ExtensiveSlackBalance: an Approach to Make Front-end Tools Aware of Clock Skew Scheduling

Kui Wang, Lian Duan, Xu Cheng
Dept. of Computer Science, Peking University
Beijing, China
{wangkui,duanlian,chengxu}@mprc.pku.edu.cn

ABSTRACT

In the traditional ASIC flow, clock skew scheduling (CSS), as a method to improve timing, is usually employed during the CTS (clock tree synthesis) step while front-end tools do not take clock skew as a manageable resource. This limits the potential of the subsequent CSS. To overcome such limitations, we design an enhanced CSS algorithm *ExtensiveSlackBalance* and integrate it into the back-annotation and re-optimization iterations of the current industrial flow. Experiment results show that, the clock frequency can be improved to 26.2% on average compared to 6.4% in the traditional ASIC flow.

Categories and Subject Descriptors: B.6.3 [Design Aids]: Optimization

General Terms: Algorithms

Keywords: clock skew, skew scheduling, logic synthesis, back-annotation

1. INTRODUCTION

Clock skew scheduling (CSS) [3] is an important way to improve circuit performance. It can minimize the clock cycle time by adjusting the individual delays for the clock signals of registers. Both binary searching [6] and graph-theoretic approaches [8] were developed for CSS. The technology of CSS has been mature even from the industrial stand-point [9] and is usually incorporated into commercial tools as the so called "useful skew" feature.

However, in the standard industrial ASIC flow, front-end tools (logic synthesis and placement tools) lack interaction with CSS. Front-end tools are not aware that the subsequent CSS can change clock signal arrival times and CSS algorithms are not aware that re-optimization iterations can change path delays. Thus, viewed from the entire design flow, the performance improvement through CSS is limited. On one hand, CSS may be obstructed by short paths and poorly-optimized long paths which are produced by front-end tools with zero-skew assumption. On the other hand, front-end tools may spend too much resources to fix setup violations that would be fixed by CSS cheaply.

Some front-end tools, such as Synopsys DesignCompiler, allow users to set different clock arrival times to given registers in the design constrain file for synthesis (SDC). To solve above problems, designers usually utilize this feature for back-annotating to front-end tools a CSS-plan that is manually designed or generated by a CSS algorithm for CTS tools. To this day, no mainstream EDA tools can automatically generate a CSS-plan dedicatedly for the design constrain used during synthesis.

In recent years, some academic approaches have also been proposed. Huang et al [4] present an algorithm which incorporates optimal CSS and delay insertion to overcome hold time violations caused by short paths. Hurst et al [5] introduce a placement algorithm which is based on a tight integration of sequential timing analysis in the inner loop of an analytic solver to minimize the maximum mean delay on any circuit loop. That makes placement tools utilize the full optimization potential of CSS and in-place retiming. However, these approaches need to modify back-end or front-end tools and are hard to employ in the existing industrial flow.

We try to make *existing* front-end tools be "CSS-aware" and maximize the timing improvement achieved through CSS in *current* industrial ASIC flow. We simply insert an enhanced CSS algorithm *ExtensiveSlackBalance* into back-annotation and re-optimization iterations in the existing industrial flow. Our algorithm is dedicatedly designed for generating design constraint for front-end tools: it balances all the critical cycles rather than balancing only the slacks of edges in the most critical cycle like traditional CSS algorithms designed for CTS tools. Experiment results show that our method can greatly improve clock skew optimization. Frequency improvements through CSS are averagely 26.2% compared to 6.4% of traditional design flow.

The organization of this paper is as follows. Section 2 introduces the basic terminologies and theories of CSS. Section 3 shows how our CSS algorithm is integrated into the industrial ASIC flow. Section 4 presents the detail of our algorithm. Analytic results are presented in Section 5, and some conclusions are drawn in Section 6.

2. PROBLEM FORMULATION

A sequential circuit can be modeled as a finite directed multi-graph $G(V, E)$, where the set of vertices V corresponds to the registers in the circuit and E presents the set of edges. The paths connecting two registers R_i and R_j are represented as a single edge e_{v_i,v_j}, $e \in E$, $v_i, v_j \in V$, and the

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

maximum and minimum delays of these paths are denoted as $T_{PDmax}(e_{v_i,v_j})$ and $T_{PDmin}(e_{v_i,v_j})$, respectively.

Let $T_{CD} : V \rightarrow \mathbb{R}$ assign a clock propagation delay to each vertex and let T_{CP} be the clock period. To ensure the right function of synchronized circuit, double-clocking and zero-clocking conditions must be prevented, which are also known as hold time and setup time violations, the following constraints must be satisfied respectively [6]:

$$T_{CD}(v_i) - T_{CD}(v_j) \geq -T_{PDmin}(e_{v_i,v_j}) \quad (1)$$

$$T_{CD}(v_i) - T_{CD}(v_j) \leq T_{CP} - T_{PDmax}(e_{v_i,v_j}) \quad (2)$$

Given T_{PDmax}, T_{PDmin} and T_{CD}, to satisfy (1) and (2) for each vertex, there is a lower bound for T_{CP}, which is denoted as T_{CPL}. By adjusting T_{CD}, CSS algorithms try to reduce T_{CPL}. However, such optimization is limited by the maximum mean cycle (MMC), also known as maximum average-delay cycle or the most critical cycle. It has been proved that even only considering setup time violations, for a given set of T_{PDmax}, the T_{CPL} can be no less than the average-delay of MMC [7]:

$$T_{CPL} \geq \max_{c_k \in C}[T_{AD}(c_k)] \quad (3)$$

where:

$$T_{AD}(c_k) = \frac{\sum_{e \in c_k} [T_{PDmax}(e)]}{|c_k|} \quad (4)$$

c_k is the set of all the edges in a cycle, which is a loop constructed by cascaded edges connected one after another. $T_{AD}(c_k)$ is the average delay of edges in c_k. C denotes the set of cycles in G. The number of the edges in c_k is represented by $|c_k|$.

To ease our presentation, we use the concept *slack balancing* to describe the process of CSS.

An edge's slack is defined as:

$$T_{slack}(e_{v_i,v_j}) = T_S - T_{PDmax}(e_{v_i,v_j}) - T_{CD}(v_i) + T_{CD}(v_j) \quad (5)$$

where T_S denotes the desired clock period in a circuit's specification. T_{CPL} is determined by the edge with minimum T_{slack}.

We also define the average slack of c_k as:

$$T_{AS}(c_k) = \frac{\sum_{e \in c_k} [T_{slack}(e)]}{|c_k|} \quad (6)$$

And we use the terminology *critical cycles* to indicate the cycles with negative T_{AS}.

According to the definitions (4), (5) and (6), we have:

$$T_{AS}(c_k) = T_S - T_{AD}(c_k) \quad (7)$$

From this equation, we know that the maximum average-delay cycle is also the minimum average-slack cycle. After CSS, the slacks of the edges in the MMC are all equal to $T_{AS}(MMC)$, that is, they are balanced.

3. THE PROPOSED FLOW

During synthesis and placement, the front-end tools make trade-offs between timing, power, area and congestion. Optimizations of timing-critical paths would cause penalty in power, area and congestion. Since resources are limited, increasing slacks on too many paths would not be feasible. As our experiments will show, in many cases, CSS can increase slacks on a significant proportion of the paths with setup violations. Thus front-end tools have less paths to optimize and can spend the limited resources on the "real" timing critical paths whose slacks can not be increased by CSS. In most cases, "less to optimize" leads to better results.

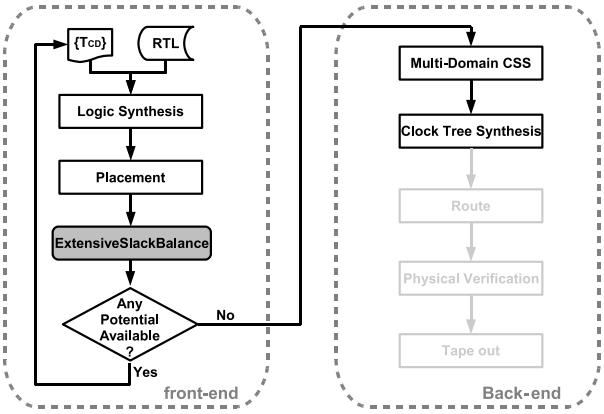

Figure 1: Design Flow

We try to make existing front-end tools interact with CSS to overcome the limitations mentioned above. Instead of integrating our algorithm into the executables of industrial tools, we integrate it into the design flow as a separate step that generates back-annotation constraints for re-optimization.

Fig. 1 shows how our algorithm is integrated into the industrial design flow. Starting with a RTL representation, under the zero-skew assumption, we use Synopsys PhysicalCompiler to generate a placed-netlist. Then our enhanced CSS algorithm *ExtensiveSlackBalance* is performed on the placed-netlist. The resulted $\{T_{CD}(v)\}$ (the set of clock arrival times to registers) is saved in a SDC file and then back-annotated to PhysicalCompiler as re-optimization constraints.

ExtensiveSlackBalance turn the poorly-optimized paths which obstructed CSS into critical paths. During the reoptimization, front-end tools will spend more efforts on them. Front-tools will also fix the hold time violations introduced by CSS, using techniques such as delay insertion, cell sizing and logic restructuring.

Because our CSS algorithm is designed for front-end tools, there are different requirements from the traditional algorithms designed for CTS tools. Firstly, it need not consider hold time violations. Such violations would be fixed by front-end tools in the consequent re-optimization. Secondly, it fixes as many setup time violations as possible. So the front-end tools can spend the limited resources on the "real" critical paths. Thirdly, it exposes to front-end tools the poorly-optimized long paths which would obstruct CSS optimization. If these paths are not exposed and then left untouched during the re-optimization iteration, they will surely obstruct the CSS again after re-optimization.

Although many traditional CSS algorithms can be easily modified to meet the first requirement, to the best of our knowledge, there is no algorithm that meets the other two. Many traditional CSS algorithms terminate when the MMC is balanced—they only fix setup time violations on the edges whose slacks are less than $T_{AS}(MMC)$ and only expose

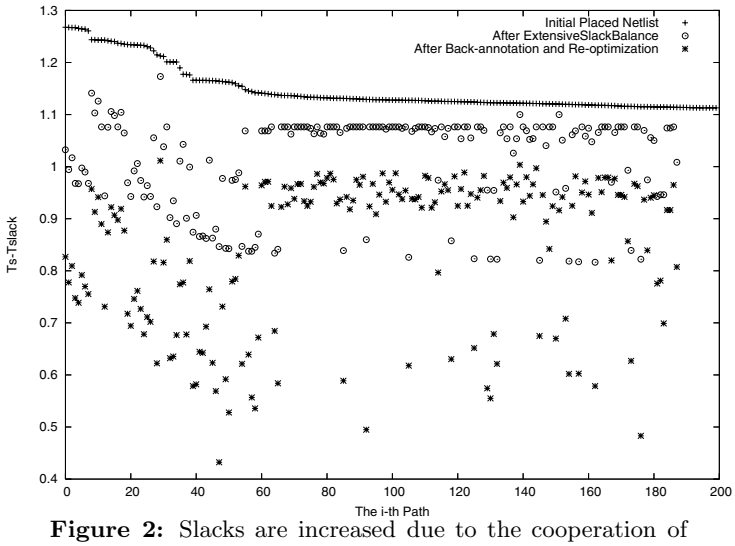

Figure 2: Slacks are increased due to the cooperation of our algorithm and front-end tools.

the poorly-optimized long paths in the MMC to front-end tools. Some algorithms go beyond this for the tolerance of process violation[1], but not adequately for re-optimization. So when generating back-annotation constraints, traditional CSS algorithms are inefficient.

Our algorithm meets the second and the third requirements by balancing all the circles whose average slacks are less than a certain value instead of the most critical circle. So we name it as *ExtensiveSlackBalance*.

Fig. 2, which is taken from the benchmark s38584 in our experiments, shows how the slacks of paths are increased in our flow ($T_S - T_{slack}$ decreased). We record the top 200 paths with the worst negative slacks in the initial placed netlist and then track the changes of their slacks during the subsequent two steps (*ExtensiveSlackBalance* and re-optimization). *ExtensiveSlackBalance* increases slacks on a wide range of paths and leaves only a small number of "real" critical paths to front-end tools. Since there are fewer paths to optimize, front-end tools can further increase the slacks on the remaining paths.

According to our experience, with back-annotation constraints generated by our algorithm, only up to two iterations of re-optimization are needed to make front-end tools and CSS tools realize the benefits of the interaction between them. After that, the placed-netlist will be handed over to the back-end.

Due to these two iterations, the run time of front-end tools is at least two times more than before. However, this overhead is ignorable in real design flows. Physical implementation usually starts during the final phase of verification, when only minor bug fixes would take place. Physical designers work out the floorplan and CSS-plan simultaneously with the verification team. After RTL freezen, both the floorplan and CSS-plan can be re-used directly and there would be no run time overhead.

When realizing our CSS-plan at back-end, we choose multi-domain clock skew scheduling presented by Ravindran[8] and the CTS algorithm based on delay target ordered merging[2].

1. *ExtensiveSlackBalance*($G(V,E)$, T_S)
3. MMC=*BalanceSlackOfMMC*(G);
2. **if** $T_{AS}(MMC) \geq 0$ **then** return;
4. G'=*IsolateMMC*(G, MMC);
3. *ExtensiveSlackBalance*(G', T_S);
4. G=*RestoreGraph*(G');
6. **end** *ExtensiveSlackBalance*

Figure 3: Pseudo code of *ExtensiveSlackBalance*

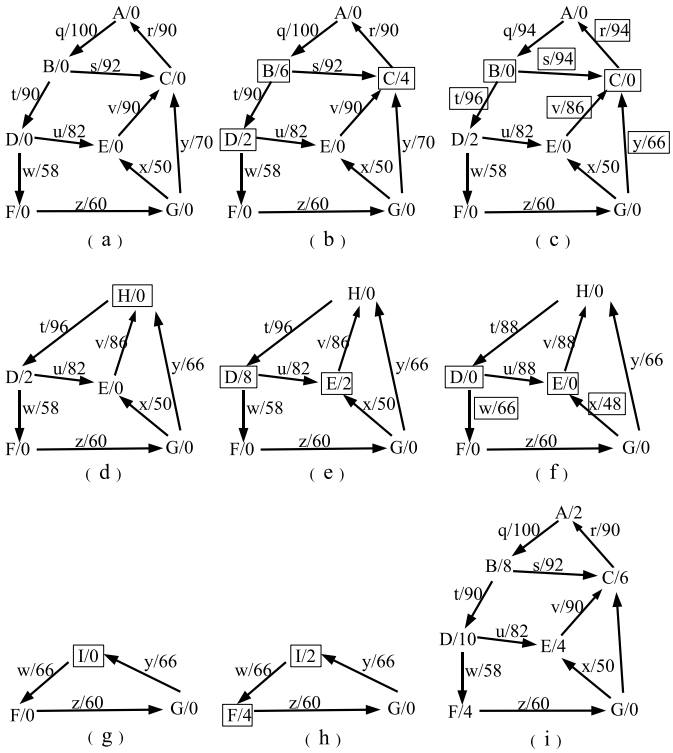

Figure 4: The process of *ExtensiveSlackBalance*. The changed parameters are marked with boxes. (a) the initial graph. (b) after *BalanceSlackOfMMC*, the MMC is A-B-C and $T_{CPL} \geq 94$ (c) after the parameter adjustment. (d) after topology adjustment, The MMC A-B-C is packed into a newly created vertex H. (e) after the second call of *BalanceSlackOfMMC*, the MMC is H-D-E and $T_{CPL} \geq 88$ (f) after the parameter adjustment. (g) after topology adjustment, The MMC H-D-E is packed into a newly created vertex I. Note that the edge x is dropped because its slack is bigger than y. (h) after the third call of *BalanceSlackOfMMC*, the MMC is I-F-G and $T_{CPL} \geq 64$. (i) The topology of the initial graph is restored.

4. THE ALGORITHM

The idea of our algorithm is simple: since traditional algorithms can balance the most critical circle, when we "isolate" the most critical one out of the graph, they will take the second as the most critical and balance it. If we recursively find and balance the maximum mean cycle and isolate it from the graph, all the circles with average slacks less than a certain value can be balanced. The outline of our algorithm is presented in Fig. 3.

BalanceSlackOfMMC implements a traditional CSS algorithm. Given a graph G, it finds and balances the current

Circuit Name	Number of Edges	Initial Phase				Iteration 1				Iteration 2				Improvement	
		P_0	P'_0	T_0	ΔS_0	P_1	P'_1	T_1	ΔS_1	P_2	P'_2	T_2	ΔS_2	P_0/P'_0	P_0/P'_2
s9234	1876	788	755	1.03	8.2	670	667	0.72	0.3	650	648	0.81	0.1	4.4%	21.6%
s13207	3195	1156	1040	1.68	11.1	875	870	1.33	3.1	804	803	1.46	1.8	11.2%	44.0%
s35932	6685	624	610	20.99	15.0	589	587	21.98	12.0	558	557	21.87	7.0	2.3%	12.0%
s15850	11777	1322	1267	2.75	3.4	1084	1070	2.34	0.7	1044	1022	1.96	1.4	4.3%	29.6%
s38584	14893	1268	1210	5.08	44.8	1030	1022	6.19	4.4	981	967	6.10	0.1	4.8%	31.1%
B21	38431	2101	1915	9.13	73.5	1698	1680	10.96	4.7	1677	1661	10.32	3.6	9.7%	26.5%
B22	60907	2103	1937	21.70	74.6	1811	1772	21.50	19.6	1736	1700	27.21	13.3	8.6%	23.7%
B17	143012	1904	1794	100.25	73.7	1730	1634	87.77	23.3	1623	1576	113.88	51.6	6.1%	20.8%

Table 1: Experiment Result

maximum mean cycle c_k of G. Then *IsolateMMC* performs parameter and topology adjustment to exclude the c_k from G and yields a new Graph G'. In the subsequent iterations, *ExtensiveSlackBalance* will find and balance the remaining critical cycles. When all the critical cycles are balanced, i.e., all the circles with negative T_{AS} are balanced, *RestoreGraph* will include back the MMCs to restore the original graph.

Fig. 4 shows the process of performing our algorithm on an example graph, where vertexes are denoted by uppercase letters and edges are denoted by lowercase letters. T_{CD} and T_{PD} are marked near the vertexes and edges, respectively. Parameter adjustment of *IsolateMMC* (Fig. 4(c) and Fig. 4(f)) clears the T_{CD} of the vertexes in MMC to zero and modify the T_{PD} of the connected edges to keep the slacks of these edges unchanged. Topology adjustment of *IsolateMMC* (Fig. 4(d) and Fig. 4(g)) packs the vertexes in the MMC into a newly created vertex such that subsequent *BalanceSlackOfMMC* will not see the edges of the MMC.

5. EXPERIMENTAL RESULTS

We selected five circuits (s9234, s13207, s35932, s15850 and s38584) from ISCAS'89 suite and three circuits (B21, B22 and B17) from ISCAS'99 suite to test the efficiency of our algorithm. The benchmarks are synthesized and placed by PhysicalCompiler2005.09, using TSMC generic 0.13um process. We implemented our algorithm in Java and measured the CPU run time on a 2.0GHz AMD opteron processor. Table 1 shows the detailed results.

In the initial phase, the RTL structural representations of the benchmarks are synthesized into placed-netlists with zero-skew assumption. In Table. 1, the clock period of the placed netlist is presented by P and after performing our algorithm, the clock period is P', both in picosecond. T shows the run-time of our algorithm in second and ΔS presents the total increase of all the T_{CD} by our algorithm in picosecond. For the placed netlists generated in the initial phase, as well as the first and second re-optimization iteration, the P, P', T and ΔS parameters are shown with the subscript 0, 1 and 2. For a consistent measure of the run time across all these three phases, we multiply the clock period with a fixed factor (0.8) to get T_S, that is, $T_{Si} = 0.8P_i,\ i \in \{0, 1, 2\}$.

Our algorithm is very effective in helping front-end tools perform better optimization after back-annotation. From P'_0 to P_1 and from P'_1 to P_2, front-end tools make considerable reductions, although CSS itself contributes less to timing improvements (from P_i to $P'_i,\ i \in \{0, 1, 2\}$). Note that without re-optimization iterations, our algorithm optimizes the timing as well as traditional CSS algorithms. So P_0/P'_0 also presents the timing improvements of the traditional algorithms.

6. CONCLUSIONS

The traditional industrial ASIC flows usually apply CSS as a post-placement technique and adopt zero-skew assumption at front-end. They commonly ignore the mutual influence between front-end tools and CSS, thus limit their optimization potential. We have proposed a design flow with back-annotation and re-optimization iterations, and also developed an enhanced CSS algorithm *ExtensiveSlackBalancing* to generate constraints for the back-annotation. In our CSS-feedback-resynthesis design flow, front-end tools can produce a better result when considering the constraints generated by *ExtensiveSlackBalancing*. Experimental results show that more improvements in clock frequency are achieved in our design flow than in traditional flow.

Acknowledgments

The authors would like to thank Xianfeng Li, Xiaomin Yang and Yuanrui Zhang for their support and help.

7. REFERENCES

[1] C.Albrecht, B.Korte, J.Schietke, and J.Vygen. Cycle time and slack optimization for vlsi-chips. *Proc. IEEE International Conference of Computer-Aided Design*, pages 232–238, 1999.

[2] R. Chaturvedi and J. Hu. A simple yet effective merging scheme for prescribed-skew clock routing. *Proceedings of the 21st International Conference on Computer Design*, pages 282–287, 2003.

[3] J. P. Fishburn. Clock skew optimization. *IEEE Transactions on Computers*, 39(7):945–951, July 1990.

[4] S.-H. Huang and Y. T. Nieh. Clock period minimization of non-zero clock skew circuits. *Proc. ICCAD*, pages 809–812, 2003.

[5] A. P. Hurst, P. Chong, and A. Kuehlmann. Physical placement driven by sequential timing analysis. *Proc. ICCAD*, pages 379–386, 2004.

[6] J. L. Neves and E. G. Friedman. Optimal clock skew scheduling tolerant to process variations. *Proceedings of the 33rd IEEE Design Automation Comference*, pages 623–628, 1996.

[7] M. C. Papaefthymiou. Understanding retiming through maximum average-delay cycles. *Proceedings of the 3rd ACM symposium on Parallel algorithms and architectures*, pages 65–84, 1994.

[8] K. Ravindran, A. Kuehlmann, and E. Sentovich. Multi-domain clock skew scheduling. *Proc. ICCAD*, pages 801–808, 2003.

[9] D. Velenis, K. T. Tang, I. S. Kourtev, V. Adler, F. Baez, and E. G. Friedman. Demonstration of speed enhancements on an industrial circuit through application of non-zero clock skew scheduling. *Proc. ICECS*, 2:1021–1025, Sept. 2001.