Breadth-First with Depth-First BDD Construction: A Hybrid Approach

Yirng-An Chen Bwolen Yang Randal E. Bryant

March 1997
CMU-CS-97-120

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper presents the technique of operator sifting as a new way of understanding both breadth-first and depth-first approaches to BDD construction. A new algorithm is also proposed to capture the breadth-first approach's advantage of memory access locality, while keeping the depth-first approach's advantage of low memory overhead. Our preliminary experimental results show that our approach is generally faster than other implementations that rely exclusively on either breadth-first or depth-first approaches while keeping memory overhead comparable to that of depth-first approaches.

Yirng-An Chen and Randal E. Bryant are sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Bwolen Yang is supported 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 (NSF) under Grant CMS-9318163, and in part by a grant from the Intel Corporation.

Keywords: Breadth-First, Depth-First, Hybrid Approach, Binary Decision Diagram, Formal Verification

1 Introduction

Binary Decision Diagrams (BDDs) have been proven successful in representing and manipulating Boolean functions symbolically [5] in a variety of application domains. This success has led to several efforts [3, 13, 14, 1, 11, 10, 15] to provide efficient BDD implementations. Conventional BDD algorithms [3, 11] are based on depth-first traversal of the BDD graphs. This approach has the advantage of small memory overhead. Recently, there have been many implementations based on breadth-first traversal [13, 14, 1, 10, 15], which have found that the breadth-first approach has better memory access locality and thus better performance. However, the breadth-first approach can have a large memory overhead, especially when the BDDs involved are large. This extra memory overhead can result in an increase of the number of page faults.

In this paper, we present the technique of operator sifting as a new way of understanding both breadth-first and depth-first approaches to BDD construction. This operator sifting concept is inspired by MORE [10] which constructs BDDs by sifting down temporary variables as existential quantifiers.

We then describe a new hybrid algorithm for BDD construction by combining depth-first and breadth-first approaches. This algorithm has good memory access locality while keeping the memory overhead below a fixed fraction of the total memory usage. Combining of breadth-first and depth-first approaches to bound memory overhead has been proven successful in the parallel computation communities [7, 2, 12]. Experimental results on ISCAS85 [4] and multiplier circuits [6] show that our new approach is generally faster than other breadth-first and depth-first implementations, while keeping memory overhead comparable to the depth-first approach. In particular, for the 13-bit multiplier circuit, our package finished building the output BDDs in about 2 hours of elapsed time while other packages did not finish after 6 hours.

The rest of this paper is as follows: Section 2 presents our view of the depth-first and the breadth-first approaches to BDD construction. In Section 3, we qualitatively compare the depth-first and the breadth-first approaches to BDD construction. Section 4 describes our new algorithm. We compares our design with two other breadth-first approaches in Section 5. Section 6 presents preliminary performance results. Finally, in Section 7, we conclude and present some directions for future work.

2 Depth-First and Breadth-First BDD Constructions

In this section, we introduce a framework under which both the breadth-first and the depth-first approaches to BDD construction can be viewed as sifting down Boolean operators. Given an ordering of variables and a Boolean operation $r = f <op> g$, the result BDD for r is constructed based on the Shannon expansion

$$r = f <op> g \quad = \quad \overline{x} \cdot (f_{x=0} <op> g_{x=0}) + x \cdot (f_{x=1} <op> g_{x=1}) \qquad (1)$$

where x is the variable (**top variable**) with the highest precedence among all the variables of f and g. $f_{x=0}$ and $f_{x=1}$ are the cofactor functions of f with respect to the Boolean variable x (i.e., the function f with the variable x restricted to 0 and 1, respectively). Similarly, $g_{x=0}$ and $g_{x=1}$ are the cofactor functions of g.

Following the given variable order, this expansion process repeats recursively for all the Boolean variables in f and g. The base case (also called the **terminal case**) of this recursive process is when the result of the operation can be trivially evaluated. For example, the Boolean operation "f *AND* f" is a terminal case because it can be trivially evaluated to f. Similarly, "f *AND* 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 process, there may be unreduced subexpressions like $(\overline{x} \cdot h + x \cdot h)$. Thus, a reduction step is necessary to reduce $(\overline{x} \cdot h + x \cdot h)$ to h.

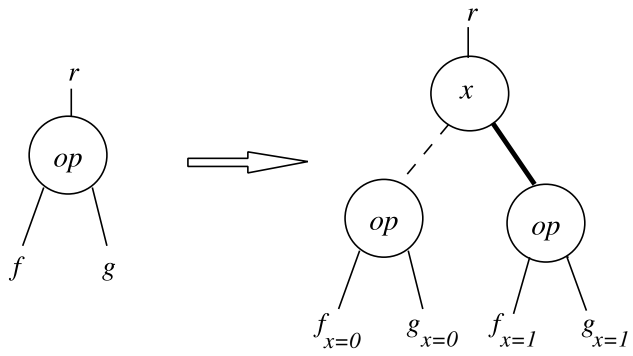

Figure 1: **Operator Sifting View of BDD Construction:** The dashed edge represent the 0-branch of a variable and the thick solid edge represents the 1-branch

Figure 1 illustrates the Shannon expansion (Equation 1) 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. Notice that this expansion is very much like **sifting down** the operator node along both the left and right branches. After this sifting process, the original operator node becomes an **unreduced node** (since neither of the children is a reduced BDD node). This sifting process is very similar to the **level exchange** method [9] in dynamic variable reordering. The main difference here is that the level exchange method sifts down variable nodes, while BDD construction sifts down operator nodes.

During the recursive step, we can choose to sift down the operator nodes in any order we like. In particular, if we always choose the operator node with the greatest depth, then we will be performing the Shannon expansion in the depth-first manner. Similarly, if we always choose to sift down the operator node with lowest depth, then we will be sifting the operations in the the breadth-first manner.

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

2.1 Depth-First BDD Construction

df_op(op, f,g)
1 if (terminal case) return simplified result;
2 if (op,f,g) is in computed cache, return result found in cache.
3 let x be the top variables of f and g.
4 e =: df_op(op, $f_{x=0}$, $g_{x=0}$);
5 t =: df_op(op, $f_{x=1}$, $g_{x=1}$);
6 if (t == e) return t;
7 result =: find or add (x, t, e) in the unique table;
8 insert (op, f, g, result) into the computed cache;
9 return result;

Figure 2: Depth-First BDD Algorithm

A typical depth-first BDD algorithm is shown in Figure 2. This algorithm constructs BDDs by recursively sifting down operations in the depth-first manner. The depth-first algorithm does not explicitly store the operations in operator nodes. Instead, the operation is implicitly stored in the arguments to the recursive calls. This recursive process ends when the new operation created is a terminal case (line 1) or when it is cached in the **computed cache** (line 2). In the depth-first approach, the computed cache preserves the previous computation results (line 8) to avoid redundant computations (line 2). Line 6 is the reduction step which ensures the BDD result is a reduced BDD node.

2.2 Breadth-First BDD Constructions

A typical breadth-first BDD algorithm is shown in Figure 3. The breadth-first approach constructs BDDs by repeatedly sifting down operator nodes in breadth-first manner. The breadth-first algorithm can be divided into two phases: a sift-down phase (line 3, 4, and 5) which repeatly sifts down the operator nodes starting from the top variable, and a reduction phase (line 6, 7, 8, and 9) which applies reduction rules to the unreduced nodes to ensure the result BDD is in reduced form. In this algorithm, there is one **operator queue** associated with each variable. Every operation is queued in its top variable's operator queue. These operator queues enable the sift-down phase to process the operator nodes in the breadth-first manner based on the fixed variable order.

Figure 4 illustrates the breadth-first algorithm using an example operation r = f *OR* g. The shaded nodes in this figure are the newly allocated nodes. In Figure 4(a), an operator node (the shaded node) is created to store the information for this operation and it is inserted into the operator queue of variable x_1 (line 2). During the sift-down phase, the operator nodes are sifted down beginning from the top variable of this operation (line 3). Each sift will create two operations as shown in Figure 1. New operator nodes will be created only if the operations generated are new and an **operator unique table** is used to guarantee this. Each of the new operator nodes created will be inserted into the operator queue of its top variable (line 5). Figure 4(b) shows the unreduced BDD graph after all the operator nodes are sifted down.

During the reduction phase (line 6, 7, 8, and 9), when the result of a reduction already exists, the

```
  bf_op(op, f, g)
1     if (terminal case) return simplified result;
2     create an operator node r = (op, f, g) and insert it into
          the operator queue for the top variable τ of f and g;
3     for each variable's operator queue q starting from variable τ,
4         sift down all operator nodes in q and
5         if the newly created operations are unique,
          queue the operations based on the top variable of the operands.
6     for each variable starting from the bottom variable,
7         Apply reduction to each unreduced BDD node of that variable;
8         if the result bdd from the reduction already exists, set the unreduced node to a forwarding node;
9         else insert the result into the BDD unique table.
  return r;
```

Figure 3: Breadth-First BDD Algorithm

unreduced BDD node is changed to a **forwarding node** (line 8) which forwards the reference to the existing BDD node. Otherwise, the corresponding unreduced BDD node is used to store new BDD result and is inserted to the BDD unique table (line 9).

Figure 4(c) shows the BDD graph after applying the reduction to variable x_3 where the shaded x_3 node in Figure 4(b) is changed to the forwarding node "F" in the Figure 4(c). Figure 4(d) shows the result of applying reduction to variable x_2. Here, one of the unreduced x_2 nodes in the Figure 4(c) is changed to a forwarding node because the result BDD (marked with "*") already exists as a subgraph (marked with "*" in Figure 4(a)) of f's BDD. Finally, Figure 4(e) shows the result after the reduction phase.

The above algorithm can be modified to allow queuing multiple top level operations (**multi-issue**) before beginning the sift-down phase. Figure 5 shows a graph of 11 operations issued before performing the actual breadth-first BDD computations. During the breadth-first BDD computations, the operator nodes will be sifted down layer by layer starting from the lowest layer (layer 1). Simultaneously sifting down multiple top level operations on the same layer is called is called **superscalarity** in [15]. For example, in Figure 5, operator nodes op_0, op_1, op_2, op_3, op_4, op_{10}, and op_{11} in layer 1 can be sifted down simultaneously (with the assumption that all the operations in layer 1 have the same top variable). The main advantage of the superscalarity is the complete caching among these top level operations.

With multi-issue, multiple layers of operator nodes can be sifted down in a **pipelined** fashion [15]. For example, to pipeline two layers of operator nodes in Figure 5, we will first sift down the operator nodes in layer 1 by one variable (under the assumption that all the operations in layer 1 have the same top variable). We will then sift down the operator nodes in layer 2 on the unreduced nodes which were just produced by sifting layer 1's operator nodes. Then we will sift the operator nodes generated from sifting layer 1's operator nodes. And then we will sift the operator nodes generated from sifting layer 2's operator nodes. This process repeats until all of the operator nodes generated by sifting layer 1 and 2 are sifted. Then, we pipeline any remaining layers two at a time. The main advantage of pipelining

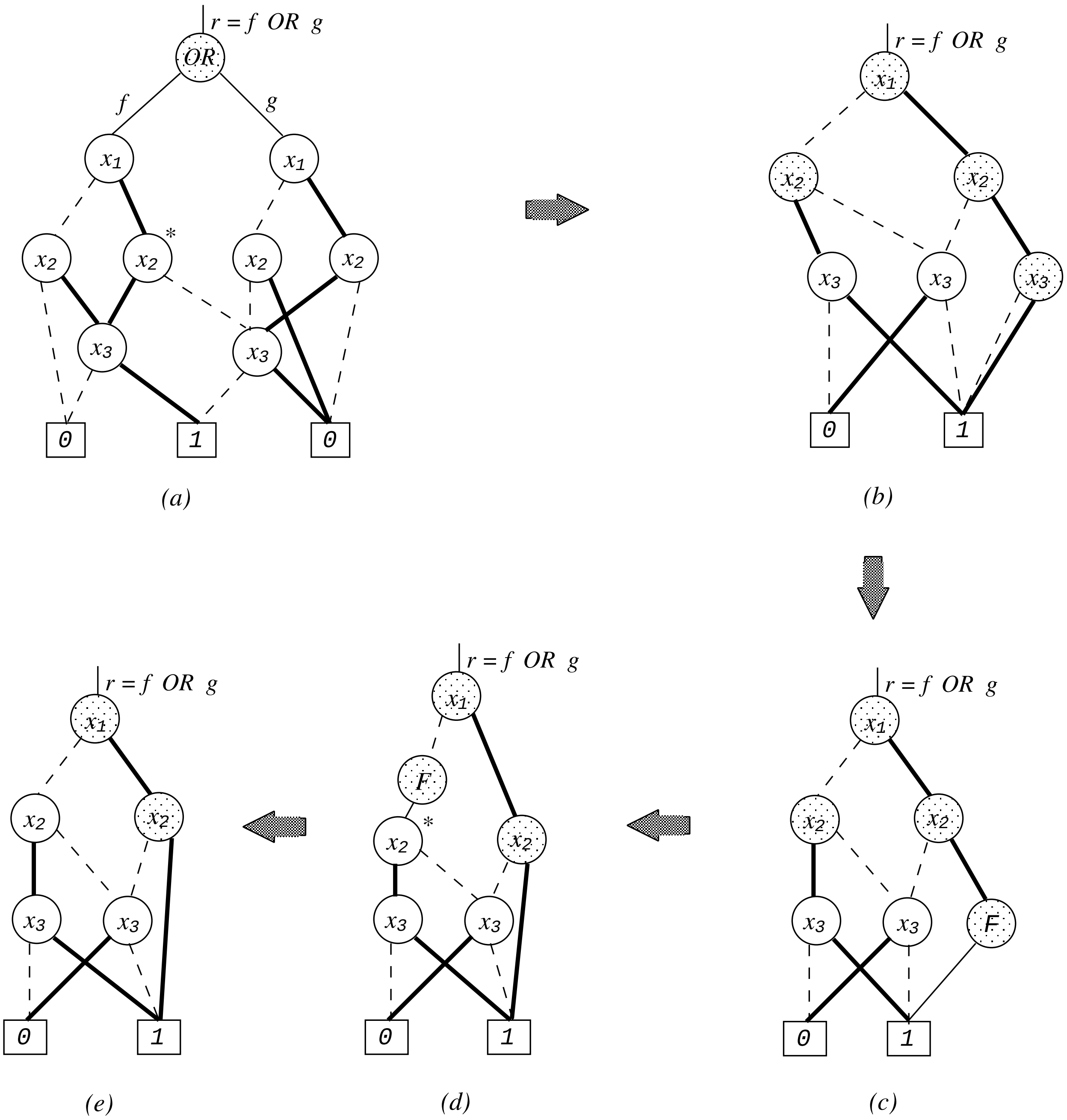

Figure 4: **Example of Breadth-First BDD Construction.** The shaded nodes are newly allocated nodes. (a) the initial graph (b) the unreduced graph after the sift-down phase (c) the graph after applying reduction on variable x_3 (d) the graph after applying reduction on variable x_2 (e) the final reduced graph

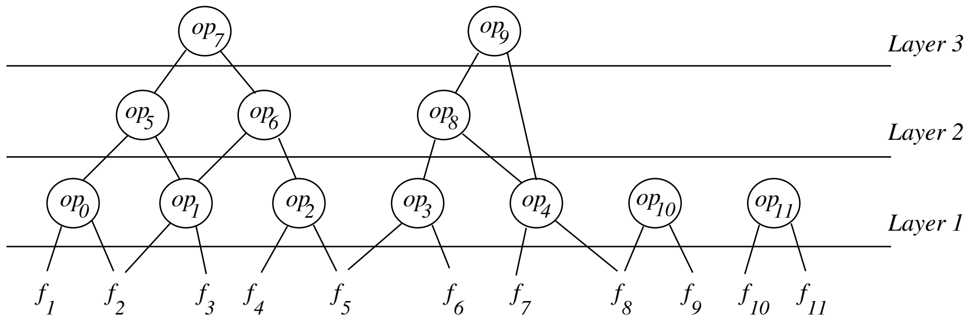

Figure 5: Example of Multi-Issue

is better locality of memory access because the operations in the higher layers of the pipeline will be processing unreduced nodes which were just produced by sifting the operations in the lower layers.

3 Depth First vs. Breadth-First

The main advantage of the depth-first approach is that memory overhead is low. The **memory overhead** in BDD construction is all the extra memory usage that is not associated with storing BDD nodes. In the depth-first approach, the two sources of memory overhead are the computed cache and the recursion stack. The size of the computed cache can be kept as a small fraction (e.g., 10%) of the total memory allocated and the size of the recursion stack is bounded by the number of Boolean variables. Thus, the memory overhead of the depth-first approach can be kept below a fixed percentage of total memory usage.

In contrast, even though the breadth-first approach does not keep a computed cache, it must create one operator node for each distinct operation and the number of the distinct operations in a BDD construction can be quadratic in the size of BDD graphs. The main source of memory overhead associated with the breadth-first approach is memory allocated to those operator nodes whose corresponding operations do not result in new BDD nodes. (Those operator nodes whose corresponding operations do result in new BDD nodes can directly reuse the operator nodes by changing the operator nodes into BDD nodes. Therefore, they do not incur memory overhead.) In practice, we observe at least 20% of the operator nodes do not result in new BDD nodes. This is the main reason why the breadth-first approach generally uses more memory than the depth-first approach.

Without a computed cache, the breadth-first approach cannot avoid duplicating computations across different sift-down phases. Thus, simultaneously sifting down a large number of top level operations is necessary to ensure good performance. Sifting down a large number of top level operations increases the number of operator nodes generated, and thus the memory overhead will also increase. Because pipelined operations work directly on unreduced BDD nodes, the number of the operator nodes will increase. Thus pipelining will further increase the memory overhead.

The main advantage of the breadth-first approach is memory access locality. During a breadth-first sift-down phase, all the BDD nodes of the same variable are accessed together. Thus, the memory allocated for all the BDD nodes of the same variable can be clustered to improve locality.

Another advantage of the breadth-first approach is the **intelligent caching** of the computation results. Within a sift-down phase, the breadth-first approach does not generate duplicate operator nodes. The uniqueness of operator nodes is often implemented using an operator unique table. This unique table is identical to a complete computed cache in depth-first approach. (Note that this complete cache behavior only holds within a sift-down phase and not across different sift-down phases.) Furthermore, the variable-by-variable top-down sifting of the breadth-first approach guarantees that no new operations generated will involve variables already sifted. Once an operation is sifted, its corresponding entry in the operator unique table can be removed. Thus, the operator unique table intelligently caches only what is necessary within a sift down phase. Since the depth-first approach does not process all the operations of one variable together, it cannot cache as intelligently.

4 Our Approach to BDD Construction

Our design is a hybrid of the depth-first and the breadth-first approaches which captures the best of both worlds — memory access locality and low memory overhead. When the memory overhead is low, we rely on the breadth-first approach with the help of a computed cache. When the memory overhead is high (due to large number of operator nodes created by the breadth-first phase), we switch to the depth-first approach to avoid excessive memory overhead and thus avoid page faults.

In the following, we will first present the hybrid algorithm in more detail. Then we will describe the top level nodes and our memory management strategy, both of which enable us to implement this algorithm efficiently.

4.1 Memory Access Locality: Breadth-First Approach

The core of our design is to use the traditional breadth-first approach to capture memory access locality. In traditional breadth-first implementations (as described in Section 2.2), whenever an operation's result is a new BDD node, this operation's corresponding operator node is reused as this new BDD node and all other operator nodes will become forwarding nodes. Thus, after the reduction phase, memory blocks become fragmented because the newly created BDD nodes will be inter-mixed with the forwarding nodes. Thus the newly created BDD nodes will not be as close together in memory as possible. Furthermore, since the forwarding nodes are not part of any BDDs, they will eventually be freed. When the memory of these freed nodes is subsequently reused, the nodes allocated will not be contiguous.

To avoid memory fragmentation, when an operation's result BDD is new, we do not reuse the corresponding operator node's memory to represent the new BDD node as is done in the traditional breadth-first implementations. Instead, we allocate new memory for this new BDD node and change the operator node into a forwarding node to the new BDD node. Thus, all operator nodes will become forwarding nodes at the end of the reduction phase and can be all freed at the same time. By allocating memory for new BDD nodes with the help of a specialized memory manager, we can ensure that the new BDD nodes will be contiguous in memory and that subsequent access to these nodes will have good memory locality. However, since we always create new BDD nodes, the memory overhead of our approach is all of the memory allocated to the operator nodes which can be quadratic in the size of the

BDD graph. In Section 4.3, we will present a way to bound this memory overhead.

4.2 Reduce Memory Usage: Computed Cache

As discussed in Section 3, the traditional breadth-first approach's higher memory usage is partially due to simultaneously sifting down a large number of top level operations. To reduce memory usage without sacrificing performance, we add a computed cache to cache the previous computation results. By using this computed cache, we can avoid duplicating some computations which were performed in the previous sift-down phases. Thus we can sift down a smaller number of the top-level operations simultaneously without sacrificing performance.

The operator unique table in the traditional breadth-first approach can be naturally extended to implement the computed cache. In our implementation, there are two types of cache nodes: the **operator cache nodes** store the operations that are currently in progress as in the breadth-first approach and the **computed cache nodes** store previously computed operations with their BDD results as in the depth-first approach. The main difference between these two types of cache nodes is that for its corresponding operation, an operator cache node stores a handle that will eventually be forwarded to a BDD result while a computed cache node stores the actual BDD result.

During the sift-down phase, if an operation is not in the computed cache, a new operator cache node is inserted to avoid duplicating computations within the same sift-down phase. During the reduction phase, when the result BDD of an operation is constructed, this operation's corresponding operator cache node will be changed to a computed cache node by storing this result BDD into the cache node. Future cache lookups (from subsequent sift-down phases) on this operation will directly return the BDD result.

The computed cache is implemented based on bucket hashing. Cache nodes that hash to the same bucket will be chain together into a linked list. This hash table is automatically resized when the average length of the chains exceeds a constant threshold.

4.3 Upper Bound on the Memory Usage: Depth-First Approach

During a sift-down phase, if the memory overhead of the operator nodes becomes too high, the number of page faults will increase and the performance will degrade dramatically. We overcome this problem by limiting the number of operator nodes to a fixed fraction of the physical memory. When the number of operator nodes exceeds this limit, we sift down the remaining operator nodes in the depth-first manner. Thus we can avoid page faults caused by breadth-first's excessive memory overhead.

In Section 4.1, we pointed out that our approach could potentially require more memory than traditional breadth-first implementations because we always allocate new memory for new BDD nodes instead of reusing the operator nodes. By limiting the number of the operator nodes, we can place an upper bound on how much extra memory our package will use. To be precise, the memory overhead of our design will be bounded by the memory allocated for the computed cache (which is a small percentage of the total memory usage) and the operator nodes (which has a constant upper bound). In practice, we found that we can limit the computed cache to be less than 1% of total memory usage and the number of operator nodes to less than 20% of available physical memory without adversely affecting running time in BDD construction.

4.4 Separation from the External World: Top Level Nodes

Top level nodes are handles to BDD nodes used by external users. This level of indirection allows us to transparently change which node a given handle refers to. Thus, after the reduction phase, we can automatically update all the top level nodes to reference BDD nodes and then free all memory blocks associated with the operator nodes.

Without this level of indirection, the external users will need to directly refer to the operator nodes to issue multiple top level operations. Since some of these exported operator nodes can become forwarding nodes, we cannot free all the memory associated with the operator nodes after the reduction phase. Therefore, memory fragmentation can occur.

In the future, this mechanism will allow us to relocate BDD nodes during memory compaction when memory becomes fragmented after garbage collection or dynamic variable reordering.

4.5 Memory Management

To ensure memory access locality, we use memory managers to customize memory allocation. We associate a memory manager for each of the following four node types: top level, BDD, operator, and cache. A memory manager tracks all the memory blocks allocated to its associated node type. When the size of the associated node type is a multiple of 16 bytes, the memory manager ensures each node allocated is 16-byte aligned. This alignment enables us to use the last 4 bits of each pointer for complement and marking flags. In our current implementation, the sizes of the top level nodes, the BDD nodes, and the operator nodes are 16 bytes, and the cache nodes are 20 bytes.

We use different garbage collection strategies for different node types:

Top Level: Garbage collection of top level nodes is based on reference counting. When a top level node's reference count reaches zero, it is placed in its memory manager's free list to be reused later.

Operator: Right after the reduction phase, all the top level nodes are updated to refer to BDD nodes and all the operator cache nodes are changed to the computed cache nodes so that there are no more references to the operator nodes. Therefore, all of the memory blocks for operator nodes can be freed at the end of this updating process.

Cache: In our design, cache garbage collection occurs whenever the cache's memory usage increases by a fixed constant. Cache nodes are garbage collected by freeing the oldest memory blocks one at a time until the memory usage of the cache nodes is less than a fixed percentage of the total memory usage of the entire BDD package.

Currently, we allow a larger cache in the depth-first mode than in the breadth-first mode because the depth-first approach has less temporal locality in processing operations. The breadth-first approach processes all the operations for one variable at a time and thus does not require a large computed cache. Since the depth-first approach has less memory overhead than the breadth-first approach, a bigger cache will not adversely increase memory requirement in comparison to a pure breadth-first approach.

In our implementation, the default behavior in the breadth-first approach is to garbage collect when the computed cache's memory usage increases by 2 MBytes and the garbage collector frees the computed cache's memory blocks until the computed cache uses less than 1% of the total memory usage. The default behavior in the depth-first approach is to garbage collect when the computed cache's memory usage increased by 8 MBytes and the garbage collector frees the computed cache's memory blocks until it uses less than 1% of the total memory usage.

All cache nodes are managed by one memory manager without distinguishing between the operator cache nodes and the computed cache nodes. Thus, our breadth-first approach no longer has a complete cache as in other breadth-first packages.

BDD: We use a mark-and-sweep garbage collector to recycle BDD nodes. We chose the mark-and-sweep over the reference-count method because the mark-and-sweep uses less memory per node (no need to store the reference count) and is less bug-prone and cumbersome to implement. To avoid dangling references, we also garbage collect any cache node that references garbage collected BDD nodes.

4.6 Implementation Support for Future Extensions

We designed this BDD package to facilitate future extension to KFDD [8] and *BMD [6]. To support *BMD without exceeding 16 bytes per BDD node on 32-bit machines, we are unable to improve memory access locality by placing both left and right children's variable orders in a BDD node as it was done in [15]. To support *BMD and KFDD, we allow simultaneously sifting down different types of operations. For BDD, we currently support *AND* and *XOR*. For *BMD, we will support *ADD* and *MULT*.

5 Comparison with Two Other Breadth-First Implementations

Our design is influenced by both of the MORE package [10] and the CAL package [15]. The concept of viewing BDD constructions (either breadth-first or depth-first) as sifting down operators is inspired by MORE. MORE introduces new temporary variables to represent existential quantification and these new temporary variables are sifted down to construct BDDs. This sifting scheme is very similar to sifting down an *OR* operator. Our framework generalizes this sifting scheme to cover any operation. This generalization is especially important in supporting *BMD [6] and KFDD [8]. MORE's support for parallel operations is identical to breadth-first's multi-issue with superscalarity and pipelining. MORE's ability to allow dynamic reordering without loss of intermediate results is quite novel and useful. We plan to incorporate the same mechanism into our breadth-first stage of BDD construction.

CAL's design and implementation to explore memory locality has greatly influenced our design and implementation. Our memory management strategy is very similar to that of CAL's. We have also used CAL's power-of-2 hashing strategy to replace the usual *MOD* operation with the bit-wise *AND* and have observed a visible performance gain. The superscalar and pipeline concepts come directly from CAL's design. However, unlike our package, the CAL package does not support simultaneously sifting down different types of Boolean operations. Therefore, the CAL package cannot support the *BMD's *MULT* operation, which generates *ADD* operations as it is sifted down.

The introduction of the top level nodes, the computed cache, and the breadth-first/depth-first hybrid approach sets our work apart from all other breadth-first implementations. Currently, our package supports superscalar feature without pipelining.

6 Experimental Results

We performed our experiments on a Sun Sparc20 workstation with 64 MB RAM of main memory and 300 MB of swap space. In order to compare our results with other BDD packages, we built a platform to ensure that all the BDD operations will be issued in the same order. Our measurements include initialization of the BDD package and the computation of the BDD outputs. The initialization of our platform and parsing the input files are not part of the measurements. The memory usage we measured is the heap usage of the BDD packages. The time limit for our measurements is 6 hours of elapsed time.

We chose two packages as the basis for comparison. The choice was made based on the results of a recent study by Sentovich [16]. The first package is CAL version 1.1 from UC Berkeley. This package implements the breadth-first algorithm for the BDD construction. In the Sentovich study, CAL generally outperforms other depth-first packages. The second package chosen is CUDD version 1.1.1 from the University of Colorado at Boulder. This package implements the depth-first algorithm for BDD construction and in the Sentovich study, it generally performs better than Long's depth-first BDD package [11] for constructing output BDDs. For both CUDD and CAL, we used all the default settings, but we turned off their dynamic variable reordering features.

For the CAL package, we used the pipeline and superscalar features in the same way as the SIP package (which is released with the CAL package and uses CAL to perform BDD computations). The SIP package first multi-issues *all* Boolean operations simultaneously and then starts the breadth-first BDD computation. For pipelining, the default pipeline depth is four. Since CAL's multi-issue feature cannot support issuing different types of Boolean operations at the same time, we decomposed all Boolean operations into *AND* with the complement edge for negation.

Currently, our package has support for the superscalar feature, but no support for pipelining. It is our experience that the addition of the computed cache eliminates the need for the superscalarity. Therefore, all of the results for our package are obtained without superscalarity and pipelining. In the breadth-first mode, we garbage collect the computed cache whenever the cache's memory usage increases by 2 MBytes to keep its memory usage under 1% of total memory usage. In the breadth-first mode, we garbage collect the computed cache whenever the cache's memory usage increases by 8 MBytes to keep its memory usage under 1% of total memory usage. We switch from the breadth-first mode to the depth-first mode when the operator nodes use more than 10 MBytes (15% of available physical memory). The computed cache table will automatically grow when its average bucket chain length is greater than 4.

The following sections show the performance result for the ISCAS85 circuits [4] and some multiplier circuits. For the ISCAS85 circuits, we used two variable orderings: one is obtained from Hett [10] and the other is generated by *order_dfs* in SIS [17]. The purpose of using two different variable orderings is to observe whether the relative performance is influenced by the different orderings.

6.1 Small Circuits: ISCAS85

This section presents results for ISCAS85 circuits to study the effectiveness of our design for small circuits where our package does not switch into depth-first mode. First, we use Hett's variable orderings to perform our experiments for ISCAS85 circuits. Table 1 shows the CPU time in seconds and memory requirements in megabytes (MB). Normalization of these results based on our results are shown in parenthesis.

Circuit	CPU Time (seconds)			Memory (MB)		
	Ours	CAL	CUDD	Ours	CAL	CUDD
C432	2.10 (1)	2.46 (1.17)	4.29 (2.04)	6.56 (1)	7.43 (1.13)	6.99 (1.06)
C499	0.50 (1)	1.29 (2.58)	0.95 (1.90)	3.01 (1)	6.97 (2.31)	1.67 (0.55)
C880	0.27 (1)	0.31 (1.14)	0.55 (2.03)	1.94 (1)	4.15 (2.13)	1.38 (0.71)
C1355	2.19 (1)	3.11 (1.42)	4.48 (2.04)	6.75 (1)	12.6 (1.87)	7.10 (1.05)
C1908	0.60 (1)	0.76 (1.26)	1.38 (2.30)	3.37 (1)	4.68 (1.38)	1.88 (0.55)
C2670	0.34 (1)	0.60 (1.76)	0.55 (1.61)	2.27 (1)	7.54 (3.32)	2.85 (1.25)
C3540	13.8 (1)	19.5 (1.41)	24.5 (1.76)	21.0 (1)	28.0 (1.33)	20.1 (0.95)
C5315	2.36 (1)	2.83 (1.19)	4.88 (2.06)	7.37 (1)	11.5 (1.56)	7.50 (1.01)
C7552	0.62 (1)	0.70 (1.16)	0.85 (1.37)	3.35 (1)	6.84 (2.04)	3.01 (0.89)

Table 1: **ISCAS85 Results with Hett's Orderings.** The parenthesized number is the result normalized based on our package.

Our package consistently performs better than the other packages in terms of CPU time. On average, the CUDD package is at least 50% slower than ours because our package has better memory locality. The CAL package is about 20% slower than ours, on average. This is mainly due to lower memory usage because we rely on the computed cache instead of multi-issue to avoid redundant computations across different sift-down phases.

For C499, the CAL package is more than 100% slower than ours. This is partially caused by a large number of *XOR* operations, which are decomposed into three *AND* operations in the CAL package. (Note: The only other circuit in ISCAS85 that has *XOR* operations is C432.) Since both CUDD and our package can handle different operations, no decomposition is necessary.

The CAL package consistently uses more memory than our package, especially for C2670 (3.32 times). Its large memory usage is caused by the multi-issue feature. In comparison with the CUDD package, our package uses comparable memory size except when the memory usage is small (< 2MB). This is mainly due to a larger constant overhead in the breadth-first approach.

Next, we used the variable ordering generated by SIS to do another round of experiments on the ISCAS85 circuits. None of the packages were able to run C2670, C3540, and C7552 due to memory or time limitations. Table 2 shows the results of these packages. In general, the relative performance is the same as using Hett's variable ordering.

Circuit	CPU Time (Seconds)			Memory (MB)		
	Ours	CAL	CUDD	Ours	CAL	CUDD
C432	2.10 (1)	2.46 (1.17)	4.29 (2.04)	6.56 (1)	7.43 (1.13)	6.99 (1.06)
C499	0.47 (1)	1.07 (2.27)	0.82 (1.74)	2.67 (1)	6.04 (2.26)	1.63 (0.61)
C880	0.26 (1)	0.31 (1.19)	0.49 (1.88)	1.78 (1)	4.15 (2.33)	1.28 (0.71)
C1355	1.13(1)	1.73 (1.53)	2.40 (2.12)	4.65 (1)	8.67 (1.86)	3.79 (0.81)
C1908	0.55 (1)	0.65(1.18)	1.25 (2.27)	3.09 (1)	4.38 (1.41)	1.65 (0.53)
C5315	0.69 (1)	0.79 (1.14)	1.21 (1.75)	3.64 (1)	7.02 (1.92)	2.79 (0.76)

Table 2: **ISCAS85 Results with SIS's orderings.** The parenthesized number is the normalized result based on our package.

Above results show that CAL and our package generally perform better than CUDD because the breadth-first approach has better memory locality. Our package consistently runs faster than CAL's pure breadth-first implementation because the addition of the computed cache is effective in reducing memory usage.

6.2 Multipliers

We used array multiplier circuits [6] to study the effects of building very large BDDs. Table 3 shows results for the multiplier circuits from 10 to 13 bits, in terms of CPU time, elapsed time, memory usage, number of page faults, and number of total BDD operations performed.

In general, our package uses around 10% more memory than the CUDD package and uses much less memory than the CAL package. CAL performs at least 50% more operations than ours mainly due to the pipelining and the *XOR* decomposition. CUDD performs fewer operations than our package, because of our smaller cache size (1% of total memory usage). For the CPU time and the elapsed time, our package is at least 3 times faster than the CAL package and is generally better than the CUDD package, except for the multipler of size 12. For this circuit, CUDD's memory usage is low enough that it barely fit into the main memory and thus there are no page faults. In contrast, our package uses about 3 MB more than CUDD does and incurs over 1000 page faults. However, because we have better memory locality, our elapsed time is only about 5% more than CUDD's. For the same circuit, the CAL package uses 90MB which causes more than 155000 page faults. This excessive memory usage is the reason why CAL's elapsed time is so much higher than both that of CUDD and our package. The 12-bit multiplier circuit clearly demonstrates the need to minimize memory overhead.

Size		10	11	12	13
CPU Time (seconds)	Ours	8.50 (1)	36.98 (1)	142.40 (1)	505.66
	CAL	38.20 (4.49)	136.26 (3.68)	491.14 (3.44)	>1443
	CUDD	12.45 (1.46)	40.57 (1.09)	162.47 (1.14)	>1101
Elapsed Time (seconds)	Ours	9 (1)	39 (1)	183 (1)	7943
	CAL	42 (4.66)	142 (3.64)	2457 (13.42)	> 6.5 hours
	CUDD	13 (1.44)	43 (1.10)	177 (0.96)	>6 hours
Memory (MB)	Ours	9.15 (1)	20.27 (1)	52.92 (1)	117.73
	CAL	13.48 (1.47)	34.22 (1.68)	90.80 (1.71)	> 188.60
	CUDD	8.63 (0.94)	19.19 (0.94)	49.02 (0.92)	> 85.95
Page Faults	Ours	0	0	1302	671704
	CAL	0	0	155405	N/A
	CUDD	0	0	0	N/A
Total Operations	Ours	1750084 (1)	7242125 (1)	24657288 (1)	78779575
	CAL	3980442 (2.27)	12344081 (1.70)	38078748 (1.54)	N/A
	CUDD	1720270 (0.98)	5418655 (0.74)	21056572 (.85)	N/A

Table 3: **Multiplier Results.** The number in parenthesis is normalized based on our results. **N/A** are statistics that are not available because the time limit was exceeded.

For the 13-bit multiplier, our package finished the computation in about 2 hours of elapsed time. The other two packages did not finished the job after 6 hours of elapsed time, and had used at least twice as much CPU time as our package did. Furthermore, the fact that their CPU time is only around 1000 seconds after more than 6 hours of computation indicates these two packages have very poor paging behavior. In all of our experiments, this is the only circuit that generated a large enough number of operator nodes for our package to switch into depth-first mode. This result strongly suggests that our hybrid design to limit the memory usage and improve memory locality is effective in avoiding excessive page faults.

7 Conclusions and Future Work

We have presented the technique of sifting as a different way of understanding BDD construction for both breadth-first and depth-first traversal. We have also described a new hybrid approach for building BDDs by combining the breadth-first approach with the computed cache and switching to the depth-first approach when the breadth-first approach's memory overhead becomes too high. Experimental results showed that in all but one case our package are better than the CUDD's depth-first implementation because our approach has better memory locality. For small circuits, the addition of the computed cache to the breadth-first approach is effective in reducing memory usage which is the reason why our package consistently runs faster than CAL's pure breadth-first implementation. For the large circuit, the results show that bounding the memory with our hybrid approach is crucial to capturing memory locality and minimizing page faults.

In the future, we plan to extend our implementation to cover other features like KFDD, *BMD, and dynamic variable reordering. We also plan to perform more studies on the effectiveness of this hybrid design and different switching heuristics between breadth-first and depth-first modes.

References

[1] 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.

[2] BLELLOCH, G. E., GIBBONS, P. B., AND MATIAS, Y. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 1995 ACM Symposium on Parallel Algorithms and Architectures* (Santa Barbara, July 1995), pp. 420–430.

[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), pp. 8:677–691.

[6] BRYANT, R. E., AND CHEN, Y.-A. Verification of arithmetic circuits with binary moment diagram. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (1995), pp. 535–144.

[7] BURTON, F. W., AND HUNTBACH, M. M. Virtual tree machines. In *IEEE Transactions on Computers* (March 1984), pp. 278–280.

[8] 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.

[9] FUJITA, M., MATSUNGA, Y., AND KAKUDA, T. On variable ordering of binary descision diagrams for the application of multi-level synthesis. In *Proceedings of the European Design Automation Conference* (1991), pp. 50–54.

[10] 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).

[11] LONG, D. E. Robdd package., November 1993.

[12] NARLIKAR, G. N., AND BLELLOCH, G. E. Space-efficient implementation of nested parallel languages. *Draft (available from the authors)* (1996).

[13] 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.

[14] 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.

[15] 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.

[16] Sentovich, E. M. A breif study of bdd package performance. In *Proceedings of the Formal Methods on Computer-Aided Design* (November 1996).

[17] 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.