International Symposium on Circuits and Systems (ISCAS'99)
pp. IV:436-439, Orlando, May 1999

CREATING HARD PROBLEM INSTANCES IN LOGIC SYNTHESIS USING EXACT MINIMIZATION

Wolfgang Günther *Rolf Drechsler*

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany
{guenther,drechsle}@informatik.uni-freiburg.de

ABSTRACT

To evaluate synthesis algorithms, usually benchmark circuits are used. Since for these circuits no exact synthesis results are known, we propose the use of exact minimization to generate hard problem instances. By this we evaluate a standard synthesis tool on different classes of circuits.

1. INTRODUCTION

Logic synthesis is one of the most important steps in the design process of ICs. Many heuristic synthesis approaches for circuit design have been proposed in the past.

While for two-level realizations powerful tools for exact minimization have been developed [4], exact multi-level optimization has been considered only in the 1960s [8, 5]. But these approaches were of limited practical interest since they were only applicable to very small functions. However, meanwhile the computational power of modern workstations and the knowledge of efficient data structures has extremely increased. Therefore exact circuit synthesis becomes an interesting topic again. To demonstrate this we consider a full adder with 3 inputs and two outputs. Even for this small function SIS [9] produces a result that is more than 50% worse than the optimal solution, i.e. SIS needs 15 NAND gates, while there exists a circuit with only 9 (see Figure 1), using the commands

```
collapse
source script.rugged
map -m 0
```

to map the circuit with SIS. For generic 2-input gates, a similar observation can be made. In this case, SIS needs 9 gates (using `xl_split -n 2`), whereas the optimum solution has only 5 gates. One reason for it is that SIS first uses a technology independent factorization to minimize a literal cost function and then maps to the target library.

To evaluate synthesis algorithms usually benchmark circuits have been used, like LGSynth91 [10], ISCAS85 [3], and ISCAS89 [1]. However, recent studies have shown that the 'standard' benchmark paradigm used today is not sufficient to evaluate state-of-the-art tools in VLSI CAD [7]. For this, there is a large need for *Design of Experiments* (DoE) methods.

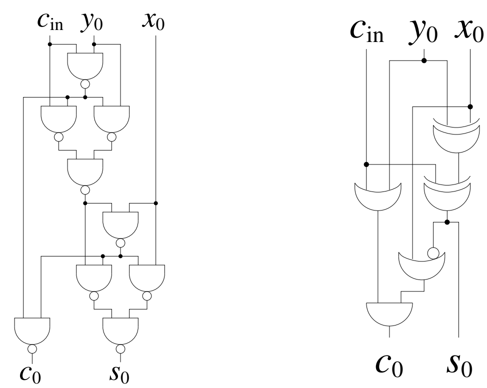


Figure 1: Optimal NAND and generic 2-input realization of a full adder

In this paper we focus on the domain of logic synthesis. First we describe an exact algorithm to find the optimal netlist for a given Boolean function, i.e. the minimal netlist among all netlist representing the function. Based on this algorithm we create problem instances that are used as input to a standard tool, i.e. SIS [9]. Using this approach it is possible to compare the exact results to SIS. It turns out that even for small functions, i.e. 4-input functions, SIS is often up to a factor of two away from the optimal results, for larger functions even more.

2. PRELIMINARIES

In general, a *Combinational Logic Circuit* (CLC) is defined over a fixed library and is modeled as a directed acyclic graph $C = (V, E)$ with some additional properties. Often a simple library consisting of primary input (PI) and output ports (PO) and the 2-input, 1-output *NAND* gate is used, denoted by Lib_{NAND}= {*PI, PO, NAND*}. Alternatively, we can have Lib_{2input}, consisting of all possible 2-input, 1-output gates, i.e. Lib_{2input}= {*PI, PO, NAND, NOR, AND, OR, EXOR, EXNOR, ANDI, ORI*}, or any of its subsets.

Definition 1 *A minimal CLC over library* Lib *for a given Boolean function f is a netlist with the least number of gates from library* Lib *among all netlists representing function f.*

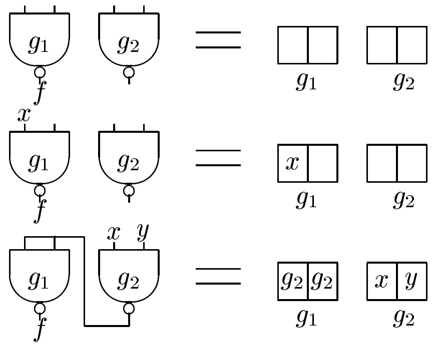


Figure 2: Initial situation, situation after first assignment, and synthesized network for $f(x, y) = x \cdot y$.

The minimal netlist is not unique in general, i.e. there may exist different netlists with the same number of gates. We next introduce an exact algorithm which returns the first instance of the minimal netlists.

3. EXACT MINIMIZATION ALGORITHM

In this section an exact algorithm is described (see also [6]). The basic idea of graph enumeration by backtracking is introduced in Section 3.1. Then the pruning technique is outlined in Section 3.2

For simplicity of the presentation throughout this section library *Lib*$_{NAND}$ is used, i.e. the exact algorithm determines the optimal *NAND* realization of a Boolean function. The extension to more complex libraries is straightforward.

3.1. Graph Enumeration by Backtracking

In each step, a realization for a given function is searched with a maximum number of gates. Starting with 1, this limit is incremented until a solution has been found. To find a realization with a given limit on the number of gates, iteratively all possible netlists are generated.

The enumeration of all possible netlists with a given number of gates is done as follows (see Figure 2). In a first step, the gates are generated. The first gates are output gates and finally have to represent the given function, the other gates are internal gates. Initially, all gate inputs are unspecified. In successive steps, either primary inputs or outputs of other gates are assigned to these inputs, starting from the first input of the first gate. If during the assignments it turns out that it is not possible to complete the netlist such that it represents the correct function, a backtracking step has to be performed. Checking this criteria is the core of the algorithm (see Section 3.2).

Finding for a given netlist the subsequent one is done in two parts:

A: Gate inputs are assigned. Naturally, assigning inputs is only done for netlists which still have unassigned inputs.

```
compute_minimal_NAND_netlist(f, gatelimit)
    initialize NAND gates;
    do {
        assign_input();
        if (backtracking_necessary())
            do_backtracking();
    } until (netlist computes function f
          or the enumeration is complete);
    return netlist, if found;
```

Figure 3: Sketch of the algorithm

B: Assignments are modified. In a first step the highest gate input which can be incremented is determined. If no such input exists, all netlists with the given number of gates have been considered. Otherwise, the input value is incremented and all higher gate input assignments are deleted.

A sketch of the basic algorithm is given in Figure 3. For each change of an assignment, it has to be checked whether the resulting netlist is still acyclic. These cycles are easy to identify: a gate h is a possible input to gate g if g is not contained in the set of successors of gate h.

3.2. Pruning Techniques

In the trivial approach, backtracking would be called only in the case that all gate inputs are assigned. The structure of a partial netlist often provides additional information to decide whether a completion of the netlist can lead to a feasible solution. These criterias can be divided into the following four classes:

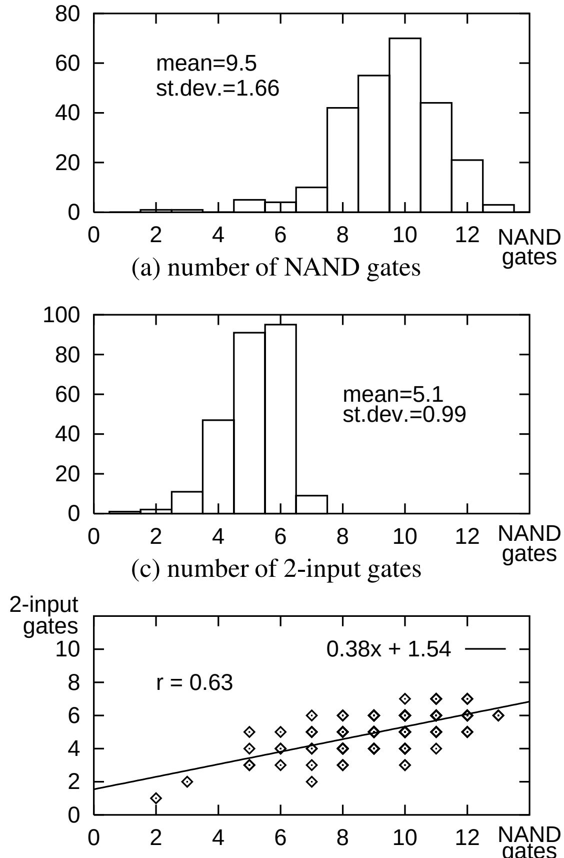
1. Topological constraints
2. Symmetry constraints
3. Functional constraints
4. Library-specific constraints

Backtracking can be called each time one of the constraints is not fulfilled. As early calls of backtracking result in the pruning of large parts of the search space, this is very desirable. Note that by using all these constraints, the minimality of the result is not lost, i.e. correctness of the algorithm is preserved. A more detailed description of these constraints can be found in [6].

4. EXPERIMENTAL RESULTS

We use the exact algorithm to study the behavior of SIS [9] using script *rugged*, mapping for area optimization. In a first part, we let the exact algorithm run on random problem instances. In a second part, results of SIS are compared on these instances. Finally, larger instances are examined.

In a first series of experiments, we study 256 randomly chosen 4-input functions. We use the exact algorithm to determine the optimal results for different libraries (see Figure 4). The distribution for library *Lib*$_{NAND}$ is given in (a). For



(a) number of NAND gates

(c) number of 2-input gates

(d) 2-input gates vs. NAND gates

Figure 4: Exact algorithm

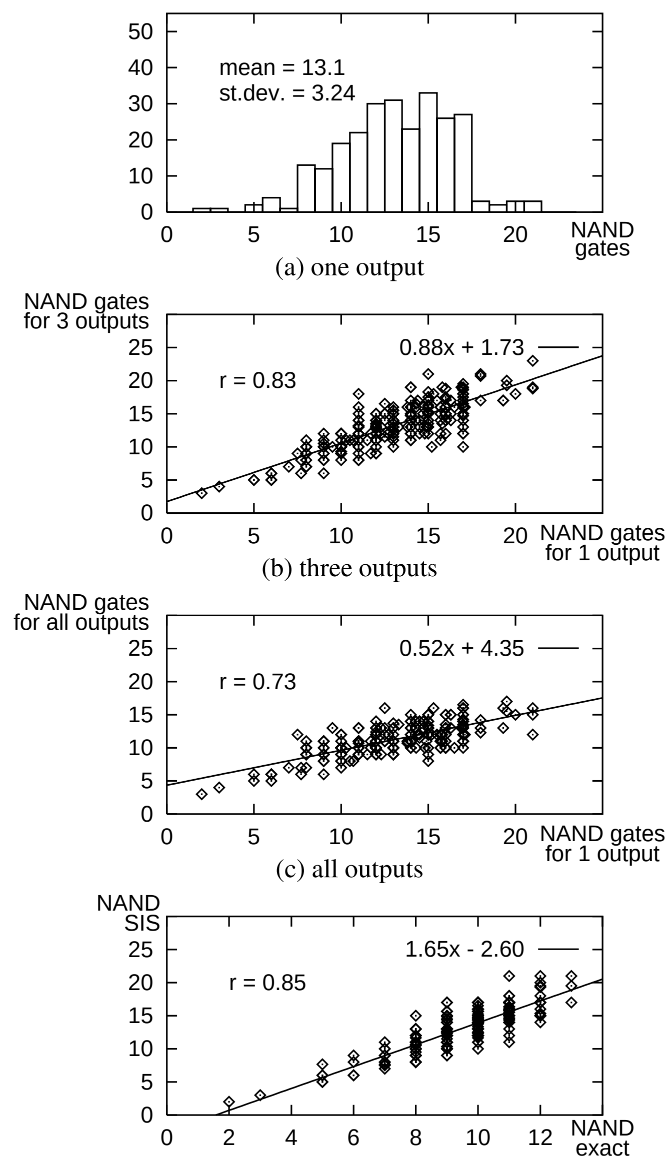
library Lib_{2input}, a similar distribution is obtained (b), and the sizes are correlated (c) with $r = 0.63$[1]. To simplify the readability, in all further experiments, library Lib_{NAND} is used.

To evaluate SIS, the problem that the results of SIS depend on the representation of the input data has to be addressed. The exact minimum does not depend on the order of the input variables chosen in the input file, as the algorithm is independent of the representation of the Boolean function. However, the results of SIS depend on that order (see also [2]). Therefore, we use equivalence classes to evaluate SIS. For each circuit, different permutations of the input variables are used as an input to SIS.

Results for SIS are given in Figure 5: For a circuit, the average of the SIS results on all $4! = 24$ orders of the input variables is computed. The distribution is similar to the exact results, but the average value is much larger.

Furthermore, we make some internal gates of an optimal netlist to additional outputs in order to find out whether SIS makes use of this "extra information". (We use the optimal netlist from the experiment above.) Results for two additional outputs (placed beneath the original output) and

[1] The correlation coefficient can be used to measure how much the sizes depend on each other. A value of 1 means strong dependence, a value of 0 means that there is no correlation.



(a) one output

(b) three outputs

(c) all outputs

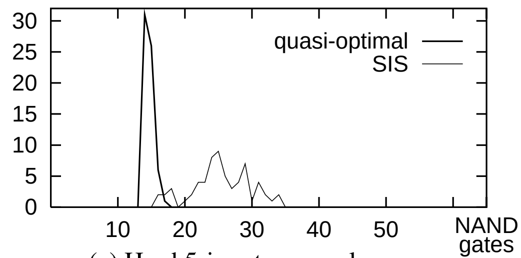
(d) SIS average (one output) vs. exact

Figure 5: SIS results using additional outputs

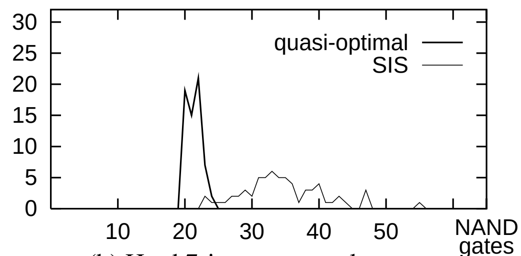
for the case that all internal gates of the optimal netlist become output are given in Figure 5 (b) and (c), respectively. Note that the optimal solution does not increase thereby. It turns out that by using two additional outputs, the quality of SIS cannot be improved on average; if all gates are declared "output", results get better, however in some cases they even get worse in spite of that "hint".

A more detailed comparison of the exact solutions to the results of SIS (average value for each equivalence class) is given in Figure 5 (d). Exact results and average SIS results are correlated with $r = 0.85$. It has been observed that for these functions, maximum and minimum results are very close together.

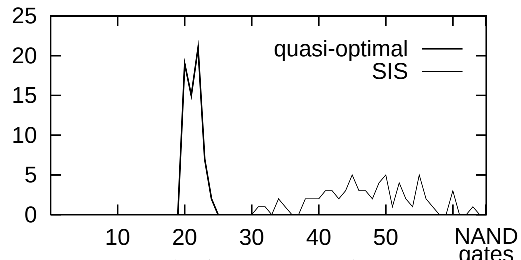
In a second series of experiments, we evaluate the quality of SIS on larger, difficult examples (Figure 6). We se-



(a) Hard 5-inputs examples



(b) Hard 7-inputs examples



(c) Hard 8-inputs examples

Figure 6: Larger examples (SIS vs. quasi-optimal results)

lect some of the 4-input function of the first experiment for which SIS produces very bad solutions. These "hard" functions are then used as basic blocks to construct 64 larger circuits by (a) adding a 2-input EXOR gate at the output of one block, also driven by an additional input (5 inputs), (b) one 4-input block driving another one (7 inputs), (c) two 4-input blocks driving one EXOR gate (8 inputs). As the exact algorithm cannot be run on these larger examples within reasonable runtime, we compute a trivial upper bound for the new function using the known sizes of the 4-input blocks and simply adding the cost for the additional EXOR gates, yielding quasi-optimal estimations for the gate count.

For larger functions it is almost impossible to enumerate all permutations of the inputs. Therefore, 128 elements of the equivalence class of input permutations are randomly chosen and average results are reported. It can be seen that for larger functions results get much worse than for the smaller ones. In some cases this increase almost reaches a factor or 3. For these circuits, also larger differences of the resulting sizes within the equivalence classes can be observed.

5. CONCLUSIONS

Based on an exact algorithm, we studied the quality of SIS, a standard synthesis tool, on hard problem instances. It turned out that for random functions using library Lib_{NAND}, results of SIS are often much worse than the exact results (more than a factor of 1.5). Using additional outputs from the exact solution does not help SIS unless most gates become outputs.

For larger functions, the differences to the almost exact results also increase. Also the differences within the equivalence classes becomes larger.

Using the exact algorithm it became for the first time possible to compare the quality of heuristic solutions to the exact result. It turned out that also for these small functions with only a few gates, i.e. less than 25, SIS is in some cases up to a factor of 3 away from the optimal results.

Acknowledgment The authors would like to thank Franc Brglez for many helpful discussions and comments.

6. REFERENCES

[1] F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *Int'l Symp. Circ. and Systems*, pages 1929–1934, 1989.

[2] F. Brglez and R. Drechsler. Design of Experiments in EDA: Context and New Data Sets for ISCAS'99. In *IEEE 1999 International Symposium on Circuits and Systems – ISCAS '99*, May 1999.

[3] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational circuits and a target translator in fortran. In *Int'l Symp. Circ. and Systems, Special Sess. on ATPG and Fault Simulation*, pages 663–698, 1985.

[4] O. Coudert. Two-level logic minimization: an overview. *Integration the VLSI Jour.*, 17(2):97–140, 1994.

[5] E. Davidson. An algorithm for NAND decomposition under network constraints. *IEEE Trans. on Comp.*, 18:1098–1109, 1969.

[6] R. Drechsler and W. Günther. Exact circuit synthesis. In *Int'l Workshop on Logic Synth.*, 1998.

[7] J.E. Harlow and F. Brglez. Design of experiments in BDD variable ordering: Lessons learned. In *Int'l Conf. on CAD*, pages 646–652, 1998.

[8] E.L. Lawler. An approach to multilevel boolean minimization. *Journal of the ACM*, 11:283–295, 1964.

[9] E. Sentovich, K. Singh, L. Lavagno, Ch. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, University of Berkeley, 1992.

[10] S. Yang. Logic synthesis and optimization benchmarks user guide. Technical Report 1/95, Microelectronic Center of North Carolina, Jan. 1991.