Parallel Breadth-First BDD Construction

Bwolen Yang and David R. O'Hallaron
Carnegie Mellon University
Pittsburgh, PA 15213
{bwolen,droh}@cs.cmu.edu

Abstract

With the increasing complexity of protocol and circuit designs, formal verification has become an important research area and binary decision diagrams (BDDs) have been shown to be a powerful tool in formal verification. This paper presents a parallel algorithm for BDD construction targeted at shared memory multiprocessors and distributed shared memory systems. This algorithm focuses on improving memory access locality through specialized memory managers and partial breadth-first expansion, and on improving processor utilization through dynamic load balancing. The results on a shared memory system show speedups of over two on four processors and speedups of up to four on eight processors. The measured results clearly identify the main source of bottlenecks and point out some interesting directions for further improvements.

1 Introduction

With the increasing complexity of protocol and circuit designs, formal verification has become an important research area. As an example, in 1994, Intel's famous Pentium division bug (which caused a pretax charge of $475 million to Intel's revenue) clearly demonstrated the demands for more powerful verification tools. Binary decision diagrams (BDDs) have been shown to be a powerful tool in formal verification [5]. BDDs have proven to be so useful because they provide a unique and compact symbolic representation of a Boolean function. Compactness is important because it allows us to represent large functions. Uniqueness is important because it allows us to easily test two Boolean functions for equivalence. For example, both the specification and the implementation of a circuit can be converted to BDDs and then compared to determine if they actually compute the same function. Because of the uniqueness property, comparing two BDDs for equivalence reduces to simply comparing a pair of pointers. Another use of BDDs in verification is in providing counterexamples when an implementation fails to match the specification. Counterexamples are important in verification as they can give insights about the location of the faults. Using BDDs, counterexamples can be obtained by XOR-ing the BDD representations of the implementation and the specification.

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 on reducing the size of the graph with techniques like new compact representations for specific classes of functions (KFDD [9] and K*BMD [8]), divide-and-conquer (POBDD [12] and ACV [6]), function abstraction (aBdd [13]), and variable ordering [21]. Despite these efforts, large graphs can still naturally arise for more irregular functions or for incorrect implementation of specification. Incorrect implementation can break the structure of a function and thus can greatly increase the graph size. For example, even though K*BMD representation for integer multiplication is linear, a mistake in the implementation of integer multiplication logic can cause an exponential explosion of the resulting graph. Thus, the ability to handle large graphs efficiently can enable us to represent more irregular functions and to provide counterexamples for incorrect implementation.

Parallelization offers a way to complement graph reduction research efforts by enabling verification of larger problem sizes. With parallelization, BDD construction not only can benefit from more computation power, but more importantly, it can also benefit from pooling the memory of multiple machines together. There is a great deal of interest in parallel BDD construction algorithms [14, 18, 10, 23]. However, it is nontrivial to parallelize BDD construction efficiently, primarily because the construction process involves numerous memory references to small data structures with little computational work to amortize the cost of each reference. Furthermore, BDD construction has irregular control flows and memory access patterns, similar to the n-body problem. However, for the n-body problem, the locality of reference and efficient parallelization can be derived based on the physical locations of the particles and thus can be effi-

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.

ciently parallelized. In contrast, the memory access pattern and the control flow of building a function's BDD is completely determined by interactions between the function's inherent structure and the graph representation used. These interactions are much more difficult to understand and exploit.

This paper explores a new parallel algorithm for BDD construction for shared memory multiprocessors and distributed shared memory (DSM) systems. The algorithm uses a novel form of *partial breadth-first expansion* that improves locality of reference by controlling the working set size and thus reducing overhead due to page faults, communication, and synchronization. The algorithm also incorporates dynamic load balancing to deal with the fact that the processing required for a BDD operation can range from constant to quadratic in the size of the BDD operands, and it is impossible to predict before runtime.

The implementation of the partial breadth-first algorithm is based on extending a hybrid implementation introduced in [7], which has comparable or better performance than other sequential BDD packages. Results on a shared memory system show speedups of over two on four processors and speedups of up to four on eight processors. These results also point out some directions for future improvements.

The rest of this paper is as follows: Section 2 gives an brief overview of BDDs and how they are constructed. Section 3 describes the new parallel partial breadth-first algorithm, and Section 4 describes the measured results on a shared memory system. Section 5 describes related work for both sequential and parallel BDD construction. Finally, Section 6 offers some concluding remarks and directions for future work.

2 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. Figure 1(a) shows the truth table for a Boolean expression and Figure 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.

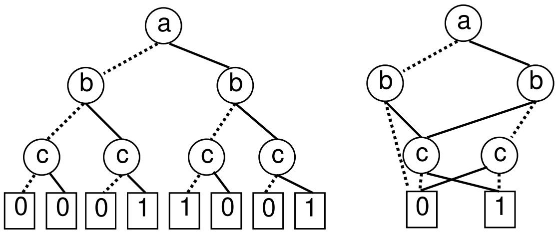
A binary decision diagram (BDD) is a directed acyclic graph (DAG) representation of a binary decision tree where equivalent Boolean subexpressions are uniquely represented by a BDD. Figure 1(c) shows the BDD representation of the binary decision tree in Figure 1(b). Due to the uniqueness of its representation,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

One criteria 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 that are on a path of any BDD, if $x > y$ based on the variable ordering, then x must appear before y on this path. Note that 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.

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

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)

Figure 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.

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 function f with respect to Boolean variable x, where $f_{x=0}$ is equal to f with the variable x restricted to 0, and $f_{x=1}$ is equal to f with x restricted 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. Similar to nodes in the binary decision tree, each BDD node consists of a variable and references to its two children: the 0-branch corresponds to the case where the variable is assigned 0, and the 1-branch corresponds to the case where the variable is assigned 1. Given a BDD b, the function f represented by b is recursively defined by

$$f = \overline{x} \cdot f_{x=0} + x \cdot f_{x=1} \qquad (1)$$

where x is the variable in 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.

2.1 Basis for BDD Construction

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

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

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. The BDDs for these cofactors can be easily obtained by the following: if the top variable is the root of a graph, then the BDD representations for its cofactor functions are simply the children of that root node. Otherwise, by definition, the top variable does not appear in the graph; thus the BDD representation for both cofactor functions is just the graph itself.

In the top-down *expansion phase*, this Shannon expansion process repeats recursively following the given variable order 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. The recursive process will terminate because restricting all the variables of a function produces a constant function and all binary Boolean operations involving constant operand(s) can be trivially evaluated. 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 *reduction phase* is necessary to reduce expressions like $(\overline{x} \cdot h + x \cdot h)$ to h. This bottom-up reduction phase is performed in the reverse order of the expansion phase.

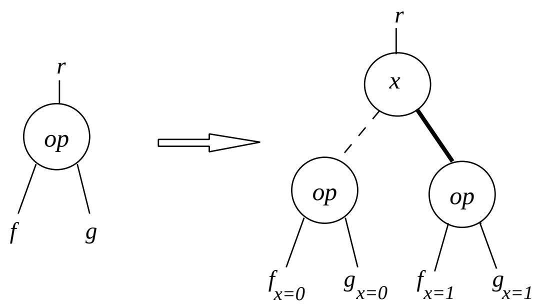


Figure 2: Shannon Expansion: The dashed edge represent the 0-branch of a variable and the thick solid edge represents the 1-branch

Figure 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. In this figure, the dashed edge is the 0-branch and the thick solid edge is the 1-branch. 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; similarly, the breadth-first construction expands the operator nodes with the smallest depth.

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.

2.2 BDD Construction

A typical depth-first BDD algorithm is shown in Figure 3. This algorithm takes an operator (op) and its two BDD operands (f and g) as inputs and returns a BDD for the result of the operation $f\ op\ g$. The result BDD is constructed by recursively performing Shannon expansion in the depth-first manner. This recursive expansion ends when the new operation created is a terminal case (line 1) or when it is found in the *computed cache* (line 2 and 3). A computed cache stores previously computed results to avoid repeating work that was done before. Line 4 determines the top variable of f and g. Line 5 and 6 recursively perform Shannon expansion on the cofactors. At the end of this recursive expansion, the reduction step (line 7) ensures that the BDD result is a reduced BDD node. Then the uniqueness of the resulting BDD node is checked against a *unique table* which records all existing BDD nodes (lines 8 to 12). Finally, the operation with its result is inserted into the computed cache (line 13 and 14) and the BDD result is returned (line 15). Typically, both the computed cache and the unique table are implemented with hash tables. The computed cache replacement policy varies from direct map to LRU. 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.

```
    df_op(op, f, g)
1     if terminal case, return simplified result
2     if the operation (op, f, g) is in computed cache,
3        return result found in cache
4     τ ← top variable of f and g
5     res0 ← df_op(op, f_τ=0, g_τ=0)
6     res1 ← df_op(op, f_τ=1, g_τ=1)
7     if (res0 == res1), return res0
8     b ← BDD node (τ, res0, res1)
9     result ← lookup(unique table, b)
10    if BDD node b does not exist in the unique table,
11       insert b into the unique table
12       result ← b
13    insert this operation (op, f, g) and its result result
14       into the computed cache
15    return result
```

Figure 3: Depth-First BDD Algorithm

For the breadth-first BDD algorithm, the Shannon expansion for a top level operation will be performed top-down from the highest to the lowest variable order. Thus, all operations with the same variable order will be expanded at the same time. The reduction phase is performed bottom-up from the lowest to the highest variable order where all operations of the same variable order will be reduced at the same time.

2.3 Memory Overhead and Access Locality

BDD construction can be very memory intensive, especially when large graphs are involved. It not only requires a lot of memory, it also requires frequent accesses to a lot of small data structures (the node size is typically 16 bytes on 32-bit machines). Conventional BDD construction algorithms are based on depth-first traversal of the BDDs [3]. This approach has poor memory behavior because of irregular control flows and memory access pattern. 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 by 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 [16, 17, 2, 11, 20, 7]. 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 a very small constant, 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 after the reduction phase. 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 the physical memory while the breadth-first construction does not, the performance of the breadth-first construction can degrade significantly due to page faults.

To limit memory overhead, [7] introduces a hybrid approach which performs breadth-first expansion until a fixed threshold is reached. This threshold is set based on the amount of available physical memory. After reaching the threshold, the algorithm switches to the depth-first construction to limit memory overhead. This work also introduces a new cache called a *compute cache* which caches both the *uncomputed operations* (those that are still awaiting results during the breadth-first expansion phase) and computed operations. Previous breadth-first algorithms do not cache the computed operations and also generally maintain a complete cache of the uncomputed operations. The compute cache combines the depth-first computed cache with the breadth-first cache of uncomputed operations. To bound the memory overhead, this hybrid algorithm does not maintain a complete cache of either the computed or the uncomputed operations. The results show that this approach consistently has comparable or better performance than other leading sequential depth-first and breadth-first BDD packages (CUDD version 1.1.1 from University of Colorado at Boulder and CAL version 1.1 from U.C. Berkeley) while maintaining low memory overhead. This hybrid package forms the basis of our parallel implementation.

3 Parallel BDD Algorithm

In our algorithm, BDD construction is parallelized by distributing operations among processors during the expansion phase. Once the operations are assigned, each processor independently constructs corresponding BDDs using a technique known as *partial breadth-first expansion*, which carefully controls the working set size in order to minimize accesses beyond a processor's own local memory. When a processor becomes idle, work loads are redistributed to keep the load balanced. The target architectures of this algorithm are shared memory multiprocessors [15] and DSM systems [1]. The rest of this section will describe each part of the algorithm in more detail.

3.1 Partial Breadth-First 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. In the sequential world, good memory access locality results in good hardware cache locality and reduced page faults. In the parallel world, memory access locality has the additional significance of minimizing communication and synchronization overhead.

For the pure breadth-first construction (which normally has good memory locality), if the BDD operands do not fit in the main 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. For the hybrid construction, when a BDD operation is much larger than the threshold, this hybrid approach will be dominated by the depth-first portion and thus have poor memory behavior.

To overcome both drawbacks while bounding the memory overhead, we introduce a so-called *partial breadth-first expansion* based on multiple evaluation contexts. 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 small 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 available physical memory, we can bound the number of BDD nodes and compute cache nodes created and accessed.

Figure 4 shows the top level procedure and a helper function for this partial breadth-first construction. For each variable, there is an operator queue and a reduction queue. An *operator 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), as in the depth-first case. If not, it queues this operation (line 18) to the top variable's operator queue 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.

Figure 5 shows the expansion phase. This top-down expansion phase processes operations queued from the variable

```
    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 have operations, then
6         take a group of operations from the top context
7         add each operation to its top variable's operator 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 operator queue
19     insert opNode into the compute cache
20     return opNode
```

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

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 control the size of the current evaluation context and when it exceeds a constant evaluation threshold *evalThreshold*, a new child context is started (lines 9 to 14).

```
    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 operator 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
12                 onto the context stack
13              start a new evaluation context start with variable x
14              return
```

Figure 5: Partial Breadth-First Construction: Expansion Phase

Figure 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 as in the depth-first algorithm. The result of a reduction is stored in the *opNode.result* field of an operator node (Lines 13 and 19).

```
    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
```

Figure 6: Reduction Phase of the Partial Breadth-First BDD Algorithm

To take advantage of structured access in the partial breadth-first approach, we associate a specialized BDD-node manager for each variable as in [20]. 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 concept so that there is one operator-node manager for each variable and these node managers are also used as both the operator and the reduction queues. Thus, during both the expansion and the reduction phase, operator nodes are accessed by walking contiguously within each memory block. 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 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. Note that having one unique table per variable is also an important part of parallelizing the reduction phase which we will describe in Section 3.2.

All these techniques work together to control the working set size and have a significant impact on both the sequential and parallel BDD implementations. A detailed study of the impact on working set size is beyond the scope of this paper and will be described in a separate paper.

3.2 Parallel Expansion and Reduction

For BDD construction, both the expansion and the reduction phases have a large degree of parallelism. However, to efficiently utilize this parallelism, careful memory layout is necessary to reduce synchronization cost. In this algorithm, each process independently maintains its own copy of BDD node managers, operator node managers, and the compute cache. This data layout allows each process to proceed independently of each other during the expansion phase. During the reduction phase, synchronization is necessary to prevent concurrent modification to the BDD unique tables.

In the expansion phase, only operator nodes and their corresponding compute cache entries will be created. By requiring each process to maintain its own operator nodes and compute cache, a process can expand its assigned share of operations without synchronizing or communicating with other processors.

In the reduction phase, new BDD nodes will be created and inserted into the unique tables. To avoid concurrent modification of the unique tables, one semaphore lock is associated with each variable's unique table. Before creating a new BDD node and inserting it into its unique table, a process must acquire the corresponding lock. Since in the breadth-first reduction, all the new BDD nodes of the same variable are produced together, a process can obtain the lock and produce all the BDD nodes for that variable before releasing the lock. In the worst case where the number of processes is close to the number of the variables, this approach will still benefit from pipeline parallelism out of the reduction phase. Note that this pipeline parallelism will only exist if no one stage of pipeline dominates the entire computation. As we will show in the result section, it is our experience that most of the BDD nodes tend to concentrate on very few variables, and thus a pure breadth-first approach would not be able to benefit too much from this pipeline parallelism. Using the partial breadth-first approach, we can control the number of operator nodes expanded per variable to reduce the variance in amount of work among different stages to get better parallelism from the reduction stage.

The main drawback of this data layout is that since the compute cache is not shared, a process will not be able to take advantage of another's compute cache. Thus, the same work might be duplicated in different processes. Another drawback of the per-process data structures is that memory is used less efficiently as free space in blocks allocated by one process is not available to another process. We will present measurements for these overheads in Section 4.

3.3 Work Distribution

As processing required for a BDD operation can range from constant to quadratic in the size of the BDD operands, it is impossible to distribute the load evenly through static allocation. In this parallel BDD algorithm, the load is dynamically balanced based on stealing unexpanded operations from processes' context stacks. In the sequential partial breadth-first construction, the context stack is used to reduce memory overhead and increase locality. For the parallel version, processes' context stacks also double as distributed work queues. When a process is idle, it tries to steal unexpanded operations from busy processes' context stacks. If an idle process fails to find any work, it notifies busy processes to create more sharable work by context switching. Upon successfully stealing work, the thief process produces the results for the stolen operations and return these results to the original owner. During the reduction phase, a process stalls if the results it needed for the reduction have not yet been returned by the thief processes. This stalled process then becomes a thief and tries to steal work from other processes' context stacks.

3.4 Garbage Collection

No BDD package is complete without a good garbage collector. External users of a BDD package can free references to BDDs and since BDD construction is a very memory intensive application, reusing the space of unreferenced BDD nodes is important. Most of the sequential BDD packages uses the reference count and maintains a free list of unreferenced nodes. This approach has several drawbacks, most notably poor memory locality as the free-list approach scatters newly created BDD nodes everywhere in memory. An alternative approach is to use a mark-and-sweep garbage collector with memory compaction. Our preliminary experience shows that on a very large application which uses over three times the physical memory size, the version with a memory compacting garbage collector reduces the total running running to half of the free-list approach. And for small test cases, memory compaction does not introduce much more overhead. Detailed study on these different garbage collection strategies is beyond the scope of this paper.

Our parallel garbage collection algorithm uses mark-and-sweep garbage collection with memory compaction. It is separated into three phases. The mark phase marks the root BDD nodes we have to keep due to external references and then recursively marks all the reachable BDD nodes in the top-down breadth-first manner one variable at a time. In this phase, each process will synchronize at each variable and mark the children of all the mark nodes in its BDD node manager of the corresponding variable. At the same time, it will perform memory compaction for each mark node it processes. Here, synchronization on each variable is necessary as a BDD node's parent(s) can belong to any of the processes and cannot be compacted away without being sure that all its parents are not reachable.

The second phase is to fix references to BDD nodes as these nodes are relocated due to memory compaction. In this phase, each process independently fixes the BDD references for nodes that it owns without any synchronization.

The third phase is rehashing. Since the hash function depends on the location of a BDD node's children, memory compaction will require all the BDD nodes to be rehashed. In this phase, each process rehashes its own BDD nodes one variable at a time. For each variable, a process will need to obtain the lock for the corresponding hash table before inserting its nodes. If it finds the lock is currently held by others, it will try to rehash BDD nodes for other variables first.

Note that there are a number of drawbacks in this approach. First of all, this algorithm requires a process to be responsible to work on nodes that it owns during all three phases. Since we currently do not guarantee even distribution of the BDD nodes, the load for all three phases of garbage collection can be very imbalanced. The second drawback is that as BDD nodes tend to cluster in very few variables, the final rehashing phase may be sequentialized as the work of rehashing nodes for these few variables dominates this phase. We will discuss this in more detail in the results section.

4 Results

The performance results for our algorithm were obtained on a twelve-processor SGI Power Challenge with 1Gb of shared memory. Each processor is a 195 MHz MIPS R10000. On this machine, we have studied results up to 8 processors, which is sufficient to identify the strengths and the weaknesses of our parallel algorithm. The test cases used are from the ISCAS85 benchmarks [4] which contains netlists of ten circuits used in industry. The variable ordering used is generated by *order_dfs* in SIS [22]. Using these variable orderings, only four out of these ten circuits take more than 5 seconds to construct and only two (C2670 and C3540) complete in less than one hour on this machine. One of the very long running circuits is a 16-bit multiplier (C6288). To get more test cases, we generate 13- and 14-bit multiplier circuits based on this circuit. The results presented in this section are obtained from these four circuits (C2670, C2540, mult-13, and mult-14). This section will first show the overall performance and then analyze different costs in more detail.

4.1 Overall Performance

This section shows the overall performance of the parallel implementation in comparison to the sequential case. For the following figures, the sequential numbers are shown in the row labeled "Seq", and when applicable, the sequential numbers are plotted as zero processors.

Figure 7 shows the elapsed time for building BDDs with different number of processors and Figure 8 plots these numbers to show speedups of the parallel implementation over the sequential case. Despite the irregularity of BDD construction, our parallel algorithm is able to achieve speedups of over two on four processors and speedups of up to four on eight processors. In Section 4.2, we further break down the cost of each component to identify the bottlenecks.

	Elapsed Time (seconds)			
# Procs	C2670	C3540	mult-13	mult-14
Seq	208	215	256	935
1	204	220	293	1092
2	120	132	173	633
4	76	81	114	383
8	52	58	96	301

Figure 7: Elapsed Time for building BDDs for each circuit with different number of processors. "Seq" represents the sequential case.

Figure 9 shows the memory usage in MBytes. Figure 10 plots these numbers. From these numbers, we can see that using per-processor data structures increases the total memory usage by up to roughly 100% for the eight processor case. However, these numbers also show that this algorithm will be effective in pooling memory together for a DSM system (e.g., cluster of workstations running Treadmarks [1]) to avoid page faults; e.g., on a DSM with 8 processors (i.e., 8 times the memory), this memory usage would be equivalent to having 4 times the memory on the 1 processor case. It is worth noting that part of the extra memory overhead is due to the fact that the sequential case is detecting the condition for garbage collection more aggressively than the parallel case. Garbage collection for the parallel case requires synchronization and thus currently, we check whether or not to garbage collect only after we complete a set of top level operations we queued. At this point, there is an implicit barrier among all processors and thus it is safe to do garbage collection. In comparison, the sequential case checks the garbage collection condition more aggressively after each reduction phase. Currently, we do not have any results on how this difference contributes to the extra memory overhead.

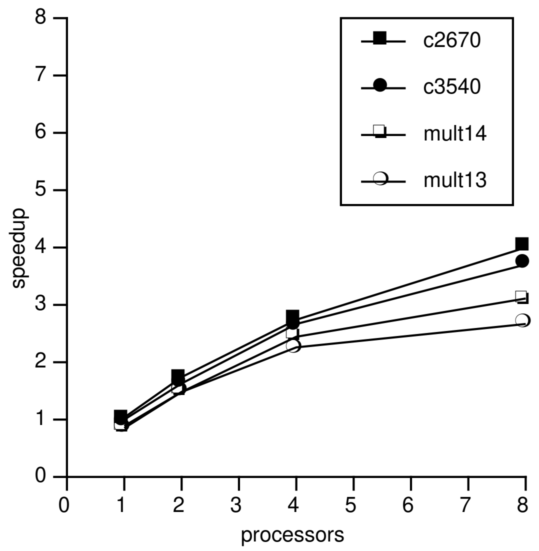


Figure 8: Speedups over the sequential running time.

	Memory (MBytes)			
# Procs	C2670	C3540	mult-13	mult-14
Seq	210	377	91	183
1	210	407	90	243
2	243	428	102	258
4	353	486	123	296
8	449	566	155	342

Figure 9: Memory Usage in MBytes. "Seq" represents the sequential case.

Figure 11 and Figure 12 show the total number operations (i.e., number of the Shannon expansion steps) for different circuits. These results show that despite the fact that the compute cache is not being shared, the total number of operations does not increase much as the number of processors increases.

4.2 Bottlenecks

Given that the total number of operations does not increase dramatically as number of processors increases, where is the source of the inefficiency? To answer this question, we will focus on analyzing the behavior of the mult-14 circuit. Figure 13 shows a breakdown of running time for the mult-14 circuit for three phases: the expansion phase, the reduction phase, and the garbage collection phase. These numbers are measurements of the first processor's work load. Figure 14 plots the speedups over the one processor numbers. There are quite a few interesting points here. First, the expansion phase scales nicely (a speedup of 6 on 8 processors). Both the reduction phase and the garbage collection phase have nice speedups for two processors but scale poorly beyond that point. Another interesting point is that for the one processor case, the expansion phase is the most expensive phase and contributes to over 50% of the running time. Thus, it

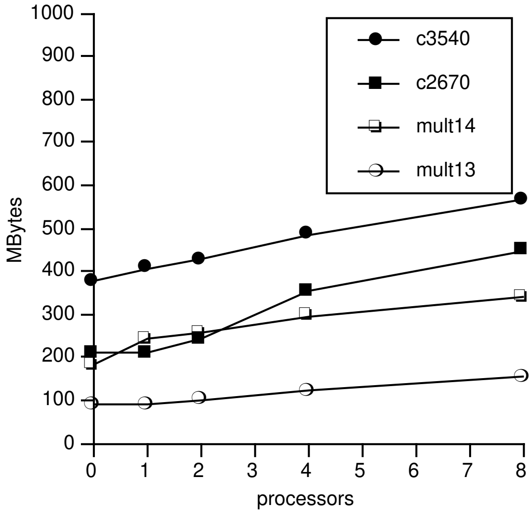


Figure 10: Memory usage in MBytes plotted against number of processors. The 0 processor number is the data from the sequential case.

	Total # of Operations (Millions)			
# Procs	C2670	C3540	mult-13	mult-14
Seq	92.5	68.1	72.8	245
1	92.5	68.1	83.5	296
2	98.4	68.8	84.2	294
4	110.1	71.6	86.7	297
8	125.1	76.2	87.8	305

Figure 11: Total Number of Operations in Millions. "Seq" represents the sequential case.

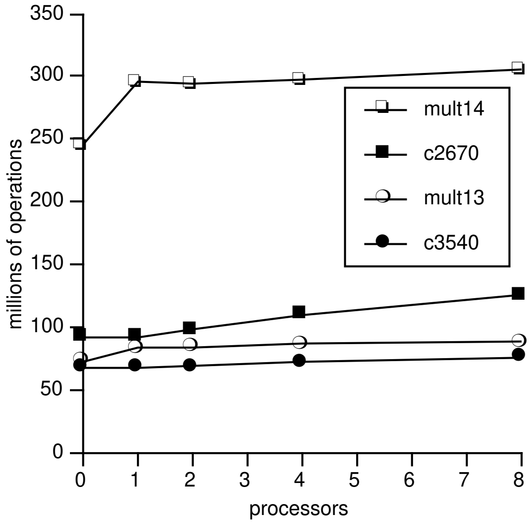


Figure 12: Total Number of Operations in Millions. The sequential number is plotted as the zero processor case.

is good that this phase is scaling very nicely. On the other hand, for the one processor case, the reduction phase constitutes about 40% of total running time. Thus, poor scaling of the reduction phase is the major limiting factor in overall performance. Finally, the garbage collection has the worst speedups overall. However, it only contributes to around 10% of total running time for the one processor case. For the rest of this section, we will first focus on analyzing the bottlenecks in the reduction phase and then briefly discuss the results for garbage collection.

	Time (seconds) breakdown		
# Procs	Expansion	Reduction	GC
1	595.0	419.7	77.1
2	324.0	253.6	47.2
4	166.4	166.6	33.0
8	97.4	147.2	30.5

Figure 13: Elapsed Time of mult-14 circuit for expansion, reduction, and garbage collection phases on the first processor.

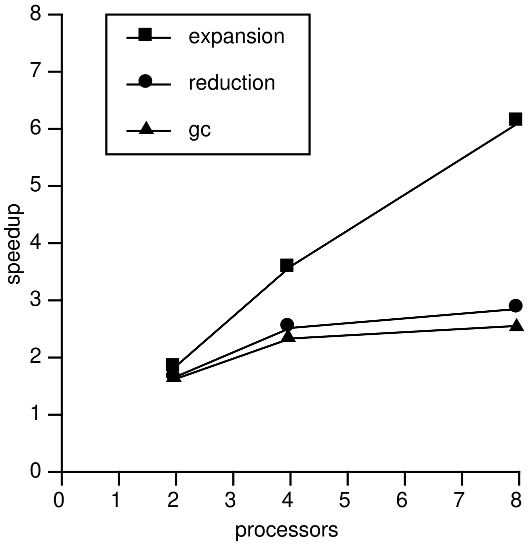


Figure 14: Speedups of mult-14 circuit for expansion, reduction, and garbage collection phases over the one processor result.

Since the reduction phase creates new BDD nodes, we can get some insight into potential for bottlenecks in the reduction phase by considering the maximum number of BDD nodes in each variable's unique table during the reduction phase. These numbers obtained from the mult-14 circuit running on one processor are plotted in Figure 15. This figure shows that the majority of the BDD nodes concentrate on very few variables, namely, variable 6 to variable 8. Thus, in the reduction phase where each processor needs exclusive access to the corresponding unique table to produce unique BDD nodes, there can be a huge number of lock contentions on these three variables.

Figure 16 plots the total time spent waiting to acquire each variable's lock during the reduction phase for different number of processors. This figure shows that there is a large

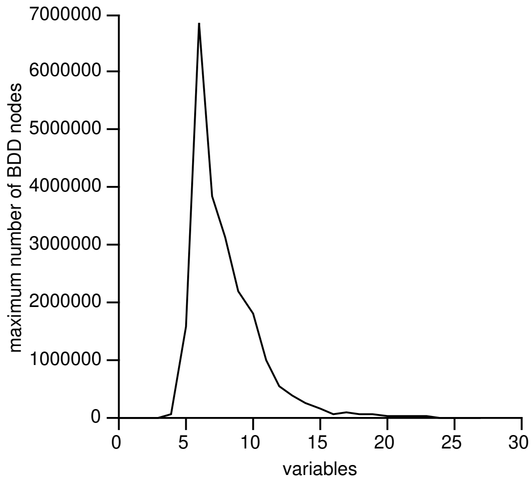


Figure 15: Maximum Number of BDD nodes for each variable. These numbers are obtain from mult-14 for one processor run.

number of contentions for acquiring the locks for these three variables, especially for the eight processor case. These contentions cause the reduction pipeline to become less effective as the number of processors increases. Figure 17 plots this lock acquiring overhead as a ratio over the total cost of the reduction phase. For the eight processor case, the lock acquiring time is 50% of the total reduction phase; i.e., over 20% of the total running time!

Thus to improve the efficiency of parallelism, we need to have a better distributed hashing methodology with finer-grain locking and without incurring too much synchronization overhead.

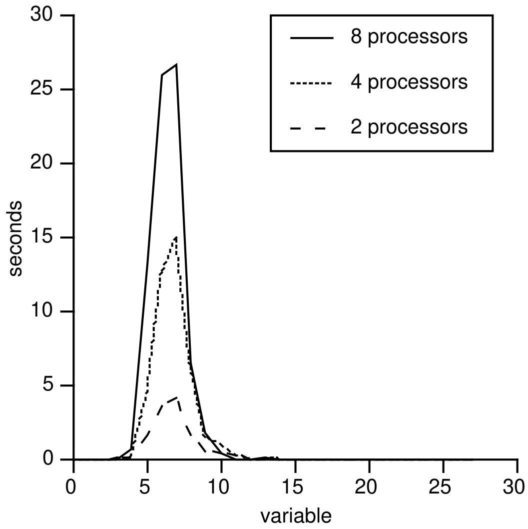


Figure 16: Total lock acquiring time on each variable for the mult-14 circuit.

To better understand the cost of garbage collection, we present measurements for each of its three phases: the mark phase (mark) where reachable nodes are marked, the fixing phase (fix) where the references of BDD children are updated, and the rehashing phase (rehash) where all the nodes are rehashed. Figure 18 shows this breakdown for the first processor on the mult-14 circuit. Figure 19 shows the speedups of each of these three phases based on the one processor results. For the two processor case, all three phases have speedups of over 1.5. However, beyond two processors, all these phases scale poorly. Currently, we do not have concrete evidence to explain these numbers. A possible explanation for the poor scaling on both the mark and the fix phase is poor load balance. This is especially likely for the fix phase which is performed completely in parallel without any synchronization, and thus, if the load is perfectly balanced, it should have linear speedups. As for the rehashing phase, we suspect that the problem is the same as the reduction phase; i.e., since the BDD nodes are all concentrated on a very small number of variables, rehashing time is going to concentrate on those three variables. Therefore, there is not enough parallelism to scale. If this is the case, solutions to the reduction phase should also solve the scaling problem of the rehashing phase in garbage collection.

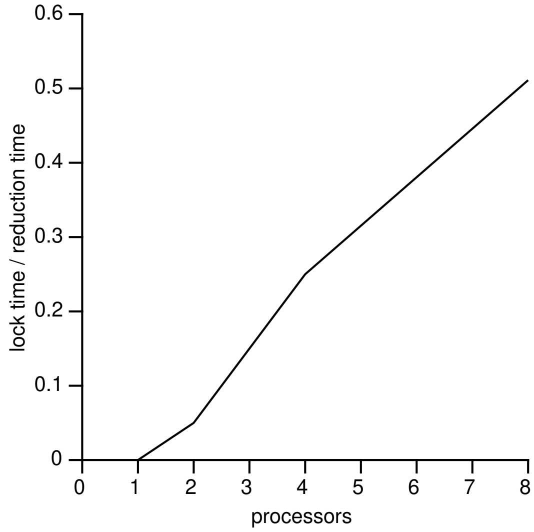


Figure 17: Ratio of lock acquiring time over the total time of the reduction phase for the mult-14 circuit.

	Time (seconds) breakdown		
# Procs	Mark	Fix	Rehash
1	34.3	19.3	23.5
2	20.6	12.6	13.9
4	13.6	9.0	10.5
8	11.2	8.0	11.3

Figure 18: Elapsed Time of garbage collection phase of the mult-14 circuit for mark, fix, and rehash phases on the first processor.

Finally, other circuits all exhibit similar clustering behavior. This provides a strong evidence that the reduction phase is the major source of the bottlenecks for parallelizing BDD algorithms efficiently.

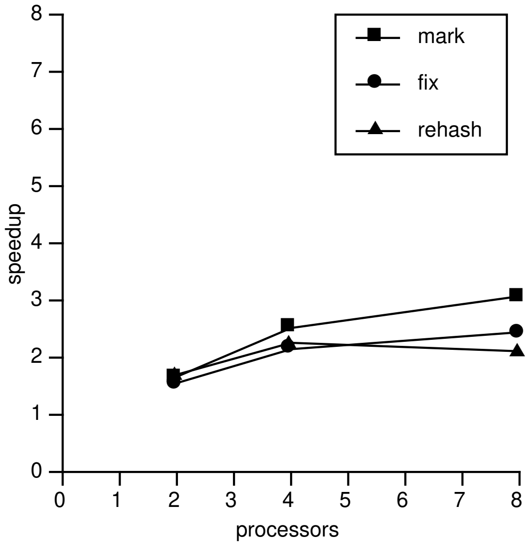


Figure 19: Speedups of three phases of garbage collection over the one processor run for the first processor on mult-14 circuit.

5 Related Work

There are many different parallel BDD implementations on many different architectures. In this section we will briefly describe each implementation. We will also briefly note related advances in sequential BDD techniques to place each parallel implementation in its proper context.

In 1986, [5] first proposes the use BDDs to manipulate Boolean functions. In 1990, [3] describes some techniques which can lead to an order of magnitude improvement in execution time and memory usage. This is the basis of most of the depth-first BDD packages today.

In 1990, [14] explores parallelism in operation sequences on shared memory systems. Their experimental results on a 10-bit multiplier show a speedup of 10 on a 16-processors Encore-Multimax shared memory machine. The BDDs are constructed through building and minimizing finite automata. This work does not consider dynamic load balancing.

In 1991, [16] describes a breadth-first algorithm for BDD construction for vector machines. Later in 1993, [17] shows how to use the breadth-first approach to construct very large BDDs. In 1994, [2] shows an efficient implementation of the breadth-first approach by building a block index table of size proportional to the address space.

In 1994, [18] parallelizes Bryant's original 1986 BDD algorithm on the CM5 using DSM. The data is arbitrarily distributed by the DSM system. The computation is distributed using a distributed stack. During the expansion phase, new operations are pushed onto the distributed stack. Each processor obtains work by getting operations from the distributed stack. The results on 32 processors show speedups of 20-32 on the cases where the problem size fits in memory and superlinear speedups when the problem size does not fit in memory. This approach does not consider memory access locality and will require a good DSM system in order to perform well.

In 1995, [10] introduces a data parallel BDD package for massively parallel SIMD machine. On a 16K-node MasPar, they reported a speedup of around 10.

In 1996, [23] parallelizes depth-first BDD construction on a network of workstations with a specialized network and a communication co-processor on each workstation. The computational model is "owner-computes" with BDD nodes evenly distributed. This method has the advantage that load is perfectly balanced. This work shows that pooling together the memory of several workstations can be a main factor in achieving speedup. In the cases where the problem does not fit into the memory of one machine, a superlinear speedup is observed.

In 1996, [20] describes a breadth-first BDD algorithm and introduces the concept of issuing superscalarity and pipelining in expanding multiple top level operations. Later in the same year, [19] extends this work and describes a sequential algorithm with the speedup obtained by pooling memory of a network of workstations. In their approach, each workstation owns a disjoint set of consecutive variables and each BDD node is assigned to the owner the corresponding variable. BDD construction is performed in breadth-first manner following the owner-compute rule. Thus, a workstation will process work for all the variables it owns and then pass it on to the workstation which owns the next set of variables. This approach can be extended to obtain pipelined parallelism for both the expansion and the reduction phase. However, as BDD nodes tend to cluster on a very small number of variables (as our experiments have shown in Section 4), this approach is very ineffective as both the expansion and the reduction phase will be stalled by processing operations on these handful of variables.

In 1997, [7] describes a hybrid BDD algorithm which also incorporates the computed cache and depth-first approach into the breath-first algorithm to bound memory overhead. Its performance is generally better than other leading BDD packages. This package forms the basis of our parallelized partial breadth-first BDD package.

6 Conclusions and Future Work

We have presented a parallel algorithm for BDD construction. This algorithm achieves good memory access locality by using specialized memory managers and exploiting the novel idea of partial breadth-first expansion. We have also presented a way of dynamically load balancing BDD construction which resulted in a very good scaling behavior of the expansion phase.

The algorithm is based on one of the fastest sequential breadth-first BDD packages yet developed. Results obtained on a shared memory multiprocessor show speedups of over two on four processors and speedups of up to four on eight processors. We have also shown that this parallel algorithm will be effective in pooling the memory of a DSM system together to avoid page faults.

From the measurements, we have identified the reduction phase as the main source of inefficiency. The cause of this inefficiency is the heavy clustering of BDD nodes in a very small number of variables. Thus, the reduction pipeline is stalled due to lock contentions to create new BDD nodes for these variables. This synchronization cost is about 20% of total running time for the 8 processor case. Thus, in order to solve the scaling problem for BDD construction, a better distributed hashing algorithm is necessary to reduce this synchronization cost.

As for future work, we plan to look for better distributed hashing algorithms to improve the scalability of this application. Since BDD construction is very memory intensive, we would also like to study the effects of bus contentions on shared memory systems. Also, we plan to port our parallel implementation to a DSM system and study the effects of pooling memory together to avoid page faults.

Acknowledgement

We thank Yirng-An Chen for numerous discussions on efficient sequential BDD implementation and for being a valuable information source in sequential BDD research efforts. We also thank Claudson F. Bornstein and Henry R. Rowley for discussions and suggestions on both efficient sequential and parallel BDD implementations. 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] Amza, C., Cox, A., Dwarkadas, S., Hyams, C., Li, Z., and Zwaenepoel, W. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer 29*, 2 (Feb 1996), 18–28.

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

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

[4] Brglez, F., and Fujiwara, H. A neutral netlist of 10 combinational benchmark circuits and a target translator in Fortran. In *1985 International Symposium on Circuits And Systems* (1985).

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

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

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

[8] Drechsler, R., Becker, B., and Ruppertz, S. K*BMDs: a new data structure for verification. In *Proceedings of European Design and Test Conference* (March 1996), pp. 2–8.

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

[10] Gai, S., Rebaudengo, M., and Reorda, M. S. A data parallel algorithm for Boolean function manipulation. In *Proceedings of Fifth Symposium on Frontiers of Massively Parallel Computation* (February 1995), pp. 28–34.

[11] Hett, A., Frechsler, R., and Becker, B. MORE: Alternative Implementation of BDD-Packages by Multi-Operand Synthesis. In *Proceedings of the European Design Automation Conference* (1996).

[12] Jain, J., Narayan, A., Coelho, C., Khatri, S., Sangiovanni-Vincentelli, A., and R.K. Brayton, M. F. Decomposition techniques for efficient ROBDD construction. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996), pp. 419–434.

[13] Jha, S., Lu, Y., Minea, M., and Clarke, E. M. Equivalence checking using abstract BDDs. Submitted to 1997 IEEE International Conference on Computer Design, 1997.

[14] Kimura, S., and Clarke, E. M. A parallel algorithm for constructing binary decision diagrams. In *1990 IEEE Proceedings of the International Conference on Computer Design* (Sept 1990), pp. 220–223.

[15] Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., and Lam, M. The Stanford Dash multiprocessor. *IEEE Computer 25*, 3 (Mar. 1992), 63–79.

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

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

[18] Parasuram, Y., Stabler, E., and Chin, S.-K. Parallel implementation of BDD algorithms using a distributed shared memory. In *Proceedings of 27th Hawaii International Conference on Systems Sciences* (January 1994), pp. 16–25.

[19] Ranjan, R. K., Sanghavi, J. V., Brayton, R. K., and Sangiovanni-Vincentelli, A. Decision diagrams on network of workstations. In *1996 IEEE Proceedings of the International Conference on Computer Design* (October 1996).

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

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

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

[23] Sotrnetta, T., and Brewer, F. Implementation of an efficient parallel BDD package. In *Proceedings of the 33rd ACM/IEEE Design Automation Conference* (June 1996), pp. 641–644.

[24] Subhlok, J., and Yang, B. A new model for integrated nested task and data parallel programming. In *Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June 1997). To Appear.