Delay Optimization Using SOP Balancing

Alan Mishchenko Robert Brayton
Department of EECS, University of California, Berkeley
{alanmi, brayton}@eecs.berkeley.edu

Stephen Jang
LogicMill Technology
sjang@logic-mill.com

Victor Kravets
IBM Corporation
kravets@us.ibm.com

ABSTRACT

Reducing delay of a digital circuit is an important topic in logic synthesis for standard cells and LUT-based FPGAs. This paper presents a simple, fast, and very efficient synthesis algorithm to improve the delay after technology mapping. The algorithm scales to large designs and is implemented in a publicly-available technology mapper. The code is available online. Experimental results on industrial designs show that the method can improve delay by 30% with the increase in area 2.4%, or by 41% with the increase in area by 3.9%, on top of a high-effort synthesis and mapping flow.

1. INTRODUCTION

Delay optimization has been studied extensively since the early days of logic design, as part of both technology independent [22][2][12][21] and technology dependent synthesis [10][15][9][5]. However, existing methods for delay optimization have several known limitations:

- Numerous local changes to the network may be applied, with no guarantee that the delay is globally improved or that additional area has been effectively spent for delay improvements.
- Algorithms of high computational complexity are often used, leading to prohibitive runtime on large designs. Much effort is spent on deciding where to make the changes.
- Structural flexibilities that are available during synthesis and potentially capable of producing a delay improvement may not be exploited by technology mapping.

The method described in this paper overcomes these limitations. Unlike previous methods, it does not perform a sequence of local changes, each one updating the mapped network and then running incremental timing analysis after each change. Instead, the proposed method transforms the subject graph before technology mapping, by minimizing the number of logic levels. A subject graph with structural choices [10][4] can be used as input to the algorithm, resulting in improved quality of results.

The method has been implemented as a straight-forward extension of the publicly available priority-cut-based technology mapper [18]. The extension is described in this paper. The resulting source code is publicly available for unrestricted use and as a benchmark for future comparisons.

The new logic structures for delay optimization are created by transforming logic structure of the cuts in the timing-critical areas. The technology mapper [18] allows for efficient area recovery in the regions where area inevitably grows due to initial logic duplication.

Previous methods in delay-oriented restructuring focused on MUX-based resynthesis, e.g. [2][19], generalized select transform (GST), e.g. [12][21], and various BDD-based techniques, e.g. [5][6]. The proposed method is simpler, scales better, and leads to competitive quality of results. It can also be extended to work for the sequential case, similar to the way delay optimization is done in [23].

The rest of this paper is organized as follows. Section 2 describes the background. Section 3 describes the algorithm. Section 4 reports experimental results. Section 5 concludes the paper and outlines future work.

2. BACKGROUND

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. In this paper, we consider only combinational Boolean networks.

A node *n* has zero or more *fanins,* i.e. nodes that are driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. A *fanin* (*fanout*) *cone* of node *n* is a subset of all nodes of the network, reachable through the fanin (fanout) edges of the node.

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these two-input ANDs are added to the AIG manager in a topological order. The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations of an AIG is to reduce both area and delay.

Structural hashing of AIGs ensures that all constants are propagated and, for each pair of nodes, there is at most one two-input AND with them as fanins (up to a permutation). Structural hashing is performed by hash-table lookups when AND nodes are created and added to an AIG manager. Structural hashing can be applied on-the-fly during AIG construction, which reduces the AIG size.

A *cut C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the *root* of cut *C*. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. A cut is *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

Area of a cut is the number of AIG nodes found on the path between the root and the leaves, including the root and excluding the leaves. The concepts of area and the number of AIG nodes are used interchangeably in this paper.

Delay of a cut is the number of AIG nodes on the longest path between the root of the cut and a primary input of the AIG. The concepts of delay, depth, and logic level are used interchangeably in this paper.

A *local* function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*. The *global function* of an AIG node is its function expressed in terms of the PIs of the AIG.

AIGs can efficiently represent both local and global functions. Because of their low memory usage, speed of manipulation and scalability, AIGs have recently emerged as a widely-used data-structure for various applications in logic synthesis and formal verification.

If Boolean functions in some application depend on 16 or fewer inputs, it is often more convenient to use truth tables to represent and manipulate them. For example, a truth table can be efficiently converted into an irredundant Sum-of-Products (ISOP) using a truth-table implementation of the Minato-Morreale algorithm [13][14].

Additional information can be found in the following publications: AIGs [11][3], AIG-based synthesis, [16][17], cut-based technology mapping, delay optimization, and area recovery can be found in [8][7][20][18].

3. ALGORITHM

This section introduces AND- and SOP-balancing, which are the key ingredients of the proposed algorithm, followed by the overall pseudo-code of the algorithm.

3.1 AND-balancing

AND-balancing of an AIG is a well-known fast transform that reduces the number of AIG levels. AND-balancing is performed in two steps: covering and tree-balancing.

The covering step identifies large multi-input ANDs in the AIG by grouping together two-input ANDs that have no complemented attributes in between and no external fanout, except possibly at the root node of each multi-input AND. The covering step is illustrated in Figure 3.1.1. The circles stand for two-input ANDs and the small bubbles on the edges stand for the complemented attributes.

The tree-balancing step decomposes each multi-input AND into two-input ANDs while trying to reduce the total number of AIG levels. As the result of this step, a new structure of two-input ANDs is created. This structure is constructed to minimize the delay while taking into account logic levels of the inputs. The tree-balancing step is illustrated in Figure 3.1.2.

It should be noted that the covering step is unique, while the tree-balancing step is not unique and depends on the grouping of the inputs with equal delay, while transforming multi-input ANDs into trees of two-input ANDs.

Because the covering step stops at the multiple-fanout nodes, AND-balancing cannot increase the total number of two-input AND nodes. However, some nodes can be reduced when AND-balancing is applied to a large AIG and logic sharing is created in the process.

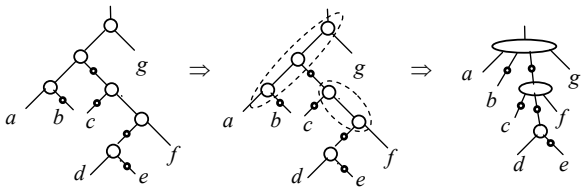

Figure 3.1.1: Illustration of the covering step.

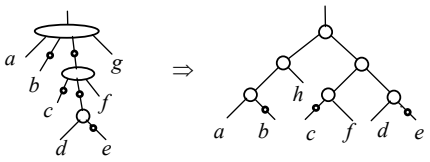

Figure 3.1.2: Illustration of the tree-balancing step.

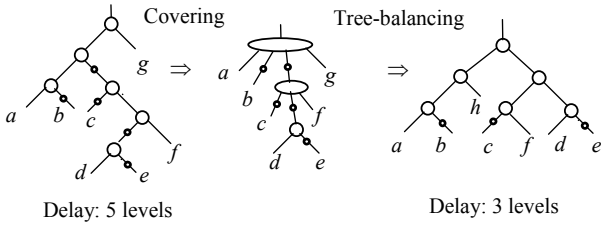

Figure 3.1.3: Illustration of AND-balancing.

Figure 3.1.3 illustrates AND-balancing, which combines covering and tree-balancing. In the above fitures, the delays of the PIs are assumed to be 0. The total delay of the AIG in this example is reduced from 5 to 3 levels.

AND-balancing described in this section is implemented in ABC [1] as command *balance*.

3.2 SOP-balancing of a small AIG

In this paper, an AIG is considered *small* if it depends on roughly 10 or less inputs. A small AIG can be converted into an SOP, and then AND-balancing can be applied to each product and the sum. In doing so, the products and the sum are treated as multi-input ANDs and decomposed to minimize the delay of the output node.

Figure 3.2.1 illustrates SOP-balancing for a small AIG, where the delays of the PIs are equal to 0. The total delay of

the AIG in this example is reduced from 4 to 3. Note that AND-balancing cannot reduce the delay in this example.

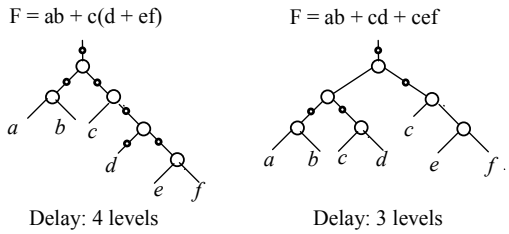

Figure 3.2.1: Illustration of SOP-balancing.

In general, AND-balancing is limited to multi-input ANDs, while SOP-balancing looks at larger functions. As a result, in many cases, SOP-balancing can reduce delay when AND-balancing cannot.

3.3 SOP-balancing of a large AIG

A large AIG, for example, the AIG representing combinational logic of an industrial design, can contain millions of AIG nodes. It is impossible to apply SOP-balancing to such an AIG as a whole, but it is possible to break it down into parts, try SOP-balancing for each part, and if the delay is improved, locally update the large AIG with the structure derived by SOP-balancing.

The latter is, in essence, the SOP-balancing algorithm described in this paper. A self-explanatory pseudo-code is given in Figure 3.3 below.

```
subject_graph performSopBalancing (
   subject_graph S,   // S is an And-Inverter Graph
   int K,             // K is the cut size
   int C  )           // C is the number of cuts at each node
{
   for each node n in S, in a topological order  {
      compute C structural K-input cuts of n;
      for each cut {
         compute truth table;
         compute irredundant SOP;
         perform delay-optimal balancing of the SOP;
         if ( the cut has smaller AIG level than the best cut )
            save the cut as the best cut;
      }
      if ( root node AIG level is reduced using the best cut )
         update AIG structure;
   }
   return S;
}
```

Figure 3.3. Pseudo-code of SOP-balancing.

4. EXPERIMENTAL RESULTS

The proposed algorithm is implemented in ABC [1][3] as command sequence (*if –g –K <num> -C <num>*; *st*), where

- *if* is the priority-cut-based FPGA mapper [18],
- *-g* enables SOP-balancing for cut evaluation,
- *-K <num>* specifies the cuts size and,
- *-C <num>* is the number of cuts used at a node,
- *st* transforms the mapped network back into an AIG.

The input of the command sequence is an AIG. The output is a delay-optimized AIG, with the reduced number of logic levels on any path from the PIs to the POs.

The following cost functions are used to prioritize the cuts in the priority-cut-based mapper:

- *Delay* of a cut the root node level, counting from the PIs of the AIG, after SOP-balancing was applied to the Boolean function of the cut.
- *Area* of a cut is the number of two-input ANDs derived after SOP-balancing was applied to the Boolean function of the cut.

Mapping into standard cells was performed by command *map* [4] in ABC. Experiments targeting standard-cell library *mcnc.genlib* from SIS distribution [24] were run on a workstation with Intel Xeon Quad Core CPU and 48Gb RAM. Only one thread and less than 1Gb of RAM were used for the largest design in our experiments. The resulting networks were verified by a SAT-based combinational equivalence checker (command *cec* in ABC).

The experimental results were collected using a suite of industrial designs optimized in three different ways:

- Reference run: $(st; dch; map)^4$
- Run 1: $(st; if -g -K\ 6 -C\ 8)(st; dch; map)^4$
- Run 2: $(st; if -g -K\ 6 -C\ 8)^2(st; dch; map)^6$

The reference run is a typical synthesis and mapping flow targeting standard-cells. It consists of four iterations. Each iteration derives an AIG (*st*), followed by AIG-based synthesis with choices (*dch*), followed by cut-based technology mapping (*map*). This or a very similar flow is currently used by most of the industrial users of ABC.

Run 1 performs one iteration of delay optimization followed by the reference flow (4 iterations).

Run 2 performs 2 iterations of delay optimization followed by the 1½ reference flows (6 iterations). The increased effort of the reference flow was needed to mitigate area increase.

The results for the three experimental runs are reported in Table 4.1. Two outlier designs were removed from the table because the delay improvement exceeded 50%, and this would skew the general conclusions. The table shows that, compared to the reference flow, Run 1 reduces delay by 30% with area increase of 2.4%, Run 2 reduced delay by 41% with area increase of 3.9%.

Table 4.2 shows the detailed break-down of delay improvements for one design in our test suite. The table lists area and delay after standard-cell mapping, level count in the AIG before mapping and in the resulting mapped network, as well as the runtime, in seconds, for each step of each of the optimization flows (Reference, Run 1, Run 2). These table indicates that the proposed method is very efficient in reducing the total number of AIG levels as well as the number of levels in the mapped network, which leads to delay reduction after technology mapping. The table also shows that area increase can be further reduced by performing more iterations of logic synthesis with choices.

In another experiment, we applied the proposed delay optimization based on SOP-balancing to MCNC benchmarks. The delay improvements were similar to those in Table 4.1 for industrial designs, but the area penalty was higher. We speculate that this is because the ratio of the critical path to the total amount of logic is relatively high in these benchmarks. The detailed results for MCNC benchmarks are not reported here because they are not representative of realistic circuits synthesized these days.

Finally, a similar flow was applied to FPGA mapping, but the delay improvements were not as substantial as for standard cells reported in this paper. We speculate that this is due to LUT mapping being less sensitive to the number of levels and more sensitive to the logic density on the critical path.

4.1 Discussion

It is important to note that the delay model used by the ABC mapper is approximate. Therefore some part of the improvement will be lost, when the mapped netlist is post-processed by a typical industrial physical synthesis flow, which performs gate-sizing, buffering, gate-duplication, and other steps, followed by place-and-route. However, given the high margin of improvement, it is likely that some of the delay reduction will persist even after place-and-route.

To support this, in a separate experiment, an industrial collaborator applied the proposed method to several test cases, followed by the full physical synthesis flow, including place-and-route for standard cells. This led to an improvement close to 5% is delay, compared to the typical high-effort flow used in that company.

We hope to be able to list the detailed results of this experiment in the final version of the paper.

5. CONCLUSIONS

This paper introduces a simple, fast, and efficient algorithm for delay optimization after technology mapping.

The proposed algorithm preprocesses the subject graph represented as an AIG to reduce the number of levels of two-input ANDs. It is implemented as a straight-forward modification of the publicly-available priority-cut-based technology mapper [18] and its runtime is close to that one run of the mapper. The area increase due to logic duplication is relatively small because of the efficient area recovery done as part of the logic synthesis flow.

Future work may include: (a) improving the quality of the algorithm by pre-computing the smallest delay AIG subgraphs, instead of deriving them using SOP balancing. (b) measuring the improvements in delay after place-and-route, (c) extending the algorithm to work for sequential circuits, as suggested in [23].

6. REFERENCES

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. http://www-cad.eecs.berkeley.edu/~alanmi/abc

[2] C. L. Berman, D. J. Hathaway, A. S. LaPaugh, and L. H. Trevillyan, “Efficient techniques for timing correction”, *Proc. ISCAS ’90*.

[3] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool", *Proc. CAV'10*, LNCS 6174, pp. 24-40.

[4] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, “Reducing structural bias in technology mapping”, *ICCAD '05*.

[5] L. Cheng, D. Chen, and D.F. Wong, “DDBDD: Delay-driven BDD synthesis for FPGAs”, *Proc. DAC’07*, pp. 910-915. http://www.icims.csl.uiuc.edu/~dchen/ddbdd.pdf

[6] M. Choudhury and K. Mohanram, “Bi-decomposition of large Boolean functions using blocking edge graphs”, *Proc. ICCAD’10*.

[7] J. Cong and Y. Ding, “FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs”, *IEEE Trans. CAD*, vol. 13(1), Jan. 1994, pp. 1-12.

[8] R. J. Francis, J. Rose, and K. Chung, ”Chortle: A technology mapping program for lookup table-based field programmable gate arrays”, *Proc. DAC ’90*, pp. 613-619.

[9] V. N. Kravets and P. Kudva, “Implicit enumeration of structural changes in circuit optimization”, *Proc. DAC ’04*, pp. 438-441.

[10] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness, “Logic decomposition during technology mapping,” *IEEE Trans. CAD*, Vol. 16(8), Aug. 1997, pp. 813-833.

[11] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust boolean reasoning for equivalence checking and functional property verification”, *IEEE TCAD*, Vol. 21(12), Dec. 2002, pp. 1377-1394.

[12] P. McGeer, R. K. Brayton, A. L. Sangiovanni-Vincentelli, and S. K. Sahni, “Performance enhancement through the generalized bypass transform”, *Proc. ICCAD‘91*, pp. 184-187.

[13] S. Minato, “Fast generation of irredundant sum-of-products forms from binary decision diagrams”. *Proc. of SASIMI'92 (Synthesis and Simulation Meeting and International Interchange)*, Kobe, Japan, pp. 64-73.

[14] E. Morreale, “Recursive Operators for Prime Implicant and Irredundant Normal Form Determination”. *IEEE Trans. Comp*., C-19(6), 1970, pp. 504-509.

[15] A. Mishchenko, X. Wang, and T. Kam, "A new enhanced constructive decomposition and mapping algorithm", *DAC '03*, pp. 143-148.

[16] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536.

[17] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure", Proc. IWLS '06, pp. 15-22.

[18] A. Mishchenko, S. Cho, S. Chatterjee, R. Brayton, “Combinational and sequential mapping with priority cuts”, *Proc. ICCAD ’07*, pp. 354-361.

[19] A. Mishchenko, R. Brayton, and S. Jang, "Global delay optimization using structural choices", *Proc. FPGA'10*, pp. 181-184.

[20] P. Pan and C.-C. Lin, “A new retiming-based technology mapping algorithm for LUT-based FPGAs,” *Proc. FPGA’98*, pp. 35-42.

[21] A. Saldanha, H. Harkness, P.C. McGeer, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Performance optimization using exact sensitization”, *Proc. DAC’94*, pp. 425-429.

[22] K. J. Singh, A. R. Wang, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Timing optimization of combinational logic”. *Proc. ICCAD ‘88*, pp. 282- 285.

[23] C. Soviani, O. Tardieu, and S. A. Edwards, “Optimizing sequential cycles through Shannon decomposition and retiming”, *IEEE Trans. CAD*, Vol. 26(3), March 2007, pp. 456-467.

[24] 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,* UCB/ERI, M92/41, ERL, Dept. of EECS, UC Berkeley, 1992.

Table 4.1. Experimental evaluation of the proposed algorithm on industrial circuits after standard-cell mapping.

Design	Reference		Run 1		Run 2	
	Area	Delay	Area	Delay	Area	Delay
D01	180978	34.2	180002	30.0	178099	27.7
D02	16296	15.0	16540	12.8	16082	12.3
D03	50431	41.4	56212	38.6	56212	38.6
D04	16296	15.0	16540	12.8	16082	12.3
D05	509984	74.1	554324	31.4	562109	25.7
D06	443913	37.9	443573	23.9	443181	20.2
D07	80939	21.4	82438	19.9	80347	18.6
D08	257609	31.3	263519	20.8	257917	21.4
D09	597980	81.2	620415	48.0	626055	42.8
D10	612608	32.1	621065	22.3	621838	19.8
D11	73191	46.0	74413	19.8	76346	14.5
D12	429761	48.4	443453	32.9	449604	25.2
D13	236783	26.2	239248	17.5	237456	14.4
D14	848678	54.4	885102	40.4	873752	39.3
D15	13066	54.4	13385	34.0	14561	26.0
D16	220757	80.9	216977	56.0	224621	26.3
D17	316893	19.7	314956	18.7	310999	18.6
Geomean	**158148**	**36.9**	**161990**	**25.86**	**180418**	**21.8**
Ratio	**1**	**1**	**1.024**	**0.70**	**1.039**	**0.59**

Table 4.2. Detailed breakdown of delay improvement achieved on one design in the test suite.

Experiments performed	Sequence of optimization steps	Final Mapped Area	Final Mapped Delay	Starting AIG Level	Final Mapped Level	Runtime, sec
Reference flow	st; dch; map	224079	92.90	164	89	222
	st; dch; map	221866	82.10	160	75	143
	st; dch; map	220757	80.90	112	71	136
Run 1	st; if -K 6 -g -C 8	n/a	n/a	164	n/a	66
	st; dch; map	230138	45.00	55	40	208
	st; dch; map	221435	44.60	58	39	149
	st; dch; map	220171	44.60	57	29	143
Run 2	st; if -K 6 -g -C 8	n/a	n/a	164	n/a	66
	st; if -K 6 -g -C 8	n/a	n/a	55	n/a	63
	st; dch; map	240809	25.30	30	24	227
	st; dch; map	232301	25.30	36	25	165
	st; dch; map	230393	25.20	40	24	160
	st; dch; map	229464	24.90	39	24	155
	st; dch; map	228302	25.60	38	24	158
	st; dch; map	227636	25.70	39	24	154