Global Delay Optimization using Structural Choices

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

Stephen Jang
non-affiliated
San Jose, CA
stephenjang@yahoo.com

ABSTRACT

This paper presents a fast global method for delay optimization after technology mapping. Timing analysis is used to identify timing-critical areas in the mapped network where new structures are synthesized to favor late-arriving signals. Unlike previous methods that make incremental local changes to the mapped network, the proposed method records many alternative structures and defers the final decision to the technology mapper. Experimental results for networks mapped into 6-input look-up tables (6-LUTs) show that the delay is, on average, improved 14% using a realistic delay library for LUTs with variable-pin delays and wire-delay estimation. The area penalty after the delay optimization is about 2% and can be eliminated by area-oriented resynthesis. When the algorithm is compared with DAOmap, the experimental results show 27% logic level reduction while the area is increased by only 1%. The algorithm is also fast and applicable to very large networks. The runtime of the proposed algorithm is 16x and 3x faster than DAOmap for large industrial and academic designs, respectively.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design Aids – *Optimization*; B.7.1 [**Integrated Circuits**]: Types and Design Styles – *Gate Arrays*; J.6 [**Computer-Aided Engineering**]: *Computer-Aided Design (CAD)*

General Terms: Algorithms, Performance, Experimentation

Keywords: FPGA, Technology Mapping, Logic Optimization, Windowing, Boolean Satisfiability, Interpolation

1. INTRODUCTION

Technology mapping converts a logic network, called the *subject graph*, into a network of logic nodes, which correspond to primitives in a particular technology. For example, technology mapping for FPGAs [4][6][7] transforms the network into *K*-input lookup tables (*K*-LUTs) that can implement any Boolean function of *K* or less variables. The subject graph is often an And-Inverter Graph (AIG), composed of two-input ANDs and inverters.

Logic restructuring with the goal of reducing delay of a mapped network has long been an important part of both technology independent [17][2][12][16] and technology dependent synthesis [10][13][9][5]. However, existing methods for delay-oriented logic restructuring have the following drawbacks:

- Numerous local changes to the network may be applied, but with no guarantee that the delay is globally improved or that area has been effectively spent for delay improvements.
- Algorithms of high computational complexity are often used, leading to excessive runtime. Much effort is spent on deciding where to make the changes.
- Structural flexibilities that may be available during mapping are typically not used during post-mapping resynthesis.

The proposed method mitigates these limitations. Unlike the 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. It need not analyze cuts of a set of critical paths nor distribute slacks on off-critical paths. Instead, it computes timing information only once. This is analyzed and a set of new candidate structures, a subset of which may lead to reducing delay after the next iteration of mapping, are recorded in the subject graph using *choice* nodes [10][3].

Decisions about which subset of structures to use are deferred to the technology mapper. The motivation is that the mapper has a global picture of both delay and area, as well as a good view of structural flexibilities presented during mapping, and thus can better decide what structures to use.

The new logic structures are created by cofactoring logic cones, in timing critical regions, with respect to timing-critical variables. The cofactors are combined using multiplexers controlled by the critical variables and then decomposed and simplified as an AIG, resulting in a logic structure that favors the late-arriving variables. A similar method was discussed in [2] and called the generalized select transform (GST) [12][16]. This was used recently for timing optimization of sequential circuits [18]. The proposed algorithm can be extended to work for the sequential case as well.

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. If the network is sequential, the memory elements are assumed to be D-flip-flops with initial states.

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 *node* is a logic component having a propagation delay. An *edge*, also called *wire*, is the pin-to-pin connection between two adjacent nodes. The delay of a path includes *logic delays* and *wire delays*. The *logic delay* occurs in a logic component, such as a LUT. The *wire delay* occurs on the edges. In modern FPGAs, the delay for each pin in a LUT is different, so a variable-pin-delay

FPGA'10, February 21–23, 2010, Monterey, California, USA.
Copyright 2010 ACM 978-1-60558-911-4/10/02...$10.00.

model is used in this paper. Wire delays are usually not known until placement and routing. To approximate wire delays, a fixed delay is added to the delay of all pins of the LUTs in the library.

3. ALGORITHM

A self-explanatory pseudo-code is given in Figure 3.1. The algorithm takes the mapped netlist and the subject graph used for mapping. The netlist is analyzed to detect timing-critical areas for logic restructuring while the subject graph is used to accumulate the alternative logic structures for the next round of mapping. Several parameters are used in the computation: the timing window (*w*) determines the range of slacks of the near-timing-critical nodes to be restructured; the logic depth (*l*) of cones selected for delay-oriented restructuring; and the limit (*p*) on the number of timing-critical edges of the cone to consider.

```
mapped netlist performSpeedup (
  subject graph S,     // S is an And-Inverter Graph
  mapped netlist M, // M was previously derived by tech-mapping of S
  timing window w, // w is used to detect the critical path
  logic depth l,       // l is used to detect a logic cone rooted at a node
  edge count p         // p limits the number critical edges of the cone  {
     perform timing analysis of M with unit-delay or LUT-library model;
     detect the critical section of M as nodes n such that $0 \leq \text{slack}(n) \leq w$;
     detect timing-critical edges connecting these nodes;
     for each timing critical node n   {
        find cone C of M that extends l levels down from n;
        detect the set of timing-critical edges V feeding into C;
        if the number of edges in V exceeds p
             continue;
        find logic cone C' in S corresponding to C in M;
        find variables V' in S corresponding to V in M;
        derive cofactors of the function of C' w.r.t. variables in V';
        build multiplexer tree C'' of the cofactors using variables in V';
        add structural choice C'= C'' to the subject graph S;
     }
     derive netlist M' by mapping subject graph S with added choices;
     return M';
}
```

Figure 3.1. Overall pseudo-code of the algorithm.

The following subsections provide the details on timing analysis and identification of timing-critical edges (Section 3.1), logic restructuring for delay (Section 3.2), and using structural choices in technology mapping (Section 3.3).

3.1 Timing analysis

The purpose of *timing analysis* is to determine the *critical nodes* and *critical edges* by running a *delay trace*. Timing analysis for a mapped netlist consists in computing arrival times, required times, and slacks of all nodes and edges in the netlist. An *arrival time* of a node is the longest time for a signal to travel from a PI to the node. A *required time* of a node is the latest time for the node to produce its value, so that when it propagates to the POs, their arrival times there do not exceed the delay requirements.

Timing analysis is performed in two passes. In the first pass, the arrival times are computed for each node and its fanin edges, in a topological order from the PIs to the POs. The arrival time of an edge is the arrival time of its driving (fanin) node plus the pin and wire delay. In the second pass, the required times are propagated in a reverse topological order from the POs to the PIs. The required time of an edge is the required time of its fanout node minus the pin and wire delay. The required time of a node is the minimum required time of its fanout edges.

The *slack* of a node (edge) is the difference between its required time and its arrival time. Nodes (edges) with the slack close to zero are called *timing-critical*. A *timing-critical edge* is an edge, i.e., a fanin/fanout connection between two timing-critical nodes, such that reducing the arrival time of the fanin is necessary (but not sufficient) to reduce the arrival time of the fanout. Not every edge between two timing-critical nodes is timing-critical.

3.2 Logic restructuring for delay

Cone selection for logic structuring is governed by several heuristics aimed at maximizing the chances of improving delay.

For each timing-critical node, we consider one cone rooted in the given node and reaching a fixed amount of logic levels towards the PIs. Considering more than one cone per node can lead to a large number of structural choices, which may degrade the quality of mapping. Limiting the number of levels included in the cone prevents restructuring from duplicating too much area.

If the cone has more than a fixed number *p* (say, *p*=2) of timing-critical edges, it is not considered because restructuring of this cone cannot give delay improvement. This is because the tree of 2:1 MUXes added on top to assemble the cofactors has a delay that may offset the gains due to restructuring.

Finally, the timing-critical edges of the cone are ordered in decreasing order of criticality. The variables corresponding to these edges are used for cofactoring and become control variables of the MUXes. The more timing-critical that a variable is, it is put closer to the output of the cone after restructuring. For example, if variable *x* arrives later than *y*, it is used to control the top-most multiplexer on the right of Figure 3.2.

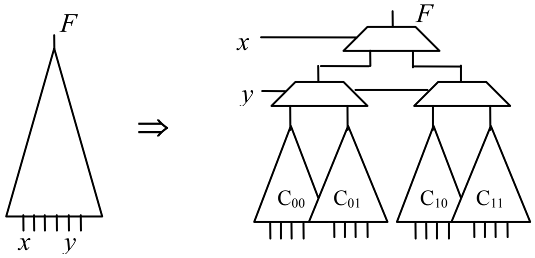

Figure 3.2. Illustration of the delay-oriented restructuring.

3.3 Adding choices and re-mapping

Multiple logic structures can be recorded in the AIG representing the network, using structural choices. A *structural choice* is a set of AIG nodes that are functionally equivalent, up to complementation. The first node in the topological order among the set is called the *representative*. The representative is the only node in the set that has fanouts. In the implementation of the AIG package, each node belonging to a choice uses two additional pointers: the first gives the representative while the second is used to link-list the nodes belonging to the same choice.

The choices added by the proposed delay-optimization algorithm are AIG structures derived by cofactoring with respect to timing-critical nodes. Given an AIG cone with *N* timing-critical nodes, 2^N cofactors are computed. Next, a 2^N-to-1 MUX is created with the cofactors as data-inputs and the cofactoring variables as controls, as shown in Figure 3.2. The root AIG node of the MUX is added as a choice of the root of the original cone. Structural hashing of the AIGs quickly removes structurally equivalent parts of the cofactor logic cones. Since logic synthesis and mapping are often iterated, a more elaborate logic synthesis of the cofactor logic cones is deferred to after the next round of synthesis.

Structural FPGA technology mapping was pioneered in [8][6]. Technology mapping with structural choices was introduced in

[10] and recently developed in [3]. In the latter case, the subject graph is an AIG and efficient equivalence checking [11] is used to detect functionally-equivalent nodes. To use structural choices, only one aspect of the mapper needs to be modified, namely, cut enumeration. Whether complete [15] or partial [14] cut enumeration is used, cuts are computed in a topological order, by merging the fanins' cut sets to produce the cut set of the choice node. In the presence of structural choices, the cut set of the representative node is computed as the union of cut sets of other nodes in the choice set. Recall that only the representative has fanout. Therefore, cuts computed for all nodes in a choice are propagated to the fanouts through the representative, and used to compute the cuts of the fanout nodes.

The advantage of using choices is that technology mapping can favor delay-oriented choices on the critical path and area-oriented choices elsewhere. The total number of different structures explored is exponential in the number of choices because decisions at each choice are made independently by the mapper.

4. EXPERIMENTAL RESULTS

The proposed delay-optimization algorithm was implemented as command *speedup* in the synthesis and verification system ABC [1]. Its technology independent algorithms are based on rewriting AIGs and iterating this many times. Mapping is performed by the priority-cut-based LUT-mapper [14] (command *if*), which attempts to minimize delay and recover area off the critical paths. Experiments targeting FPGAs with 6-LUTs were run on an Intel Xeon 2-CPU 4-core computer with 8Gb RAM. The resulting networks were verified using a SAT-based combinational equivalence checker (command *cec* in ABC).

The LUT library, for $K = 6$, used in the experiments is shown in Figure 4.1 using the ABC LUT-library format. The LUT sizes, listed first on each line, should be in increasing order. Listed next is the LUT area followed by the delays of each pin of the LUT, also in increasing order. There are as many delay numbers as there are input pins of the given LUT.

Figure 4.1 shows the variable-pin-delay LUT library used in the experiments. To make the delay realistic, a constant value (0.2) was added to all the pin delays to simulate delays in the wires. If this value is not added, two 2-LUTs are faster than one 4-LUT, which is not true in practice. It can be observed that, as the added wire delay value is increased, the variable-pin-delay model becomes more like a unit-area unit-delay model. For the unit-delay mode, all pin delays have the same value, 1.0.

We performed two experiments. In the first one, the results of the proposed algorithm, called *speedup*, are compared with the baseline flow using the variable-pin-delay model. In the second experiment, *speedup* is compared with DAOmap [4], a state-of-the-art mapper, using the unit- delay model.

4.1 Speedup vs. Baseline

Experiments on 20 industrial benchmarks ranging in size from 1K to 50K 6-LUTs are reported in Table 4.1. The following notation is used: columns denoted "PI", "PO", and "Reg" list the number of primary inputs, primary outputs, and registers in the design. Columns "LUT", "Lev", and "Delay" report the number of 6-LUTs, the maximum number of logic levels and the maximum delay. Columns "Time1" and "Time2" give, respectively, the runtime (seconds) of command *speedup* and the runtime of the total synthesis/mapping flow.

The baseline synthesis/mapping flow reported in the table was done by the ABC script: (*st; dchoice; if -C* 16 *-F* 2)^{8}, where the exponent (here, 8) shows how many times the script is iterated. At the end of the script, the best result is selected among all the results observed at the end of each iteration. The command *st* converts the circuit into an AIG and structurally hashes it. The command *dchoice* uses various methods for rewriting the AIG to minimize the number of AIG nodes while not increasing the number of its levels. In particular, *dchoice* strives for smaller delay by "balancing", which decreases the number of AIG levels by decomposing "wide-input" AND gates in a balanced way. During its synthesis, *dchoice* saves three snapshots of the circuit seen during this flow, the initial, an intermediate and the final. The command *if* is a cut-based FPGA mapper, which first minimizes the delays along all paths, using the delay model in the library, and then executes several iterations to recover area on the off-critical paths. Thus, the baseline technology independent synthesis mostly targets small area while reducing the AIG level somewhat. Most of the delay optimization in the baseline script is in the mapper, which makes the choices between area and delay during the mapping phase using the library delay model.

Delay-optimization (columns "Speedup") was done with the following script: (*st; dchoice; if -C* 16 *-F* 2)^{4} (*speedup; if -C* 16 *-F* 2)^{3} (*st; dchoice; if -C* 16 *-F* 2)^{4}. The *speedup* command is our implementation of the algorithm described in Section 3. The three iterations of *speedup* were performed with parameter *p* equal to 1, 2, and 3, respectively, which controls the number of timing critical edges allowed in the restructuring of a logic cone.

```
1   1.0   0.4
2   1.0   0.4 0.5
3   1.0   0.4 0.5 0.6
4   1.0   0.4 0.5 0.6 0.65
5   1.0   0.4 0.5 0.6 0.65 0.75
6   1.0   0.4 0.5 0.6 0.65 0.75 0.85
```

Figure 4.1. A variable-pin-delay LUT library with wire-delays.

The experimental results (Table 4.1) show that the delay, compared to the baseline result, was on average reduced by 14%. Area was increased by 2%; however, in a separate experiment, not shown, this area increase was eliminated by several iterations of area-oriented resynthesis (commands *mfs* and *lutpack* in ABC).

The runtime of *speedup* is fast, only about 10-12% of the total runtime of the delay-optimization flow. The total runtime of this flow is dominated by command *dchoice*, which performs 15 iterations of AIG-based logic synthesis, saves three snapshots of the network, derives structural choices, and uses them for technology mapping performed by command *if.* The total runtime the delay-optimization flow is roughly the same as that of the baseline plus the runtime of command *speedup.* This is because the number of additional choices produced by *speedup* is relatively small and does not impact the runtime of mapping. The total runtime of delay optimization, which included 8 iterations of *dchoice* and 3 iterations of *speedup*, was about 6 minutes for the largest reported benchmark with 50K 6-LUTs.

5. CONCLUSIONS

This paper proposes a simple and efficient algorithm for improving delay after technology mapping. The algorithm creates alternative structures for speeding up a circuit. The location of these structures is not as critical as in classical methods because their possible use is postponed until the next round of mapping. By iterating the speedup and technology mapping several times, subsequent delay improvements can be obtained. The algorithm is efficient because only one delay analysis is done during each speedup synthesis phase, while other methods iterate delay analysis during each new structure found.

Future work should include: (a) quantifying the improvements in delay after place-and-route, (b) extending the algorithm to work for sequential circuits as suggested in [18], (c) applying similar optimizations for cost functions other than delay. Also, the algorithm should be equally applicable to mapping for standard cells as well as FPGAs, and this should be developed.

6. ACKNOWLEDGMENTS

This work is supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the industrial sponsors Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

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

[4] D. Chen and J. Cong, “DAOmap: A depth-optimal area optimization mapping algorithms for FPGA designs”, *Proc. ICCAD*’04

[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] 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.

[7] J. Cong, J. Peck, and Y. Ding, "RASP: A General Logic Synthesis System for SRAM-based FPGAs," *Proc. FPGA*, pp. 137-143, 1996.

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

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

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

[16] 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.

[17] 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.

[18] 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.

Table 4.1. Experimental evaluation of *speedup* on industrial circuits using variable-pin delay model.

Design	Profile			Baseline				Speedup				
	PI	PO	Reg	LUT	Lev	Delay	Total	LUT	Lev	Delay	Time1, s	Time2, s
1	2,420	1,243	1,963	4,956	17	7.00	40.28	5,175	11	4.80	4.12	45.79
2	13,827	9,528	7,111	19,222	18	7.45	113.52	19,349	17	7.25	14.32	128.35
3	37	28	9,829	11,775	8	3.65	47.00	12,134	8	3.50	8.92	57.38
4	643	918	7,177	9,056	8	3.65	32.97	9,172	6	2.70	4.13	37.92
5	8,927	10,761	26,246	38,734	8	3.45	151.46	39,030	7	3.05	17.77	171.06
6	378	395	1,297	3,289	6	2.85	19.56	3,282	6	2.85	2.60	22.55
7	730	583	3,330	5,532	16	6.70	36.66	5,859	14	6.35	5.87	43.06
8	367	154	2,606	5,371	14	5.85	36.66	5,403	12	5.35	4.11	40.97
9	966	1,434	12,733	18,258	8	3.75	88.15	18,302	8	3.45	10.62	97.65
10	2,061	1,897	13,950	16,531	7	3.15	77.38	16,652	7	2.95	9.16	85.48
11	2,061	1,897	13,950	16,531	7	3.15	77.70	16,652	7	2.95	9.33	87.95
12	50	68	1,358	3,284	19	8.40	23.88	3,371	16	7.00	3.46	28.68
13	1,044	1,098	2,074	7,147	23	9.35	74.39	7,789	16	6.65	7.37	86.71
14	391	129	1,049	7,526	14	6.05	251.11	7,573	14	6.05	27.29	280.41
15	749	777	7,348	16,086	10	4.35	169.25	16,097	9	4.00	18.48	188.00
16	1,041	736	1,063	3,611	11	4.70	19.63	3,621	11	4.65	2.77	22.71
17	3,512	2,992	3,425	12,533	20	8.45	178.58	12,830	17	7.40	13.19	199.36
18	11,456	10,791	10,114	27,622	15	6.25	160.22	28,857	10	4.35	22.29	184.63
19	11,292	11,454	20,184	49,871	12	5.00	317.79	50,283	9	3.75	37.83	355.19
20	131	129	26258	13,811	8	3.65	72.17	14,186	5	2.45	8.23	81.60
Geomean				**10,804**	**11.49**	**4.99**	**72.13**	**11,023**	**9.80**	**4.29**	**8.77**	**82.29**
Ratio 1				**1.000**	**1.000**	**1.000**		**1.020**	**0.854**	**0.860**		
Ratio 2											**0.107**	**1.000**