Basic Research in Computer Science

BRICS RS-96-26 Klarlund & Rauhe: BDD Algortihms and Cache Misses

BDD Algortihms and Cache Misses

Nils Klarlund
Theis Rauhe

BRICS Report Series **RS-96-26**

ISSN 0909-0878 **July 1996**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

BDD Algorithms and Cache Misses

Nils Klarlund* Theis Rauhe†

Abstract

Within the last few years, CPU speed has greatly overtaken memory speed. For this reason, implementation of symbolic algorithms—with their extensive use of pointers and hashing—must be reexamined.

In this paper, we introduce the concept of *cache miss complexity* as an analytical tool for evaluating algorithms depending on pointer chasing. Such algorithms are typical of symbolic computation found in verification.

We show how this measure suggests new data structures and algorithms for multi-terminal BDDs. Our ideas have been implemented in a BDD package, which is used in a decision procedure for the Monadic Second-order Logic on strings.

Experimental results show that on large examples involving e.g the verification of concurrent programs, our implementation runs 4 to 5 times faster than a widely used BDD implementation.

We believe that the method of cache miss complexity is of general interest to any implementor of symbolic algorithms used in verification.

1 Introduction

On a modern computer with a RISC architecture, the goal is to write programs that allow one instruction to be executed per cycle. In fact, superscalar CPUs allow even two or three instructions to be executed per cycle. With clock rates of 50-300 MHz, such CPUs should be able to carry out the symbolic computations at an astounding rate.

When we look at the basic *apply* routines used to manipulate Binary Decision Diagrams (BDDs), it appears that something on the order of a

*AT&T Bell Laboratories, Room 2C-410, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: klarlund@research.att.com. This work was mainly carried out while the author was with BRICS, Aarhus.

†BRICS, Basic Research in Computer Science, Centre of the Danish Research Foundation, Dept. of Computer Science, University of Aarhus.

hundred machine instructions are executed for each *apply step*, which is an instance or iteration of the recursive procedure as defined in e.g. [3]. So we would expect a step to take a microsecond or so.

Unfortunately, BDD decision procedures run much slower in practice. On a Sparc 1000, we have measured each apply step to last up to 30 microseconds with the widely used BDD package [8] written by David Long.

The fundamental problem is that non-local access to memory is very slow: typically, 10 cycles if the data resides in the Level 2 (L2) cache and, for a multi-processor machine, 100 cycles for a L2 cache miss. If the data resides in the primary cache, there is no penalty, but this cache is only 8kB-32kB. The L2 cache is typically 256kB-1Mb.

In this paper, we suggest data structures and algorithms that aim at optimizing the use of the L2 cache and minimizing pointer chasing. To do this, we suggest a *cache miss complexity* concept to measure the running time of an algorithm. We analyze a traditional BDD implementation and calculate its cache miss complexities.

We suggest alternative implementations and calculate their cache miss complexity. According to this measure, the new algorithms are two to three times faster than the traditional implementations.

Our main improvements to BDD algorithms are

- For the unary apply routine, we use an extra field in a BDD node for intermediate results and thus avoid a hash table look-up.
- For the binary apply routine, we have found a property about the structure of the resulting BDD, which implies that hashing of BDD nodes is unnecessary for injective leaf functions.
- We store BDD nodes directly in the hash table—a technique that greatly complicates certain operations, but cuts in half the time to access a node.

In general, we hope that the concept of cache miss complexity as an analytical method that can be useful to others who seek to improve algorithms in verification.

Related work

Studies of cache miss and CPU pipeline performance have been carried out for C and Fortran programs in [4] and for ML programs in [5]. These studies show that pipeline utilization of only 25% to 35% are common, especially in

the pointer-oriented code generated by an ML compiler. Such low performance is mainly due to data and instruction cache misses.

The relationship between cache misses and BDD performance has not to our knowledge been studied before. But the related issue of designing fast BDD packages for data sets that do not fit into RAM has been studied from a practical point of view in [10, 9], where algorithms reducing page faults are described.

Theoretical lower bounds and optimal algorithms are discussed in [1]. The discrepancy between accessing RAM and disk is much higher than the discrepancy between accessing cache (whether it be L1 or L2) and memory. Also, a page size is usually bigger than a cache line size. Thus different considerations guide the design of data structures and algorithms in the two situations.

Overview

All of our design decisions are on measuring the complexity of an algorithm by its expected number of cache misses. We discuss the cache miss complexity in Section 2 and estimate the cache miss complexity of the BDD implementation by David Long. In Section 3, we introduce new techniques to reduce the cache miss complexity, and in Section 4, we report on our implementation. In Section 5, we discuss our experimental results. In Section 6, we summarize our work.

2 Cache-complexity measure

In the worst case scenario described in the introduction, a cache miss costs 100 cycles. Even when it only costs 10 cycles, a cache miss is the limiting factor in symbolic computations, where the CPU essentially functions as a throughway for exchange of pointers and does not carry out much arithmetic. Thus we suggest designing BDD algorithms solely based on reducing what can be called *pointer chasing*, i.e. the use of an address that has likely not been used recently.

Examples of pointer chasing are:

- Looking up an entry in a hash table (which usually would not fit into the primary cache).
- Following a hash table entry that points to a dynamically created object.

- Following a left or right successor of a BDD node.

But pointer chasing is not:

- Looking up a variable in the current activation record (since activation records are accessed according to a stack discipline).
- Looking up or writing a record in an array as part of copying the whole array. Here we assume that the sequential access of the copying algorithm loads enough records per cache line to allow us to disregard the time it takes to load the line from memory.

In practice, it is almost impossible to foresee what the distribution of memory accesses is with respect to hitting the different levels of the memory architecture. Thus the cache miss complexity cannot be used to precisely calculate the running time.

In addition, our neglecting the cost of sequential access is an approximation to reality that is reasonable with an amount of such code that is in little or constant proportion to the amount of random access code.

Also, even if it is evident that CPU speed is less important for performance of pointer rich algorithms, certain operations such as hashing may play a role as well.

Thus our contention is only that the cache miss complexity can act as an important guide to the construction of data structures and algorithms.

Cache miss complexity of conventional implementation

Let us consider BDDs of n variables. A truth assignment x maps each variable to a value in $\mathbf{B} = \{0, 1\}$. For simplicity, we assume that a BDD f represents a function $\mathbf{B}^n \to \mathbf{N}$, which we also denote f. Here $\mathbf{N}$ is the set of natural numbers (but could be any finite or countable set). The binary apply routine $\mathsf{Apply2}$ combines BDD f, BDD g, and a *leaf function* $\lambda : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ into a new BDD h such that $h(x) = \lambda(f(x), g(x))$. We write $h = \mathsf{Apply2}(f, g, \lambda)$.

Traditional algorithm

An algorithm for the apply routine is expressed in terms of a function $\mathsf{Apply2\_step}(p, q, \lambda)$, which takes as arguments a pointer p to a BDD node in f and a pointer q to a BDD node in g. The function returns a pointer r to a node representing the product of the BDDs starting in p and q. Nodes are stored in a hashed table T. The node (l, r, i) with index i, with left successor

l, and with right successor r is stored according to a hash value calculated from (l, r, i).

We assume that an explicit garbage collection scheme is employed: free nodes are linked together in a *free list*, and reference counts are used to keep track of which nodes are referred to.

A *result table* R is used to record the pairs (p, q) which have been met together with the result r of the call. Usually, this table is implemented as a hash table, where the key is (p, q). Looking up this key results in one cache miss. We assume that the key and the result is stored under the address calculated by the hash function (and we ignored the additional penalty of looking through overflow lists). If the pair is not in the table, the nodes p and q must be examined. This costs two additional cache misses (again assuming that any node is contained in a cache line). The Apply2_step routine is then called recursively on left and right successors of p and q. Let the results of these calls be r' and r'', respectively.

The new node (r', r'', i), where i is the minimum of the indices of p and q, is created dynamically and its address r becomes the contents of the field designated by the hash value. These operations require two cache misses: one for the look up and one for obtaining a node from the free list. (We here make the simplifying assumption that a new node is created every time; in practice, this happens not quite as frequent, but in any case, one cache miss is unavoidable.)

At this point, the algorithm must insert the result r in the result table. This requires another cache miss, since many nodes may have been calculated since the initial invocation that the address of the key (p, q) is lost from the cache.

In total, there are then six cache misses when a pair (p, q) is not found in the result table. Our experience is that this situation occurs in two-thirds of the calls of Apply2_step. Thus the expected number of cache misses is approximately 4.3.

Garbage collection

In addition, we have to account for the work involved in removing nodes when they are no longer needed. We here make an assumption that nodes are deleted at the same pace that they are created. Then we conclude that the deletion of a node requires at least one cache miss.

Doubling

We must also account for the most complicated aspect of a BDD implementation: the doubling of tables and consequent rehashing. In our proposed scheme, nodes themselves never need to be moved, but when the node hash table becomes too big, then the table must be doubled. All nodes must be inserted in the new table. This rehashing costs two cache misses per node: one for accessing the node and one for the insertion of its address in the hash table.

In the worst case, every node is on average rehashed once, and the average case is not much different assuming that we start with a small table.

In total, the cache miss complexity of this suggested implementation is 7.3 per Apply2 step.

For a similar implementation of Apply1, we calculate the cache miss complexity to 6.7.

Proposition 1 *For the conventional algorithms, the cache miss complexity of Apply1 is* 6.7 *per step and that of Apply2 is* 7.3 *per step.*

3 Improved BDD algorithms

We propose in this section new algorithms that offer cache miss complexities less than half the complexities of the conventional algorithms.

First, we mention a couple of general techniques that should be employed.

A well-recognized way of reducing cache miss complexity is to use a memory management technique of collecting nodes with the same life-span in a contiguous block of memory, which can be released in a unit time operation. Thus for the Apply2 routine, we seek to use a new memory block for the result of the apply operation, while the two memory blocks containing the argument BDDs are released in a constant time operation after the result has been calculated.

Of course, it is often important to be able to write several BDDs to the same block so that comparing BDDs become a unit time operation. In fact, such shared BDDs are essential to the use in the Mona decision procedure, where the life-span of shared BDDs is naturally reflected by the automata algorithms.

Another key technique is to store BDD nodes directly in the hash table, which then is the same as the node table. Then we need only one pointer chase instead of two for a look up.

Next, we introduce a couple of new techniques to further reduce the cache miss complexity.

3.1 Speeding up unary apply

The unary apply calculating $h = \mathsf{Apply1}(f, \lambda)$, enjoys the fortunate property that a good estimate can be given on the size of the resulting BDD h, namely, the size of f. Thus, we need only to allocate memory for h once.

The next observation is that the result table can be avoided if we keep an extra field in each node p, called mark that contains the result of the apply operation on p, if already visited, and 0, otherwise. The node table can be initialized without cache miss penalties (in practice we use the memzero function of C, which is often implemented especially fast in hardware).

Our experience with the unary apply is that on the average, as with the binary apply, a result is not found in the result table every two out of three times.

With this scheme, looking up in the result table and looking up the node is the same thing and so the cache miss complexity becomes $1/3 \cdot 1 + 2/3 \cdot (1 + 1)$, where the last 1 is the cache miss incurred when the result is stored in the node upon return from the recursive calls (we assume that the node has disappeared from the cache during these calls).

Proposition 2 *The cache miss complexity of the Apply1 routine above is* 1.7 *per step instead of* 6.7*.*

3.2 Hashed binary apply

Our general design decisions above almost specifies the algorithm. When the node table is full, we double it by application of the Apply2 operation. In addition, we must rehash the result cache, which can be shown to be doable with an extra cache miss per node. With these doublings, there is almost nothing gained. But if we can give a correct size estimate, not entailing doubling, the algorithm becomes twice as fast:

Proposition 3 *The cache miss complexity of the Apply2 routine above is* 5.7 *per step instead of* 6.7*. With a correct size estimate, the complexity becomes only* 3 *per step.*

3.3 Sequential binary apply

For injective leaf functions, we can do better than 5.7 per step even when the size of the resulting BDD is not known. We need the following terminology.

The high and low successors of a BDD node p are denoted $p \cdot 0$ and $p \cdot 1$, respectively. We assume that the n variables are number $0, \ldots, n-1$. The index, $\iota(p)$, of a node is in $\{0, \ldots, n-1\}$ if it is a decision node and is n if it is a leaf. A truth assignment $\vec{x}$ assigns a truth value to each variable. If p is a decision node, then $p \cdot \vec{x}$ denotes the value of the leaf that is reached by following decision nodes from p according to $\vec{x}$.

Lemma 1 *Assume that the leaf function λ is injective. If, during the traditional algorithm, the pair (p, q) is explored in the apply step, but is not in the result table, then the node calculated is not already present in the node table.*

Proof Let $r(p, q)$ denote the node calculated by the apply step on (p, q). The Lemma follows from:

Claim 1 *For all explored (p, q) and (p', q') it holds that $(p, q) \neq (p', q')$ implies $r(p, q) \neq r(p', q')$.*

Proof of claim Assume $(p, q) \neq (p', q')$. Since the BDDs are canonical, there is some $\vec{x}$ such that $(p \cdot \vec{x}, q \cdot \vec{x}) \neq (p' \cdot \vec{x}, q' \cdot \vec{x})$. By injectivity of λ, $r(p, q) \cdot \vec{x} = \lambda(p \cdot \vec{x}, q \cdot \vec{x}) \neq \lambda(p' \cdot \vec{x}, q' \cdot \vec{x}) = r(p', q') \cdot \vec{x}$. Thus $r(p, q) \neq r(p', q')$. □

Lemma 1 implies that we do not need to hash into the node table. Thus when the pair (p, q) is not in the result table, we allocate sequentially a new node r and its address is put into the result table before p and q are explored. In this way, one cache miss is incurred for the look-up in the result table, two misses are incurred for examining p and q, and one miss is incurred when the results of the recursive calls are stored in r. When the node table is full, we copy it sequentially to a new table twice the size. The result table can be rehashed sequentially if we use tables that have sizes 2^m and if we use the m least significant bits of the hash function $h(p, q)$. When we rehash into the new table of size 2^{m+1}, an entry at address i is entered at address i or $i + 2^m$ in the new table.

Proposition 4 *For injective leaf functions, the cache miss complexity of the sequential Apply2 routine is 3.3 per step.*

4 Our BDD implementation

The algorithms of the preceding Section have been implemented in C.

Data representation

On the Sparc architecture, the cache line size is 32 bytes. Thus it seems important to squeeze a BDD node into 16 bytes. This is feasible if we represent BDD pointers as three byte unsigned integers (that are used as array indices) and if we use two byte unsigned integers to represent node indices. These 8 bytes are packed into a field lri consisting of two 32 bit integers. In this way, we can at most handle 2^{24} (approximately 16 million) BDD nodes with up to 65,000 variables. We have judged as insignificant the time it takes to pack and unpack these components.

We need two additional fields: the mark field is used by the unary apply routine to hold the result as described above and the next field, which is used for hashed insertion. The C declaration is:

```
struct bdd_record_
  {unsigned lri[2];
   unsigned next;
   unsigned mark;
};
```

which defines a structure of 16 bytes.

For hashed insertion, we use the node table in a two-way associative manner: the hash function calculates an even index k for a node (l, h, i) and the node is found either at k or at $k + 1$ or in the overflow list denoted by the next field of the node at k. We make sure to align the nodes at k and at $k + 1$ so that they fit into the same cache line.

The result table is organized in a similar manner.

As hash function, we use multiplication by a prime number (which is only four cycles on the Sparc architecture) followed by an "and" operation to capture the appropriate number of least significant bits as described in Section 3.3.

BDD managers

Each node table is managed through a *BDD manager* data structure. The manager defines a list of roots so that shared BDDs can be built. No pointers are returned as a result of an apply operation, since such a pointer would be valid only as long as the node table has not been doubled. Instead, the result is added to the list of roots, and this list is updated whenever a doubling takes place.

The binary apply operation requires the following arguments:

- a BDD manager bddm_p and a node pointer p (to a node in the table managed by bddm_p);
- a BDD manager bddm_q and a node pointer q
- a BDD manager bddm_r, where the result of the apply operation is built; and
- the leaf function.

The hashed binary apply code is complicated, since activation records on the call stack contain pointers that change during doubling and pointers in the result table are also changed. (We use the result table of the unary apply operation used to double as a translation table between old and new pointers; we have disregarded this work in our previous cache miss complexity analysis, since the call stack is usually small compared to be overall size of the BDDs.) The situation becomes even more complicated, when the bddm_r manager is the same as e.g. bddm_p, that is, when the new nodes are added to the nodes of the table of the p argument.

5 Experimental results

We compared our BDD package to the MTBDD (multi-terminal BDD) routines in the BDD package by David Long. The unary apply routine is used in the Mona [6] decision procedure for Monadic Second-order Logic to minimize BDD-represented automata. The minimization routine calls unary apply repeatedly over the same BDD with leaf functions that represent finer and finer partitions. There is almost no other heavy computational work. We used the Mona program, which is written in ML, to parse long formulas from which long sequences of automata-theoretic operations are calculated. The minimization procedure is written in C and was interfaced with the old BDD package by Long and our new package. The table below (left) shows the running times for two examples that required 1.12 and 8.33 million apply steps. Example 1 is a verification of timing properties of a flipflop [2]. Example 2 is a formula that arises during the verification of a concurrent system against another concurrent system [7]. We originally used the Sparc 1000 multiprocessor, but repeated the tests on a uniprocessor Sparc 4.[1]

[1] All reported times are the minimum recorded in several trials on a machine with little load. The times for the Sparc 1000 varied with the load, even if there seemingly are no other active users. For this machine, the usual running times are 30% or so slower. For the Long package, we were able to obtain slightly better times (5%) by adjusting

The Sparc 1000 features slow RAM access and slow CPUs (our version has four of them), but one megabyte L2 cache per processor. The Sparc 4 features RAM access that is several times faster and a processor that runs approximately twice as fast, but it has no L2 cache.

The table to the right shows the relative performance gain obtained by our package.

	Apply1 (sec./step)			
	Example 1		Example 2	
	Ours	Long	Ours	Long
Sparc 1000	3.5	14.7	2.8	16.7
Sparc 4	3.5	10.6	2.6	11.2

	Apply1 (relative)	
	Ex. 1	Ex. 2
Sparc 1000	4.0	5.9
Sparc 4	3.1	4.3

The time per step for the Example 1 is significantly higher than for Example 2. The reason is that three quarters of the 2146 BDD tables created have less than 32 nodes. (The largest table contains on the order of 2^{16} nodes.) We have measured that the amount of time involved in creating the tables and BDD managers constitute 40% of the time. If this time is discounted, the time per step is about 25% less than for Example 2. We believe that the situation in Example 1 is atypical of most BDD usage.

In contrast, approximately half the apply steps in Example 2 occur when the number of nodes in the tables is between 2^{13} and 2^{18}. On the uniprocessor Sparc 4, we have measured the average apply step in BDDs with more than a 10^5 nodes to take only $2.1\mu s$ whereas the corresponding number for the multiprocessor Sparc 1000—with its much slower RAM—increases to $3.5\mu s$.

To test our sequential apply routine, we used the automaton product routine that relies on repeated calls of the binary apply with an injective leaf function (namely, the pairing function). The number of apply steps in the examples are .125 and .956 million, respectively. The results were

the internally defined *cache load factor*. The times reported may not precisely cover the time per operation since garbage collection of nodes produced under one operation can find place during another. For both packages, significant time was spent in a subset construction that involves both unary and binary apply operations. The gain for this operation is in between that of the unary and binary apply.

	Apply2 (seconds)			
	Example 1		Example 2	
	Ours	Long	Ours	Long
Sparc 1000	9.6	25.6	6.2	33.2
Sparc 4	9.6	20.0	5.2	21.1

	Apply2 (relative)	
	Ex. 1	Ex. 2
Sparc 1000	2.7	5.3
Sparc 4	2.1	4.0

For comparison of hardware performance, we give running times for our package on a Pentium PC (133MHz with 64 megabytes RAM) running Linux.

	Apply1 (sec./step)			Apply2 (sec./step)	
	Ex. 1	Ex. 2		Ex. 1	Ex. 2
Pentium 133	1.8	1.4	Pentium 133	4.6	3.5

Hashed binary apply

We measured performance for the hashed binary apply under four sets of circumstances.

First, we tried running the binary apply with a unit size initial node and result table, which necessitate the maximal amount of doubling. We also ran the binary apply with a good estimate of the resulting table size (4× the maximal size of the two operands), which implied that less than a quarter of all nodes became involved in doubling.

Second, we ran the apply for two different strategies of dealing with the result table during doubling: either to erase the whole table, thus forgetting about previous results, or to rehash the table.

We present the running times for Example 2 on the Sparc 4. (On the multiprocessor Sparc 1000, we encountered extremely fluctuating running times even on an otherwise unloaded machine.)

	Apply2 (sec./step)	
	Res. erase	Res. double
Size est. 1	13.5	14.4
Size est. 4×	6.9	7.4

As can be seen, the hashed apply routine is almost as fast as the sequential apply when the size of the resulting BDD area can be predicted in most cases. We also note that it seems to be a waste of time to rehash the result table.

Real time versus cache miss complexity

Even though we have proposed the cache miss complexity measure only as a guide to implementation, we give below the number of micro seconds per apply step divided by the cache miss complexity for our principal algorithms

	μs/c.m.c.
Apply 1	1.5
Sequential Apply 2	1.6
Hashed Apply 2 (w/o size estimate)	2.0
Hashed Apply 2 (with size estimate)	2.5

Thus the highest fraction is 66% greater than the lowest fraction. Under ideal circumstances, we would expect the fraction to be constant, but we have already explained why this is unlikely to be the case.

We should compare these times to the actual memory access time. On the Sparc 1000, we have measured a load from a random address to take almost $2\mu s$, whereas such an access take only approximately $.5\mu s$ on the Sparc 4. Unfortunately, we have not been able to obtained statistics on the cache miss ratios, which require specialized hardware.

We note that the Long BDD package is much slower than the cache miss analysis of the traditional algorithms showed above. A reason is that there are more pointer indirection in this package than assumed in our analysis. Also, the Long package implements dynamic variable reordering, which introduces pointer chasing in critical sections of the code even if reordering is not used.

Memory consumption

We have not performed detailed measurements on the amount of memory that our implementation demands, but information from the "top" Unix program reveals that the memory used by our program is approximately the same as that used by the implementation by David Long. Although our node representation demands fewer bytes, we pay a penalty by storing nodes in an array, which is often only half full.

6 Conclusion

We have presented new implementation techniques for Binary Decision Diagrams with multiple leaves. Our guide has been our cache miss complexity concept. Although the complex memory architecture of modern computers

cannot be precisely summarized in such a simple concept, we have nevertheless obtained substantial improvements in the running times of basic BDD routines.

We have shown that our BDD performance on two very different architectures, the Sparc 4 and the Sparc 1000, is 4, respectively 5 times faster than with the Long BDD package. We have also argued that on uniprocessor machines the performance gain is even bigger for BDDs that have thousands of nodes or more since our results are skewed by the presence in our benchmarks of many very small BDDs for which initialization is expensive.

With our algorithms and the right choice of computer—a home PC—we have achieved a 10-fold speed-up over the Long BDD package run on a multiprocessor computer. In fact, we are getting close to our goal of running an apply step in one microsecond: on the Pentium, the unary apply takes 1.3 μs, the binary sequential apply 3.5μs, and the binary hashed apply (with good size estimates) 4.3 μs (all measured for Example 2).

We believe that our results can also be used to significantly improve the performance of programs that rely on binary-valued BDDs. In fact, our unary apply can be used in the projection routine and our hashed binary apply routine can be used for the usual Boolean connectives. It remains to be seen to which extent the addition of dynamic variable ordering affects the gains reported here.

When used with the Mona decision procedure, memory management using our blocks of BDDs is simpler than with the reference count technique of the Long package. We do not know whether this will hold for other BDD uses.

Acknowledgments

Rowan Davis dual skills in ML and C made it possible to hook up BDD routines in C to the Mona program. He also expertly rewrote most of the automata routines of the ML program.

Discussions with Lars Arge, Christian Fecht, Lal George, and David Long helped us gain valuable insight into the complexities of memory bound performance.

References

[1] L. Arge. The i/o-complexity of ordered binary-decision diagram manipulation. In *Proc. of 6th Annual International Symposium on Algorithms and Computation (ISAAC'95), LNCS 1004*, pages 82–91, 1995.

[2] D. Basin and N. Klarlund. Hardware verification using monadic second-order logic. In *Computer aided verification : 7th International Conference, CAV '95, LNCS 939*, 1995.

[3] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.

[4] Z. Cvetanoic and D. Bhandarkar. Characterization of alpha axp performance using TP and SPEC. In *Proc. of the 21st annual Int. Symp. on Computer Architecture*, pages 60–70. ACM, 1994. Also, Computer Arch. News, Vol 22, No. 2, April 1994.

[5] Lal George and George Necula. Accounting for the performance of Standard ML on the DEC Alpha. Technical report, AT&T Bell Labs., Sept. 1994.

[6] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. Technical Report RS-95-21, BRICS, Department of Computer Science, University of Aarhus, 1995. Accepted for the TACAS Workshop, 1995; available through `http://www.brics.dk/~klarlund`.

[7] N. Klarlund, M. Nielsen, and K. Sunesen. Automated logical verification based on trace abstraction. Technical Report RS-95-53, BRICS, 1995. To appear in Proceedings of PODC '96.

[8] D. Long. Bdd library. Available by FTP from emc.cs.cmu.edu.

[9] H. Ohci, N. Ishiura, and S. Yajima. Breadth-first manipulation of sbdd of boolean functions for vector processing. In *Proc. 28th ACM/IEEE Design Automation Conference*, pages 413–416. IEEE, 1991.

[10] Ashar P. and Cheong M. Efficient breadth-first manipulation of Binary Decision Diagrams. In *Proc. International Conference on CAD*, pages 622–627. IEEE, 1994.

Recent Publications in the BRICS Report Series

RS-96-26 **Nils Klarlund and Theis Rauhe.** ***BDD Algortihms and Cache Misses.*** **July 1996. 15 pp.**

RS-96-25 **Devdatt Dubhashi and Desh Ranjan.** ***Balls and Bins: A Study in Negative Dependence.*** **July 1996. 27 pp.**

RS-96-24 **Henrik Ejersbo Jensen, Kim G. Larsen, and Arne Skou.** ***Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL.*** **July 1996. 20 pp.**

RS-96-23 **Luca Aceto, Wan J. Fokkink, and Anna Ingólfsdóttir.** ***A Menagerie of Non-Finitely Based Process Semantics over BPA^*: From Ready Simulation Semantics to Completed Tracs.*** **July 1996. 38 pp.**

RS-96-22 **Luca Aceto and Wan J. Fokkink.** ***An Equational Axiomatization for Multi-Exit Iteration.*** **June 1996. 30 pp.**

RS-96-21 **Dany Breslauer, Tao Jiang, and Zhigen Jiang.** ***Rotation of Periodic Strings and Short Superstrings.*** **June 1996. 14 pp.**

RS-96-20 **Olivier Danvy and Julia L. Lawall.** ***Back to Direct Style II: First-Class Continuations.*** **June 1996. 36 pp. A preliminary version of this paper appeared in the proceedings of the 1992 ACM Conference on Lisp and Functional Programming, William Clinger, editor, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.**

RS-96-19 **John Hatcliff and Olivier Danvy.** ***Thunks and the λ-Calculus.*** **June 1996. 22 pp. To appear in** ***Journal of Functional Programming*****.**

RS-96-18 **Thomas Troels Hildebrandt and Vladimiro Sassone.** ***Comparing Transition Systems with Independence and Asynchronous Transition Systems.*** **June 1996. 14 pp. To appear in Montanari and Sassone, editors,** ***Concurrency Theory: 7th International Conference,*** **CONCUR '96 Proceedings, LNCS 1119, 1996.**