not fit into physical memory. We

Circuit	CPU Time(seconds)			Memory(MBytes)		
	PBF	CAL	CUDD	PBF	CAL	CUDD
C432	1.08	0.94	1.02	5.5	3.7	2.7
C499	0.25	0.45	0.19	2.9	1.9	0.9
C880	0.25	0.23	0.11	2.5	1.8	1.0
C1355	0.74	0.83	0.57	5.4	3.0	2.0
C1908	0.39	0.66	0.30	3.0	1.9	1.6
C5315	0.90	0.86	0.32	5.5	3.1	2.4
C2670	240	573 292§	795	180	217 372§	148
C3540	251	658 536§	403	165	176 325§	169
C6288	n/a	n/a	n/a	n/a	n/a	n/a
C7552	n/a	n/a	n/a	n/a	n/a	n/a
MCRA14	837	2016	1004	139	207	152
MCRA15	3104	7383	3425	365	646	482

Fig. 10. Performance comparison when the test cases fit in physical memory. Both C6288 and C7552 cases exceeded the 1 GBytes memory limit and thus the results are not available. Numbers marked with § are CAL's results using superscalarity of 10 with automatic pipelining.

repeated the experiments on a smaller system — a 200MHz Pentium Pro with 256 KBytes L2 Cache and 128 MBytes of 60ns EDO DRAM. This system is running Linux 2.0.30 with 32-bit address space. All measurements were obtained under single user mode. Timing results reported are elapsed time and time limit is set to be 24 hours of elapsed time. For this experiment, we chose the test cases which use more memory than available physical memory (128 MBytes).

Fig.11 shows that our approach (PBF) consistently outperforms both CAL and CUDD with speedups ranging from 1.51 (C2670) to 13.2 (MCRA14) in comparison to the best of CAL and CUDD. The significant speedup of MCRA14 is mainly due to the fact that our approach's memory usage for this case is only slightly more than the available physical memory. This case demonstrates the importance of limiting the memory overhead. Another interesting point to note that both the PBF (our approach) and the CAL (breadth-first) approach have much better paging locality than the CUDD (depth-first) approach. For the C3540 circuit, this locality resulted in an order of magnitude difference in performance.

Circuit	Elapsed Time(seconds)			Memory(MBytes)		
	PBF	CAL	CUDD	PBF	CAL	CUDD
C2670	1169	1773	7071	169	217	148
C3540	1058	1925	22629	157	176	169
MCRA14	1173	15506	22135	134	207	152
MCRA15	n/a	n/a	n/a	n/a	n/a	n/a

Fig. 11. Performance comparison when the test cases do not fit into physical memory. MCRA15 case exceeded the time limit of 24 hours for all three packages. CAL's numbers are measured without its superscality nor pipelining features to reduce the memory usage and minimize paging.

V. Array Multipliers

In this section, we demonstrate the effectiveness of our techniques by building very large output BDDs of two types of integer multiplication circuits. The first type is based on C6288 from ISCAS85 benchmark. C6288 is a 16-bit array multiplier using carry save adders. Based on its design, we derived corresponding circuits from 1 to 15 bits. The second type is an array multiplier with carry ripple adder (MCRA) as in Section IV. In this study, we characterize both multipliers from 1 to 16 bits.

The system used for this evaluation is an SGI Power Challenge with 4 GBytes of physical memory. This system has 16 processors running IRIX 6.2 with 64-bit address space. Each processor is a 194MHz MIPS R10000. We perform our experiments under dedicated mode using one processor. Note that for BDD applications, memory usage on 64-bit machines is generally twice that of 32-bit machines.

Fig.12 shows the results for this experiment. Fig.13 plots the memory usage of output BDDs and memory usage for constructing C6288 and MCRA circuits in a semi-log graph. Note that the output BDD sizes grows exponentially at a factor of about 2.87 per bit of word size.

Fig.13 also shows that other than the initial overhead, which affects the memory usage of smaller circuits, the total memory usage grows at the same rate as the output BDDs' memory usage. This plot is a semi-log plot to clearly show the numbers for small cases. However, it is worth noting that even though the total memory usage for the 16-bit multiplier is about a factor of three to four over the size of output BDDs, this semi-log plot deemphasizes this difference.

To better understand the memory usage, we analyze the BDD construction for building the C6288 circuit. The maximum memory usage for building this circuit is 3803 MBytes. The maximum number of BDD nodes that exist simultaneously during the BDD construction process is about 110 million (3352 MBytes). To accommodate these BDD nodes, the unique tables have a combined total of 48 million bins (366 MBytes). Thus the memory overhead of the operator nodes, the compute cache, and other auxiliary data structures is 85 MBytes which is only 2.2% of the total memory usage. This result demonstrates that our approach has very little memory overhead. As far as we know, this is the first time that the entire C6288 circuit has been built using conventional BDD representations.

VI. Related Work

There are many research efforts based on breadth-first BDD construction [14, 15, 1, 10, 18]. However, none of these propose how to bound the memory overhead of the breadth-first construction. To address this issue, we introduced a hybrid algorithm which performs the breadth-first construction to exploit memory locality and switches to the depth-first construction when the memory overhead becomes too high [8]. This hybrid approach has the drawback that when a BDD operation is much larger than the switch-over threshold, this hybrid approach will be dominated by the depth-first portion and thus

# of Bits	Output Size (# of nodes)	CPU Time(seconds)		Memory(MBytes)	
		C6288	MCRA	C6288	MCRA
1	3	0.01	0.01	0.4	0.5
2	14	0.01	0.01	0.7	0.7
3	46	0.04	0.03	2.7	2.0
4	140	0.06	0.08	3.9	5.2
5	404	0.10	0.10	5.9	6.0
6	1156	0.15	0.15	7.4	8.0
7	3256	0.27	0.23	9.3	9.3
8	9258	0.59	0.55	10.4	11.9
9	26,217	2.02	1.72	17.4	15.3
10	74,456	6.96	5.87	26.3	24.8
11	212,088	26.97	19.70	37.1	33.2
12	605,883	108.63	70.57	70.2	54.9
13	1,733,156	403.15	288.47	162.6	134.6
14	4,955,083	1483.66	996.63	438.0	320.5
15	14,181,971	5529.94	3378.62	1277.2	974.7
16	40,563,945	22175.23	12257.76	3803.7	2795.6

Fig. 12. Results for multiplier circuits. Note that since a 64-bit machine is used for this study, the memory usage is roughly twice as big as results on a 32-bit machine.

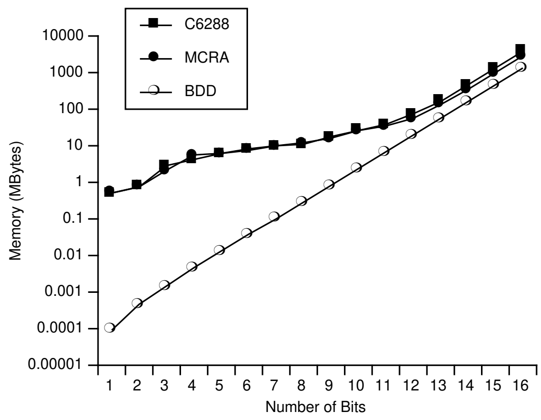


Fig. 13. Maximum memory usage for both C6288 and MCRA compared with memory usage of output BDDs (labeled as BDD).

have poor memory behavior. Note that this hybrid is similar to the mixed depth-first and breadth-first approach that prunes unnecessary recursion branches for the quantification and relational product operations [18].

SMV [13]'s BDD package uses mark-and-sweep garbage collector without memory compaction. In [15, 1, 17], memory compaction is used to avoid memory fragmentation. These three approaches are all based on reference counting. In [15], the compaction algorithm is *stable* (i.e., linear ordering of the nodes is maintained) and does not require additional memory. Our approach is quite similar to this. In [1], the garbage collection uses a free-list and when memory fragmentation becomes high, a separate memory compaction algorithm based on copying is used. In [17], garbage collection phase is also free-list based and memory compaction is performed after garbage collection only when memory fragmentation becomes high. This compaction is performed by moving the newest set of live nodes to fill the holes left behind by the oldest set of dead nodes; thus, no additional memory is required. This algorithm has the advantage of moving minimum number of nodes necessary but it does not maintain the linear ordering of the live nodes. The performance impact of this tradeoff deserves further study. Our approach combines many attributes of the approaches above by integrating a mark-and-sweep garbage collector with a stable memory compaction without any additional memory overhead.

VII. Summary and Conclusions

This paper has introduced three techniques to control the working set size by limiting memory overhead and improving both temporal and spatial locality. First, we have introduced a novel BDD construction algorithm based on partial breadth-first expansion. This approach has the good memory locality of the breadth-first BDD construction while maintaining the low memory overhead of the depth-first approach. Second, we have described how memory management on a per-variable basis can improve spatial locality of BDD construction at all levels, including expansion, reduction, and rehashing. Finally, we have introduced a memory compacting garbage collection algorithm to avoid memory fragmentation due to unreachable BDD nodes. These algorithms work together in controlling the working set size to gain better memory access locality with little memory overhead. As these techniques exploit inherent properties of BDD construction, graph reduction techniques (like *BMD, POBDD, and variable reordering) can be incorporated into our algorithms to further expand the usefulness of these algorithms.

Experimental results show that by controlling the evaluation threshold, the partial-breadth approach can reduce the memory usage by 60% in comparison to our pure breadth-first case (∞ evaluation threshold). In the performance comparison study, the results show that when the applications fit in physical memory, our approach is consistently faster for larger cases ($>$ 2 seconds) with speedups of up to 1.6 in comparison to the leading depth-first (CUDD) and breadth-first (CAL) packages. When the applications do not fit into physical memory, our approach outperforms both CUDD and CAL by up to an order of magnitude. Furthermore, to demonstrate how our techniques can efficiently build very large graphs, we constructed the output BDDs for the C6288 multiplication circuit from the ISCAS85 benchmark and showed that the memory overhead of our approach is 2.2%. These results show that our techniques have successfully achieved better memory locality while reducing the memory overhead.

Beyond the sequential world, another advantage of the partial breadth-first algorithm is that it can be parallelized by using each processor's context stack as a distributed work queue [22]. This approach achieves speedups of up to four on eight processors of a shared memory system.

Acknowledgement

We thank Claudson F. Bornstein and Henry R. Rowley for numerous discussions on efficient BDD implementations. We also thank Rajeev K. Ranjan for his help in setting up our performance study with CAL package. This work utilized Silicon Graphics Power Challenge shared memory machines on both the Pittsburgh Supercomputing Center and the National Center for Supercomputing Applications at Urbana-Champaign. We are very grateful to the wonderful support staff in both supercomputing centers.

References

[1] R. 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, November 1994.

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

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

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

[5] R. E. Bryant. On the complexity of VLSI implementations and graph representations of Boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, 40(2):205–213, Feburary 1991.

[6] R. E. Bryant and Y.-A. Chen. Verification of arithmetic circuits with binary moment diagrams. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference*, pages 535–541, June 1995.

[7] Y.-A. Chen and R. E. Bryant. ACV: An arithmetic circuit verifier. In *Proceedings of the International Conference on Computer-Aided Design*, pages 361–365, November 1996.

[8] Y.-A. Chen, B. Yang, and R. E. Bryant. Breadth-first with depth-first BDD construction: A hybrid approach. Technical Report CMU-CS-97-120, School of Computer Science, Carnegie Mellon University, 1997.

[9] R. Drechsler, A. Sarabi, M. Theobald, B. Becker, and M. A. Perkowski. Efficient representation and manipulation of switching functions based on ordered kronecker functional decision diagrams. In *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pages 415–419, June 1994.

[10] A. Hett, R. Frechsler, and B. Becker. MORE: Alternative implementation of BDD-packages by multi-operand synthesis. In *Proceedings of the European Design Automation Conference*, pages 16–20, September 1996.

[11] J. Jain, J. Bitner, J. A. Abraham, and D. S. Fussell. Functional partitioning for verification and related problems. In *Proceedings of the Brown/MIT VLSI Conference*, pages 210–226, March 1992.

[12] S. Jha, Y. Lu, M. Minea, and E. M. Clarke. Equivalence checking using abstract BDDs. In *1997 IEEE Proceedings of the International Conference on Computer Design*, pages 332–337, October 1997.

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

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

[15] 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, November 1993.

[16] R. K. Ranjan, W. Gosti, R. K. Brayton, and A. Sangiovanni-Vincentelli. Dynamic reordering in a breadth-first manipulation based BDD package: Challenges and solutions. In *1997 IEEE Proceedings of the International Conference on Computer Design*, pages 344–357, October 1997.

[17] R. K. Ranjan and J. Sanghavi. CAL-2.0: Breadth-first Manipulation Based BDD Library. Public software. University of California, Berkeley, CA, June 1997. http://www-cad.eecs.berkeley.edu/Research/cal_bdd/.

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

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

[20] 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. Technical Report UCB/ERL M92/41, Electronics Research Lab, University of California, May 1992.

[21] F. Somenzi. *CUDD-2.1.2: CU Decision Diagram Package*, April 1997. ftp://vlsi.colorado.edu/pub/cudd-2.1.2.tar.gz.

[22] B. Yang and D. R. O'Hallaron. Parallel breadth-first BDD construction. In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 145–156, June 1997.