Efficient Breadth-First Manipulation of Binary Decision Diagrams

Pranav Ashar Matthew Cheong
C&C Research Labs, NEC USA
Princeton, NJ 08540

Abstract

We propose new techniques for efficient breadth-first iterative manipulation of ROBDDs. Breadth-first iterative ROBDD manipulation can potentially reduce the total elapsed time by multiple orders of magnitude compared to the conventional depth-first recursive algorithms when the memory requirement exceeds the available physical memory. However, the breadth-first manipulation algorithms proposed so far [5] have had a large enough overhead associated with them to make them impractical. Our techniques are geared towards minimizing the overhead without sacrificing the speed up potential. Experimental results indicate considerable success in that regard.

1 Drawbacks of Conventional DF Recursive ROBDD Manipulation

There is a need today for manipulating ROBDDs with tens to hundreds of millions of nodes which cannot be met by means of conventional depth-first (DF) recursive algorithms. There are two good reasons why DF recursive algorithms have been the algorithms of choice for ROBDD manipulation until now. One is that the recursive formulation for ROBDD manipulation[2] lends itself naturally to a compact depth-first recursive implementation. An outline of such an implementation (from [1]) of the ITE(F, G, H) operation is illustrated in Figure 1.[1] In addition, the DF recursive paradigm has been exploited [1] to eliminate the temporary creation of redundant ROBDD nodes by performing the isomorphism check on the nodes on the fly - a new node is created only if a node with the same attributes does not already exist (Line 12 in Figure 1). However, the use of DF algorithms has its downside for very large ROBDDs arising from an extremely disorderly memory access pattern[5].

The depth-first approach is characterized by the fact that a new ITE computation request with some top-variable can be issued only after the final results of all the previous ITE requests with the same top-variable are known. It is apparent from Figure 1 that successive memory accesses correspond to successive nodes on paths in the ROBDDs F, G and H. Given that a typical node in a large ROBDD generally has a large indegree, it is impossible to ensure that an arbitrary pair of nodes next to each other on some path in an ROBDD are located at contiguous memory addresses or even in the same page.[2] The latency of fetching a page from secondary storage is multiple orders of magnitude greater than fetching a word from main memory. With current technology, a page fetch takes of the order of 10ms. If the process size exceeds the available main memory, the part of the process that is needed immediately can be moved from secondary storage to main memory only at the expense of moving some part of it out from main memory to secondary storage. In the case of ROBDD manipulation, if the ROBDDs that are being traversed are too large to fit in main memory, it is unlikely that the desired node will ever be in main memory. Therefore, each time an ROBDD node is visited, the complete page containing the ROBDD node must be fetched from secondary storage in the worst case. This would cause hundreds of millions of page faults for ROBDDs of the size we are interested in, making it virtually impossible to manipulate/create them using DF algorithms.

df_ite(F, G, H) {
1. if (terminal case(F, G, H)) {
2. return result ;
3. } else if (computed table has the entry (F, G, H)) {
4. return result ;
5. } else {
6. Determine x, the top variable of (F, G, H) ;
7. T = **df_ite**(F_x, G_x, H_x) ;
8. E = **df_ite**$(F_{\overline{x}}, G_{\overline{x}}, H_{\overline{x}})$;
9. if (T equals E)
10. return T ;
11. R = find_or_add_unique_table(x, T, E) ;
12. insert_computed_table((F, G, H), R) ;
13. return R ;
14. }
}

Figure 1: Depth-First Recursive ROBDD Manipulation

2 BF Iterative ROBDD Manipulation

Ochi *et al.*[5] have proposed that the disorderly memory access pattern can be corrected by the use of breadth-first (BF) iterative algorithms for ROBDD manipulation. Essentially, instead of the ROBDD operations being executed path-by-path, they are executed level-by-level where each level is associated with a specific variable index in the ROBDD. A direct side-effect of the BF approach is that the isomorphism check mentioned above cannot be done on the fly any more and it becomes necessary to temporarily generate redundant nodes. But the consequent overhead incurred by the generation of redundant nodes is small compared to the orders of magnitude of savings in run time resulting from the regular memory access pattern. A major and fundamental drawback of their algorithm is that "pad nodes" need to be added to the ROBDD so that successive nodes on any path in the new BDD differ in their index by exactly 1. Since successive nodes along a path in the original ROBDD can differ in their index by an arbitrary amount, it is likely that a large number of pad nodes may have to be added. We have implemented their algorithm and find that the pad nodes can increase the node count by multiple factors for many circuits. This drawback manifests itself in two ways: (1) Significantly increases the run time since the pad nodes are treated like the original nodes and must be fetched from memory. (2) Considerably limits the size of ROBDDs that can be built given an address space limit. We find that the pad node approach is an impractical solution for manipulating large ROBDDs.

Our contribution has been to propose a BF algorithm that avoids the need for pad nodes. The algorithm achieves this with a negligible penalty in CPU time, and an insignificant perturbation of the regular memory access pattern. Our experiments indicate that for some large industrial circuits with greater than 10K gates for

[1] For basic ROBDD related terminology and the recursive formulation of ROBDD manipulation, please refer to [1].

[2] In UNIX memory management, read and writes from secondary storage are always in units of one *page*. While the size of a page may depend on the environment, a page is usually a 4KB block of memory located at 4KB boundaries in UNIX on current processors.

```
bf_ite(F, G, H) {
1.  if (terminal case(F, G, H)) {
2.      return result ;
3.  } else {
4.      bf_ite_apply(F, G, H) ;
5.      R = bf_ite_reduce() ;
6.      return R;
7.  }
}
```

Figure 2: Outline of BF ROBDD Manipulation

which our algorithm finishes in about 1 hour of total elapsed time, our faithful implementation of the algorithm of Ochi *et al.* does not complete because it runs out of more than 1 Giga Bytes of secondary storage. On some circuits for which our algorithm finishes in about 10 minutes, the pad node approach requires many hours. Our algorithm runs faster than the pad node approach by multiple factors consistently for circuits on which both approaches finish. Our algorithm opens up the possibility of creating BDDs for very large portions of chips, something considered unviable until now. Our approach is machine independent and has been ported with no modifications to SPARC, SGI and NEC EWS based machines.

3 Basic Algorithm for BF ROBDD Manipulation

In this section, we first describe how ROBDD manipulation can be performed in a BF manner, and highlight the requirements (labeled as Problems 1, 2 and 3 in Sections 3.1 and 3.2) that need to be met in order to ensure locality in the memory access pattern. We then present our own solution and compare it to the approach of Ochi *et al.*[5].

The basic outline of an algorithm for the BF computation of ITE is shown in Figures 2, 3 and 4. The same code with minor modifications can be used to execute in a BF manner any Boolean operation with an arbitrary number of arguments. The first phase of the algorithm, the *apply* phase, is where the result BDD is created. The essential difference between the DF and BF approaches is that in the BF approach a new ITE computation request with some top-variable is issued before the final results of all previous ITE requests with the same top-variable are known. As a result, isomorphism check cannot be done on the fly, and the result BDD may contain redundant nodes. The second phase of the algorithm, the *reduce* phase, removes redundant nodes from the BDD and generates the final ROBDD. Let us analyze the memory access patterns generated during *apply* and *reduce*.

3.1 Memory Access Pattern During BF *Apply*

The basic operation during the *apply* phase (Figure 3) is the top down (from the root variable to the leaves) processing of outstanding requests to compute the ITE of ROBDD triples. In general, two new ITE requests are issued each time an ITE request is processed. The result for a new request is directly available if a terminal case is encountered. Otherwise, a new node is allocated for a new request if an identical request has not already been issued in the past. Processing an ITE request requires that the root node of each of its argument ROBDDs be fetched if the top variable of that ROBDD is the same as the top variable of the argument triple.

The essence of the BF algorithm is that the outstanding ITE requests are processed strictly in increasing[3] order of their top variable indices. This implies that the outstanding ITE requests that have the same top variable index are processed consecutively. In turn, this means that there is temporal locality in the access of the ROBDD

[3]We follow the convention that the variable index increases from the root to leaves.

```
bf_ite_apply(F, G, H) {
1. min_index = determine_top_variable_index(F, G, H) ;
2. R = new_bdd_node(min_index) ;
3. create_new_request(F, G, H, R) ;
4. add (F, G, H, R) to queue[min_index] ;
5. for (index = min_index; index ≤ max_index; index ++) {
6.    x is the variable corresponding to index index ;
7.    do {
8.       (F, G, H, R) = fetch next request from queue[index] ;
9.       if (terminal case(F_x, G_x, H_x)) {
10.         R→T = result ;
11.      } else {
12.         next_index = determine_top_variable_index(F_x, G_x, H_x) ;
13.         if (request corresponding to (F_x, G_x, H_x) already occurs
            in queue[next_index]) {
14.            fetch THEN node corresponding to (F_x, G_x, H_x)
               from queue[next_index] ;
15.            R→T = THEN ;
16.         } else {
17.            THEN = new_bdd_node(next_index) ;
18.            R→T = THEN ;
19.            create_new_request(F_x, G_x, H_x, THEN) ;
20.            add (F_x, G_x, H_x, THEN) to queue[next_index] ;
21.         }
22.      }
23.      if (terminal case(F_x̄, G_x̄, H_x̄)) {
24.         R→E = result ;
25.      } else {
26.         next_index = determine_top_variable_index(F_x̄, G_x̄, H_x̄) ;
27.         if (request corresponding to (F_x̄, G_x̄, H_x̄) already occurs
            in queue[next_index]) {
28.            fetch ELSE node corresponding to (F_x̄, G_x̄, H_x̄)
               from queue[next_index] ;
29.            R→E = ELSE ;
30.         } else {
31.            ELSE = new_bdd_node(next_index) ;
32.            R→E = ELSE ;
33.            create_new_request(F_x̄, G_x̄, H_x̄, ELSE) ;
34.            add (F_x̄, G_x̄, H_x̄, ELSE) to queue[next_index] ;
35.         }
36.      }
37.   } while (queue[index] is not empty) ;
38}
}
```

Figure 3: Outline of BF *Apply*

nodes corresponding to a given variable index. Now if we can ensure that the ROBDD nodes for each variable index are stored in contiguous locations in memory, the temporal locality translates to spatial locality. This ability to introduce spatial locality in the memory accesses during ROBDD creation is the fundamental reason for using the BF approach.

A levelized *request queue* enables the processing of outstanding ITE requests in appropriate order. One queue is created per variable index. Each time a new request is generated, it is placed in the appropriate queue corresponding to the top variable of its argument triple. Obviously, a new request can only be placed in a queue with index greater than the current index. The queues themselves are processed in the order of increasing variable index.

Two more critical problems must be resolved to ensure the absence of randomness in the access pattern.

3.1.1 The Problem of Computing Variable Indices

Problem (1) One issue is the computation of the top-variable index. Variable index computation represents a problem because the variable index associated with an ROBDD node is normally stored as an entry in the node structure itself. Consequently, the variable index for a node cannot be computed without fetching the node

```
bf_ite_reduce() {
1.  for (index = max_index; index ≥ min_index; index − −) {
2.      x is the variable corresponding to index index ;
3.      do {
4.          R = fetch next temporary node from queue[index] ;
5.          if (R→T has been forwarded to T′) {
6.              R→T = T′ ;
7.          }
8.          if (R→E has been forwarded to E′) {
9.              R→E = E′ ;
10.         }
11.         if (R→T equals R→E) {
12.             forward R to R→T ;
13.         } else if (unique_table[index] has
            entry (x, R→T, R→E)) {
14.             let R′ be the BDD node stored
                at the entry (x, R→T, R→E) ;
15.             forward R to R′ ;
16.         } else {
17.             add R to unique_table[index] under
                entry (x, R→T, R→E) ;
18.         }
20.     } while (queue[index] is not empty) ;
21. }
22. return either R or the node to which it is forwarded ;
}
```

Figure 4: Outline of BF *Reduce*

from memory. The problem manifests itself on Lines 8, 12 and 26 in Figure 3. On Line 8, the next request to be processed is fetched from the request queue. Say that x is the top-variable of the triple (F, G, H). We already know the top-variable index of the triple. But, the root nodes of the individual ROBDDs F, G, and H do not necessarily have the same variable index as x. If the variable index of the root node of, say, F is greater than that of x, then $F_x = F$ and $F_{\overline{x}} = F$, meaning that we don't need to fetch the node F from memory to obtain the cofactors. But in order to determine that, we had to compute the top variable index of F, and thereby had to fetch the node F from memory. Either way, the node for F gets fetched from memory. If the top variable index of F is greater than that of x, fetching it from memory corresponds to an out-of-order memory access.

On Lines 12 and 26, we need to determine the top-variable indices corresponding to the positive and negative cofactor triples, respectively, for the newly generated requests. We need to know the top-variable indices so that the new requests can be placed in the appropriate queues. The computation of the top-variable indices of these two triples is required because no relationship is imposed in an ROBDD between the indices of parent and child nodes except that the index of the child must be greater than that of the parent. In order to compute the indices, we must fetch each of the six cofactors in the two triples. Since these are fetches of nodes with indices greater than that of the current index, they represent out-of-order memory accesses. Ideally, we would like to be able to compute the two top-variable indices without out-of-order accesses.

3.1.2 The Problem of Checking for Duplicate Requests

Problem (2) The second critical issue to be resolved manifests itself on Lines 13 and 28 in Figure 3. It is concerned with accessing the queue associated with a newly issued request. Before a new request is issued, it must first be checked whether an identical request has been issued in the past. A table lookup in the appropriate queue with the index $next\_index$ is performed for this check. If there is a duplicate request in the queue, it must be fetched. If a duplicate request does not exist, a new request must be issued and inserted into the queue. There is no restriction on $next\_index$ except that it be greater than the current index $index$. In addition, there is no relationship between the top-variable indices for successively issued requests. This lack of relationship creates the potential for randomness in the memory access pattern here. Ideally, we would like the look ups into the queues to be done in the order of increasing index.

It is in the solution to the two above stated problems of computing the variable indices and checking for duplicate requests without introducing randomness in the memory accesses that our approach differs from that of Ochi *et al.* Our approach solves these two problems with a significantly lower penalty in terms of additional memory usage and at the expense of a negligible overhead in CPU time.

3.2 Memory Access Pattern During BF *Reduce*

The *reduce* phase (Figure 4) removes redundant nodes from the BDD by doing a bottom-up traversal of the BDD nodes. A redundant BDD node is a node with identical $THEN$ and $ELSE$ nodes, or a node such that another node with identical attributes already exists in the unique table. The corresponding checks are performed in Lines 11 and 13 in Figure 4. If a node is found to be redundant, it is forwarded to the node that should take its place. In terms of programming, an easy way (also suggested in [5]) to implement the forwarding is the following: Say that R is the node to be forwarded to R'. Set $R{\rightarrow}E$ to some predefined constant, and set $R{\rightarrow}T$ to R'. To determine if a node R has been forwarded, one first checks $R{\rightarrow}E$ for the predefined value.

As in the case of *apply*, the nodes to be processed are accessed from the levelized queue, but in the order of decreasing variable index. Therefore, if the nodes belonging to the same level are stored in contiguous memory locations, we have spatial locality of address when fetching these nodes. Even so, there is still potential for randomness in the memory access pattern as described below:

3.2.1 The Problem of Checking for Node Forwarding

Problem (3): The first step in the processing of a node, say R, involves checking if $R{\rightarrow}T$ and $R{\rightarrow}E$ have been forwarded (Lines 5 to 10 in Figure 4). If, say, $R{\rightarrow}T$ has been forwarded, then it must be reassigned to the node to which it has been forwarded. Given the way the forwarding of nodes is implemented (as indicated in the previous paragraph), checking if $R{\rightarrow}T$ has been forwarded requires that $R{\rightarrow}T$ be fetched from memory. Since the index for $R{\rightarrow}T$ can be arbitrarily greater than the index for R, and since there is no relationship between the variable indices of two nodes checked for forwarding one after the other, this fetch introduces randomness in the memory access pattern. This potential for random access must be removed if we don't want the performance of the algorithm to degrade rapidly once the BDD sizes reach a certain point. Ideally, all the checks for forwarding of nodes belonging to a given level should be done consecutively. As in the case of *apply*, our solution to this problem differs from the solution of Ochi *et al.*

3.3 The Pad Node Solution

The common reason that causes the Problems 1, 2 and 3 in Sections 3.1 and 3.2 is that the index of the child node can be arbitrarily greater than the index of the parent. In their solution, Ochi *et al.* [5] proposed that additional nodes be introduced in the ROBDD until the index of each child node is either exactly equal to one plus the index of its parent node, or the child node is a terminal node. The solution is simple but naive. In effect, the solution potentially increases the memory requirement by multiple factors in order to remove the randomness in memory access. In practice, the increased memory requirement nullifies the advantage of the regular memory access.

4 Our BF Approach

In this section, we demonstrate that orderly page access during BF ROBDD manipulation can be achieved using the basic BF algorithms outlined in Section 3 with a few enhancements and without

the need for pad nodes and associated overheads. The key ideas that make this possible are (1) a new way of determining the variable index of an ROBDD node (2) appropriate sorting of the requests and nodes to be processed at a given level during *apply* and *reduce*, respectively. These ideas are described in some detail below.

4.1 Determining the Variable Index of an ROBDD Node

We know from earlier sections that in order to ensure spatial locality in memory accesses, we must ensure that ROBDD nodes with the same top-variable index are stored in contiguous memory locations. To make this possible, the memory manager must be able to allocate memory in the form of appropriately sized blocks with each block being associated with a particular variable index. Memory for a new ROBDD node is allocated from within the block associated with the variable index of the node. An additional block is allocated for a variable index when all previously allocated blocks for that index are filled up.

A key side effect of such an organization of ROBDD nodes in memory is that given the address of a node, one can easily determine the block of memory to which it belongs and thereby also easily determine its variable index. Note that this way, the variable index of the node is determined directly from the address (pointer) of the node. The node itself does not need to be fetched from memory. This ability to determine the variable index without fetching the ROBDD node from memory enables us to solve Problem 1 described in Section 3.1.1, and hence removes the first bottleneck to ensuring orderly page access during BF manipulation.

Of course, this method of computing the variable index is not completely free of overhead. But we show in this section that the overhead is small enough that it can be neglected for all practical purposes.

Since the goal of the BF approach is to maximally utilize each page access, a block size of one page (4 KBytes on most current UNIX systems) is used. Note that since we can determine the variable index directly from the address of a node, we do not need the corresponding field in the ROBDD node structure any more. The remaining fields in the node structure are (1) REFERENCE_COUNT (2) THEN (3) ELSE (4) NEXT. The REFERENCE_COUNT field maintains a count of the fanins to the node, the THEN and ELSE fields store pointers to the THEN and ELSE nodes, respectively, and the NEXT field stores a pointer to another node with the same variable index and is used to maintain the unique table as described in [1]. Each of these fields is 4 bytes wide, making the total size of each ROBDD node equal to 16 bytes. A 4 KByte block would, therefore, accommodate 256 ROBDD nodes. Therefore, fetching a page from secondary storage puts into main memory 256 ROBDD nodes.

The variable index is computed in the following manner: Given a 32 bit address space (corresponding to 4 GBytes of maximum per process addressable memory as provided by most microprocessors), a 4 KByte block size implies that there can be at most 1 M blocks at any given time. In other words, the higher 20 bits of the address of a node determine the block to which the node belongs. The correspondence between a block and the variable index to which it corresponds is maintained by means of a table (call it the block-index table) with 1 M entries[4]. This table can be located anywhere with the restriction that the 1 M entries be contiguous. A table with only 1 M entries is small enough that it is relatively easy to find room for it. In addition, given its small size and the large number of times that variable indices need to be computed, the table is almost guaranteed to always remain in main memory and never get swapped out to secondary storage. To compute the variable index, the ROBDD node address is first shifted to the right until the 20 bits identifying the block occupy the appropriate positions. These shifted 20 bits are now used as an offset address to index into the block-index table to fetch the variable index. On a typical CPU architecture, the right shift requires one instruction, and adding an offset to the base address requires another instruction. Therefore, our approach requires two instructions in addition to the actual memory fetch.

[4]Each entry is a short integer corresponding to a variable index. It is, therefore, two bytes wide.

4.1.1 Overhead of Index Computation

How much more expensive is it to compute the variable index in this manner rather than by a pointer indirection assuming that the appropriate ROBDD node is already in main memory? In the pointer indirection method, if the index field is the first field in the ROBDD node structure, obtaining it would require a memory fetch with no offset computation. In such a case, our method requires two non-memory instructions more than the index determination by indirection. If the index field is not the first field in the node structure, then obtaining it by pointer indirection requires one instruction for adding an offset to the top address of the structure prior to the memory fetch. In this case, our method requires one instruction more than index computation by pointer indirection. Therefore, in the worst case, we need to pay a penalty of only two non-memory instructions to determine the variable index from the block-index table. Given that the latency of a memory fetch is much higher than the latency of a shift or an add instruction, the two additional instructions correspond to a very low real-time penalty in practice. What this means is that in creating small ROBDDs that fit completely in main memory, our BF approach will not be slowed down by our method of computing variable indices compared to the conventional depth-first approach, everything else being equal.

Now consider the penalty of our way of computing the variable index compared to the the BF approach using pad nodes. No index computation is required when using the pad node approach. We know that using the basic BF approach outlined in Figures 3 and 4, index computation is required at most 9 times[5] in each iteration of the core loop in the *apply* phase. The 9 index computations correspond to 20 additional non-memory instructions and 9 additional memory fetches (from the block-index table) which are practically guaranteed to be from main memory. These 27 additional instructions for index computation correspond to an insignificant fraction of the total number of instructions for the rest of the complete loop. This leads us to conclude that even discounting the overhead of using additional nodes for padding, computing the variable indices from the block-index table would result in an insignificant run time penalty compared to the pad node method.

4.2 Sorted Processing of Requests During *Apply*

The next problem to be resolved is the bottleneck associated with checking for duplicate requests during *apply* (Problem 2 in Section 3.1.2). Given a queue of requests to process at the current level, our goal is to remove the randomness in page access. In order to achieve this we process the requests at the current level in the order of increasing variable indices of the two new requests that are issued from each of them. This is done in the following manner: In the first pass through the current request queue, requests such that the new requests issued from them belong to the level immediately below are processed immediately. Other requests are stored in an array of lists, with each list corresponding to a level below the current level[6]. After the first pass is complete, the requests in the lists in the newly created array are processed in the order of increasing level. Note that since there is no relationship even between the top-variable indices of the two new requests issued from the same request, a request may appear in two lists at the same time, one for the new request corresponding to the positive cofactor, and the other for the negative cofactor. Processing the requests in this manner ensures that all the look ups into a particular queue are done consecutively, thereby removing the randomness.

The randomness is removed at the cost of doing more than one pass through the current request queue. Even so, the effective number of passes required is only some number between 1 and 2 depending on the number of requests that get processed in the first pass itself. Also, the new array of lists is at most of size equal to the number of levels, and is therefore, very small. The creation of a new list per level does not cost any memory since the requests were already in a list before (the list corresponding to

[5]In Lines 8, 12 and 26 of Figure 3

[6]Basically, this corresponds to a counting sort.

the queue to which they belong[7]). They just need to be removed from the original list and put in a new list. A request may need to be duplicated if the top-variable indices of the two new requests issued from it (corresponding to the positive and negative cofactors) are different. The duplication is required since the request must be placed simultaneously in the two lists corresponding to the two top-variable indices. In spite of the potential for some duplicate requests, this approach is superior to the pad node approach where, effectively, n copies of a request are created if the *actual* top-variable index of a newly created request is n levels below the current level.

4.3 Sorted Processing of Nodes During *Reduce*

The final problem to be resolved is the potential for random page-access during the check for forwarded nodes during *reduce* (Problem 3 in Section 3.2.1). This problem and the solution to it are analogous to the case of checking for duplicate requests during *apply*. This problem arises because of the lack of any relationship in the indices of the nodes that are successively checked for forwarding. As in the previous section, we use more than one pass, and maintain an array of lists (one list for each level below the current level) for nodes to be processed in the second pass. $R \rightarrow T$ (*c.f.* Section 3.2.1) is processed during the first pass only if the level of $R \rightarrow T$ is immediately below the current level. Otherwise, R is placed in the list corresponding to the level of $R \rightarrow T$. Similarly for $R \rightarrow E$. As in the previous section, R must effectively be placed in two lists if the levels of $R \rightarrow T$ and $R \rightarrow E$ are different. With this approach, all checks for forwarding of nodes belonging to a given level get done consecutively. Therefore, there is no randomness in the page-access pattern.

The costs associated with this solution are similar to the costs associated with avoiding random access when checking for duplicate nodes. Again, the array size is very small and no extra memory is needed for the new lists. The reason no new memory is needed is that the ROBDD node structure already has a NEXT field to be used to maintain the lists in the unique table. Since a node is not placed in the unique table until it has been processed during *reduce*, the NEXT field can be used to maintain the desired lists.

4.4 Adaptive Garbage Collection

Garbage collection should serve two purposes: (1) free up memory for subsequent use and (2) prevent fragmentation of the memory used by a single ROBDD. We use an adaptive scheme, consisting of a combination of two well known garbage collection schemes, to realize both these goals. Our scheme is described below. A number of Boolean operations must be performed before the ROBDDs for the primary outputs are created. Dead nodes are created as a result of the freeing of these intermediate ROBDDs as well as by the freeing of redundant nodes. The space occupied by the dead nodes can be reclaimed for use by new nodes. An effective scheme for reclaiming the memory used by dead nodes is called the *reference-count* garbage collection strategy. In this strategy, one maintains a list of nodes (called a FREE_LIST) that can be reused. When a new node is to be allocated, one first checks the FREE_LIST for available nodes. If a dead node is available, it is removed from the FREE_LIST and reused. A new node is allocated if no dead node is available. Given the levelized organization of nodes in memory in our algorithm, we maintain a separate FREE_LIST for each level. In order to be able to identify dead nodes, we maintain a REFERENCE_COUNT field in the node data structure. The REFERENCE_COUNT field maintains the number of nodes that refer to this particular node. A node is considered dead when its REFERENCE_COUNT becomes zero. Once a node is declared dead, the REFERENCE_COUNT fields of its two children must be decremented by one. Therefore, labeling a node to be dead has a potentially cascading effect down the ROBDD. The task of the *reference-count* garbage collector is to traverse the ROBDDs top-down, and if their REFERENCE_COUNT is zero then mark them dead and decrement the REFERENCE_COUNT fields of their children. As in the case of *apply* and *reduce*, we must do this traversal in a levelized manner to avoid random page-access. The same strategy of using levelized queues as in *apply* and *reduce* is used for the purpose. The queue at a level consists of the dead nodes at that level and the nodes whose REFERENCE_COUNT is to be decremented. Also, the same strategy of sorted traversal described in Sections 4.2 and 4.3 is used here to ensure complete removal of randomness in page access. The *reference-count* garbage collector is called at periodic intervals, *e.g.* every time the number of nodes doubles. The advantage of *reference-count* is that it is very fast.

A potential problem with the *reference-count* scheme is that it can lead to a fragmentation of the memory used by a particular ROBDD. Consider the scenario where *reference-count* frees up 20% of the nodes at a given level. These 20% of the nodes can potentially be scattered over a number of pages allocated for that level. When a new ROBDD is created, it must first use up these dead nodes scattered all over before it can allocate new nodes for that level. Therefore, spatial locality is partially lost for 20% of the nodes at that level in the new ROBDD. In order to avoid such a possibility, we use the *reference-count* scheme in combination with another scheme called the *stop-and-copy* garbage collection scheme. The *stop-and-copy* scheme copies to a new address space only those nodes that are alive. The space that was being used before the copy is then completely discarded. This scheme serves to remove the fragmentation caused by the *reference-count* scheme. But, since it must copy all the nodes that are alive, it is potentially much slower than *reference-count*. Therefore, it is called only when *reference-count* produces a large number of dead nodes. In our implementation, the *stop-and-copy* scheme is only called for those levels with a large number of dead nodes, and not for all the levels.

A similar adaptive strategy was also used in [5].

4.5 Implementation Details

4.5.1 ITE-Request Data Structure

The ITE request structure requires five fields: the arguments F, G, and H, the result node R, and the NEXT field used for maintaining lists. If we used a separate type for the ITE requests, we would need to allocate 20 additional bytes for every BDD node allocated during *apply*. These additional bytes would stay alive until the request is processed. We avoid this penalty by using the BDD node structure for the ITE requests also and overloading the meanings of the various fields. The REFERENCE_COUNT field maps to F, THEN to G, ELSE to H, NEXT to NEXT, and the BDD node itself is the result field R. The same 16 bytes are, therefore, initially used for the ITE request and subsequently for the BDD node.

4.5.2 Block-Index Table

With an address space of 32 bits and a page size of 4 KB, the block-index table would have at most 1 M entries. This number is small enough that a flat statically allocated table can be constructed easily. That is what we do in our implementation. For future machines with 64 bits of address space, a flat table would be infeasible. One solution would be to make the table hierarchical in nature, allowing the memory used by the table to be increased dynamically. The penalty would be increased overhead since more intstructions would be required for index computation.

5 Experimental Results

We wish to illustrate the following points concerning our BF approach in this section: (1) It is orders of magnitude faster than DF implementations when BDD sizes exceed main memory. (2) Its overhead compared to DF implementations for ROBDDs that fit in main memory is manageable. (3) It is superior to the pad node approach in terms of memory requirement and run time. It should be noted that we are not addressing the variable-ordering problem for ROBDDs here. In fact, good variable orderings exist for all the circuits for which we report results[6]. The comparisons between the BF aproach and the other approaches are for a specific ordering. Most of the circuits used for our experiments are from the IWLS '93 benchmark set. The two circuits `Indust1` and `Indust2` are from the industry.

[7] Therefore, the requests already have a NEXT field in their structure which can be used to point to the next request in a list

Circuits	# I/O/G	# Nodes $\times 10^6$	Machine Config.	ET
C2670	233/140/1161	2.53	64MB sparc 10/41	0:23:34
			128MB sparc 2	0:19:21
C3540	50/22/1667	4.22	64MB sparc 10/41	0:20:56
			128MB sparc 2	0:11:01
C6288-X	32/32/2416	20.2	64MB sparc 10/41	9:20:26
s9234	247/250/5597	3.85	64MB sparc 10/41	0:06:01
			32MB sparc 2	0:24:43
			128MB sparc 2	0:06:38
s38417-X	1664/1742/22397	103.8	64MB sparc 10/41	26:07:06
Indust1	1133/1106/21698	5.37	64MB sparc 10/41	0:19:23
			128MB sparc 2	0:20:58
Indust2-X	2131/2304/42617	29.6	64MB sparc 10/41	9:20:24

I/O/G: Primary Inputs/Primary Outputs/Gates
ET: Total Elapsed Time (hours:minutes:seconds)
For the circuits suffixed with "-X", ROBDDs were only built for a subset of the gates -
C6288: for the first 1229 gates in dfs;
s38417: for the first 13933 gates in dfs;
Indust2: for the first 7509 gates in dfs

Table 1: Results of our BF Approach for Large ROBDDs

Circuits	# I/O/G	# Nodes	Elapsed Time (s)		
			Ours	DF	PN
C1355	41/32/514	175925	13	8.8	26
C1908	33/25/880	42929	6.1	2.9	17
C432	36/7/160	146384	13	9.7	35
C499	41/32/202	55493	10	6.5	20
C5315	178/123/2290	59912	8	2.4	63
C880	60/26/357	26032	1	0.9	7
s13207	700/790/8027	28594	14	2.7	349
s1423	91/79/657	55572	3	2	22
s15850	611/684/597	136330	21	7.1	618
s35932	1763/2048/16353	26681	61	5.3	>> 1 hr
s38584	1464/1730/19407	108494	61	8.3	>> 1 hr

I/O/G: Primary Inputs/Primary Outputs/Gates; PN: Pad Node

Table 2: Results of our BF Approach for Small ROBDDs

What is the overhead of our BF approach compared to the current DF implementations? The elapsed times on a SPARC 10/41 for circuits whose ROBDDs fit in main memory are provided in Table 2. The comparison is with a current generation DF ROBDD package [7], and with our fair implementation of the pad node approach of [5]. Many of the ROBDDs are too small for the comparison to be meaningful and we only provide comparisons for circuits with greater than 10000 nodes. It can be seen that for these small ROBDDs, our approach is consistently faster than the pad node approach indicating a lower overhead. For some examples, our overhead is markedly smaller than the pad node approach. The numbers also demonstrate that our approach does have an overhead compared to the DF approach for these small ROBDDs, but that the overhead is not inordinately large unlike the pad node approach. Given that and the fact that the absolute elapsed times involved for these small ROBDDs are of the order of tens of seconds, we feel that the overhead of our approach for small ROBDDs is a reasonable penalty to pay for optimizing the times for large ROBDDs.

Elapsed times for large ROBDDs are provided in Table 1. The numbers clearly demonstrate the ability of our BF approach to build and manipulate very large ROBDDs in short run times. For example, the 3.85 million node ROBDDs for `s9234` were built on a SPARC 2 with only 32 MB of main memory in about 25 minutes. To make the same ROBDDs using the current generation ROBDD package from CMU[3] requires about 48 hours. That corresponds to a speed up by a factor of 120. Similarly, we can build ROBDDs with about 104 million nodes for the first 7509 gates of `s38417` in 26 hours on a SPARC 10/41 with 64 MB. The CMU ROBDD package can only build ROBDDs with 7.8 million nodes for the first 4807 gates in 43 hours on the same machine. The BF approach is faster by a factor of about 40 for the first 4807 gates. Similar speed up is obtained consistently when the ROBDD sizes are much greater than the available main memory.[8] For our experiments, the orderings were generated using the ordering algorithm of [4] implemented in SIS.

Finally, as in the case of the small ROBDDs, our approach consistently outperforms the pad node approach for large ROBDDs also. For example, the pad node approach quickly exhausted the 2 GB of available swap space for `Indust1` and `Indust2` as a result of the pad node overhead. 51 times more nodes were required to pad the ROBDDs for `Indust1` before exhausting the swap space. Such overheads are very common using the pad node approach for random-logic circuits. In addition, whenever the pad node approaches manages to build the ROBDDs for a circuit, it is much slower than our approach. For example, our approach requires 21 minutes on a 64MB sparc 10/41 while the pad node approach requires more than 3.5 hours.

6 Acknowledgments

Various discussions with Rick Rudell were helpful.

References

[1] K. Brace, R. Rudell, and R. Bryant. An efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, June 1990.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[3] D. Long. *ROBDD Package*. Carnegie Mellon University, 1993.

[4] S. Malik, A. R. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–9, November 1988.

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

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

[7] R. Rudell. *Personal Communication.* 1994.

[8] Unfortunately, we do not have available at this time the run times using the DF ROBDD package of Rudell on the same computing platform on which we have run our experiments for the larger ROBDDs. We do have the run times for `C2670`, `C3540` and `s9234` using that ROBDD package on a SPARC 10/41 with 128 MB of main memory. They are, respectively, 0:04:08, 0:05:18, and 0:03:58. These numbers, however, are not useful for comparison because the ROBDDs for all the 3 circuits can completely reside in 128MB of main memory.