RC220000 (98829) 19 March 2001 **Computer Science**

IBM Research Report

Design of a Pointerless BDD Package

Geert Janssen

IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

Design of a Pointerless BDD Package

Submitted to 10th Intl. Workshop on Logic & Synthesis
Granlibakken, Lake Tahoe, CA, June 12−15, 2001

Geert Janssen
IBM T.J. Watson Research Center
geert@watson.ibm.com

Abstract

In this paper a pointerless BDD package is proposed. The new design is inspired by the 1998 ICCAD paper of David Long [Long98]. The main idea is to enforce a strict ordering on the BDD node identifiers and cleverly reap the advantageous consequences of that decision, such as a better memory locality of the nodes and faster unique table lookup. The down side is a more complicated garbage collection scheme, although it offers some extra flexibility. It will also be shown how dynamic variable ordering fits into this new context. The ultimate goal is to exceed the performance of a pointer−based package and have reproducible results across different platforms.

Introduction

BDDs are effectively deployed in many EDA tools, in particular in the area of formal verification. A plethora of public domain BDD packages are available on the web [bdd−portal]. In this paper we propose a new design for a BDD package in the programming language C that uses *indices* (non−negative integers) to identify the nodes. This fact in itself is already a deviation from Long's paper; he still insists on using pointers and also defines his node ordering based on the addresses of the memory where the nodes reside. Using indices we obtain better control over node allocation and can easily achieve platform independence.

In the following sections we shall detail our decisions and discuss their consequences w.r.t. the main data structures: the unique table and the computed table. Next we show how we implement garbage collection and do dynamic variable ordering. At the end you will find some experimental results and conclusions. First we briefly mention related work.

Related work

As stated before, the main thrust to investigate a new design for a BDD package was the publication by David Long [Long98]. Long introduces the notion of node−age and rearranges his node allocation and garbage collection accordingly. He uses pointers to identify nodes and therefore runs into problems when the memory allocator issues out−of−order blocks. His reference objects (our *handles*) use a hash table to ensure uniqueness [Long99]; in our approach no intermediate data structure is necessary. We use a slightly different garbage collection algorithm. Long's paper does not address dynamic variable ordering. The BDDs as implemented in earlier versions of SMV [McMillan93], do not use node reference counts; a mark−sweep garbage collector is used. Variable reordering is using Rudell's algorithm [Rudell] but does excessive BDD traversals to calculate accurate live node counts because no explicit reference counts are used. Armin Biere's package called ABCD [Biere] focusses on compactness of representation (only 64 bits per node) and uses indices to achieve this. It does not take advantage of the node order and does not offer dynamic variable ordering. Also its memory management is rather rigid.

Preliminaries

We assume the common terminology of BDDs to be known [Bryant]: Boolean functions are canonically represented by a multi−rooted directed acyclic graph (DAG); nodes are kept in a unique table; operations are sped up by employing a computed table. Occasionally, the order of variables may be changed to reduce the total number of nodes. We furthermore assume the reader to be familiar with the "classical" way of implementing a BDD package in an imperative programming language like C, see e.g. [Brace]: nodes are C structs that contain a variable identifier and *THEN* and *ELSE* children pointers; a *NEXT* pointer strings nodes together that belong to the same collision chain in the unique table. Recycling of nodes is easily implemented by keeping a reference count for each node.

In the following we will discard most of the

classical implementation decisions and start afresh with a new set of requirements. Why would one want to do this? What's wrong with the current BDD packages? For one (and this is a very impor−tant point in many commercial applications), a pointer based BDD package almost invariably will use hashing on pointer values. This means that even on the same machine, a repeated invocation of the same program using such a package need not exhibit exactly the same behavior: nodes get allocated to different addresses, these pointers get hashed to different table indices, this causes a different pattern of hits and misses in the computed table, and ultimately the usage of nodes and the sizes of the resultant BDDs might differ because of (presumably unpredictable) calls to garbage collec−tions and dynamic variable orderings that are triggered by heuristics depending on node usage. On a different machine, the behavior might even get more erratic because of different memory alignment requirements, different direction of run−time stack growth, et cetera. All in all, it could be said that debugging such a BDD package is a programmer's nightmare.

A second important reason to deviate from a pointer based implementation is the opportunity to uniquely identify nodes by (consecutive) integer numbers and thereby achieve better control over node allocation. Being able to control the assignment of numbers to identify nodes, naturally imposes a strict order on the nodes. This order can be used to our advantage: keep the DAG ordered like a priority queue, hence searching for a particular node will obviously benefit; keep collision chains ordered, then on average less time is spent searching. Since BDDs are typically constructed bottom−up in a recursive depth−first fashion and since traversal of BDDs is a common operation that proceeds in a similar fashion, it pays off to keep adjacent (parent and children) nodes close in memory as well.

Ground rules for the new package

Clearly, our first rule should be: no more pointers. We identify a node by a number and the easiest is of course to let this number coincide with the index in the memory array where the node resides. We assume the availability of a single, contiguous array of nodes. Conceptually this is the easiest way to implement the node memory; in reality we find that repeatedly having to re−allocate such a large contiguous array is not feasible. Unlike David Long's proposal, we decided against a complicated scheme of memory blocks, but opted for a paging oriented solution. Consequently, when the need for more nodes arises, we only have to allocate a much smaller stretch of nodes. Any reference to a BDD node will be done by means of its index: the *THEN*, *ELSE*, and *NEXT* fields of a node are therefore also indices.

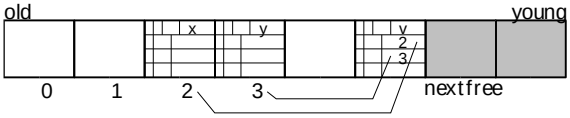

Figure 1: Node memory and the concept of node age.

We will insist that nodes be handed out in order of increasing indices, so that the newest (or youngest) node has the largest index. Think of the index of a node as its date of birth, then a young node has a recent date of birth. In the BDD DAG, where we naturally allocate the parent node later than the children nodes, we will have the counterintuitive situation that a parent node is younger than its two children (Figure 1).

Complemented edges are handled in a way that is similar to what has been used in pointer−based packages. We define complemented edges by merely flipping the most significant bit of the 32−bit unsigned integer index value of a node.

Complemented edges are of course not a necessity, but, as Fabio Somenzi [Somenzi01] pointed out, they do offer more advantages than just a simple way of representing the negated function and a trivial implementation for the logical *not* operation. Indeed, because we are able to do complementation and test for complement both in constant time, we can define more bottom cases to speed−up other operations as well, e.g. the *and* operation can now test for complemented operands and immediately return the 0 (zero/false) BDD. Rules of DeMorgan can be used at negligible cost to reduce the number of distinct cases that need be implemented. Also, some applications rely on the fact that the root nodes of a pair of complemented BDDs are in fact the same and in this way share certain attributes.

The next departure from the classical approach is to refrain from using reference counts. They would not be of much use anyway because dead nodes i.e., nodes with a 0 reference count, cannot be (directly) reused when we require node indices to obey the age requirement. The consequences are rather severe: we no longer have a precise notion of

whether a node is alive or dead. It will be harder to find useful metrics to be used in heuristics that control invocation of garbage collection and dy−namic variable ordering calls. When implementing dynamic variable ordering based on local level swaps, it is crucial to have a precise measure for the number of nodes gained by that swap. More on this in a subsequent section.

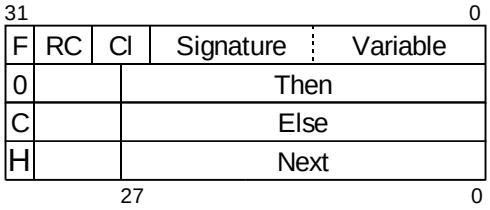

Figure 2: Memory layout of a BDD node.

Not using reference counts obviously saves space in the node (typically 16 bits are used for a reference count) and saves time: directly, because no increments and decrements need be performed; and indirectly, because the referred node itself need not be accessed and therefore we have less moves across the memory hierarchy. Actually, in some model−checking benchmarks [Yang98] it was observed that it often happens that one and the same BDD is repeatedly constructed and immediately disposed off. It was the reason of the invention of a so−called *death row* data structure. Its purpose is to queue BDDs that are candidates to be freed. The actual freeing is postponed in the hope that they might get resurrected and put to use again before the queue is full and the real freeing kicks in. Without reference counts, there is no longer a need to worry about this behavior. Figure 2 outlines the various fields in the BDD node.

In order for garbage collection to be meaningful, we do need to have a means of gathering all BDD nodes currently referenced by an application. These external references are the starting points of a node marking scheme that identifies all live nodes. Such external references are captured by *handle* objects. A handle implements the external view of a BDD. For simplicity, handles will have reference counts and therefore can easily be recycled. It is expected that the number of live handles at any time will be much less than the current number of live BDD nodes. Handles are to be interpreted as the Boolean functions of interest, and each function typically comprises many BDD nodes. Obviously, a handle should be unique w.r.t. the node it refers to. Handles are the ideal place to cache additional data about the Boolean function, e.g. its size and its set of support variables.

With the ground rules laid down, it is now time to explore their consequences. The next sections discuss the impact our new requirements have on the rest of the framework.

Unique table

Strong canonicity is achieved by ensuring that the triple $<v,T,E>$ is associated with a unique node. The triple $<v,T,E>$ consists of the variable identifier v and the edges to the *THEN* and *NEXT* sub−BDDs which themselves are assumed to consist of unique nodes. Of course in our case, T and E are the (possibly complemented) indices of the children nodes. A simple and fast implementation applies a hash function to the triple to obtain the index in the unique table of the start of a collision chain of nodes. Comparing the triple against the nodes in the chain resolves the look up. Note that instead of hashing on the children indices, hashing on a not necessarily unique signature kept in a node would work just as well, provided we do search for matching children indices in the collision chain. The reason why we would want to do this, is to make the hash key independent of the memory position of the node; when garbage collection moves the node in memory, at least its signature would stay the same. However, since we empty and rebuilt the unique table during garbage collection, there's no real need to assure hash key independ−ence. This doesn't hold for the computed table though, as we will see shortly.

Computed table

The computed table, a.k.a. operation cache, records results of operations on BDDs. Typically it stores the fact that $R = op(F,G,H)$, i.e., the BDD R is the result of the operation *op* applied to three BDD arguments F, G, and H, in a hash table. The hash key should obviously be composed of the operation and its operands. But should we use the indices of the operand BDDs? If we did, and garbage collection alters indices, we need to rehash those entries in the computed table. If the hash key were independent of the indices, then we merely need to invalidate entries that refer to dead nodes. Like David Long does, we decided to define a signature for each node which consists of the variable associated with that node and a pseudo random number (whose sequence is of course

deterministic). Computed table hashing takes the signature of the operands and their complement bit as a key. Note that incorporating the complement bit is vital: it is easy to construct a case involving the calculation of the *and* of two parity functions which exhibits exponential behavior if the complement bit is left out.

2−bit reference count

Proper utilization of the computed table is vital for efficiency. One should avoid storing facts that are either trivial (typically these concern operations where some of the operands are constant) or facts that are very unlikely to ever be retrieved. The negative effect of storing too much is that crucial facts might get overwritten. Facts concerning operands with low, in particular a single, reference count should be ignored. Unfortunately, in our new design we no longer have a reference count per node. Instead, we reintroduce a 2−bit saturating reference count with the sole purpose of indicating whether a node has no, a single, or more references. It is not necessary to keep this count accurate (we don't even bother to decrement the count when nodes are freed).

Garbage collection

Most BDD packages use a reference counting garbage collector. The idea is simple: if we keep count of the number of references to each BDD node, then a count of 0 means that the node is no longer in use and thus can be recycled. Often the recycling does not occur immediately. The dead node would have to be unlinked from the unique table and the entries in the computed table referring to this node would have to be invalidated; both of which is considered prohibitively costly. Instead, occasionally, when a certain percentage of the nodes is dead, the unique table is swept and all dead nodes are moved to a free list. The disadvantage of reference counting is the extra memory needed for each node and the overhead in increment and decrement operations. The great advantage is that at any time we have an accurate knowledge of the number of references for each node.

In a mark−sweep garbage collector, a marking phase that identifies all live nodes is proceeded by a sweeping phase that cleans up the garbage (= dead) nodes. A prerequisite for this to work is that we know all external references to the nodes. Also, if we consider garbage collection as if it were an asynchronous process (which it would be if we decided to run it as a separate thread), every intermediate BDD that occurs during a (recursive) operation needs to be explicitly protected. In our new design, there are actually two reasons why certain local variables in the program code need to be protected:

1. The obvious reason: if we do not protect a variable that holds an index to a BDD node that will be needed later, that node might be considered garbage by the collector.
2. The not so obvious reason: our garbage collector compacts the node memory and hence nodes get moved and their indices change. We need a mechanism to be able to report back the "change of address" of a node.

```
void gc(void)
{
  mark_phase();
  new_nextfree = 0;
  for (i = 0; i < nextfree; i++)
    if (marked(i)) {
      n = new_nextfree++;
      THEN(i) = FORWARD(THEN(i));
      ELSE(i) = FORWARD(ELSE(i));
      mem[n] = mem[i], but retain
      n's mark and forwarding address;
      FORWARD(i) = n;
    }
  invalidate_computed_table();
  update_phase();
  for (i = 0; i < new_nextfree; i++) {
    unmark(i);
    add i to unique table;
  }
}
```

Algorithm 1: mark−sweep−update−mark garbage collection.

We use a *mark−sweep−update−sweep* approach: the marking phase does the obvious; in the first sweep over the node memory, from lower indices to higher indices, live nodes are assigned a forwarding address at their original position and are then moved over to the left to fill the holes created by the dead nodes. At the same time, the *THEN* and *ELSE* fields are updated to reflect the new positions of the children nodes. Note that in the node memory the direction of the "pointers" *THEN*, *ELSE*, and *NEXT* is always from right to left (because of the node−age rule). During the update phase, all external references and all explicitly protected internal references are notified of any change of address. Then the second sweep phase cleans the marks and reestablishes the unique table. See Algorithm 1 for more details. The way we do

compaction has the nice property that it preserves node−age, moreover it even improves node proximity, see Figure 3. There is no obvious need to always consider all nodes for garbage collection; we allow a user to set a breakpoint below which nodes will be frozen. This might be beneficial in those applications that keep some initial BDDs around for most of their life−time, e.g. in model checking one could set the breakpoint after the next−state function BDDs.

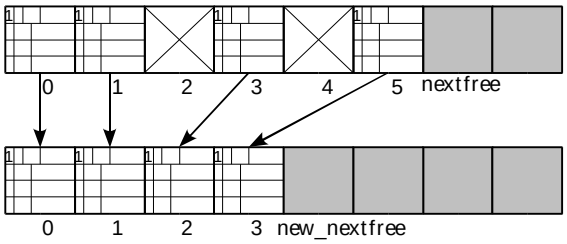

Figure 3: Node memory during garbage collection.

Dynamic variable ordering

Dynamic variable ordering is the process of changing the rank order of the variables in the presence of BDDs. This implies that all existing BDDs need to be modified to reflect the new variable positions. The objective is to decrease the overall size of the BDDs. Like garbage collection, dynamic variable ordering should best be considered an asynchronous process that potentially can be invoked at any time during the construction and manipulation of BDDs. We will here sketch how the popular Rudell sifting algorithm [Rudell] is implemented in our new package. We concentrate on the basic operation of swapping two neighboring variables. Clearly, exchanging parent and children nodes destroys the node−age property. Moreover, we need an accurate measure of the number of nodes gained (or lost) by a swap. We propose the following solutions:

- We do not rely on node−age during dynamic variable ordering; we will carefully rebuilt the BDDs afterwards to reestablish this invariant. This also means that the unique table may not assume a certain order of the nodes in its collision chains.
- All nodes are taken out of the original unique table and stored in linked lists per variable (using the *NEXT* field). The lists are stored in a table indexed by rank numbers.
- We reintroduce full−fledged reference counts for the nodes. The reference count can conveniently be stored in the space for the variable identifier.

```
void swap_levels(k, k+1)
{
  for (n in level[k+1]) mark(n);
  for (n in level[k])
    if(!marked(THEN(n))&&!marked(ELSE(n))
      move n to unique table;
  for (n in level[k]) {
    T = THEN(n); E = ELSE(n);
    n11 = marked(T)? THEN(T): T;
    n10 = marked(T)? ELSE(T): T;
    n01 = marked(E)? THEN(E): E;
    n00 = marked(E)? ELSE(E): E;
    THEN(n) = ut_lookup(vk,n11,n01);
    ELSE(n) = ut_lookup(vk,n10,n00);
  }
  for (n in level[k+1]) {
    unmark(n); remove n from level[k+1];
    if (dead(n))
      put n on freelist;
    else
      add n to level[k];
  }
  for (n in unique table)
    remove from unique table,
    add to level[k+1];
  swap(rank(vk), rank(vk+1));
}
```

Algorithm 2: Swapping of 2 neighboring variables.

In our approach we distinguish 3 phases: 1) the prologue that prepares the rank table of node lists and appropriate reference counts, 2) the processing phase that does the actual level swaps, and 3) the epilogue that reestablishes the node−age invariant and unique table. The processing phase is not much different from an implementation that uses separate unique tables per variable, except that it operates on the rank table. Algorithm 2 outlines the processing of the lists of two neighboring levels to establish a variable swap.

Note that during a level swap the unique table only stores nodes labeled with the same variable. We do not explicitly need to hash on the variable and need not store it.

Managers

All major data structures are defined relative to a manager record. In this way multiple invocations of the BDD package can coexist. Since handles are not pointers but indices into some table that is owned by a manager, we are faced with the problem of associating a manager with each handle. A simple solution is to encode a manager index (of course) into the object that holds the handle index. For now we opted for a 3−bit index allowing for 8

managers. Should the need arise to enlarge the number of managers we could always resort to 64−bit handle objects.

Experiments

To be supplied.

Acknowledgments

Thanks go to Thomas Kutzschebauch for carefully proofreading the paper. Jessie Xu should be mentioned for all her efforts in integrating the new BDD package in the Verity™ verification tool and supplying valuable feedback.

References

[bdd−portal]: Meinel, Ch. Wagner, A., www.bdd−portal.org, http://www.bdd−portal.org, 2000

[Biere]: Biere, Armin, ABCD: a compact BDD library, http://www.inf.ethz.ch/personal/biere/projects/abc, 2000

[Brace]: K. S. Brace, R. L. Rudell, R. E. Bryant, Efficient Implementation of a BDD Package, Proc. 27th DAC, 40−45, 1990

[Bryant]: R. E. Bryant, Graph−Based Algorithms for Boolean Function Manipulation, IEEE Trans. on Computers, 677−691, 1986

[Long98]: D. E. Long, The Design of a Cache−Friendly BDD Library, Proc. ICCAD, 639−645, 1998

[Long99]: D. E. Long, private (email) communication, 1999

[McMillan93]: K. L. McMillan, Symbolic Model Checking, 1993

[Rudell]: R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, Proc. ICCAD, 42−47, 1993

[Somenzi01]: F. Somenzi, Efficient Manipulation of Decision Diagrams, STTT, , 2001

[Yang98]: Bwolen Yang, et al., A Study of BDD Performance in Model Checking, Formal Methods in Computer Aided Design, 255−289, 1998