

On-the-Fly Compression of Logical Circuits

Malay K. Ganai*

Andreas Kuehlmann

Dept. of ECE

IBM T. J. Watson Research Center

The University of Texas at Austin

Yorktown Heights, NY

Austin, TX

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.

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

In this project, we develop 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.

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.

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.

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 1 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 subnet-

```

/* Function create_and takes two operand
edges  $p_1$  and  $p_2$ , and returns an edge
representing the Boolean AND of  $p_1$  and  $p_2$ . */

EDGE
Algorithm create_and (EDGE  $p_1$ , EDGE  $p_2$ ) {
  if ( $p_1 == \text{const}_0$ ) return  $\text{const}_0$ ;
  if ( $p_2 == \text{const}_0$ ) return  $\text{const}_0$ ;
  if ( $p_1 == \text{const}_1$ ) return  $p_2$ ;
  if ( $p_2 == \text{const}_1$ ) return  $p_1$ ;
  if ( $p_1 == p_2$ ) return  $p_1$ ;
  if ( $p_1 == \overline{p_2}$ ) return  $\text{const}_0$ ;

  /* check if vertex exists to avoid
  repeated recursion on same vertices */
  if ( $\text{rank}(\text{non\_inv}(p_1)) > \text{rank}(\text{non\_inv}(p_2))$ ) {
    swap ( $p_1, p_2$ );
  }
   $p = \text{hash\_lookup}(p_1, p_2)$ ;
  if ( $p \neq \text{NULL}$ ) {
    return  $p$ ;
  }
  if ( $\text{is\_var}(p_1) \ \&\& \ \text{is\_var}(p_2)$ ) { /*  $p_1, p_2$  are variables */
    return create_and_vertex ( $p_1, p_2$ );
  }
  if ( $\text{is\_var}(p_1)$ ) { /* only  $p_1$  is variable */
    return create_canonical_and3 ( $p_1, p_2$ );
  }
  if ( $\text{is\_var}(p_2)$ ) { /* only  $p_2$  is variable */
    return create_canonical_and3 ( $p_2, p_1$ );
  }
  /* neither  $p_1, p_2$  is variable */
  return create_canonical_and4 ( $p_1, p_2$ );
}

```

Figure 1: Pseudo-code for multi-input structural hashing.

works, a hash-lookup checks whether the vertex to be created already exists in which case the recursion will be broken.

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.

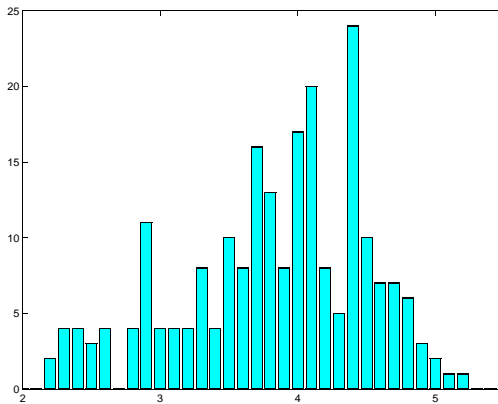


Figure 2: Distribution of circuit sizes.

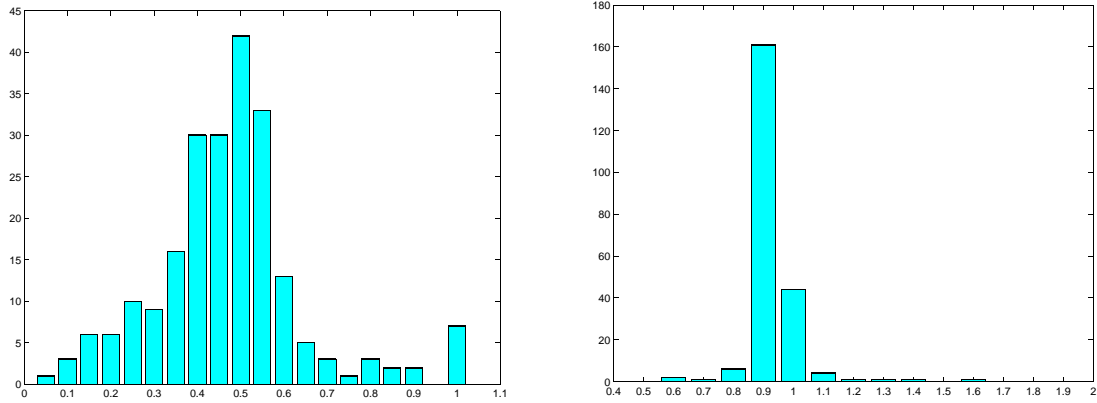
Similar to the three-input case, the construction of four-input substructures is based on a signature computation and table lookup to point to a non-redundant construction rule for the corresponding function.

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 2. For all circuits two representations are given, one derived from the RTL specification and the other extracted 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 3(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.

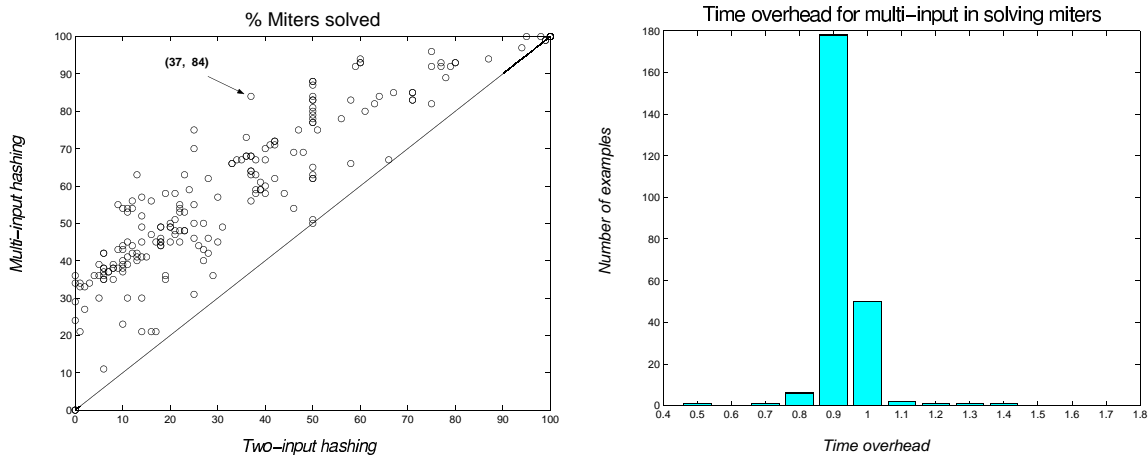
In the second experiment we build the actual Miter circuit for performing the functional equivalence



(a)

(b)

Figure 3: 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).



(a)

(b)

Figure 4: Fraction of miters solved: (a) comparison 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).

check between the RTL and transistor-level representation using 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 4(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 4(b), there is no corresponding CPU overhead.

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.