Equivalence Checking Combining a Structural SAT-Solver, BDDs, and Simulation

Viresh Paruthi
IBM Enterprise Systems Group
Austin, TX 78758

Andreas Kuehlmann*
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

This paper presents a verification technique for functional comparison of large combinational circuits using a novel combination of known approaches. The idea is based on a tight integration of a structural satisfiability (SAT) solver, BDD sweeping, and random simulation; all three working on a shared graph representation of the circuit. The BDD sweeping and SAT solver are applied in an intertwined manner both controlled by resource limits that are successively increased during each iteration. In this cooperative setting the BDD sweeping incrementally reduces the search space for the SAT solver until the problem is solved or the resource limits are exhausted. This approach improves on previous work in several ways: The integral application of the SAT solver significantly enhances the capacity and efficiency of BDD sweeping and extends its suitability for miscomparing designs. Further, the random simulation algorithm works on the compressed circuit graph and thus runs more efficiently. Our experiments demonstrate that the outlined approach is effective for a large class of equivalence checking instances by automatically adapting to the difficulty of the problem.

1 Introduction

Functional equivalence checking is ubiquitous in most contemporary design verification methodologies. Because of the computational complexity of formal equivalence checking the design methodology typically adopts specific rules to make the problem tractable for large designs. For example, matching state encoding of the specification and implementation reduces the general equivalence checking problem to the combinational domain for which many practical algorithms exist. In practice, the specification and implementation have often a large degree of structural similarity in terms of internal nets that often implement the same function. Advanced equivalence checking algorithms exploit this similarity to partition the problem into a series of smaller problems [1, 2, 3, 4, 5].

*Currently with Cadence Berkeley Labs, Berkeley, CA 94704

There are three basic approaches to functional equivalence checking. Structural methods are based on SAT solvers which search for a counterexample to prove inequivalence of the two functions. Similarly, random simulation is used to find a counterexample by random search. Functional methods are based on canonical function representations for which structural equivalence implies functional equivalence.

One of the first practical programs using a functional approach to verifying logic designs was SAS (Static Analysis System) [6]. SAS is based on the DBA (Differential Boolean Analyzer) and ESP (Equivalent Sets of Partials) algorithms, which are similar to unordered BDDs [7] in their unreduced and reduced forms, respectively. Because of their efficiency, Reduced Ordered BDDs (ROBDD) [8] became widely accepted for functional equivalence checking. The advantage of pure functional methods is their independence of structural similarities. However, in many cases they suffer from exponential memory usage. For example, it has been our experience that roughly 20% of the circuit outputs of a typical high-end microprocessor cannot practically be verified with pure BDDs.

Internal equivalence points or *cutpoints* can be used to decompose the equivalence checking problem into smaller pieces. The original approach [1] treats cutpoints as independent inputs which, in effect, dramatically reduces the overall size of the BDD representations of all pieces. However, since the possible functional correlation of these cutpoints is disregarded, *false negative* verification results may be introduced. There are a number of methods [9, 10, 11, 4, 5] to eliminate false negative verification results. Specifically, in [5] a tight integration of a SAT solver, BDDs and random simulation is suggested. In contrast to our work, the collaboration of those algorithms is limited to the identification of cutpoints and the elimination of false negatives in an iterative manner.

The concept of a "Miter" [2] portrays the combination of the two circuit cones to be compared with an XOR gate. Using this structure the actual equivalence checking problem is recast as a synthesis problem. If the Miter can be transformed into a constant “0”, functional equivalence is proven. In [2] this is done by merging the two cones starting from the inputs towards the outputs by applying a series of SAT queries for justifying the merging of internal nets. The drawback of this method is that the underlying SAT routine can only test whether two candidate nets can be merged. This requires additional effort to guess candidates which, in practice, is quite inefficient. Similarly, in [3] recursive learning is applied to prove functional equivalence using resynthesis.

A constructive method to find functionally identical nets in a Miter-based equivalence checking approach is presented in [4]. A technique called “BDD sweeping” builds BDDs for the Miter nodes from the inputs towards the outputs in a robust manner. By keeping cross references between the corresponding nodes of the BDD and Miter, functionally equivalent Miter nodes can be identified on the fly and merged immediately. This BDD sweeping technique is applied in several layers, starting from multiple internal cut frontiers. Further, a robust technique to eliminate false negatives is described. The main drawback of this approach is that BDD sweeping is not capable of proving inequivalence. Further, for hard problem instances, the sweeping algorithm might run out of memory before it can prove equivalence.

In this paper we address the above mentioned problems in two ways. First, the BDD sweeping is supplemented with a structural SAT solver working on the same circuit graph data structure. BDD sweeping and SAT search tackle the verification problem from a different angle with distinct strengths. BDD sweeping merges the two cones of the Miter structure from the inputs toward the outputs. For difficult instances it might be required to build large BDDs during sweeping before the two outputs to be compared can be merged. In contrast, the SAT solver attempts to prove equivalence or inequivalence by a structural search from the outputs toward the inputs. If only a small fraction of the two cones is shared, the search might require too many backtracks and may not finish. However, any structural cone sharing produced by BDD sweeping reduces the search space for the SAT solver dramatically and thus increases the chances of its success. We proposed an intertwined application of BDD sweeping and SAT search with iteratively increased resource limits for both algorithms. This interlaced scheme is able to dynamically adjust the effort needed by BDD sweeping to help make the SAT solver successful.

A second supplement to the BDD sweeping algorithm is a random simulator which works on the same graph representation and attempts to demonstrate functional inequivalence of the two circuits. Empirical evidence has shown that a large number of miscompares can be found by simulating the network with a relatively small number of patterns [12]. Similar to the SAT solver, the random simulator gets invoked after each BDD sweeping step.

2 Equivalence Checking Algorithm

2.1 BDD Sweeping

In this section we briefly summarize the BDD sweeping algorithm presented in [4] which is the basis for the method described here. The purpose of the BDD sweeping algorithm is to prove functional equivalence of two cones by transforming the output of the Miter into a constant “0”. When the Miter structure is built from the circuits, all Boolean operations are converted into a graph representation using two-input AND vertices and edges with an optional INVERTER attribute. Similar to efficient BDD implementations [13], each vertex is entered into a hash table using the identifier of the two predecessor vertices and their edge attributes as key. This hash table is utilized during graph construction to identify isomorphic subnetworks and to immediately merge them on the fly. After the Miter graph representation is built, BDD sweeping is applied to further compress it. The pseudo-code for the basic BDD sweeping algorithm is outlined in Figure 1. The depicted algorithm omits the specifics of multiple sweeping layers from intermediate cut frontiers and also the handling of false negatives. Details can be found in [4].

The overall idea of the sweeping algorithm is to merge the subgraphs of the two output cones using BDDs to prove functional equivalence of intermediate vertices. The BDD propagation is controlled by a sorted heap. For each primary input a BDD variable is created and entered onto the heap (see Figure 2). Then an iterative process removes the smallest BDD from the heap, processes the Boolean operation for the immediate fanouts of the corresponding circuit graph vertex, and reenters the resulting BDDs onto the heap. This process is continued until either all BDDs up to a given size limit are processed, or the heap becomes empty.

Procedure **put_on_heap** stores a BDD on the heap only if its size is smaller than the given limit, otherwise the BDD node is freed. Functionally equivalent vertices found during BDD construction are immediately merged and the subsequent parts of the circuit graph are rehashed by the routine **merge_vertices**. The rehashing is applied in depth-first order starting from the merged vertex toward the primary outputs and stops if no further reconvergence is found. Procedures **get_bdd_from_vertex** and **get_vertex_from_bdd** provide cross-references between BDD nodes and the corresponding circuit graph vertices. Inverted edge attributes are handled internally in those routines.

Similar to [4] the upper size limit for the BDD handling ensures a robust application of the algorithm and avoids

```
Algorithm bdd_sweep( $heap, v, bdd\_lower\_size\_limit,$
                                   $bdd\_upper\_size\_limit$) {
  /* check if there are any BDDs on heap with
     size($bdd$) $\leq$ $bdd\_lower\_size\_limit$ */
  while (!is_heap_empty($heap$, $bdd\_lower\_size\_limit$)) do {
     $bdd$   = get_smallest_bdd($heap$);
     $v$     = get_vertex_from_bdd($bdd$);

     /* check if previously encountered */
     if (get_bdd_from_vertex($v$)) continue;
     store_bdd_at_vertex($v, bdd$);
     for all fanout_vertices $v_{out}$ of $v$ do {
        $bdd_{left}$  = get_bdd_from_vertex(get_left_child($v_{out}$));
        $bdd_{right}$ = get_bdd_from_vertex(get_right_child($v_{out}$));
        $bdd_{res}$   = bdd_and($bdd_{left}, bdd_{right}$);
        $v_{res}$     = get_vertex_from_bdd($bdd_{res}$);
        if ($v_{res}$) {
            merge_vertices($v_{res}, v_{out}$);
            /* return if Miter solved */
            if      ($v$ == $const\_0$) return 0;
            else if ($v$ == $const\_1$) return 1;
        } else {
            store_vertex_at_bdd($bdd_{res}, v_{res}$);
        }
        put_on_heap($heap, bdd_{res}, bdd\_upper\_size\_limit$);
     }
  }
  return $-1$;
}
```

Figure 1: BDD sweeping algorithm.

memory blow-ups. To allow reinvocation of BDD sweeping in an iterative manner, a lower size limit is added to the heap handling. The successive invocation of **bdd_sweep** with increased lower limits essentially "hides" bigger BDDs from the current run and automatically restarts the sweeping when these BDDs "reappear" in the next iteration.

2.2 SAT Solver

The integrated SAT solver is based on the Davis-Putnam procedure [14] and is implemented to operate on the AND/INVERTER circuit graph. The algorithm starts by asserting the output of the Miter to "1". It then successively iterates through a series of implication and case splitting steps until a solution is found. If a conflict occurs, the algorithm backtracks to the previous splitting level and continues with the next choice. If all choices are exhausted, the algorithm proves non-satisfiability of the Miter output, which implies equivalence of the two cones being compared. On the other hand, if a solution is found, it demonstrates non-equivalence by providing a counterexample.

The search heuristics of the given implementation are similar to [15]. The resources of the algorithm are controlled by a given backtracking limit. If this limit is exceeded, the algorithm returns with an undecided result. A particular setup of the backtracking stack allows the search process to restart, after it exceeds the backtracking limit, from the point it stopped. This feature is used in the intertwined invocation of the BDD sweeping algorithm and the SAT solver and avoids restarting the SAT search from scratch for each iteration.

2.3 Random Simulator

Similar to the SAT solver, a bit-parallel random simulator is implemented on the AND/INVERTER circuit graph. Our uniform circuit graph using identical vertex functions permits an efficient implementation of the simulation algorithm using bit-parallel operations for the AND-function and inversion. In case of input constraints, the SAT solver is used to generate valid input stimuli.

In order to save the memory needed to store the simulation results at intermediate vertices, "pattern-space dragging" [12] is used. This scheme uses a reference counting mechanism to determine when the simulation pattern at a vertex has been propagated to all destinations. When the last destination queries the pattern of a vertex, the pattern space can be "dragged" to the destination vertex by performing the simulation operation in-place.

```
Algorithm check_equivalence($v1, v2$) {
  $v$ = XOR($v1, v2$);
  if ($v$ == $const\_0$) return equal;
  if ($v$ == $const\_1$) return not_equal;

  for all primary_inputs $I$ do {
     $bdd$ = create_bdd_variable();
     put_on_heap($heap, bdd, \infty$);
  }
  while (!is_heap_empty($heap, \infty$)) do {
     $bres$ = bdd_sweep($heap, v, bdd\_lower\_size\_limit,$
                                  $bdd\_upper\_size\_limit$);
     if      ($bres$ == 1) return not_equal;
     else if ($bres$ == 0) return equal;

     $sres$ = sim_equal($v1, v2$);
     if      ($sres$ == 1) return not_equal;
     else if ($sres$ == 0) return equal;

     $jres$ = sat_justify($v, 1, sat\_backtrack\_limit$);
     if      ($jres$ == 1) return not_equal;
     else if ($jres$ == 0) return equal;

     $bdd\_lower\_size\_limit$ += $delta\_bdd\_limit$;
     $sat\_backtrack\_limit$   += $delta\_sat\_limit$;
  }
  $jres$ = sat_justify($v$,1,$max\_sat\_backtrack\_limit$);
  if      ($jres$ == 1) return not_equal;
  else if ($jres$ == 0) return equal;

  return undecided;
}
```

Figure 2: Algorithm for equivalence checking.

BDD Size Limit	BDD Sweeping Memory [kB] / Time [sec]	SAT Search # Backtracks / Time [sec]	Total (BDD + SAT) Memory [kB] / Time [sec]
2^0	**342 / 0.00**	**2407939 / 347.17**	**342 / 347.17**
2^1	347 / 0.64	115 / 7.40	347 / 8.04
2^2	349 / 0.60	115 / 7.47	349 / 8.07
2^4	**358 / 0.66**	**87 / 6.93**	**358 / 7.59**
2^6	372 / 0.78	43 / 6.88	372 / 7.66
2^8	396 / 1.18	43 / 7.03	396 / 8.21
2^{10}	791 / 1.67	43 / 6.57	791 / 8.24
2^{12}	2212 / 4.22	43 / 6.30	2212 / 10.52
2^{14}	2219 / 6.98	43 / 6.15	2219 / 13.13
2^{16}	8381 / 12.14	27 / 5.27	8381 / 17.41
2^{17}	**8540 / 19.16**	**0 / 0.00**	**8540 / 19.16**

Table 1: Performance of BDD sweeping and SAT search for various BDD size limits.

2.4 Overall Algorithm

For each pair of cones to be compared, first the Miter structure is built using the AND/INVERTER graph structure. During construction, this structure is locally optimized using a multi-input hashing scheme [16]. Next, the algorithm outlined in Figure 2 is invoked. It first checks if the structural hashing algorithm could prove equivalence of the two cones. Note that for a large number of practical applications, this structural test is successful. For example, in a typical ASIC methodology, equivalence checking is used to compare the logic before and after insertion of the test logic. Since no logic transformations have actually changed the circuit, a simple structural check suffices to prove equivalence.

Next BDD sweeping, random simulation, and the SAT solver are invoked in an intertwined manner. During each iteration, the size limit for BDD sweeping and the backtrack limit for the SAT solver are increased. It is important to note that in the given setting, these algorithms do not just independently attempt to solve the equivalence checking problem. Each BDD sweeping iteration incrementally compresses the Miter structure from the inputs towards the outputs, which effectively reduces the search space for the SAT solver. This interleaved scheme dynamically determines the minimum effort needed by the sweeping algorithm to make the SAT search successful.

2.5 Example

In the following we demonstrate the cooperative integration of the BDD sweeping algorithm and SAT search, using a circuit example from a high-performance microprocessor design. We selected an output pair which had 97 inputs, 1322 gates for the specification and 2782 gates for the implementation. The comparison was done under a simple input constraint which is handled equally well by BDD sweeping and SAT search.

In a series of experiments the BDD sweeping algorithm was applied to the original Miter circuit with varying limits for the BDD size. After sweeping, the SAT solver was invoked on the compressed Miter structure and run until equivalence was proven. Table 1 gives the results for different limits on the BDD size. As shown, there is a clear tradeoff between the effort spent in BDD sweeping and SAT search. The optimal performance is achieved with a BDD size limit of 2^4. The application of BDD sweeping and SAT search in the described incremental and intertwined manner heuristically adjusts the effort spent on each algorithm to the difficulty of the problem.

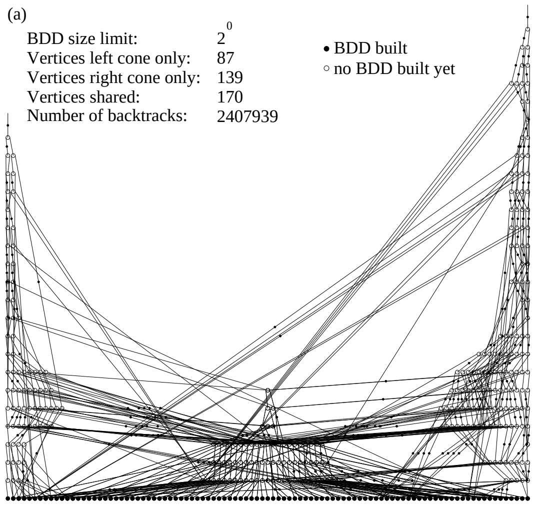
Figure 3 shows the actual Miter structures for three selected runs. In the drawing, all inputs are positioned at the bottom. The placement of the AND vertices is done based on their connectivity to the two outputs which are located at the top. AND vertices that feed only one of the two outputs are aligned on the left and right side of the picture. Vertices that are shared between both cones are placed in the middle. Further, filled circles and open circles are used to distinguish between vertices with and without BDDs, respectively. Filled dots on the edges symbolize inverters.

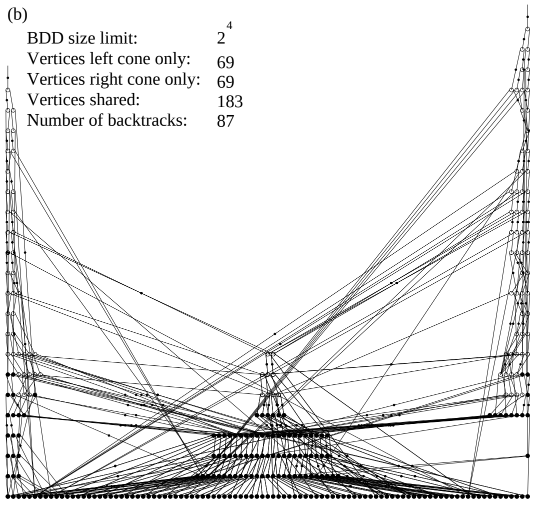
Part (a) of the picture illustrates the initial Miter structure without performing any BDD sweeping. As shown, a number of vertices are shared as a result of structural hashing (including the scheme described in [16]). In order to prove equivalence at this stage, the SAT solver would need about 2.5 million backtracks. Figure 3(b) shows the Miter structure after performing a modest BDD sweep with a size limit of 16 BDD nodes. It is clear that many more vertices are shared at this point. The SAT solver can now prove equivalence using only 87 backtracks. The last part of the picture displays the Miter structure when it is completely merged by BDD sweeping. As shown, the equivalence proof required building BDDs for all Miter vertices.

3 Experimental Results

We implemented the presented algorithms in the equivalence checking tool Verity [17]. In order to evaluate its effectiveness we applied the new algorithm to check the equivalence of 132 circuits randomly selected from a currently developed high-performance microprocessor. The circuits range in size from a few 100 to $100K$ gates, the size distribution is shown in Figure 4. The number of output comparisons per circuit ranges from a few 100 to more than 10000. All designs were developed in a custom design methodology, for which the RTL specification is often significantly different than the transistor-level implementation. The experiments were performed on a RS/6000 model 270 with a 64-bit, two-way Power3 processor running at 375 MHz and 8 GBytes of main memory.

For each circuit we first run the default setting of Verity


(a)
BDD size limit: 2^0
Vertices left cone only: 87
Vertices right cone only: 139
Vertices shared: 170
Number of backtracks: 2407939
• BDD built
○ no BDD built yet



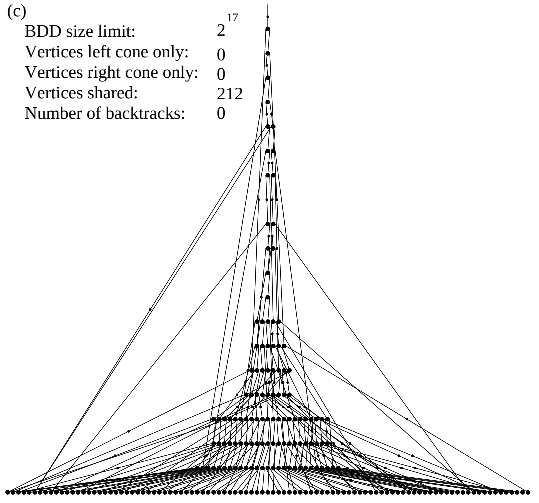


Figure 3: Example Miter structure at different stages of BDD sweeping: (a) No sweeping performed, (b) sweeping result with BDD size limit of 2^4, (c) sweeping result with BDD size limit of 2^{17}.

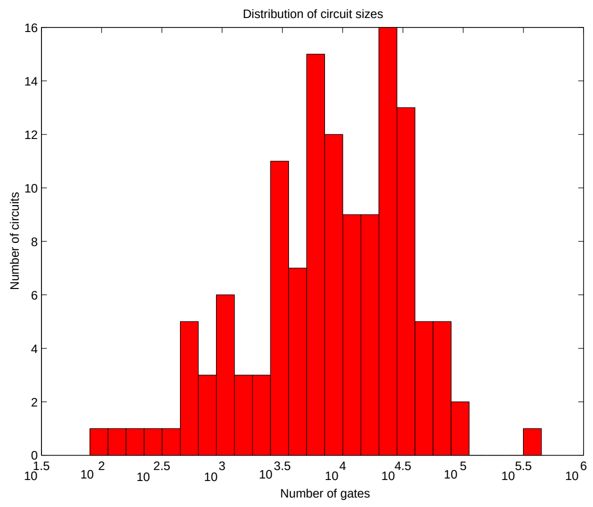


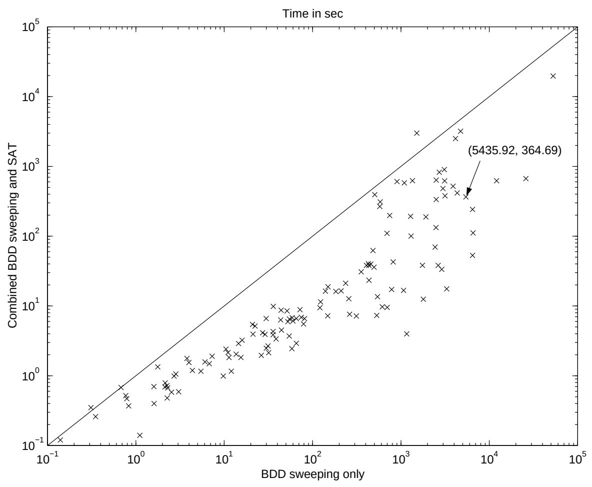
Figure 4: Distribution of circuit sizes for the experiment.

which invokes as standard engine the original BDD sweeping algorithm presented in [4]. Note, that the performance of the standard setting can often be improved by selecting a circuit-specific option. We chose the standard setting for all circuits to avoid skewing the comparison. Further, one of the criteria we would like to measure in this experiment is the robustness of the new approach for various circuits. The second Verity run used a newly developed engine based on the algorithm described in this paper.

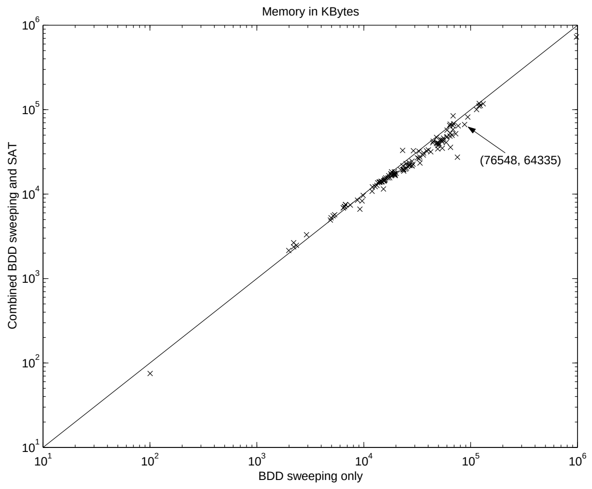
Overall, the new algorithm performs significantly better than the original version of the BDD sweeping technique. Figures 5 (a) and (b) display a comparison of the runtime and memory, respectively. Each marker represents a circuit, with the runtime/memory of the original BDD sweeping algorithm on the x-axis and the runtime/memory of the new algorithm on the y-axis. As shown, the majority of circuits could be compared using significantly less time, sometimes by two orders of magnitude faster. The memory consumption remained about the same, mostly using less memory than before. The performance for a particularly functionally complex circuit is marked in both diagrams. This designs contains 92,043 gates, 97 primary inputs, 65 outputs, and 4,642 latches. The verification run included 4,707 comparisons and 166,194 consistency checks and could be accomplished in 364 versus 5,435 seconds using 64 versus 77 MBytes for the new and old engine, respectively.

4 Conclusions

This paper presents a new approach to functional comparison of large combinational circuits using a combination of BDD sweeping, structural SAT search, and random simulation. All three techniques are implemented on a uniform AND/INVERTER circuit graph that helps to maxi-



(a)



(b)

Figure 5: Comparison of the original BDD sweeping algorithm with the new algorithm: (a) Runtime comparison, (b) Memory comparison.

mize the overall performance and allows a unique combination of their individual strengths. In particular, by applying the algorithms in an intertwined manner, each BDD sweeping step incrementally reduces the problem size for the SAT solver until it can be solved. This scheme leads to a robust verification algorithm that can solve a large class of equivalence checking problems with reduced overall effort.

5 Acknowledgements

The authors would like to thank Geert Janssen from IBM Thomas J. Watson Research Center and Malay K. Ganai from the University of Texas at Austin for their technical contributions. They also wish to thank Victor Rodriguez from IBM Austin for supporting the project.

References

[1] C. L. Berman and L. H. Trevillyan, “Functional comparison of logic designs for VLSI circuits,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 456–459, IEEE, November 1989.

[2] D. Brand, “Verification of large synthesized designs,” in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 534–537, IEEE, November 1993.

[3] W. Kunz and D. Pradhan, “Recursive learning: a new implication technique for efficient solutions to cad problems-test, verification, and optimization,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 125, pp. 684–694, May 1993.

[4] A. Kuehlmann and F. Krohm, “Equivalence checking using cuts and heaps,” in *Proceedings of the 34th ACM/IEEE Design Automation Conference*, (Anaheim, CA), pp. 263–268, ACM/IEEE, June 1997.

[5] J. R. Burch and V. Singhal, “Tight integration of combinational verification methods,” in *Proceedings of the 34th ACM/IEEE International Conference on Computer Design*, (Austin, TX), pp. 570–576, ACM/IEEE, November 1998.

[6] G. L. Smith, R. J. Bahnsen, and H. Halliwell, “Boolean comparison of hardware and flowcharts,” *IBM Journal of Research and Development*, vol. 26, pp. 106–116, January 1982.

[7] S. B. Akers, “Binary decision diagrams,” *IEEE Transactions on Computers*, vol. 27, pp. 509–516, June 1978.

[8] R. E. Bryant, “Graph-based algorithms for Boolean function manipulation,” *IEEE Transactions on Computers*, vol. 35, pp. 677–691, August 1986.

[9] C. van Eijk and G. Janssen, “Exploiting structural similarities in a BDD-based verification method,” in *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design*, pp. 110–125, 1995.

[10] S. M. Reddy, W. Kunz, and D. K. Pradhan, “Novel verification framework combining structural and OBDD methods in a synthesis environment,” in *Proceedings of the 32th ACM/IEEE Design Automation Conference*, (San Francisco), pp. 414–419, ACM/IEEE, June 1995.

[11] Y. Matsunaga, “An efficient equivalence checker for combinatorial circuits,” in *Proceedings of the 33th ACM/IEEE Design Automation Conference*, (Las Vegas), pp. 629–634, ACM/IEEE, June 1996.

[12] F. Krohm, A. Kuehlmann, and A. Mets, “The use of random simulation in formal verification,” in *Proceedings of the IEEE International Conference on Computer Design*, (Austin, TX), pp. 371–376, IEEE, October 1996.

[13] K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient implementation of a BDD package,” in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, (Orlando), pp. 40–45, ACM/IEEE, June 1990.

[14] M. Davis and H. Putnam, “A computing procedure for quantification theory,” *Journal of of the Association of for Computing Machinery*, vol. 7, pp. 102–215, 1960.

[15] S. Kundu, L. M. Huisman, I. Nair, V. Ivengar, and S. M. Reddy, “A small test generator for large designs,” in *Proceedings of the International Test Conference*, pp. 30–40, 1992.

[16] M. K. Ganai and A. Kuehlmann, “On-the-fly compression of logical circuits,” in *Proceedings of the International Workshop on Logic Synthesis*, (Dana Point, CA), June 2000.

[17] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, “Verity - a formal verification program for custom CMOS circuits,” *IBM Journal of Research and Development*, vol. 39, pp. 149–165, January/March 1995.