Circuit-based Boolean Reasoning

Andreas Kuehlmann*
Cadence Berkeley Labs
Berkeley, CA 94704

Malay K. Ganai
The University of Texas at Austin
Austin, TX 78750

Viresh Paruthi
IBM Enterprise Systems Group
Austin, TX 78758

Abstract

Many tasks in CAD, such as equivalence checking, property checking, logic synthesis, and false paths analysis require efficient Boolean reasoning for problems derived from circuit structures. Traditionally, canonical representations, e.g., BDDs, or SAT-based search methods are used to solve a particular class of problems. In this paper we present a combination of techniques for Boolean reasoning based on BDDs, structural transformations, and a SAT procedure natively working on a shared graph representation of the problem. The described intertwined integration of the three techniques results in a robust summation of their orthogonal strengths. Our experiments demonstrate the effectiveness of the approach.

1 Introduction

Many tasks in computer-aided design such as equivalence or property checking, logic synthesis, static timing analysis, and automatic test-pattern generation require Boolean reasoning on problems derived from circuit structures. There are two main approaches used alternatively for such applications. First, by converting the problem into a functionally canonical form such as BDDs, the solution can be obtained from the resulting structure. Second, a SAT search performed on the original circuit representation either encounters individual solutions or, if all cases have been enumerated, concludes that no solution exists. Both approaches generally suffer from exponential worst-case complexity. However, they have distinct strengths and weaknesses which make them applicable to different classes of practical problems. A monolithic integration of SAT and BDD-based techniques could combine their individual strengths and result in a robust solution for a wider range of applications.

Many practical problems derived from the above mentioned applications have a high degree of structural redundancy. There are three main sources for this redundancy: First, the primary netlist derived from an RTL specification contains redundancies generated by language parsing and processing. For example, in industrial designs, between 30 and 50% of generated netlist gates are redundant [1]. A second source of structural redundancy is inherent to the actual problem formulation. For example, a Miter structure [2], built for equivalence checking, is globally redundant. It also contains many local redundancies in terms of identical substructures used in both designs to be compared. A third source of structural redundancy originates from repeated invocations of Boolean reasoning on similar problems derived from overlapping parts of the design. For example, the individual paths checked during false paths analysis are composed of shared sub-paths which get repeatedly included in subsequent checks. Similarly, a combinational equivalence check of large designs is decomposed into a series of individual checks of output and next state functions which often share a large part of their structure. An approach that detects and reuses structural and local functional redundancies during problem construction could significantly reduce the overhead of repeated processing of identical structures. Further, a tight integration with the actual reasoning process can increase its performance by providing a mechanism to efficiently handle local decisions.

In this paper we present an approach that integrates BDD sweeping [3], local structural transformations, and a circuit-based SAT procedure in one framework. All three techniques work on a shared AND/INVERTER graph represention of the problem. BDD sweeping and SAT solver are applied in an intertwined manner both controlled by resource limits that are increased during each iteration [4]. BDD sweeping incrementally simplifies the graph structure, which effectively reduces the search space of the SAT solver until the problem can be solved. The set of local transformations based on functional hashing gets invoked when the sweeping algorithm causes a structural change, potentially solving the problem or simplifying the graph for the SAT search.

2 Previous Work

SAT search has been extensively researched in several communities. Many of the published approaches are based on the Davis-Putnam procedure [5] which executes a systematic case split to perform an exhaustive search in the solution space. Over the years, many search tactic improvements have been published, the most noticeable implementation is GRASP, a CNF-based SAT solver [6]. Grasp uses an implication graph for initiating non-chronological backtracks during the search process and for learning additional CNF clauses to avoid symmetrical conflict cases in the future. Classic SAT solvers such as GRASP are difficult to integrate with BDD sweeping and dynamic circuit transformations because of the CNF representation of the problem. In this paper we present an efficient implementation of a SAT procedure working on an

*This work was mostly done while author was with IBM.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

AND/INVERTER graph allowing a tight interaction with sweeping and local circuit graph transformations. We describe a modified implementation of the non-chronological backtracking and conflict-based learning and present an efficient means to learn static implications which reuses the hash table of the AND/INVERTER graph.

Trading off compactness of Boolean function representations with canonicity for efficient reasoning in CAD applications has been the subject of many publications. BDDs [7] represent one extreme of the spectrum which map Boolean functions onto a canonical graph represention. Deciding whether a function is a tautology can be done in constant time, at the possible expense of an exponential graph size. XBDDs [8] propose to divert from the strict functional canonicity by adding function nodes to the graph. The node function is controlled by an attribute at the referencing arc and can represent an AND or OR operation. Similar to BDDs, complementation is expressed by a second arc attribute and structural hashing identifies isomorphic subgraphs on the fly. The proposed tautology check is similar to a technique presented in [9] and is based on recursive inspection of all cofactors. This scheme effectively checks the BDD branching structure sequentially, resulting in exponential runtime for problems where BDDs are excessively large.

Another form of a non-canonical function graph representation are BEDs [10]. BEDs use a circuit graph with six possible vertex operations. The innovative component of BEDs is the application of local functional hashing, which maps any four-input sub-structure onto a canonical represention. The proposed tautology check is based on converting the BED structure into a BDD by moving the variable from the bottom to the top of the structure. Similar to many pure cutpoint-based methods, this approach is highly sensitive to the ordering in which the variables are pushed up. In our approach, we apply an extended functional hashing scheme to an AND/INVERTER graph representation. Since our graph preserves the AND clustering, the hashing can take advantage of its commutativity which makes it less sensitive to the order in which the structure is built. If the structural method fails, we apply BDD sweeping on the circuit graph for checking tautology. Due to the multiple frontiers, it is significantly more robust than the BED to BDD conversion.

Several publications have suggested an integration of SAT and BDD techniques for Boolean reasoning. In [11] a cutpoint-based equivalence checking approach is presented. First, a partial output BDD is built using the cutset as auxiliary inputs. Then the onset cubes of this BDD are enumerated and a SAT search is applied for justifying those cubes from the primary inputs. This approach becomes intractable if the BDD includes many cubes in its onset. Further, the actual justification of individual cubes may timeout if the cutset is chosen unwisely. A modification of this approach [12] suggests searching through all co-factors of the BDD instead of enumerating all cubes. Another proposal to combine BDD and SAT is based on partitioning the circuit structure into a set of components [13]. As most cutpoint-based methods, this approach is highly sensitive to the chosen partitioning.

A common problem with the mentioned integration approaches is the insertion of BDD operations into the inner loop of a structural SAT search. Structural SAT is efficient if the underlying problem structure can be exploited for effective local search heuristics. BDDs work well if redundancy of the problem structure eludes an exponential growth during construction. A spatial partitioning of the application space for BDDs and SAT blurs their individual global scope and separates the application of their orthogonal strengths to different parts.

3 On-the-Fly Reduction of Circuit Graphs

In this section we describe the basic AND/INVERTER graph representations and the set of local transformations that allow an efficient structural reduction. The data structure for all three algorithms is based on a graph using two-input AND vertices and INVERTER attributes on the edges. Similar to BDDs a hash table is used to remove structural redundancy during construction. Fur-

```
Algorithm and(p1, p2) {
  /* constant folding */
  if (p1 == CONST_0) return CONST_0;
  if (p2 == CONST_0) return CONST_0;
  if (p1 == CONST_1) return p2;
  if (p2 == CONST_1) return p1;
  if (p1 == p2)      return p1;
  if (p1 == !p2)     return CONST_0;
  /* rank order inputs */
  if (rank(p1) > rank(p2))  swap(p1, p2);
  /* check for isomorphic entry in hash table */
  if (p = hash_lookup(p1, p2)) return p;
  /* 3 cases depending on position in circuit */
  if (is_var(p1) && is_var(p2))
                    return new_and_vertex(p1, p2);
  else if (is_var(p1)) return and_3(p1, p2);
  else if (is_var(p2)) return and_3(p2, p1);
  else              return and_4(p1, p2);
}

Algorithm and_4(l, r) {
  ll    = left_child(l);
  lr    = right_child(l);
  rl    = left_child(r);
  rr    = right_child(r);
  index = get_canonical_case(l, r);
  switch(index) {
  /* First set:  grandchildren are shared */
    ...
    case 144: /* example of Figure 2 */
      return ¬and(rl, rr);
    ...
  /* Second set:  grandchildren are not shared */
    case 244: /* l and r both non-inverted */
      ...
    case 245: /* only l inverted */
      if (share(l, rl) || share(ll, rl) || share(lr, rl)) {
        return and(and(l, rl), rr);
      }
      if (share(l, rr) || share(ll, rr) || share(lr, rr)) {
        return and(and(l, rr), rl);
      }
      if (share(ll, r) || share(lr, r)) {
        return ¬and(¬and(¬ll, r), ¬and(¬lr, r));
      }
      break;
    case 246: /* only r inverted */
      ...
  }
  /* no reduction or rewriting */
  return new_and_vertex(l, r);
}

Algorithm new_and_vertex(p1, p2) {
  p = alloc_vertex(p1, p2);
  add_to_hash_table(p, p1, p2);
  /* learn implication shortcuts for SAT */
  if ((b1 = hash_lookup(¬p1, p2)) != NULL)  learn(p, b1);
  if ((b2 = hash_lookup(p1, ¬p2)) != NULL)  learn(p, b2);
  /* reschedule for BDD sweeping */
  put_on_heap(bdd_from_vertex(p1), upper_size_limit);
  put_on_heap(bdd_from_vertex(p2), upper_size_limit);
  return p;
}
```

Figure 1: Algorithm for circuit graph construction including on-the-fly reduction.

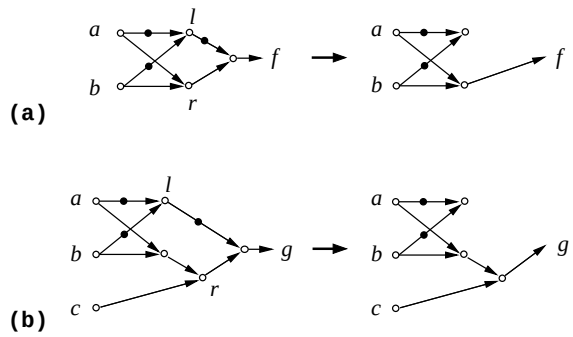

Figure 2: Example of local reduction using algorithm **and_4** of Figure 1: (a) reduction of $f = (a \vee b) \wedge ab$ using case 144, (b) reduction of $g = (a \vee b) \wedge abc$ using case 245 and then recursively case 144.

ther, we apply a two-level lookup scheme that converts any local four-input sub-structure into a canonical representation, effectively removing local redundancy. Note that due to its finer granularity this approach is strictly more powerful than a hashing scheme using four-input vertices [1]. If the local two-level lookup does not apply, we use simple rewriting rules to further reduce the circuit graph.

The algorithm **and** in Figure 1 shows the pseudo-code for the basic AND operation. The $\neg$ symbol denotes complementation of the inverter edge attribute. Similar to typical BDD data structures, this attribute is implemented by a single bit of the reference pointer.

The first part performs constant folding and a table lookup for isomorphic pre-existing vertices. Then a two-level lookup scheme converts the local function of the four grandchildren into a canonical representation. During circuit construction from the primary inputs, the first level of vertices does not have four grandchildren and therefore must be treated specially. If both immediate children are primary inputs, the algorithm calls the function **new_and_vertex**, which is also shown in Figure 1. It allocates the data structure, performs static learning for the SAT search (see Section 4), and reinitializes BDD sweeping for processing new fanouts (see Section 5). If only one of the children is a primary input, a canonical three-input substructure is created using the function **and_3**, in all other cases the routine **and_4** is called. The function of these two routines is very similar, thus we will limit our descriptions to the routine **and_4**, which is given in Figure 1. The function **and_3** simply implements a subset of the presented cases.

The algorithm **and_4** first analyzes the local structure and computes a signature which uniquely reflects (1) the equality relationship of the four grandchildren and (2) the inverter attributes of the six edges. The signature is then categorized into two sets of cases:

- The first set includes the local structures with shared grandchildren. Here all substructures with identical or complemented Boolean functions get mapped to an isomorphic implementation, effectively removing local redundancy. The pseudo-code in Figure 1 provides an example for case 144 illustrating this transformation for the function $\bar{x}y \vee x\bar{y}$. Figure 2a shows a particular instance of such a structure. In general, this mechanism is called recursively which often results in a significant global reduction of the graph. More details of this method can be found in [1].
- The second set of cases do not share any grandchildren. The signatures distinguish between the four cases of different polarity combinations of the children. For each case, if any sharing is found between grandchildren or great-grandchildren, specific rewriting rules are applied. Case 245 in Figure 1 illustrates that scheme for inverted left children. The algorithm **share** determines if any of the immediate predecessors of the arguments are shared. For example, if any of the children of r share a predecessor of l, the AND expression is rearranged such that a two-level hashing case of the first set can be applied. Figure 2b shows a rewriting example that first rearranged the structure and then applied the previous example for simplification. Note that recursive rewriting potentially loops unless special precaution is taken.

4 SAT Solver

The SAT solver implements a basic Davis-Putnam procedure on the presented AND/INVERTER graph. It attempts to find a set of consistent assignments that result in a logical 0 at the output. Tautology is proven if an exhaustive enumeration does not result in such an assignment. Figure 3 provides the basic flow of the algorithm.

```
Algorithm sat_justify(vertex, value, limit) {
  if (!imply(vertex, value)) return UNSAT;
  return justify(limit);
}

Algorithm justify(limit) {
  if (backtracks > limit) return UNDECIDED;
  mark = tail_pointer(assignment_list);
  if (v = next_vertex_from_queue(justification_queue)) {
    if (imply(v->left, 0) {
      if (res = justify(limit) != UNSAT) return res;
    }
    undo_assignments(mark);
    if (imply(v->right, 0) {
      if (res = justify(limit) != UNSAT) return res;
    }
    undo_assignments(mark);
  }
  return SAT; /* queue is empty */
}

Algorithm imply(vertex, value) {
  assign(vertex, value);
  lvalue     = get_value(vertex->left);
  rvalue     = get_value(vertex->right);
  next_state= lookup(value, lvalue, rvalue);
  switch (next_state) {
    case CONFLICT:
      return 0;
    case CASE_SPLIT:
      add_vertex_to_queue(vertex, justification_queue);
      return 1;
      ...
    case PROP_LEFT_AND_RIGHT:
      if(imply(vertex->left, next_state->lvalue &&
          imply(vertex->right, next_state->rvalue) {
            return 1;
      }
      return 0;
      ...
  }
  return 1;
}
```

Figure 3: General Davis-Putnam based SAT procedure.

A circuit-based implementation of the Davis Putnam algorithm can take specific advantage of the underlying problem struc-

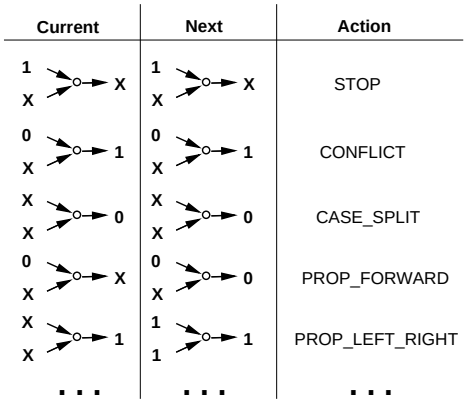

Figure 4: Lookup table for fast implication propagation.

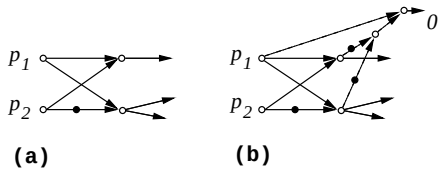

Figure 5: Learning for $(p_1 p_2 = 0) \wedge (p_1 \overline{p_2} = 0) \Rightarrow (p_1 = 0)$: (a) original structure, (b) structure with implication shortcut.

ture which allows a smooth integration into the presented overall scheme. For example, the fact that only one vertex function is used allows an efficient propagation of logic implications using a fast table lookup scheme. The algorithm **imply** iterates over the circuit graph and determines at each vertex new implied values and the directions for further processing. Figure 4 gives an excerpt from the implication lookup table. For Boolean logic only one case, a logical 0 at the output of an AND vertex, requires a new case split to be scheduled in the $justification\_queue$. All other cases cause a conflict, in which case the algorithm returns for backtracking, further implications, or a return to process the next element from the $justification\_queue$. The lookup table is programmable for different logics. For example, it can be applied to implement a parallel, one-level recursive learning scheme using nine-valued logic. Due to its uniformity and low overhead, the presented implication algorithm is highly efficient. As an indication, on a 400MHz Pentium class machine it can execute several hundred thousand backtracks per second. From a performance point of view, any gain that is potentially achieved by using additional analysis has to offset the resulting slowdown of the **imply** function. In [14] a similar reasoning is given for an efficient ATPG algorithms.

An effective mechanism to exploit the structure of the AND/INVERTER graph is illustrated in Figure 5. Using the hash table, a pair of vertices, which implement $p_1 p_2$ and $p_1 \overline{p_2}$ can be detected by two constant time lookups. This configuration occurs quite often in practical designs, for example, in multiplexer circuits to exchange data streams between two sources and two destinations. By adding two additional vertices to the graph, an implication shortcut can utilize the existing **imply** function. If a logical 0 is scheduled for both output vertices, the implication procedure can immediately justify the entire structure and bypass the two case splits. This learning structure is created statically and integrated into the circuit construction process as shown in algorithm **new_and_vertex** in Figure 1. Note that the learned vertices are built using the regular **and** routine which may cause additional circuit restructuring or learning events.

Advanced SAT solvers use conflict analysis to skip the evaluation of assignments which are symmetric to previously encountered conflicts [6]. Two mechanisms are used for this purpose: First, non-chronological backtracking skips the evaluation of case alternatives if the corresponding splitting variable was not involved in any lower-level conflict. Second, conflict-based learning creates an additional structure which reflects the assignment that have caused a particular conflict. This redundant structure causes additional implications which detect symmetric conflicts earlier.

Conflict analysis requires tracking the logical impact of case split assignments on the conflict points. Previously, this was implemented by an implication graph for which the nodes correspond to variables and edges reflect single implication steps. In the given setting, the conflict graph manipulation during each step of the **imply** routine would severely impact its performance. To reduce this penalty, we apply a mechanism that directly collects the responsible case assignments as a side function of the implication process. This mechanism uses a conflict bit-vector where each bit represents a case assignment in the decision tree. The table lookup in function **imply** is expanded to also determine the controlling sources for the propagation of the conflict bit-vectors. The actual propagation is done by bitwise OR operations along the implication sequence. This scheme reduces the speed of the implication process by typically less than 50%, a penalty that is easily offset by the average gain.

The resulting conflict bit-vectors are used to directly control the backtracking mechanism. If all choices of a decision level result in a conflict, the backtrack level is determined by the lowest level that was involved in a conflict. This is implemented efficiently by bit-vector operations. The combined conflict bit-vector reflects all responsible assignments for that part of the decision tree and is passed upward to the next backtracking level.

Further, the conflict bit-vector is examined for compact structures to learn. Similar to the learned structure of Figure 5, its setting is directly mapped into a circuit graph representing that conflict. To avoid excessively large learned structures, we apply a variable limitation similar to [6]. However, instead of just restricting the maximum number of conflict variables, we also take their assignment into account, effectively estimating the size of the eliminated decision subtrees.

The tight integration of the SAT solver into the overall framework requires an execution control by providing resource limits such as the number of backtracks. This supports an interleaved application of the SAT search with BDD sweeping. Further, by preserving the decision stack between subsequent invocations, the SAT algorithm can continue its search from the point it stopped before without repeating previous cases.

5 BDD Sweeping

For completeness, we briefly present the BDD sweeping algorithm. Figure 6 gives the general procedure; a number of details such as the BDD variable invocation, multi-layer BDD processing, and false-negative processing are omitted and can be found in [3]. The task of BDD sweeping is to efficiently find vertices

that are functionally identical or complemented. If a pair of equivalent vertices is found, the algorithm **merge_vertices** merges them and rebuilds their fanout structure forward until no change occurs. This rebuilding is based on the **and** function presented in Section 3 including the local restructuring scheme. The merging step may combine children or grandchildren of formerly unrelated vertices, which potentially results in a significant forward simplification of the circuit graph structure. This scheme is particularly effective for equivalence checking to "crush" the Miter circuit until equivalence is proven.

To enable re-invocation of BDD sweeping the original algorithm is modified. Instead of deleting the BDDs which exceed the size limit, they are "hidden" in the heap. When the sweeping algorithm is then reinvoked with a higher size limit, these BDDs "reappear" for further processing. An overall $bdd\_upper\_size\_limit$ is used to remove and free excessively large BDDs.

```
Algorithm bdd_sweep(heap, v, bdd_lower_size_limit) {
  /* check if there are any BDDs on heap with
     size(bdd) ≤ 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 problem solved */
          if (v == CONST_1) return UNSAT;
          if (v == CONST_0) return SAT;
       } else {
          store_vertex_at_bdd(bdd_res, v_out);
       }
       put_on_heap(heap, bdd_res, bdd_upper_size_limit);
     }
  }
  return UNDECIDED;
}
```

Figure 6: BDD sweeping algorithm.

6 Overall Algorithm

The overall algorithm combining structural transformations, BDD sweeping, and SAT is outlined in Figure 7. It first checks if the structural hashing algorithm did solve the problem. Next BDD sweeping and the SAT solver are invoked in an intertwined manner [4]. During each iteration, the size limit for BDD sweeping and the backtrack limit for the SAT solver are increased. Note that in the given setting, these algorithms do not just independently attempt to solve the problem. Each BDD sweeping iteration incrementally compresses the structure from the inputs toward 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.

7 Experiments

In order to evaluate the effectiveness of the presented approach we performed a number of extensive experiments using 488 circuits

```
Algorithm check_SAT(v) {
  if (v == CONST_0) return SAT;
  if (v == CONST_1) return UNSAT;

  while (!is_heap_empty(heap, ∞)) do {
     res = bdd_sweep(heap, v, bdd_lower_size_limit,);
     if (res != UNDECIDED) return res;
     res = sat_justify(v, 0, sat_backtrack_limit);
     if (res != UNDECIDED) return res ;

     bdd_lower_size_limit += delta_bdd_limit;
     sat_backtrack_limit  += delta_sat_limit;
  }
  return sat_justify(v,0,max_sat_backtrack_limit);
}
```

Figure 7: Overall algorithm integrating BDD sweeping and SAT search.

randomly selected from a number of microprocessors designs. The circuits range in size from a few 100 to $100K$ gates with a size distribution given in Figure 8. The number of outputs and inputs per circuit range from a few 100 to more than 10,000. 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.

In the first experiment, we evaluated the effectiveness of the AND/INVERTER graph structure and the multi-level hashing scheme. For this we constructed the circuit graph for the design specification and compared the results of the simple hashing method with the size generated by the new multi-input method presented in Section 3. The histogram for the size reduction of the circuit graphs is plotted in Figure 9. As shown, on average the given sample of circuit representations can be reduced by 50%, the runtime overhead for all runs was negligible. In a few cases, the size actually grows, however, this increase is easily offset by savings in other parts of the circuit. The results suggests that the presented on-the-fly circuit compression method could be useful for general applications that process netlists.

In the second experiment, we evaluated the effectiveness of the combination of BDD sweeping, structural transformations and SAT in the context of equivalence checking. For the given set of designs we ran a full equivalence check using the presented approach and compared it with the original BDD sweeping algorithm presented in [3]. The results are given in Figure 10. 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. The performance for a particularly complex circuit is marked in both diagrams. This design contains 55,096 gates, 302 primary inputs, 2,876 outputs, and 2,200 latches. The verification run included 5,076 comparisons and 231,232 consistency checks and could be accomplished in 246 seconds versus 8.3 hours using 82 MBytes versus 357 MBytes for the new and old method, respectively.

8 Conclusions

In this paper we presented a combination of techniques for Boolean reasoning using BDD sweeping, structural transformations, and a SAT solver in a tight integration. All three methods work on a shared AND/INVERTER graph representation of the problem and are invoked in an intertwined manner. This unique

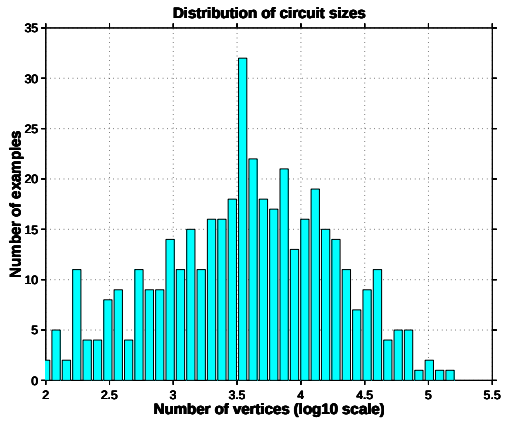

Figure 8: Distribution of circuit sizes for the experiments.

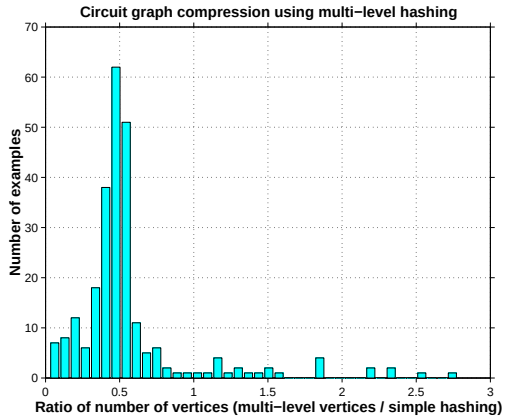

Figure 9: Comparison of graph reduction of simple hashing versus multi-input hashing.

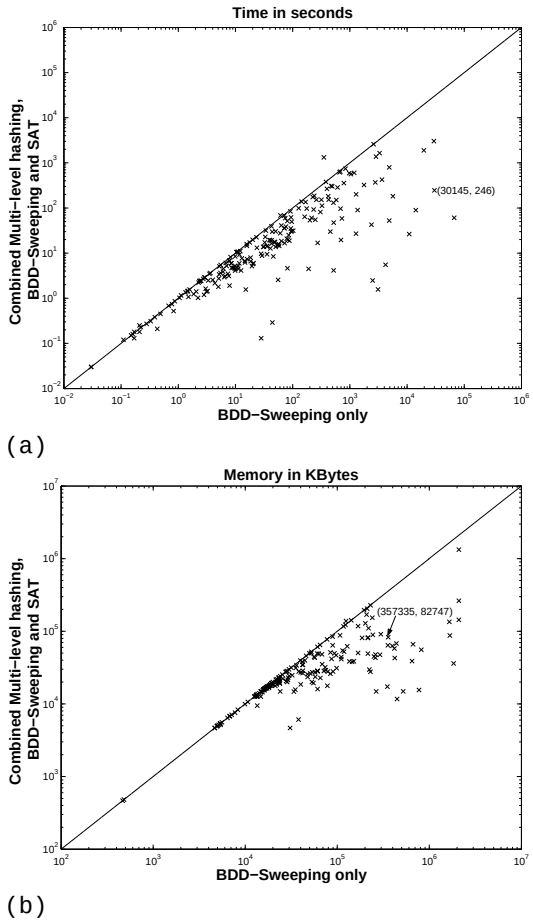

Figure 10: Comparison of the original BDD sweeping algorithm with the new algorithm for equivalence checking: (a) runtime comparison, (b) memory comparison.

integration results in a robust summation of their natively orthogonal strength. The described approach expands on previous work by presenting new details, in particular in the area of structural transformations, the implementation of the SAT solver, and the overall integration of the components. The evaluation of the approach using an extensive set of industrial problems demonstrated its effectiveness for a wide range of applications.

References

[1] M. K. Ganai and A. Kuehlmann, "On-the-fly compression of logical circuits," Tech. Rep. Computer Science, RC 21704, IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, March 2000.

[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] 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.

[4] V. Paruthi and A. Kuehlmann, "Equivalance checking combining a structural SAT-solver, BDDs, and simulation," in *Proceedings of the IEEE International Conference on Computer Design*, (Austin, TX), pp. 459–464, IEEE, September 2000.

[5] 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.

[6] J. Marques-Silva and K. Sakallah, "GRASP: a search algorithm for propositional satisfiability," *IEEE Transactions on Computers*, vol. 48, no. 5, pp. 506–521, 1999.

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

[8] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi, "Extended BDD's: Trading off canonicity for structure in verification algorithms," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 464–467, IEEE, November 1991.

[9] 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.

[10] H. Hulgaard, P. Williams, and H. Andersen, "Equivalence checking of combinational circuits using boolean expression," *IEEE Transactions on Computer-Aided Design*, vol. 18, July 1999.

[11] 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.

[12] A. Gupta and P. Ashar, "Integrating a boolean satisfiability checker and BDDs for combinational equivalence checking," in *Proc. of the Int. Conference on VLSI Design*, 1998.

[13] J. R. Burch and V. Singhal, "Tight integration of combinational verification methods," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (San Jose, CA), pp. 570–576, ACM/IEEE, November 1998.

[14] 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.