The Design of a Cache-Friendly BDD Library

David E. Long
Bell Laboratories, Lucent Technologies

Abstract
We describe the architecture for a new BDD library that is designed to be cache-friendly. The library incorporates a novel technique for terminating searches early during find operations together with a regrouping garbage collector. These features lead to a factor of two improvement in speed on typical examples compared to existing libraries.

1 Introduction
We describe a new BDD [3, 4] library that offers several improvements compared to existing implementations [8, 11, 12].

Better locality We propose a novel scheme for reducing the number of nodes searched during each find operation. This is combined with a garbage collector that packs related nodes into nearby addresses. These improvements result in significantly greater locality of reference, which translates into fewer data cache misses and significantly improved performance on modern machines.

Machine independence Address-dependent hashing is avoided during operation cache manipulations, leading to machine-independent results even in the presence of reordering [10].

Generational garbage collection Groups of long-lived BDDs (such as the transition functions in a state space search application) are automatically recognized as non-garbage and are not swept by the garbage collector.

We implemented a library based on these ideas and tested it by building BDDs for the standard combinational benchmark circuits. The new library is typically about twice as fast as existing implementations [8, 11, 12]. We assume that the reader is familiar with standard depth-first BDD implementation techniques [2].

2 Data Cache Miss Patterns
With today’s high speed CPUs, the biggest performance bottleneck in BDD packages is the long latency of main memory. Thus, the main architectural decisions for the new library are motivated by the desire to be as cache-friendly as possible. To understand these decisions, we begin by examining the data cache miss (DCM) pattern of a typical BDD operation (assuming a depth-first implementation).

Base cases Testing for base cases usually requires no DCMs.

Operation cache lookup Computing the location of the relevant operation cache entry does not require accessing the operands if the hashing is based on memory address. However, making the library machine-independent is problematic in this case. Since access to the fields of the operands is required after an operation cache miss, using these fields to compute an address-independent hash value will not cause many additional DCMs. However, the operation cache lookup itself will almost certainly cause a DCM.

Cofactoring When the cache lookup fails, cofactoring requires access to the fields of the operands. The first time an operand node is encountered, this will usually cause a DCM.
Finding the result node  A find operation requires checking to see if a node with given index and children exists. If not, the node must be created. Each find requires a unique table lookup. The hash function here can be address-dependent without affecting externally visible machine independence, so the fields of the children need not be accessed. However, the nodes in the unique table with the same hash key must be accessed to determine whether they match the desired node. Each of these accesses is usually a DCM.

Operation cache insert  Insertion of the operands and result into the operation cache involves only writes, and hence does not stall the processor on many machines even if the operation cache entry is not in the processor’s cache.

To summarize, we have one DCM for the operation cache lookup, about one DCM per operand node as the operands are traversed, and potentially several DCMs per find.

3  The Unique Table

There are some immediate possibilities for reducing DCMs due to finds. One is to make the unique table large to reduce hash collisions, but this is wasteful of memory and requires more frequent rehashing of the unique table. Instead, we make use of the observation that the children of a node must be chronologically older than the node itself. If node age can be determined, then during a find operation, we need only look at nodes that have the same hash key and that are also younger than both of the desired child nodes. The problem then is how to determine node age. One possibility is to use a field in the node, but this has a nontrivial memory cost. A small field could perhaps be packed into the node structure without increasing the size, but this compromises the ability to tell the age of a node accurately. Instead, we will require that chronological ordering corresponds to address ordering. On a machine where the heap grows upwards, it is most natural to have older nodes occupy lower addresses. We will assume that this is the case for the remainder of this paper. We use the notation \( f < g \) to indicate that node \( f \) is older than node \( g \).

In the unique table, collisions are resolved by separate chaining. Thus, each node contains a successor link that points to the next node with the same hash key. In our new design, this link satisfies the same invariant as the two children fields: it must point to a chronologically older node. Thus, as we traverse a chain, we check to see whether the current node is younger than both of the desired children. If not, we know that desired node does not yet exist and must be created. Only if the node is young enough do we access its fields and compare with the desired values. Schematically, the main loop inside the find operation is shown in Figure 1.

4  Garbage Collection

The decision to identify chronological order with address order imposes some constraints on the garbage collector. First, when BDDs are freed by the user, we cannot just reuse the freed nodes to build other BDDs, since the nodes are likely to have addresses that would not correspond to the chronological ordering. But because we must reclaim the storage occupied by these nodes, we are lead to the conclusion that the garbage collector must be able to copy nodes to new addresses.

Because of this, the handles passed back to the user cannot be pointers to BDD nodes, since there is no way to locate these handles reliably so that they could be adjusted after a garbage collection. Instead, the handles must be pointers to structures that contain pointers to BDD nodes. The library manages a pool of these structures that we will call “references.” After a garbage collection, we sweep through the set of all references and adjust each node pointer to reflect the node’s new address. The references also provide a convenient place to store an external reference count. These counts are not limited in size like a small in-node reference count; hence overflow is not an issue.

Since garbage collection and reordering can take place in the middle of an operation, we need a way to locate the node pointers corresponding to cofactors of operands and to partially constructed results. To do this portably requires that node addresses go on a library-managed stack that is distinct from the CPU stack. So if the operations were written in a recursive fashion, two stacks would really be maintained. To avoid this duplication of work, we simply use a library-managed “operation stack” to store everything and then write the operation in an iterative style. The garbage collector uses a “frame header” in each stack record to determine which words are pointers. Even if it were not necessary for garbage collection purposes, avoiding recursion is usually slightly faster.

We now consider the particular type of garbage collector that is required. Because address order and chronological age must correspond, it is most natural to use a sliding collector [6]. With such a collector, free space is reclaimed by sliding non-garbage nodes.
ptrdiff_t h = uniqueTableIndex(index, t, e);
BDDNode* youngest = (t < e ? e : t);
for (BDDNode* f = uniqueTable[h]; f != NULL && youngest < f; f = f->next)
  if (f->index == index && f->t == t && f->e == e)
    return f;
Create a new node for (index, t, e);

Figure 1: Main loop of the find procedure

to the low end of the heap. Allocation of new nodes is simply a matter of incrementing a pointer. Since
the original allocation order is maintained, the required correspondence is preserved. Identification of
free nodes can be done either by reference counting or by using a mark/sweep collector. We experimented
with both and found that while reference counting gave faster garbage collections, the overhead of main-
taining the counts made mark/sweep faster overall.

In either case, we can cheaply obtain some of the benefits of a generational garbage collector [6]. With
reference counting, it is only necessary to keep track of the lowest node whose reference count has dropped
to zero. Nothing below this node needs to be swept. With a mark/sweep collector, we must do a bit more:
we need to regroup nodes [6]. Consider creating a series of BDDs \(f_1, f_2, \ldots\) in an initially empty heap,
and then freeing \(f_5\). If we were to garbage collect at this point, we know that all nodes that existed just
after the creation of \(f_4\) are not garbage. Since these nodes already form a contiguous block at the start
of the heap, we can avoid marking and copying these nodes entirely. We record this type of information by
storing a “breakpoint” with each BDD reference. The breakpoint stored with a reference indicates the
youngest node that can be guaranteed to be non-
garbage if neither that reference nor any chronologi-
cally older reference has been freed by the user. Main-
taining breakpoints requires that we not use a pure
sliding collector however. In particular, after garbage
collection in the example above, it would not be guar-
anteed that the nodes for \(f_1, \ldots, f_4\) and \(f_6\) form a
contiguous block at the start of the heap. This is be-
because the BDDs \(f_7, f_8, \ldots\) may have shared nodes
with \(f_5\). We can make the idea work by regrouping:
we trace \(f_6, f_7, \ldots\) in order, moving the nodes in each
BDD in turn down as far as possible. Note that trac-
ing is already needed during the mark phase, so this
does not introduce additional overhead. An im-
portant benefit of either a sliding or a regrouping collec-
tor is that nodes that are connected via short paths
are more likely to be close in address. This tends to
reduce DCMs when operands are being traversed.

Figure 2 outlines the garbage collection procedure.
The procedure for tracing a node is shown in Figure 3. A node’s mark is stored in a one-bit field in
the node. To avoid having a separate field in each node for the forwarding address, we can use the same
link field that forms chains in the unique table. When
an unmarked node is encountered, either during trac-
ing or during the first sweep after all non-garbage
nodes have been marked, that node’s link points to
a chronologically older node. If this older node is
below the breakpoint, or if the link is null, then we
compute the unique table position of the node and
store the link into the unique table at that position.
Then after the first sweep, the only nodes remaining
in the unique table will be those that are below the
breakpoint. During the second sweep, any nodes from
above the breakpoint that are still alive are reinserted
into the unique table.

if (f <= breakpoint)
  return f;
if (f is marked)
  return f’s forwarding address;
Mark f;
if (f is not a constant) {
  f->t = Trace f->t;
  f->e = Trace f->e;
}
 f’s forwarding address = ++newLastUsed;
return f’s forwarding address;

Figure 3: Tracing a node

To be as portable as possible, we do not use a cus-
tom memory allocator. Hence we will not really be
able to have one contiguous heap. Instead, we allo-
cate large “blocks” of nodes together. When a new
node is required, if the current allocation pointer is
not at the end of a block, then the allocation pointer is
simply incremented. Otherwise the allocation pointer
must be moved to a new block. While we can always
allocate nodes within a block to maintain the ad-
dress/age correspondence, we cannot guarantee that
successively allocated blocks will also have the re-
quired relationship. So we must decide what to do
BDDNode* breakpoint = first node in heap;
int lookingForBreak = 1;
for (BDDReference* r = oldest reference to youngest reference)
  if (r->refs == 0) {
    Delete r;
    lookingForBreak = 0;
  }
  else if (lookingForBreak)
    breakpoint = r->breakpoint;
BDDNode* newLastUsed = breakpoint;
Trace BDDs for constants and BDDs for variables;
for (BDDReference* r = oldest reference to youngest reference)
  r->f = Trace r->f;
Trace BDDs on the operation stack;
Clear invalid operation cache entries;
for (BDDNode* f = breakpoint+1 to lastUsed)
  if (f is marked) {
    BDDNode* g = f’s forwarding address;
    while (g is marked && f < g) {
      Swap *f and *g;
      g = f’s forwarding address;
    }
    *g = *f;
  }
for (BDDNode* f = breakpoint+1 to newLastUsed) {
  Unmark f;
  Insert f into the unique table;
}
lastUsed = newLastUsed;

Figure 2: Outline of the garbage collection procedure

when a block is obtained whose address is less than that of some previously allocated block. We choose to
deal with the problem of out-of-order blocks by simply
deferring the use of these blocks until after the
next garbage collection. Empirically, only a small
percentage of the blocks in use are deferred. To
ensure that access to a node causes at most one DCM,
the size of a node (in bytes) is made to be a power
two, and the address of the first node in a block is
forced to be a multiple of the node size.

5 The Operation Cache

The main obstacle to machine independence is the op-
eration cache. If an address-dependent hashing func-
tion is used, then overwriting in the cache may lead
to an operation being redone on one machine (due to
its cache entry being replaced) and not on another.
With operations such as simultaneous substitution
that may produce garbage, this recomputation can
lead to gross differences in behavior by causing re-
ordering or garbage collection to occur at different
points on different machines. Machine independence
can be achieved by using a true hash table for opera-
tions like substitution, but initializing and destroying
these tables introduces extra overhead. Instead, we
choose to use a field inside the nodes to provide an
address-independent hash function. Because access
to the node will be required anyway if an operation
cache miss occurs, this does not increase DCMs sig-
ificantly.

The standard node fields that are required include
two words for child pointers, one word for linking in
the unique table, and a half word for the node’s in-
dex. Thus, in a four word node, a half word is avail-
able. We store a pseudo-randomly generated (but
deterministic) node ID in this space. By laying out
the node storage with the index and this ID adja-
cent, we make one word of the node that has the
ID in the low bits and the index in the high bits.
This word is XORed with the negation flag to form
the node’s hash key (Figure 4). Empirically, we find
that address-independent caching results in about a ten percent slowdown when compared to address-dependent caching with a simple hash function. However, simple hash functions tend to break down occasionally, and the speed difference when using a more robust address-dependent hash function is small.

![Figure 4: Node layout and hashing](image)

Since most common operations have only two arguments, a four word cache entry (holding operation code, arguments, and result) usually suffices. However, there is also the reasonably common case of an ITE, which requires three arguments. Using a five word cache entry would increase both memory and time requirements. The time penalty comes from having to manipulate larger entries and because some cache entries would span processor cache lines. However, since node addresses are aligned on node-sized boundaries, the low order bits of all node pointers are guaranteed to be zero (except for the negation flag). Hence we can distinguish between a node pointer and an operation code by ensuring that the operation code has at least one of these low order bits set. This allows us to share the operation code field in a cache entry with the third argument to an ITE.

6 Experimental Results

In this section, we compare a library based on these ideas to existing libraries [8, 11, 12]. The experiments, run on a Sun Ultra Enterprise (248 MHz UltraSPARC-II CPU, 1 GB main memory, 1 MB cache), consisted of building BDDs for the 1991 and 1993 Logic Synthesis Conference benchmarks. The benchmarks were obtained from the CBL archives [5]. Because of differences in the reordering heuristics used in the different libraries, all experiments were done without reordering. Inputs were ordered as they appeared in the file. The BLIF format versions of the 1991 benchmarks were used, and the 1993 benchmarks were converted to flat BLIF format using the supplied format conversion utility. Table 1 shows timing comparisons to the CUDD [12] (version 2.1.2) and CAL [11] (version 2.0) libraries, to Long's package [8] (version 1.0), and to a later version of Long's package (only available internally in Lucent) denoted by L3.2. All packages were compiled with the Sun native compilers at maximum optimization. No manual tuning was done for any of the packages (beyond setting the compiler flags). Benchmarks for which it was trivial or impossible to build BDDs are omitted. The benchmarks marked with a star were taken from the 1991 benchmark set; all others are from the 1993 set. For benchmarks that are in both sets, the version that required longer to build was chosen. We see about a factor of two difference in time compared to the CUDD and L3.2 libraries, and a factor of three to four compared to Long's package. The ratio between the new library and CAL are more variable since CAL uses breadth-first manipulation (all the other packages use depth-first).

To directly measure locality, the experiments were also run on an SGI Power Challenge (194 MHz R10000 CPU, 1.5 GB main memory, 1 MB cache). The R10000 has hardware registers for counting DCMs. Table 2 shows the ratios of DCMs between the new library and the others over all the benchmarks. In comparing to CUDD and Long’s package, s38584.1 was not counted because of the abnormal performance of these libraries for that benchmark. Interestingly, CAL had significantly fewer DCMs than the new library on five individual benchmarks, and was comparable on two others. Nevertheless, its time was always worse than that of the new library, which goes to show that DCMs alone are not the only determiner of performance. A breakdown of DCMs by action for the depth-first packages for a typical benchmark, C499, is shown in Table 3. (Since CAL’s internals are significantly different, it was not included in this comparison.) The differences between the libraries can be explained as follows.

**Finds** The new library does well because searches can be cut off early based on node ages.

**Cache lookups** Long’s library sets the cache size based on the number of nodes in existence, so it is not directly comparable in this category. The other libraries resize based on cache performance. Among these, the new library uses four word entries that always fit in a cache line, while CUDD and L3.2 use five word entries that sometimes span lines.

**Garbage collection** CUDD uses reference counting, so it is not directly comparable in this category. The other libraries use mark/sweep. Long’s package and L3.2 have poor locality both when marking and when traversing the unique table to collect garbage nodes. The new library has better locality during marking because of...
Table 1: Build CPU times (in seconds)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>New Time</th>
<th>CUDD Time</th>
<th>Ratio</th>
<th>CAL Time</th>
<th>CAL Ratio</th>
<th>Long’s Time</th>
<th>Long’s Ratio</th>
<th>L3.2 Time</th>
<th>L3.2 Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1355</td>
<td>10.2</td>
<td>20.5</td>
<td>2.0</td>
<td>15.1</td>
<td>1.5</td>
<td>30.7</td>
<td>3.0</td>
<td>21.8</td>
<td>2.1</td>
</tr>
<tr>
<td>C1908</td>
<td>3.5</td>
<td>7.6</td>
<td>2.2</td>
<td>6.0</td>
<td>1.7</td>
<td>12.3</td>
<td>3.6</td>
<td>7.0</td>
<td>2.0</td>
</tr>
<tr>
<td>C3540*</td>
<td>17.5</td>
<td>37.2</td>
<td>2.1</td>
<td>48.0</td>
<td>2.7</td>
<td>56.8</td>
<td>3.2</td>
<td>37.9</td>
<td>2.2</td>
</tr>
<tr>
<td>C432</td>
<td>2.0</td>
<td>5.6</td>
<td>2.8</td>
<td>4.2</td>
<td>2.1</td>
<td>8.5</td>
<td>4.3</td>
<td>5.2</td>
<td>2.7</td>
</tr>
<tr>
<td>C499</td>
<td>26.2</td>
<td>64.1</td>
<td>2.4</td>
<td>45.6</td>
<td>1.7</td>
<td>89.9</td>
<td>3.4</td>
<td>55.9</td>
<td>2.1</td>
</tr>
<tr>
<td>C880</td>
<td>4.6</td>
<td>9.9</td>
<td>2.1</td>
<td>8.8</td>
<td>1.9</td>
<td>16.0</td>
<td>3.5</td>
<td>8.9</td>
<td>1.9</td>
</tr>
<tr>
<td>clma</td>
<td>1.5</td>
<td>4.7</td>
<td>3.2</td>
<td>12.6</td>
<td>8.5</td>
<td>5.3</td>
<td>3.6</td>
<td>3.1</td>
<td>2.1</td>
</tr>
<tr>
<td>clmb*</td>
<td>1.1</td>
<td>3.7</td>
<td>3.4</td>
<td>12.1</td>
<td>11.0</td>
<td>4.4</td>
<td>4.0</td>
<td>2.4</td>
<td>2.2</td>
</tr>
<tr>
<td>cm150a*</td>
<td>1.1</td>
<td>2.0</td>
<td>1.8</td>
<td>4.7</td>
<td>4.3</td>
<td>3.2</td>
<td>2.9</td>
<td>1.8</td>
<td>1.7</td>
</tr>
<tr>
<td>comp*</td>
<td>2.9</td>
<td>7.8</td>
<td>2.7</td>
<td>4.5</td>
<td>1.6</td>
<td>9.8</td>
<td>3.4</td>
<td>5.5</td>
<td>1.9</td>
</tr>
<tr>
<td>daio_receiver</td>
<td>0.7</td>
<td>1.5</td>
<td>2.4</td>
<td>1.8</td>
<td>2.8</td>
<td>2.5</td>
<td>3.8</td>
<td>1.5</td>
<td>2.2</td>
</tr>
<tr>
<td>dalu*</td>
<td>136.4</td>
<td>324.2</td>
<td>2.4</td>
<td>419.3</td>
<td>3.1</td>
<td>462.9</td>
<td>3.4</td>
<td>276.4</td>
<td>2.0</td>
</tr>
<tr>
<td>des*</td>
<td>0.6</td>
<td>1.2</td>
<td>2.0</td>
<td>3.9</td>
<td>6.4</td>
<td>2.3</td>
<td>3.8</td>
<td>1.2</td>
<td>1.9</td>
</tr>
<tr>
<td>ecc</td>
<td>1.8</td>
<td>3.3</td>
<td>1.9</td>
<td>3.3</td>
<td>1.8</td>
<td>5.1</td>
<td>2.9</td>
<td>3.0</td>
<td>1.7</td>
</tr>
<tr>
<td>i10*</td>
<td>92.0</td>
<td>234.4</td>
<td>2.5</td>
<td>254.3</td>
<td>2.8</td>
<td>361.9</td>
<td>3.9</td>
<td>210.3</td>
<td>2.3</td>
</tr>
<tr>
<td>mm9a*</td>
<td>14.1</td>
<td>35.9</td>
<td>2.6</td>
<td>41.7</td>
<td>3.0</td>
<td>55.0</td>
<td>3.9</td>
<td>33.5</td>
<td>2.4</td>
</tr>
<tr>
<td>mm9b*</td>
<td>17.6</td>
<td>47.4</td>
<td>2.7</td>
<td>51.9</td>
<td>3.0</td>
<td>71.4</td>
<td>4.1</td>
<td>42.3</td>
<td>2.4</td>
</tr>
<tr>
<td>mult16a</td>
<td>9.3</td>
<td>19.8</td>
<td>2.1</td>
<td>18.3</td>
<td>2.0</td>
<td>31.5</td>
<td>3.4</td>
<td>20.7</td>
<td>2.2</td>
</tr>
<tr>
<td>mux*</td>
<td>1.2</td>
<td>2.0</td>
<td>1.7</td>
<td>3.6</td>
<td>2.9</td>
<td>3.5</td>
<td>2.9</td>
<td>1.9</td>
<td>1.6</td>
</tr>
<tr>
<td>my_adder*</td>
<td>3.0</td>
<td>5.5</td>
<td>1.8</td>
<td>10.3</td>
<td>3.4</td>
<td>10.7</td>
<td>3.6</td>
<td>5.1</td>
<td>1.7</td>
</tr>
<tr>
<td>rot</td>
<td>7.7</td>
<td>19.0</td>
<td>2.5</td>
<td>17.3</td>
<td>2.2</td>
<td>31.1</td>
<td>4.0</td>
<td>18.9</td>
<td>2.4</td>
</tr>
<tr>
<td>s1423</td>
<td>2.7</td>
<td>7.0</td>
<td>2.6</td>
<td>5.3</td>
<td>2.0</td>
<td>11.6</td>
<td>4.3</td>
<td>6.5</td>
<td>2.4</td>
</tr>
<tr>
<td>s15850.1*</td>
<td>1.3</td>
<td>3.4</td>
<td>2.6</td>
<td>6.3</td>
<td>4.8</td>
<td>5.2</td>
<td>4.0</td>
<td>2.8</td>
<td>2.1</td>
</tr>
<tr>
<td>s38417*</td>
<td>219.6</td>
<td>680.5</td>
<td>3.1</td>
<td>519.7</td>
<td>2.4</td>
<td>1206.7</td>
<td>5.5</td>
<td>648.3</td>
<td>3.0</td>
</tr>
<tr>
<td>s38584.1</td>
<td>1191</td>
<td>2872.9</td>
<td>24.1</td>
<td>366.1</td>
<td>3.1</td>
<td>1647.3</td>
<td>13.8</td>
<td>236.7</td>
<td>2.0</td>
</tr>
<tr>
<td>s420.1*</td>
<td>0.8</td>
<td>1.5</td>
<td>1.8</td>
<td>1.3</td>
<td>1.6</td>
<td>2.5</td>
<td>3.1</td>
<td>1.4</td>
<td>1.8</td>
</tr>
<tr>
<td>s9234.1*</td>
<td>51.5</td>
<td>121.8</td>
<td>2.4</td>
<td>135.2</td>
<td>2.6</td>
<td>195.8</td>
<td>3.8</td>
<td>96.6</td>
<td>1.9</td>
</tr>
<tr>
<td>too_large</td>
<td>1.1</td>
<td>2.5</td>
<td>2.4</td>
<td>5.2</td>
<td>4.9</td>
<td>3.4</td>
<td>3.3</td>
<td>2.1</td>
<td>1.9</td>
</tr>
</tbody>
</table>

1 Apparently the caching in CUDD and in Long’s package broke down on this example

Table 2: Total DCMs (in millions) and ratios to new library

<table>
<thead>
<tr>
<th>Library</th>
<th>DCMs</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>New</td>
<td>961</td>
<td>—</td>
</tr>
<tr>
<td>New w/o s38584.1</td>
<td>784</td>
<td>—</td>
</tr>
<tr>
<td>CUDD w/o s38584.1</td>
<td>1568</td>
<td>2.0</td>
</tr>
<tr>
<td>CAL</td>
<td>1873</td>
<td>1.9</td>
</tr>
<tr>
<td>Long’s w/o s38584.1</td>
<td>2897</td>
<td>3.7</td>
</tr>
<tr>
<td>L3.2</td>
<td>2115</td>
<td>2.2</td>
</tr>
</tbody>
</table>

The regrouping garbage collector. It uses two sweeps, but both exhibit good locality. For this application, the ability of the new library to avoid garbage collecting the entire heap each time is only of minor benefit. For C499, disabling this ability increases the DCMs associated with garbage collection by 0.7 million.

Operations The regrouping garbage collector leads to better locality during operand traversal.

7 Comparison to Related Work

Another approach to trying to improve cache behavior is described by Klarlund and Rauhe [7]. Their main idea is to store the first few nodes in a unique table chain directly in the unique table. These nodes can then be accessed with only one cache miss. Klarlund and Rauhe do not discuss overflow chain organization, but these could be cache line-sized blocks of nodes as well, either in a separate area or, with coalesced chaining, in the table itself. Regrouping
nodes to improve traversal locality is not possible with this scheme. However, there should be fewer cache misses during finds. (Empirically, our new library accesses one unique table slot and about 0.75 nodes per find operation). For a general operation (what Klarlund and Rauhe call a “hashed binary apply”), their performance data suggests a speed ratio of between two and three compared to the MTBDD routines in Long’s package. Unfortunately, the application they consider (and the application interface that they provide) is somewhat unusual, and they only compare their hashed binary apply with Long’s package on a single example.

Breadth-first BDD manipulation algorithms [1, 9, 11] have the aim of improving virtual memory performance, and as a side effect also tend to improve cache behavior. This is evidenced by the good DCM statistics for CAL shown in Table 2. But as the CPU time comparison to CAL shows, these algorithms also tend to have higher overhead. They are still the methods of choice when the BDDs are significantly larger than main memory though.

### References


