On-the-Fly Compression of Logical Circuits

Malay K. Ganai*
Dept. of ECE
The University of Texas at Austin
Austin, TX

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

Abstract

The application of CAD algorithms in logical verification and synthesis requires an efficient representation of combinational circuits in terms of a network of Boolean primitives. Typical input descriptions of such circuits contain a large amount of functional redundancy. Previous approaches simplify the network representation by graph hashing, allowing the identification and elimination of structurally isomorphic subcircuits. We propose a compression technique which extends this method by also identifying a bounded set of functionally equivalent subcircuits that are not necessarily structurally isomorphic. The described technique is based on a constant-time table lookup scheme which eliminates a larger amount of network redundancy without any appreciable computational overhead. This directly results in a reduction of the memory requirements for storing the network and an increased efficiency of the algorithms working on it.

1 Introduction

The application of CAD algorithms in logical verification and synthesis requires an efficient representation of combinational circuit in terms of a network of Boolean primitives. Typical input descriptions of such circuits contain a large amount of functional redundancy in terms of multiple internal nets that implement the same Boolean function. This redundancy originates mostly from circuit specifications using common hardware description languages and the procedures by which those languages are parsed and further processed. An extreme case of a functionally redundant circuit is the Miter [1] structure which is built for the purpose of combinational equivalence checking. The task of proving functional equivalence of two nets in this setting is identical to eliminating the redundancy from the Miter, if successful, resulting in a single constant output net. In general, identifying functionally redundant nets is computationally hard. As a special case, structurally equivalent nets in combinational loop-free circuits can be detected in linear time. As an example, in [2], a method is described that, among other techniques, employs structural hashing to identify and merge isomorphic subcircuits.

In this paper, we present a generalized hashing scheme which identifies functionally identical subcircuits of bounded size independent of their actual structural implementation. The proposed method hashes each subcircuit onto a unique functional signature which is then used to implement the function in a structural canonical manner. As a result, two structurally different but functionally identical subcircuits get mapped onto the same canonical implementation and hence are automatically merged during their construction.

2 Previous Work and Motivation

In [2], a two-step approach for identifying functionally identical nets is presented. First, during the actual network construction structural hashing is applied to consolidate isomorphic subcircuits. Then, a BDD sweeping technique is used to further identify and merge functionally equivalent nets. Although BDD sweeping is quite general and can identify all functional identical nets as long as the BDD representation does not grow too large, this method has a considerable overhead caused by the BDD manipulation. The functional hashing technique presented in this paper is not as general as the sweeping method. However, it is based on a constant-time table lookup scheme which, if applicable, completely eludes the significant overhead of the BDD processing.

Further, BDD sweeping requires the identification of intermediate nets as cutpoints from which the sweeping process restarts. Finding meaningful cutpoints is a delicate task for which no general and robust heuristic exists [3]. The presented functional hashing technique completely avoids cutpoint selection by working on all subcircuits of a fixed depth. Moreover, as exploited in [2], an

*The author carried out this work in summer 1999 as an intern at the IBM Austin Research Lab.

```
/* Function create_canonical_and2 takes two
   operand edges p1 and p2, and returns an edge
   representing the Boolean AND of p1 and p2 */

EDGE
Algorithm create_canonical_and2 (EDGE p1, EDGE p2) {
  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;

  /* p1 must be the smaller vertex for some ranking;
     rank(p) returns rank of from-vertex of p;
     non_inv(p) returns non-inverted edge */
  if (rank(non_inv(p1)) > rank(non_inv(p2))) {
     swap (p1, p2);
  }

  /* Hash lookup for vertex with children p1 and p2 */
  p = hash_lookup (p1, p2);
  if (p != NULL) {
     return p;
  }
  /* Allocate new vertex if lookup failed */
  return create_and_vertex (p1, p2);
}
```

Figure 1: Pseudo-code for structural hashing.

upfront structural compression of the network increases the average fanout of intermediate nets and as a result enhances the effectiveness of all selected cutpoints. In essence, the presented functional hashing method does not replace the cutpoint-based BDD sweeping technique, it rather increases its robustness through the additional static compression.

In the following we first revisit the structural hashing scheme presented in [2] and then describe its generalization to include functional hashing. Without loss of generality, we will focus our description on the previously proposed uniform AND/INVERTER network representation. During network construction, all Boolean operations are converted into a graph using two-input AND vertices and edges with an optional INVERTER attribute. Similar to efficient BDD implementations [4], 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 applied during graph construction to identify isomorphic subnetworks and to immediately map them onto the same subgraph. Figure 1 shows the self-explanatory pseudo-code for the structural hashing algorithm.

The construction of the circuit graph for a simple example is illustrated in Figure 2. Part (a) represents a Miter circuit built for proving equivalence of nets x and y which are functionally identical but have different structural implementations. The applied labeling identifies functional equivalent nets by using identical numbers with one or more apostrophes. Figure 2(b) shows the result of the graph construction using the algorithm *create_canonical_and2* from Figure 1. The vertices of the graphs represent AND functions and the filled dots at the edges symbolize the INVERTER attribute. Note that the functions $\overline{a \wedge b}$ (net 1) of the upper circuit and $\overline{a} \vee \overline{b}$ (net 1′) of the lower circuit are identified as structurally equivalent

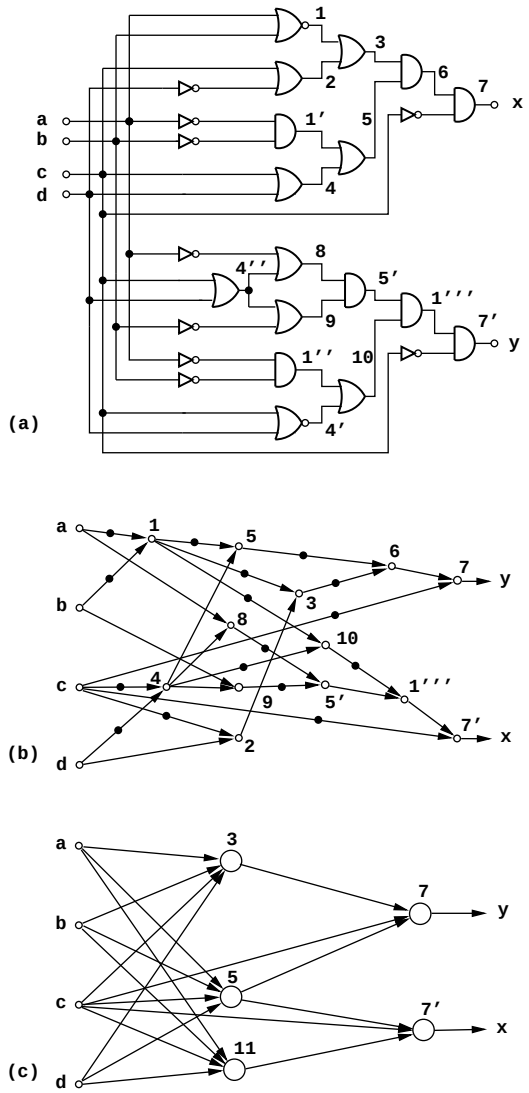

Figure 2: Example for circuit graph construction: (a) a functionally redundant Miter structure generated to check functional equivalence of outputs x and y, (b) circuit graph using structural hashing constructed according to [2] with two-input vertices, (c) circuit graph with four-input vertices.

(modulo inversion) and mapped onto the same vertex in the graph model. No other parts of the two circuits could be merged due to the limited scope of this technique.

A natural way to increase the scope of the structural hashing method is to divert from the two-input graph model and use vertices with higher fanin degree. The set of possible functions of a vertex with more than two inputs cannot be encoded with edge attributes. Instead the vertex function is represented by an index which is hashed in conjunction with the input identifiers to find structurally identical circuit parts. Since the number of possible vertex functions grows exponentially, this method is only practical for vertices with up to four inputs. For the given circuit example, Figure 2(c) shows the graph model based on vertices with a maximum fanin degree of four. Note that this method can identify the equivalence of the net pair $(5, 5')$ but still fails for pair $(7, 7')$.

In this paper we describe a method that combines the fine granularity of the original two-input graph model with the larger scope of multi-input functional hashing. We demonstrate that this approach can identify all functional equivalent vertex pairs for the given example.

3 Multi-Input Functional Hashing

The presented approach is a generalization of the structural hashing method described in [2]. Instead of considering only the two immediate inputs of a vertex for functional hashing (i.e. one level backwards), the structural analysis is extended to two levels preceding a vertex. Thus, the resulting granularity to identify functionally identical vertices is comparable to the granularity of the hashing technique based on four-input vertices. However, by applying this method on all intermediate vertices in an overlapping manner, this approach can also take advantage of structural similarities that otherwise remain internal to four-input vertices.

Figure 3 outlines the overall flow of the multi-input hashing scheme. In general, for each vertex the algorithm will produce a canonical substructure using the grandchildren as its inputs by applying the function *create_canonical_and4*. However, during network 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 directly the original function *create_canonical_and2* which is based on two-input hashing. If only one of the children is a primary input, a canonical three-input substructure is created by calling the function *create_canonical_and3*. Note that both routines, *create_canonical_and3* and *create_canonical_and4* call *create_and* recursively. To avoid the repeated construction of identical subnetworks, a hash-lookup checks whether the vertex to be created already exists in which case the recursion will be broken. In the following subsections 3.1 and 3.2, we discuss the algorithms to build canonical substructure for 3 and 4 inputs, respectively.

```
/* Function create_and takes two operand
   edges p1 and p2, and returns an edge
   representing the Boolean AND of p1 and p2. */

EDGE
Algorithm create_and (EDGE p1, EDGE p2) {
  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;

  /* check if vertex exists to avoid
     repeated recursion on same vertices */
  if (rank(non_inv(p1)) > rank(non_inv(p2))) {
     swap (p1, p2);
  }
  p = hash_lookup (p1, p2);
  if (p != NULL) {
     return p;
  }
  if (is_var(p1) && is_var(p2)) { /* p1, p2 are variables */
     return create_and_vertex (p1, p2);
  }
  if (is_var(p1)) { /* only p1 is variable */
     return create_canonical_and3 (p1, p2);
  }
  if (is_var(p2)) { /* only p2 is variable */
     return create_canonical_and3 (p2, p1);
  }
  /* neither p1, p2 is variable */
  return create_canonical_and4 (p1, p2);
}
```

Figure 3: Pseudo-code for multi-input structural hashing.

3.1 Three-Input Case

The construction of non-redundant three-input substructures is done in two steps. First, the subnetwork is mapped onto a unique signature using a simple lookup scheme. The signature is then used for an table lookup which returns an index pointing to a non-redundant implementation of the function. The idea is that the signatures of all substructures with equivalent functions will produce identical indices and therefore result in the same structural implementation.

Figure 4 gives the pseudo code for the signature computation for the three-input case. The calling routine has to ensure that the second argument passed to that pro-

```
/* Function 3_input_signature computes the signature
   for a three input substructure.
   R must have two children, L must be leaf.  */

int
Algorithm 3_input_signature (EDGE L, EDGE R) {
   /* Get children of R */
   RL  = left_child(R);
   RR  = right_child(R);

   /* Set i-th bit sig[i] if corresponding
      predicate is true */
   sig[0] = (non_inv(L) == non_inv(RL));
   sig[1] = (non_inv(L) == non_inv(RR));
   sig[2] = (rank(non_inv(L)) > rank(non_inv(RL)));
   sig[3] = (rank(non_inv(L)) > rank(non_inv(RR)));
   sig[4] = is_inv(RL);
   sig[5] = is_inv(RR);
   sig[6] = is_inv(L);
   sig[7] = is_inv(R);
   return sig;
}
```

Figure 4: Pseudo-code for *3_input_signature*.

```
EDGE
Algorithm create_canonical_and3 (EDGE L, EDGE R) {
   /* Get the non-inverted edge for L
      and the children of R */
   l  = non_inv(L);
   rl = non_inv(left_child(R));
   rr = non_inv(right_child(R));

   /* Generate structure specific signature */
   signature = 3_input_signature (L, R);

   /* Map signature to canonical structure
      by table lookup */
   index     = LOOKUP3[signature];
   switch(index) {
      ...
      case 3:
         p = create_canonical_and2 (l, rl);
         return p;
       /* Illustrating example in Figure 6 */
      case 4:
         p = l;
         return p;
      ...
      case 33:
         p = not(l);
         return p;
      ...
   }
}
```

Figure 5: Pseudo-code for *create_canonical_and3*.

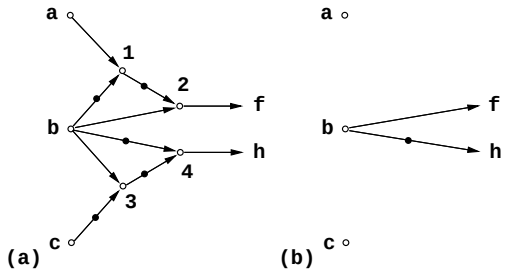

Figure 6: A 3-input example: (a) pre-lookup structure $sig_f = \langle 10100110 \rangle$ and $sig_h = \langle 11100001 \rangle$, (b) post-lookup structure $f = b$ and $h = \overline{b}$.

cedure points to a non-leaf vertex. The signature is composed of three components which identify the substructure and input order in an unambiguous manner. The first two signature bits indicate whether any of the children RL and RR are identical to L. The next two bits encode the rank order of all three inputs. This information is essential to make the merging of the substructure independent of the actual input sequence. Note that the algorithm *create_canonical_and2* (see Figure 1) ensures by construction that the childen RL and RR are different and $rank(non\_inv(RL)) < rank(non\_inv(RR))$. The last four signature bits reflect the setting of the inverter attributes within the substructure.

The actual construction of the three input substructure is outlined in the algorithm *create_canonical_and3* in Figure 5. Figure 6 provides a small example to illustrate the described mechanism. Both structures $f = b \wedge \overline{(a \wedge \overline{b})}$ and $h = \overline{b} \wedge \overline{(\overline{c} \wedge b)}$ are first characterized by the signatures $sig_f = \langle 10100110 \rangle$ and $sig_h = \langle 11100001 \rangle$ (assuming $rank(a) < rank(b) < rank(c)$) which then get mapped to the indices $index_f = 4$ and $index_h = 33$, respectively. As outlined in the pseudo-code, both structures are then reimplemented in the shown manner.

3.2 Four-Input Case

Similar to the three-input case, the construction of four-input substructures is based on a signature computation and table lookup to point to a non-redundant construction rule for the corresponding function. Figures 7 and 8 outline the pseudo-code for the algorithms. As an example, Figure 9 demonstrates how two different implementations of the XOR and XNOR function get mapped onto the same structure. Both structures $g = \overline{(a \wedge b)} \wedge \overline{(\overline{a} \wedge \overline{b})}$ and $k = \overline{(\overline{a} \wedge b)} \wedge \overline{(a \wedge \overline{b})}$ are first characterized by the signatures $sig_g = \langle 11110010000011 \rangle$ and $sig_k = \langle 11100110000011 \rangle$ (assuming $rank(a) < rank(b)$) which then get mapped to the indices $index_g = 143$ and

```
/* Function 4_input_signature computes the signature
   for a four input substructure.
   L and R must each have two children.  */

int
Algorithm 4_input_signature (EDGE L, EDGE R) {
   /* Get children of L and R */
   LL  = left_child(L);
   LR  = right_child(L);
   RL  = left_child(R);
   RR  = right_child(R);

   /* Set i-th bit sig[i] if corresponding
      predicate is true */
   sig[0]  = (non_inv(LL) == non_inv(RL));
   sig[1]  = (non_inv(LR) == non_inv(RR));
   sig[2]  = (non_inv(LL) == non_inv(RR));
   sig[3]  = (non_inv(LR) == non_inv(RL));
   sig[4]  = (rank(non_inv(LL))  > rank(non_inv(RL)));
   sig[5]  = (rank(non_inv(LR))  > rank(non_inv(RR)));
   sig[6]  = (rank(non_inv(LL))  > rank(non_inv(RR)));
   sig[7]  = (rank(non_inv(LR))  > rank(non_inv(RL)));
   sig[8]  = is_inv(LL);
   sig[9]  = is_inv(LR);
   sig[10] = is_inv(RL);
   sig[11] = is_inv(RR);
   sig[12] = is_inv(L);
   sig[13] = is_inv(R);
   return sig;
}
```

Figure 7: Pseudo-code for *4_input_signature*.

```
EDGE
Algorithm create_canonical_and4 (EDGE L, EDGE R) {
   /* Get the non-inverted children for L and R */
   ll  = non_inv(left_child(L));
   lr  = non_inv(right_child(L));
   rl  = non_inv(left_child(R));
   rr  = non_inv(right_child(R));

   /* Generate structure specific signature */
   signature = 4_input_signature(L, R);

   /* Map signature to canonical structure
      by table lookup */
   index     = LOOKUP4[signature];
   switch(index) {
      ...
       /* Illustrating example in Figure 9 */
      case 143:
         x = not(create_and(ll, lr));
         y = not(create_and(not(ll), not(lr)));
         p = create_canonical_and2(not(x),not(y));
         return p;
      ...
      case 165:
         x = not(create_and(ll, lr));
         y = not(create_and(not(ll), not(lr)));
         p = not(create_canonical_and2(not(x),not(y)));
         return p;
      ...
   }
}
```

Figure 8: Pseudo-code for *create_canonical_and4*.

$index_k = 165$, respectively. As outlined in the pseudo-code, both structures are then reimplemented in the shown manner.

3.3 Example

In this section we demonstrate that the multi-input hashing scheme can find the functional identity of the two output nets of the example given in Figure 2. Figure 10 illustrates step by step how the network representation is built. As shown in part (a), the creation of the first set of vertices 1 - 5 is identical to Figure 2(b). Figure 10(b) illustrates that the construction of nets 6 and 7 yields in an optimized structure for both vertices, removing the existing redundancy of the original description. The next part shows the creation of the vertices 8 - 10, again without any achievable reduction by the multi-level hashing scheme. In contrast to the 2-input method, the presented multi-input scheme can then identify the equivalence of vertices $5'$ and 5. This successively results in the merging of vertices $1'''$ and $7'$ with 1 and 7, respectively, which proves the equivalence of the two outputs.

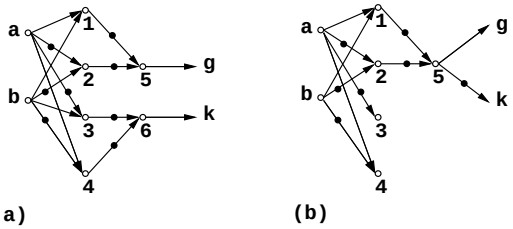

Figure 9: A 4-input example: (a) pre-lookup structure $sig_g = \langle 11110010000011 \rangle$ and $sig_k = \langle 11100110000011 \rangle$, (b) post-lookup structure $k = \overline{g}$.

4 Experiments

We implemented the presented multi-input hashing algorithm in the equivalence checking tool Verity [5]. In order to evaluate the effectiveness of the presented multi-input hashing method, we carried out two experiments using 220 industrial circuits, ranging in size from a few 100 to $100K$ gates. The actual distribution of the circuit sizes is shown in Figure 11. For all circuits two representations are given, one derived from the RTL specification and the other ex-

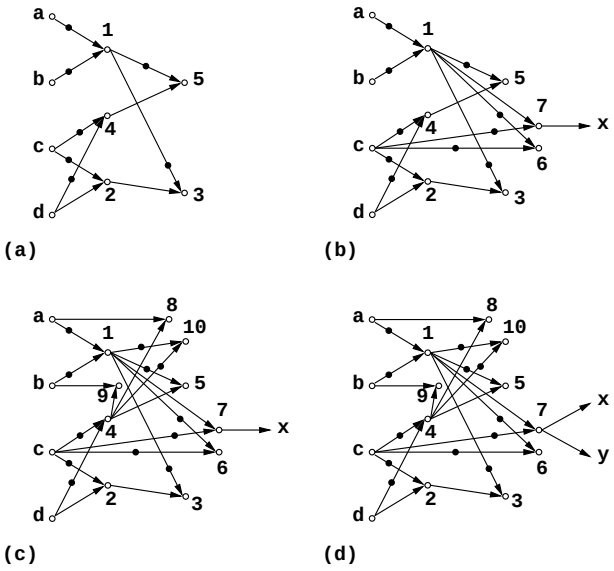

Figure 10: Example of Figure 2 for circuit graph construction using multi-input hashing: (a) vertices 1 - 5 are identical to Figure 2(b), (b) vertices 6 and 7 yield more compact structures, (c) vertices 8 - 10 yield no compaction, (d) vertices $5'$, $1'''$, and $7'$ are merged with vertices 5, 1, and 7, respectively.

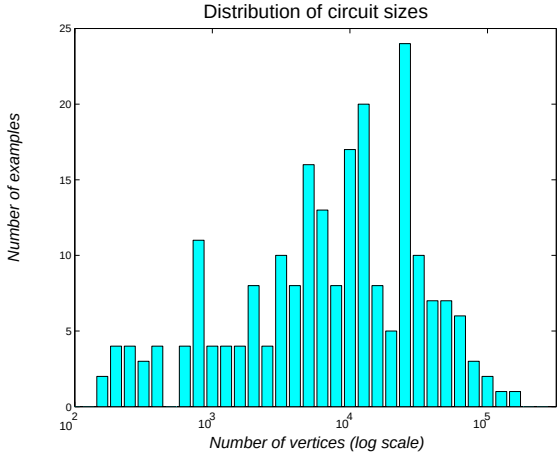

Figure 11: Distribution of circuit sizes.

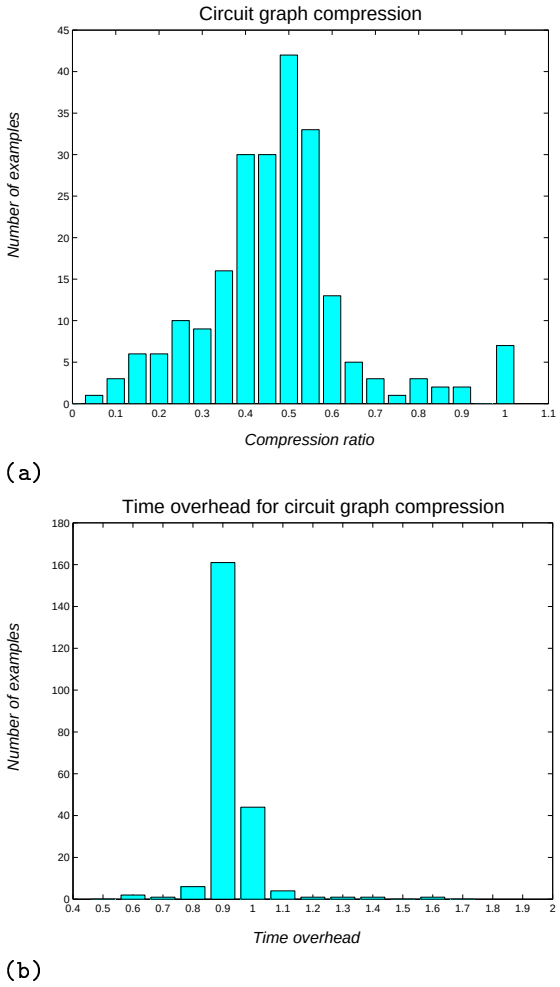

Figure 12: Circuit graph compression: (a) compression ratio (*ratio of the size of circuit built by multi-input hashing to that by 2-input hashing*), (b) time overhead (*ratio of the time taken for circuit building by multi-input hashing to that by 2-input hashing*).

tracted from the independently designed transistor-level implementation.

In the first experiment we constructed the circuit graph for the RTL specification only and compared the size of using the plain two-input method with the size generated by the newly presented multi-input method. The histogram for the size reduction of the circuit graphs is plotted in Figure 12(a). Part (b) displays the computing overhead needed to achieve this reduction. As shown, on average the given sample of circuit representations can be reduced by 50% without any overhead of computing resources.

In the second experiment we build the actual Miter circuit for performing the functional equivalence check between the RTL and transistor-level representation using

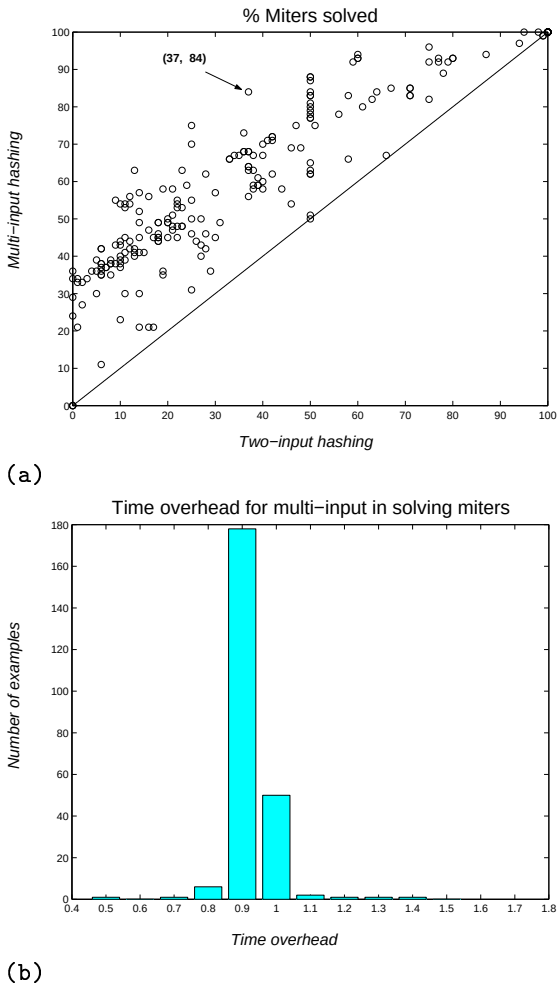

Figure 13: Fraction of miters solved: (a) comparision between multi-input and 2-input hashing, (b) time overhead (*ratio of the time taken by multi-input hashing to that by 2-input hashing*).

the two schemes. No additional equivalence checking algorithm (e.g. BDD sweeping [2]) has been applied, therefore only a subset of the outputs can be fully verified with the two structural hashing methods. Figure 13(a) shows for each circuit what fraction of the outputs can be solved with the two-input method and the multi-input method on the X-axis and Y-axis, respectively. For example, for the circuit marked with an arrow, the two-level hashing scheme could solve 37% of the outputs whereas the multi-input hashing scheme was able to solve 84% of them. It can be seen that all circles are in the upper triangle of the diagram, indicating that the multi-input method could always solve more problems without engaging more powerful verification algorithms. As shown in Figure 13(b), there is no corresponding CPU overhead.

5 Conclusion

This paper descibes an efficient algorithm to simplify logical network representations during construction. The method is based on a table lookup hashing scheme that maps each subcircuit onto a unique functional signature which is then used to implement the function in a structural canonical manner. We demonstrate that for industrial circuits we can compress the circuit representation by up to 90% with a minimum computational overhead. This compression directly results in a reduction of the memory requirements to store the network and an increased efficiency of the algorithms working on it. In particular, the application of the proposed method in functional equivalence checking results in a significantly larger number of outputs that can be verified without engaging more complex algorithms.

References

[1] D. Brand, "Verification of large synthesized designs," in *Digest of Technical Papers of the IEEE Internat. Conference on Computer-Aided Design*, pp. 534–537, IEEE, November 1993.

[2] A. Kuehlmann and F. Krohm, "Equivalence Checking Using Cuts and Heaps," in *Proc. of the Design Automation Conf.*, pp. 263–268, June 1997.

[3] 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.

[4] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient Implementation of a BDD Package," in *Proc. of the Design Automation Conf.*, pp. 40–45, June 1990.

[5] 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.