Bi-decomposition and tree-height reduction for timing optimization

Jordi Cortadella*
Universitat Politècnica de Catalunya, Barcelona, Spain, 08034
jordic@lsi.upc.es

Abstract

A novel approach for timing-driven logic decomposition is presented. It is based on the combination of two strategies: logic bi-decomposition of Boolean functions and tree-height reduction of Boolean expressions. This technology-independent approach allows to find tree-like expressions with smaller depths than the ones obtained by state-of-the-art techniques. Experimental results show an average delay reduction of more than 20% with regard to `speed_up` *in SIS.*

1 Introduction

Delay optimization has usually been considered as the step preceding technology mapping [13, 15]. Before that, Boolean networks are manipulated by multi-level logic synthesis techniques that aim at reducing the area of the circuit. Typically, the extraction of common sub-expressions is the basic step to reduce the complexity of a Boolean network.

When delay is the parameter under optimization, sharing logic is not always a good approach for logic decomposition. The degree of sharing may prevent a Boolean network from reducing the number of levels. To illustrate this fact let us analyze the DAGs G1 and G2 in Figure 1. Let us assume that the nodes represent arbitrary operations and that the expressions cannot be simplified. T1 and T2 represent the tree versions of G1 and G2, respectively. The number of paths of a DAG G, $\Pi(G)$, corresponds to the number of leaves of its tree version. The DAG nodes in the figure have been annotated with the number of paths, that can be simply calculated by adding the number of paths of the children. A lower bound on the depth of a DAG G is $\lceil \log_2 \Pi(G) \rceil$, assuming that it can be transformed by rules that cannot reduce the number of nodes [5].

Even though G1 and G2 have the same number of nodes, the lower bound on their depth is different due to their sharing degree. The tree T2' shows a possible restructuring of T2 that reduces the depth to three levels.

*This work has been funded by a grant from Intel Corp. and CICYT TIC2001-2476

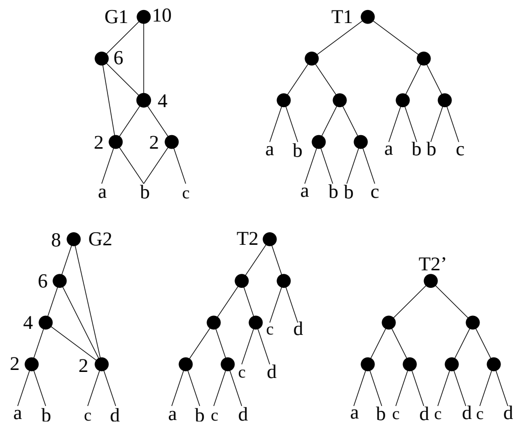

Figure 1. DAGs and tree versions of the DAGs.

This paper proposes a novel technology-independent method for timing-driven logic decomposition. The method combines two strategies:

- Tree-based bi-decomposition of Boolean functions
- Tree-height reduction of Boolean expressions

The approach aims at finding the minimum-depth tree for a Boolean function. It builds the tree from root to leaves by using bi-decomposition techniques and reduces the depth by means of rewriting rules that apply the associative, commutative and distributive laws of the Boolean algebra. Unlike the existing approaches for timing optimization, area reduction is performed as a final step without sacrificing delay.

Section 2 illustrates the impact of tree-height reduction with an example. Section 3 proposes algorithms for an efficient exploration of the transformations for tree-height reduction. Section 4 presents the main algorithm for logic decomposition. Finally, experimental results are reported in Section 5.

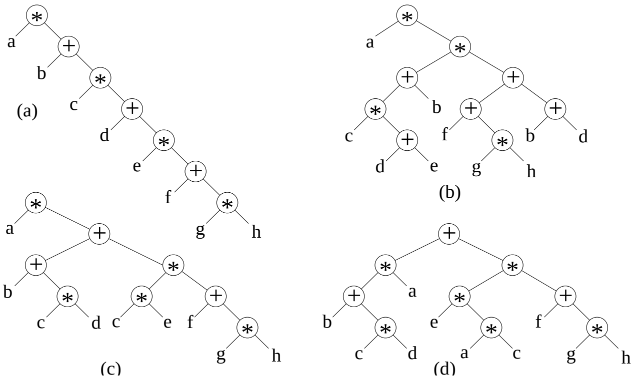

Figure 2. Equivalent factored forms.

2 Tree-height reduction: an example

Tree-height reduction [7] was originally proposed in the scope of optimizing compilers for the generation of code in multiprocessor systems. The same techniques are applicable to the optimization of combinational circuits.

Figure 2 illustrates an example. The tree in Figure 2(a) represents a factored form obtained from the Boolean expression $ab + acd + acef + acegh$. If we assume zero arrival time for all inputs and unit area ($a = 1$) and unit delay ($d = 1$) for each node, the tree is characterized by the pair ($a = 7, d = 7$).

The tree in Figure 2(b) is the one obtained by SIS after executing the `speed_up` command [13]. This tree is characterized by the pair ($a = 9, d = 5$). A more efficient implementation can be found by applying simple transformations (associative and distributive laws) to the original tree. It is shown in Figure 2(c) with ($a = 8, d = 5$). Finally, by further applying transformations, the tree in Figure 2(d) can be obtained, with ($a = 9, d = 4$). It would not be difficult to prove that the solutions shown in Figures 2(a) and 2(d) are optimal in area and delay, respectively. The tree obtained by `speed_up` is sub-optimal, since there are other equivalent trees with the same area and shorter delay (Fig. 2(d)) or the same delay and smaller area (Fig. 2(c)).

3 ACD tree-rewrite system

Boolean expressions will be represented by complement-free factored forms. Each factored form is represented by a binary tree in which the leaves are literals and the internal nodes are disjunctions ($+$) or conjunctions ($*$).

Given a binary tree T, we will refer to T as the root node or the tree itself. The following nomenclature will be used for binary trees:

```
CLUSTER (T)
   { Pre-cond: T.op ≠ ⊥. Returns a set of subtrees }
   If T.op = T.left.op then C_L := CLUSTER(T.left);
   else C_L := T.left;
   If T.op = T.right.op then C_R := CLUSTER(T.right);
   else C_R := T.right;
   return C_L ∪ C_R;
-----------------------------------------------------
MIN_DELAY_CLUSTERS (T)
   { Returns a tree with min-delay clusters }
   { Q is a list ordered by tree height }
   if T.op = ⊥ then return T;
   C := CLUSTER(T); Q := ∅;
   for each c ∈ C do
      INSERT (Q, MIN_DELAY_CLUSTERS(c));
   endfor;
   while |Q| > 1 do
      X := EXTRACT_MIN_HEIGHT(Q);
      Y := EXTRACT_MIN_HEIGHT(Q);
      INSERT(Q, (T.op  X  Y));
   endwhile;
   return EXTRACT(Q);
```

Figure 3. Algorithm for minimum-delay clusters.

T.left, T.right:		Left and right children
CHILDREN(T)	$=$	$\{T.\text{left}, T.\text{right}\}$
T.op:		Type of node: $+$, $*$ or $\perp$ (literal)
$\lvert T \rvert$:		Number of nodes of the tree
HEIGHT(T):		Height of the tree

We can also represent trees as triples:

$$T \equiv (T.\text{op} \;\; T.\text{left} \;\; T.\text{right})$$

Trees will be transformed by using the associative, commutative and distributive laws (ACD rules) of Boolean algebra.

3.1 Minimal-delay clusters (AC-rules)

This section presents algorithms for optimal tree-height reduction by applying only the associative and commutative laws (AC-rules).

Given a tree T, the topmost cluster is the set of sub-trees closer to the root that have an operation different from T. Formally, the topmost cluster of a tree is obtained by the algorithm CLUSTER in Figure 3.

Given a cluster, a minimum-delay tree can be built by combining the elements of the cluster in an appropriate way, trying the tallest sub-trees to be closer to the root. Baer and Boven [1] proposed an algorithm to build such a tree. It is an iterative algorithm that maintains all elements of the cluster in a priority queue ordered by the height of the

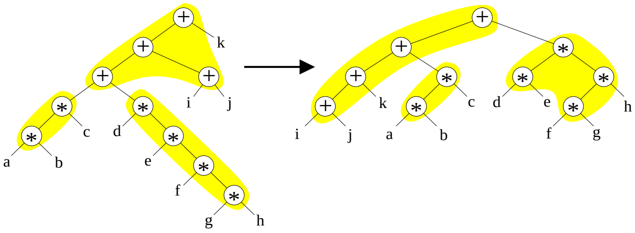

Figure 4. Application of MIN_DELAY_CLUSTERS**.**

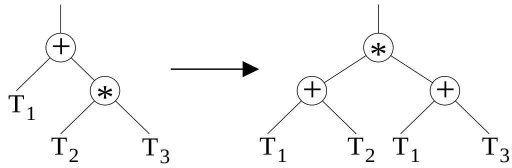

Figure 5. Distributive law.

elements. At each iteration, the two shortest elements are extracted and a new tree is built and inserted in the queue. The algorithm terminates when only one element is left in the queue, which is the returned tree. This simple algorithm was proved to be optimal in [2]. It is also the algorithm used in SIS for minimum-delay decomposition of AND and OR gates [13], although no proof of optimality was given.

The algorithm MIN_DELAY_CLUSTERS to obtain a minimum-delay tree by only using the associative and commutative laws is shown in Figure 3. The algorithm was proposed in [2] and was proved to minimize delay. It is a recursive algorithm that invokes the algorithm by Baer and Boven to build minimum delay clusters (the "`while`" loop).

Figure 4 depicts an example on the solution derived by the algorithm. The shadowed areas correspond to the clusters visited when traversing the tree. Note that the algorithm produces another tree with the same size, since the associative and commutative laws do not change the size of the tree.

3.2 Distributive law (D-rule)

The distributive law can only be applied to two nodes of the tree, n_1 and n_2, for which the following condition holds:

$$n_2 \in \text{CHILDREN}(n_1) \ \wedge \ n_1.\text{op} \neq n_2.\text{op} \ \wedge \ n_2.\text{op} \neq \perp.$$

The transformation is shown in Figure 5. By itself, the distributive law cannot provide any performance improvement, since the height of the resulting tree is not shorter than the height of the original tree. It can even produce some performance degradation if

$$\text{HEIGHT}(T_1) > \max(\text{HEIGHT}(T_2), \text{HEIGHT}(T_3))$$

However, the distributive law changes the structure of the clusters and enables the application of AC-rules that can potentially result in shorter heights. The combination of D- and AC-rules is illustrated in the example of Figure 6. After the application of a D-rule, a minimum-delay tree is obtained by running the MIN_DELAY_CLUSTERS algorithm (AC-rules).

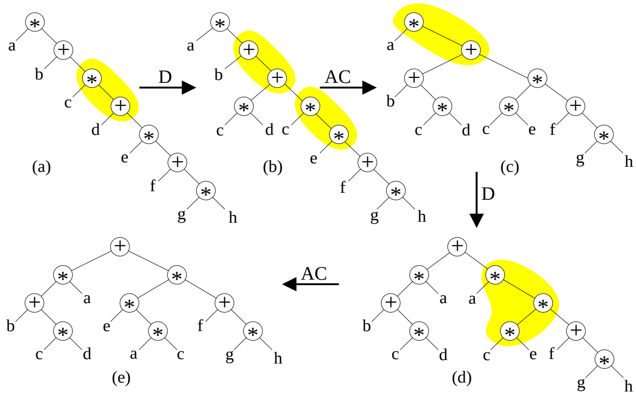

Figure 6. Application of ACD rules to optimize performance.

3.3 ACD_Speed

The solution in Figure 6(e) can only be obtained by applying the D-rule to certain nodes of the tree. One can immediately see that this solution cannot be obtained if the D-rule is applied to the root node of Figure 6(a). Therefore, the order in which rules are applied is relevant for searching optimal solutions.

Figure 7 presents an algorithm for speeding-up a tree by using ACD-rules. It assumes that F is an initial tree with minimum number of nodes, e.g. obtained by area minimization transformations on a Boolean network. The required time, in terms of number of logic levels, is also another parameter. The algorithm implements a dynamic programming approach with memoization that alternatively applies the D-rule to one of the nodes and MIN_DELAY_CLUSTERS to the tree. The set `explored` collects all the solutions generated in the algorithm.

In order to control the explosion of solutions, a frontier with limited width is selected at each layer of the search. The width of the frontier, k, is a factor that can be tuned according to the exhaustiveness of the search. The selection of "best" solutions is done by giving priority first to delay (height of the tree) and second to area (size of the tree).

```
ACD_SPEED (F, ReqTime)
  { F is a tree. Returns a tree }
  { ReqTime is the required time (in logic levels) }
  Best := MIN_DELAY_CLUSTERS (F);
  frontier := {Best}; explored := {Best};
  while depth(Best) > ReqTime  ∧  "improving" do
    new := ∅;
    for each F_r ∈ frontier do
      for each node n ∈ F_r
      such that D-rule is applicable do
        F' := APPLY_DISTRIBUTIVE (F_r, n);
        F'' := MIN_DELAY_CLUSTERS (F');
        if F'' ∉ explored then
          explored := explored ∪ {F''};
          new := new ∪ {F''};
          Best := Best_Delay_Area (Best, F'');
    frontier := Select_Best_k_Circuits (new, k);
  return Best;
```

Figure 7. Algorithm for speed-up with ACD rules.

The algorithm stops when a solution with the required time is found or when no improvement has been obtained during few iterations. The "improvement" criterion is another tunning parameter of the algorithm.

3.4 DAG representation and arrival times

Even though the theory presented in this paper uses trees as the basic object for Boolean manipulation, it can be easily extended to DAGs. By having a common manager to represent all trees, a single instance of each subtree in the manager can be guaranteed. The way to do that is similar to the approach used in BDD managers, in which a *unique table* stores all nodes in the manager.

By using this approach, the memoization of the ACD_SPEED algorithm can be simply implemented by comparing pointers in the table of explored solutions. For the sake of brevity, the details of the implementation will not be described in this paper, given that they do not differ significantly from the implementation of a BDD manager.

Additionally, the algorithms for speeding-up DAGs can be easily extended to inputs with different arrival times. Each arrival time can be considered as an attached depth to the input that cannot be modified by the transformation rules.

4 Logic decomposition

The decomposition of a Boolean function F is performed recursively from root to leaves by finding an operation op and two functions, A and B, such that $F = A \text{ op } B$. This type of decomposition is called bi-decomposition [3, 16, 10].

```
ACD_DECOMPOSE (ON, DC, ReqTime)
  { ON and DC are covers. Returns a tree }
  T_1 := BI-DECOMP (ON, DC, ReqTime, method_1);
  :
  T_n := BI-DECOMP (ON, DC, ReqTime, method_n);
  T := Choose_Best_Tree (T_1, ... , T_n);
  F_l := collapse (T.left); {cover of the left subtree}
  F_r := collapse (T.right); {cover of the right subtree}
  if depth(T.right) > depth(T.left) then swap(F_l, F_r);

  {Decompose the fastest child of the tree (left)}
  D_l := ACD_DECOMPOSE (F_l, DC, ReqTime -1);

  {Update DC for the slowest child of the tree}
  F_l := collapse (D_l); {cover of the left subtree}
  if T.op = AND then
    F_r := F_r · F_l;  DC = DC + F_l';
  else {T.op = OR}
    F_r := F_r · F_l'; DC = DC + F_l;

  {Decompose the slowest child of the tree}
  D_r := ACD_DECOMPOSE (F_r, DC, ReqTime -1);
  return (T.op, D_l, D_r);
---------------------------------------------------------
BI-DECOMP (ON, DC, ReqTime, method)
  { ON and DC are covers. Returns a tree }
  { "method" determines the decomposition strategy }
  F_d := Decompose_2input_gates (ON, DC, method);
  F_s := ACD_Speed (F_d, ReqTime);
  return F_s;
```

Figure 8. Algorithm for logic decomposition.

The main algorithm is shown in Figure 8. Since the approach attempts to explore different solutions, different bi-decomposition methods may fit in the same framework. The actual implementation uses two methods for bi-decomposition (function Decompose_2input_gates) that will be explained below.

The recursive paradigm behind the ACD_DECOMPOSE algorithm is as follows: (1) find bi-decompositions of an incompletely specified function, (2) optimize each bi-decomposition for delay (ACD_SPEED), (3) choose the best bi-decomposition and collapse the children, (4) recursively decompose the children. Note that the recursive call is done in such a way that the simplest child is decomposed first, whereas the second child is decomposed by enhancing its DC-set according to the function implemented by the other child.

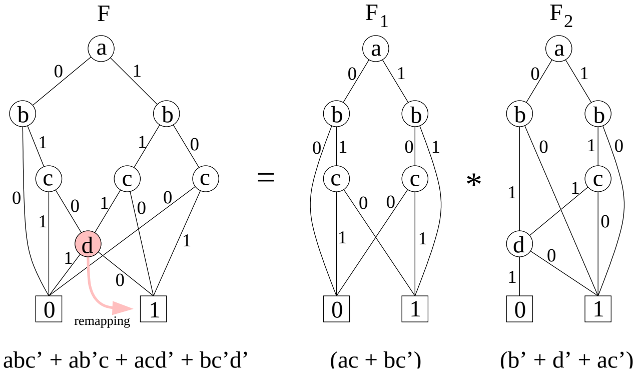

Figure 9. Bi-decomposition by approximation.

4.1 Bi-decomposition methods

Two bi-decomposition methods are used in the actual implementation of the decomposition algorithm.

The first one is a factorization based on the search of kernels and algebraic division [4]. This factorization is implemented by the function `factor_good` in SIS [12].

The second approach is computationally more expensive and based on BDD decompositions. Several approaches have been proposed in this direction. The one we have chosen has been inspired on the calculation of function approximations [11]. Fig. 9 illustrates a simple example on how a conjunctive decomposition for a function can be calculated by approximations. Given the BDD F, we want to find F_1 and F_2 such that $F = F_1 * F_2$. For that, we need F_1 and F_2 be over-approximations of F. The approach consists of *remapping* some nodes of F in such a way that the BDD size is reduced but the number of minterms of the new BDD is not increased too much (a *dense* over-approximation). In the figure, the approximation is calculated by remapping the node d into the constant 1. F is reduced by two nodes (F_1) and the number of minterms is increased by two. Once F_1 is known, F_2 can be calculated by BDD minimization: $F \subseteq F_2 \subseteq F + \overline{F}_1$. The approximations for disjunctive decomposition are similar (under-approximations must be used instead).

The actual BDD-based approach used in this paper is similar to the one in [11], but considering many more nodes as candidates for replacement (same level, children and grand-children).

It is important to notice that the approximation approach subsumes the conjunctive and disjunctive bi-decompositions proposed by other authors [9, 16], in which the BDD transformations can be reduced to re-mapping some nodes into constants or other nodes of the same BDD. Only the particular heuristics used in each approach may lead to different decomposition results in practice.

alg	**rug**	**bidec**[10]	**acd**	**acdr**
collapse				
algebraic*4	rugged*4	bidecomp	acd_decompose	
speed_up -d3			-	resub
map -n1 -AFG (library mcnc.genlib)				

Table 1. Scripts used for the experimental results.

	alg			**acd**		
circuit	delay	area	levels	delay	area	levels
9sym	18.3	378	12	11.8	178	9
apex6	23.7	1289	11	12.5	1960	8
count	16.7	403	7	11.1	537	7
frg1	16.7	213	11	10.0	101	7
lal	13.6	202	6	8.8	273	5
sct	13.1	140	6	8.5	195	5
vda	21.8	1456	9	15.9	2102	8

Table 2. Some salient examples.

5 Experimental results

The strategy presented in this paper has been implemented in SIS using DAG representations by means of a circuit manager, as explained in Section 3.4. The results have been compared with SIS and the method for bi-decomposition presented in [10]. The experiments have been run on 57 combinational circuits from the IWLS'93 benchmark set [6] using the scripts sketched in Table 1. The suffix *4 indicates that the script has been run four times (experimentally we found this number to be adequate to obtain good-quality results).

All the benchmarks were multi-level netlists. Initially, the circuits were collapsed and converted into 2-level forms. After that, the algebraic script `alg` was the one deriving the best results for SIS. The scripts `acd` and `acdr` are the ones implementing the strategy of this paper. The script `acd` derives a tree decomposition (no sharing), whereas `acdr` attempts to share as much logic as possible after decomposition, by means of algebraic re-substitution.

Table 2 reports the most remarkable results of the experiments. For some cases (9sym and frg1) area is drastically reduced due to the power of Boolean bi-decomposition. The column "levels" reports the number of levels of the circuit before technology mapping. The number of levels is counted as the depth of the circuit represented with 2-input gates (inverters are ignored). A summary of the results for the 57 benchmarks is presented in Table 3. The results have been obtained by adding all the individual results of each benchmark.

`Acd` obtains a 23% delay reduction at the expense of 49% area increase. If sharing is allowed (`acdr`) the delay reduction is 15%, but the area increase is only 18%. The

		normalized results wrt **alg**				
script	**alg**	**alg**	**rug**	**bidec**	**acd**	**acdr**
delay	688	1.00	1.23	1.03	0.77	0.85
area	12676	1.00	1.05	1.51	1.49	1.18
levels	363	1.00	1.22	1.07	0.88	0.88
cpu(sec)	305	1.00	1.93	0.33	2.10	2.12

Table 3. Summary of results for 57 benchmarks.

delay increase of `acdr` with regard to `acd` reduction is due to two factors: (1) the capacitive load of the shared nodes and, (2) sub-optimality of the tree-mapping algorithm when working on DAGs. We believe that results with delay similar to `acd` and area similar to `acdr` could be obtained by using DAG covering [8] or gate duplication [14] techniques.

The experimental results also manifest the problems of speeding-up networks that have been highly optimized for area. The results obtained by the `rugged` script are inferior, on average, than those obtained by the `algebraic` script. As an example, we took `apex6` from the benchmark suite and compared the networks before executing the `speed_up` command. Here are the results:

	algebraic		rugged	
	nodes	levels	nodes	levels
before speed_up	721	14	713	22
after speed_up	738	11	770	13

The algebraic script initially derives a slightly larger netlist (721 nodes, each node is a 2-input gate) with regard to the rugged script (713 nodes). However, the number of logic levels is much higher for the rugged script, due to the more aggressive sharing. This fact has a tangible impact when trying to speed-up the netlist. The result obtained by the rugged script ends up by having a larger number of nodes and levels. This example illustrates the phenomenon mentioned in the introduction of this paper (Fig. 1).

6 Conclusions

This paper has shown that speeding-up a Boolean network after having been reduced for area is not necessarily the best approach for synthesizing fast circuits. A novel approach for timing-driven decomposition has been presented. It combines bi-decomposition with tree-height reduction. Some specific heuristics to prune the exploration of the design space have been proposed. However, the major contribution of this work is the proposal of a strategy that aims at reducing the depth of a circuit by generating a balanced tree-like decomposition. Area reduction is performed by sharing isomorphic subtrees of the decomposition. This optimization framework can be enriched with any bi-decomposition technique proposed by other authors.

References

[1] J.L. Baer and D.P. Bovet. Compilation of arithmetic expressions for parallel computations. In *Proc. IFIP Congress*, pages 340–346, Amsterdam, 1968. North-Holland.

[2] J.C. Beatty. An axiomatic approach to code optimization for expressions. *Journal of the ACM*, 19(4):613–640, October 1972.

[3] D. Bochmann, F. Dresig, and B. Steinbach. A new decomposition method for multilevel circuit design. In *Proc. European Design Automation Conference (EURO-DAC)*, pages 374–377, 1991.

[4] R.K. Brayton and C. McMullen. The decomposition and factorization of Boolean expressions. In *Proc. of the Int. Symp. on Circuits and Systems*, pages 49–54, May 1982.

[5] A. Gibbons and W. Rytter. *Efficient parallel algorithms.* Cambridge University Press, 1988.

[6] IWLS'93 benchmark set. http://www.cbl.ncsu.edu/CBL_Docs/lgs93.html.

[7] D. Kuck. *The Structure of Computers and Computation.* John Wiley & Sons, 1978.

[8] Y. Kukimoto, R.K. Brayton, and P. Sawkar. Delay-optimal technology mapping by DAG covering. In *Design Automation Conference*, pages 348–351, 1998.

[9] Y.-T. Lai, K.-R. Pan, and M. Pedram. OBDD-based function decomposition: algorithms and implementation. *IEEE Transactions on Computer-Aided Design*, 15(8):977–990, August 1996.

[10] A. Mishchenko, B. Steinbach, and M. Perkowski. An algorithm for bi-decomposition of logic functions. In *Proc. ACM/IEEE Design Automation Conference*, pages 282–285, June 2001.

[11] K. Ravi, K.L. McMillan, T.R. Shiple, and F. Somenzi. Approximation and decomposition of binary decision diagrams. In *Design Automation Conference*, pages 445–450, 1998.

[12] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli. SIS: A system for sequential circuit synthesis. Technical report, U.C. Berkeley, May 1992.

[13] K.J. Singh, A.R. Wang, R.K. Brayton, and A. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 282–285, November 1988.

[14] A. Srivastava, R. Kastner, and M. Sarrafzadeh. On the complexity of gate duplication. *IEEE Transactions on Computer-Aided Design*, 20(9):1170–1176, September 2001.

[15] H. Touati, H. Savoj, and R.K. Brayton. Delay optimization of combinational circuits by clustering and partial collapsing. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 188–191, November 1991.

[16] C. Yang, M. Ciesielski, and V. Singhal. BDS: A BDD-based logic optimization system. In *Proc. ACM/IEEE Design Automation Conference*, pages 92–97, June 2000.