Distributed Dynamic BDD Reordering

Ziv Nevo
IBM Haifa Research Labs
Haifa, Israel
nevo@il.ibm.com

Monica Farkash
IBM Systems Group
Austin, TX
mfarkash@us.ibm.com

ABSTRACT
Dynamic BDD reordering is usually a computationally-demanding process, and may slow down BDD-based applications. We propose a novel algorithm for distributing this process over a number of computers, improving both reordering time and application time. Our algorithm is based on Rudell's popular sifting algorithm, and takes advantage of a few empirical observations we make regarding Rudell's algorithm. Experimental results show the efficiency and scalability of our approach, when applied within an industrial model checker.

Categories and Subject Descriptors
J.6 [**Computer Applications**]: Computer-Aided Engineering – *Computer-aided design (CAD)*

General Terms
Algorithms, Performance, Experimentation, Verification.

Keywords
Model checking, BDD, reordering, distributed computing.

1. INTRODUCTION
Reduced ordered Binary Decision Diagrams (BDDs) are compact data structures for the representation and manipulation of Boolean functions. BDDs are therefore widely used by many design-automation tools, despite the growing popularity of Boolean-satisfiability-based algorithms. Popular examples are formal verification and logic synthesis tools.

The size of BDDs tends to be highly sensitive to a given variable ordering, sometimes varying from linear to exponential sizes [3]. Determining an optimal order is known to be an NP-complete problem [2]. Many heuristics were therefore suggested for finding a better order for a given BDD. Some algorithms attempt to exploit structural information (e.g., [6]). Others are based on *dynamic variable reordering* [7], the most popular one being Rudell's sifting algorithm [14].

Rudell's sifting algorithm picks one BDD variable at a time and tries to find a better position for this variable, such that the resulting BDD is smaller in size. The search for a better position is based on the exchange of adjacent variables, which turns out to be an efficient operation.

Although Rudell's sifting algorithm proved to be a successful dynamic variable reordering algorithm, and although many improvements to Rudell's algorithm were suggested (e.g., [4], [5], [10], [13], [15]), it still often consumes a significant part of the runtime when used in BDD-based applications. Of course, these applications will usually run longer without performing variable reordering, or will even run out of memory.

We propose a new approach for speeding up dynamic reordering. Our algorithm is a distributed version of Rudell's algorithm, where multiple CPUs are used to find better positions for different variables in parallel. Search results from all CPUs are combined and applied to the original BDD.

The algorithm relies on a few properties of Rudell's algorithm that we have found empirically and are discussed in this paper. The main observation is that most of the time most variables are already in their optimal place, and need not to be relocated. Also, only a few relocations imply a significant decrease in BDD size. As a result, a distributed algorithm can perform the ineffective searches faster, without running into synchronization problems.

Experimental results show significant speedups. Reordering becomes up to 4 times faster using 4 additional CPUs. When incorporated into an industrial model checker, total runtime of the application improved by almost a factor of 1.8 on average (up to a factor of 3 on reordering intensive examples) using 4 additional CPUs. The algorithm is scalable; application speedup grows by almost 0.2 for each additional CPU.

The rest of the paper is organized as follows. Section 2 contains background and previous work on variable reordering. Section 3 presents a few empirical observations, demonstrating a few properties of Rudell's sifting algorithm. In Section 4 we propose a distributed variant of Rudell's algorithm, based on the findings in Section 3. Section 5 provides experimental results, showing the efficiency of our approach. The last section concludes.

2. PRELIMINARIES

2.1 Binary Decision Diagrams
Binary Decision Diagrams (BDDs) are data structures used for efficient representation and manipulation of Boolean functions. A BDD representing the function $f: \{0,1\}^n \rightarrow \{0,1\}$ is a directed acyclic graph, where each node v is labeled with a variable var(v), and a Shannon decomposition is carried out in each node with respect to var(v). We will only consider reduced ordered BDDs (Bryant [3]), where variables are encountered at most once and in the same order on every path from root to a terminal node.

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

For brevity we will use the term BDDs for reduced ordered BDDs. We will also associate each variables x_i with the set of nodes having var(v) = x_i. We will refer to these sets as *levels*.

2.2 Rudell's Sifting Algorithm

Rudell's sifting algorithm [14] is a common algorithm for dynamic BDD reordering, and is part of many BDD packages and applications (e.g., [1], [11], [16]). The algorithm successively picks BDD variables. For each variable the algorithm attempts to find the best position in the BDD order, leaving all other variables in place. The best position is the one that minimizes total BDD size. The search for a better position is performed using a sequence of exchanges of neighboring variables (swaps).

Exchanging the position of adjacent variables can be performed in time which is linear in the number of nodes labeled with either variable. This is because only local transformations to the two BDD levels are performed, and there is no change to nodes in other levels. Relying on this low-cost exchange makes Rudell's sifting algorithm so efficient.

Rudell's sifting algorithm is summarized as follows.

1. While there are unselected variables
   a. Select an unselected variable with maximal number of nodes
   b. Exchange the selected variable with its predecessor variable until it becomes the first variable in the ordering (search up)
   c. Exchange the selected variable with its successor variable until it becomes the last variable (search down)
   d. Exchange the selected variable with its predecessor until it comes back to the position where BDD size was minimal

2.3 Improvements to Rudell's Algorithm

Many improvements to Rudell's algorithm were suggested over the years. We consider here only a small collection of them, especially those that became widely used. The following improvements are considered common knowledge [4].

Upper limit: During sifting of one variable, BDD size can grow dramatically. It is possible to place an upper limit on BDD size (usually in terms of BDD size before reordering) and to stop a search in a certain direction when this limit is exceeded. This prevents huge intermediate BDDs.

Closest end: The selected variable is not always moved upwards first. Instead it is first moved to the direction where less swaps are likely to happen (or even less nodes have to be treated).

Somenzi [16] suggests keeping information about independent variables that can be swapped without changing BDD structure.

Panda and Somenzi [13] use symmetry properties to group variables together and to perform *group sifting*.

Meinal and Slobodova [10] divide the BDD into blocks of consecutive levels by analyzing the amount of communication between every two levels. A variable is then moved only within the range of its original block.

Drechsler and Gunther [4] calculate a lower bound on how small BDD size can get by further moving in the current direction. They then stop moving a variable in that direction if the lower bound exceeds the smallest BDD size already recorded. Ebendt and Drechsler [5] later derived a different lower bound, which seems to complement the previous one.

Slobodova and Meinel [15] implement a sampling-based approach, in which a representative small sample is taken from the considered BDD. The reordering problem is then solved on this sample, and the resulting order is applied to the original BDD.

A first attempt to parallelize dynamic BDD reordering was made by Milvang-Jensen and Hu [11]. They divide the BDD into blocks as in [10], then simultaneously reorder different blocks. The paper reports this method sometimes fails to find good orders and sometimes lacks enough blocks to gain significant speed-up.

3. EMPIRICAL OBSERVATIONS

In this section we study a few properties of Rudell's sifting algorithm empirically. In particular, we would like to answer the following questions.

- How many variables relocate during Rudell's algorithm?
- How does the gain in BDD size distribute among the variables that did move?
- How far away do variables move?

We define *reordering gain* as the decrease in BDD size after one application of Rudell's algorithm. We define *relocation gain* as the decrease in BDD size following a single iteration of Rudell's algorithm, in which a variable was moved to its best position.

3.1 Experimental Setting and Test Cases

For our experiments we picked 6 industrial hardware designs. These designs turned out to be reordering intensive while verifying them with the BDD-based model-checking engines of IBM's formal-verification tool RuleBase [1]. The number of BDD variables in each of the selected designs is shown in Table 1. These numbers roughly corresponds to the number of flip-flops left after applying standard reductions to the original circuit.

Table 1, Number of BDD variables

Model	A	B	C	D	E	F
#variables	95	118	188	193	218	369

A model checker usually applies reordering whenever BDD size exceeds a certain threshold. We therefore also picked for each design two reordering sessions in random. The initial number of nodes (before reordering) for each session is shown in Table 2, together with the reordering gain when using Rudell's algorithm.

Table 2, Initial BDD size and reordering gain for each reordering session

	# nodes	RG
A1	9929588	72%
A2	23784973	68%
B1	13593129	92%
B2	25334464	89%
C1	1710948	89%
C2	6635188	85%

	# nodes	RG
D1	2685574	74%
D2	6935896	20%
E1	1489149	67%
E2	3173233	34%
F1	22656155	72%
F2	24022198	47%

For all the experiments we used a 2.4GHz Intel Xeon machine with 2GB RAM.

3.2 Findings

3.2.1 Number of Relocated Variables

For each selected reordering session we measured the number of BDD variables that Rudell's algorithm brought to another position. Numbers are shown in Table 3 together with the percentage of relocated variables out of the total number of BDD variables.

Table 3, Relocated variables

	Number	Percent		Number	Percent
A1	32	33.7%	D1	53	27.4%
A2	34	35.8%	D2	39	20.2%
B1	55	46.6%	E1	62	28.4%
B2	42	35.6%	D2	60	27.5%
C1	36	19.1%	F1	121	32.8%
C2	42	22.3%	F2	171	46.3%

It is rather clear from the table that typically only about a third of the variables leave their original position. For all other variables, sifting time is simply a waste.

3.2.2 Impact of Relocated Variables

It may well be that changing position for many relocated variables had only a slight impact on BDD size. We therefore tried to measure the productivity of relocation instances.

Given a specific reordering session, we define the set of *significant variables* (SV) as the minimal set of variables, whose total relocation gain accounted for at least 70% of the total reordering gain when relocated.

We define the set of *very significant variables* (VSV) as the minimal set of variables whose total relocation gain accounts for at least 50% of the total reordering gain when relocated.

Table 4 shows the cardinality of both sets for each reordering session, as well as the percentage out of the total number of variables.

Table 4, Significant relocated variables

	SV	VSV	SV (%)	VSV (%)		SV	VSV	SV (%)	VSV (%)
A1	1	1	1.05	1.05	D1	4	2	2.07	1.04
A2	2	1	2.11	1.05	D2	2	1	1.04	0.52
B1	7	5	5.93	4.24	E1	5	2	2.29	0.92
B2	4	3	3.39	2.54	D2	3	1	1.38	0.46
C1	3	2	1.60	1.06	F1	7	3	1.36	0.81
C2	4	2	2.13	1.06	F2	5	3	1.90	0.81

Results show that a small set, containing just 1.04-5.93% of the total number of variables can achieve together 70% of total relocation gain, and that it is enough to move 0.46-4.24% of all variables, to achieve 50% of the reordering gain.

Considering only the variables that relocate, 1.6-9% of them achieve 70% of all the reordering gain and 5-5.9% of them achieve 50% of all the reordering gain.

Rudell's algorithm moves each variable through the entire BDD. Considering that it would be enough to move only 5.93% of the variables to achieve 70% of the reordering gain means that 94.07% of the work is of no consequence except for the decision not to relocate.

Note: Theoretically, it may be that for a specific variable, a significant relocation gain cannot be achieved without first relocating another variable with an **insignificant** relocation gain. Practically, the success of our distributed algorithm (Section 4) proves this is not the common case.

3.2.3 Relocation Range

Some suggested improvements to Rudell's algorithm are based on limiting the sifting range of each variable. We therefore tried to measure how far variables with high relocation gain traveled.

We considered the five variables with highest relocation gain for each reordering session. The relocation moves them over at least 1% of the BDD levels and up to 89.9%. Table 5 shows relocation ranges for the variable that relocated most (out of the 5). Figure 1 shows relocation ranges for all five.

Table 5, Maximal relocation of very significant variables

	A1	A2	B1	B2	C1	C2
%BDD	52.6	41.1	23.7	10.1	89.9	44.7

	D1	D2	E1	E2	F1	F2
%BDD	80.8	19.7	38.5	71.1	69.9	54.5

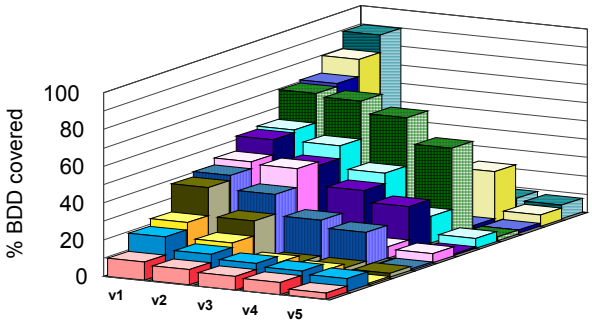

Figure 1, Relocation range of 5 most significant variables (from far to close we have C1, D1, E2, F2, F1, A1, C2, A2, E1, B1, D2, B2)

We could find no direct connection between the percentage of relocation gain achieved by a variable and its relocation range. That is, we found variables that relocate at distance but achieved insignificant percentage of the total reordering gain, and variables that achieved more than 10% of the reordering gain and relocated only 5% of the BDD. It is clear though, that limiting the relocation range should be performed wisely, otherwise it may significantly reduce total reordering gain.

4. DISTRIBUTED BDD REORDERING

Despite many improvements to Rudell's sifting algorithm, dynamic BDD reordering is still sometimes a bottleneck in BDD-based applications, taking a significant part of the computation time. In an era when hardware is cheap and huge computer farms are common, it makes sense to make use of available CPUs and distributing the demanding task of BDD reordering over a few machines.

We propose a distributed version of Rudell's sifting algorithm, allowing speeding-up dynamic reordering by utilizing idle CPUs. Our algorithm uses the common *master-slave* framework. The master is the BDD-based application which initiated the reordering session. Slaves are other idle CPUs.

The main idea is to let the slaves search for a better place for individual variables in parallel. The master's job is to distribute the variables among the slaves and then gather the results from the slaves regarding the best positions for their allocated variables. Master and slaves algorithms are as follows.

Master algorithm

1. While there are unselected variables
   a. Select an unselected variable with maximal number of nodes
   b. Send the selected variable (its current position in the order) to an idle slave, along with a copy of the current BDD
   c. If there is still an idle slave go to a.
   d. Otherwise, wait for reports from slaves, then sift variables according to a given merging algorithm (see Subsection 4.1)

Slave algorithm

1. Until told to quit
   a. Get a variable and a copy of the current BDD from master
   b. Exchange the selected variable with its predecessor variable until it becomes the first variable in the ordering
   c. Exchange the selected variable with its successor variable until it becomes the last variable in the ordering
   d. Report best found position and relocation gain to master

Note that the algorithm is insensitive to the way slaves are allocated. Slaves may be allocated per reordering session, or per application run. Moreover, slaves can be easily added or dropped during a specific reordering session. There is no limit on the number of allocated slaves, though there is no point in allocating more slaves than the number of BDD variables. Also, a slave can be allocated on the same CPU as the master with only a small penalty, as master and slaves usually do not work at the same time (depends on the merging algorithm).

An advantage of our algorithm is that slaves do not have to put variables back in their optimal place. They just have to report their findings to the master. Since many variables tend to stay where they are (refer to Section 3), we now get rid of much redundant work. This explains the fact that we can still see speedups having only one slave, as demonstrated in Section 5.

Most of the optimizations to Rudell's algorithm, discussed in Subsection 2.3 can still be applied to our distributed algorithm.

4.1 Merging Slave Results

Step d of the master's algorithm uses a merging algorithm to combine the results from the slaves. A few possible solutions are available, with varying degrees of synchronization and consistency with the original sequential algorithm. We tested two solutions. The first, called *synchronous merging*, is fully consistent with the sequential algorithm and requires a lot of synchronization. The second, called *continuous merging*, uses less synchronization but may not necessarily follow the variables choosing order of the sequential algorithm.

Synchronous merging: The master waits for answers from all slaves, and then starts applying search results on its BDD. Variables are considered one by one, according to the order they were sent to the slaves. If a search result indicates a variable should relocate, all remaining variables are considered unselected again and their search results are ignored. Thus, this algorithm complies with Rudell's order of choosing variables.

Definition: Two relocations of two different variables, one from position i to position j and the other from position k to position l are considered *intersecting* if the sections $[i, j]$ and $[k, l]$ intersect.

Claim: Given a BDD and two non-intersecting relocations r_1 and r_2, the relocation gain from performing r_2 on the BDD is the same relocation gain from performing r_2 right after performing r_1.

Proof (sketch): Since swapping of two adjacent levels changes only these levels, there is no BDD level both relocations change. Thus, relocations are completely independent of each other.

Continuous merging: The master keeps sending tasks to idle slaves and receiving results from other slaves. Only when it receives a result indicating a variable should move, it stops and waits for results from all slaves. Results are then sorted by relocation gain, and relocations are applied at that order. If a specific relocation intersects with an already applied relocation, its variable is marked unselected again, and will be later re-sent to a slave for a new search. Other relocations can be performed without losing relocation gain, as stated by the above claim.

Clearly, by using continuous merging slaves are constantly fed with new searching tasks and are rarely left idle during a reordering session. Moreover, we can set a threshold to ignore very small relocation gains, cutting idle times further.

4.2 Search Splitting

Another way of using the distributed framework is splitting the best-position search for the selected variable. One slave can perform the upwards search while another slave performs the downwards search in parallel. The master decides on the exact relocation, by considering the higher relocation gain.

The benefit from making this split is saving a considerable amount of swaps. After a slave searches in a specific direction, it no longer needs to sift the variable all the way back to its original location. It simply reports its findings and may start a new search.

The downside of performing search split is an additional synchronization effort. The master process cannot decide where to move a specific variable before it receives results from both slaves trying to relocate this variable. In addition, for variables closer to one BDD end work load may become highly imbalanced.

4.3 Efficient Transfer of BDDs

BDDs are made of distinct nodes and must be serialized before they can be sent over the network. We used a serialization algorithm, which is a variant of the algorithm introduced by Heyman et al [8]. The main difference is that we assume slaves to start every iteration with the same BDD order as the master. The master therefore does not have to send BDD order, and the slaves do not have to use Shannon expansion, while rebuilding the BDD.

The serializations and de-serialization algorithms are as follows.

Serialize

1. uniqueId = 0
2. Dump to buffer maxlevel (maximal level with nodes)
3. for levelNo = maxlevel down to 0
   a. Dump the number of nodes in level levelNo
   b. For each node in level levelNo do
      i. Associate uniqueId with node
      ii. uniqueId = uniqueId + 1
      iii. Dump to buffer unique id of left son, then unique id of right son

De-serialize

1. uniqueId = 0
2. Read maxlevel from buffer
3. for levelNo = maxlevel down to 0
   a. Read #nodes from buffer
   b. Repeat #nodes times
      i. Read from buffer left _id, right _id
      ii. Create the BDD node (levelNo, find_in_hash(left_id), find_in_hash(right_id))
      iii. Insert_to_hash(uniqueId, new node)
      iv. uniqueId = uniqueId + 1

The de-serialization algorithm uses a hash table to find an already generated node with a specific unique id. The serialization algorithm can use a hash table to do the opposite, or it can simply store the unique id within each node. For both algorithms, working from bottom levels up guarantees that when handling a specific node, its sons are already handled. Since the nodes ZERO and ONE are always present, they can be allocated a fixed unique id, and should not be transferred.

The serialization algorithm dumps two integers for each BDD node. While this may look compact, sending BDDs with hundreds of millions of nodes may still congest the network, and may deteriorate performance. We should therefore try to minimize the number of times BDDs are transferred through the network.

Fortunately, our findings in Section 3 help us again. If only a few variables move, then in many cases the master sends to a slave exactly the same BDD it sent last time. On such cases, the master can only send an indication that the BDD is unchanged, and the slave will simply rebuild the BDD from the buffer it already has.

Similarly, the master does not have to dump its BDD to a buffer every time it sends a BDD to a slave. It can run the serialization algorithm only when a BDD is changed. All other times, it can use the buffer, which is the result of the last serialization.

Having these optimizations, network utilization drops and communication overhead becomes negligible compared to the actual slaves work.

5. EXPERIMENTAL RESULTS

We implemented distributed reordering within the BDD-based model-checking engines of IBM's formal-verification tool, RuleBase [1]. We used five 2.4GHz Intel Xeon machines with 2GB RAM each. For all algorithms we applied upper bound, closest end and lower bounds as in [4] (see Subsection 2.3). A topological sort from observed outputs determined the initial BDD order for all runs.

First, we compare the two merging algorithms we suggested, as well as search splitting. Search splitting was implemented using synchronous merging. We used four slaves in addition to the master application. The six examples are the same as in Section 3. Figure 2 shows total reordering time (for all reordering sessions) for each solution. Times are presented as percentage of the reordering time using the sequential algorithm.

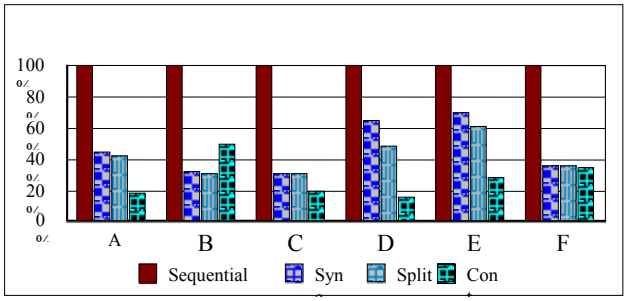

Figure 2, Reorder time using 4 slaves as % of sequential time

Results show reordering time was decreased on average to 49.6% for synchronous merging without search splitting, 44% for synchronous merging with search splitting and 23.4% for continuous merging without search splitting.

Reducing reordering time is of no use if reordering quality drops, and the application runs slower as a result. We therefore also measured the total runtime of the model-checking engine for each of the above solutions. Results are shown in Figure 3. Times are presented as percentage of the total model checking time using sequential reordering.

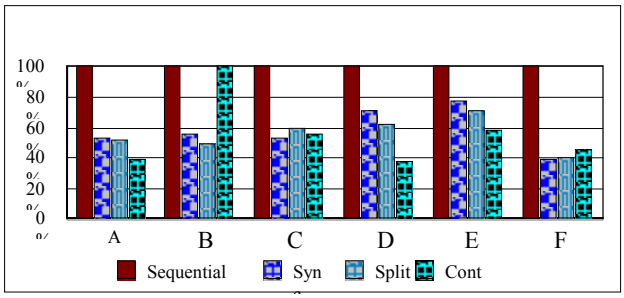

Figure 3, Total model checking time using 4 slaves as % of sequential time

Results show total model checking time was decreased on average to 58.8% for synchronous merging without search splitting, 57% for synchronous merging with search splitting and 47.2% for continuous merging.

It is also evident from the results that continuous merging does compromise reordering quality in exchange for reordering speedup. Speedups for total model checking time are not far better than the other solutions, as they were when looking at reordering speedups. Yet, this is clearly the winning solution.

Finally, we measured the scalability of our distributed solution. We ran the algorithm with one to four slaves, and compared the

results to the sequential solution. We used continuous merging, and measured only total model checking time, as this is the parameter that actually counts for the end user. We used here a much larger set of designs, containing 23 industrial hardware designs, not all of them are reordering intensive. Figure 4 shows both average speedups and global speedups.

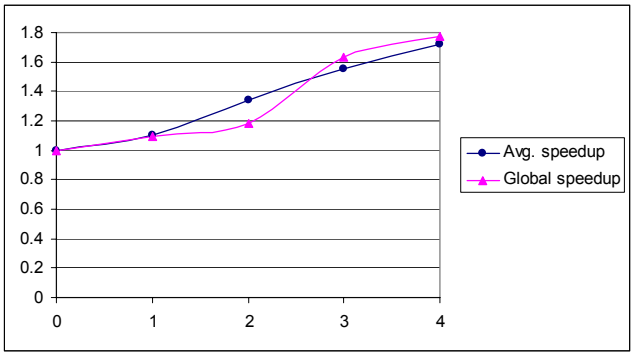

Figure 4 –Scalability of distributed solution using 0-4 slaves

Results show that the average speedup for the total model checking time grows linearly by almost 0.2 for each added slave. Absolute global times are 9:44 hours for the sequential algorithm, 8:55 using one slave, 8:15 using two slaves, 5:58 using three slaves and 5:29 using four slaves. None of the runs ran out of memory, showing that reordering quality was never significantly compromised.

Recall that these are not particularly reordering-intensive examples. Typically, model-checking engines spend most of their time in BDD operations rather than in reordering. Nevertheless, speedups are still significant.

6. CONCLUSIONS

We studied a few properties of Rudell's sifting algorithm, and discovered that typically many variables do not relocate, while many of those that do relocate do not bring a significant reduction in BDD size. Rudell's algorithm therefore spends much of its time sifting variables for nothing.

Based on the above observation we developed a distributed version of Rudell's algorithm. The algorithm allows various CPUs to try relocating different variables in parallel. Results are then gathered and applied to the original BDD.

Our approach proved to be efficient, speeding up dynamic reordering by a factor of 4 and speeding up application time by a factor of 1.8, using 4 slaves.

An interesting direction for future research would be finding better ways to combine search results, so that synchronization efforts are minimal but reordering quality is not compromised. Also, it may be interesting to decide what the optimal number of slaves is.

7. ACKNOWLEDGMENTS

Our thanks to Dan Badescu for his help in carrying out some of the experiments.

8. REFERENCES

[1] Beer, I., Ben-David, S., Eisner, C., and Landver, A. RuleBase: An Industry-oriented formal verification tool. *ACM IEEE Design Automation Conferenc*e, (1996), 655-660.

[2] Bollig, B., and Wegener, I. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transaction on Computers*, 45, 9 (September, 1996), 993-1002.

[3] Bryant, R. E. Graph-based algorithms for Boolean function manipulation. *IEEE Transaction on Computers*, C-35, 8 (August, 1986), 677-691.

[4] Drechsler, R., and Gunther, W. Using lower bounds during dynamic BDD minimization. *IEEE Transactions on CAD*, 20 (2001), 51-57.

[5] Ebendt R., and Drechsler, R. Lower bounds for dynamic BDD reordering. *Asia and South Pacific Design Automation Conference*, (2005), 579-582.

[6] Fujii, H., Ootomo, G., and Hori, C. Interleaving based variable ordering for binary decision diagrams, *Proceedings of International Conference on Computer-Aided Design*, (1993), 38-41.

[7] Fujita, M., Matsunaga, and Y., Kakuda, T. On variable ordering of binary decision diagrams for the application of multilevel synthesis. *European Conference on Design Automation* (1991), 50-54.

[8] Heyman, T., Geist, D., Grumberg, O., and Schuster, A. Achieving scalability in parallel reachability analysis of very large circuits. *Proceedings of the 12th International Conference on Computer Aided Verification*, (2000), 20-35.

[9] McMillan, K. L. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[10] Meinel, C., and Slobodova, A. Speeding up variable reordering of OBDD. *International Conference on Computer Design*, (1997), 338-343.

[11] Milvang-Jensen, K., and Hu, A. J. BDDNOW: A Parallel BDD Package. *Formal Methods in Computer-Aided Design*, (1998), 501-507.

[12] Lind-Nielsen, J. BuDDy: Binary Decision Diagram package, http://sourceforge.net/projects/buddy, 2002.

[13] Panda, S., and Somenzi, F. Who are the variables in your neighborhood. *International Conference on Computer-Aided Design*, (1995), 74-77.

[14] Rudell, R. Dynamic variable reordering for ordered binary decision diagrams. *International Conference on Computer-Aided Design*, (1993), 42-27.

[15] Slobodova, A., and Meinel, C. Sample method for minimization of OBDDs. *Proceedings of the 25th Conference on Current Trends in Theory and Practice of Informatics: Theory and Practice of Informatics*, (1998), 419-428.

[16] Somenzi, F. CU Decision Diagram Package Release 2.4.1, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html, University of Colorado at Boulder, 2005.