Lagrangian relaxation-based multi-threaded discrete gate sizer

by

Ankur Sharma

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Iowa State University

Ames, Iowa

2018

DEDICATION

I would like to dedicate this thesis to the Almighty whom we refer to in our culture as Krishna, and I hope and pray that the skills and knowledge that I have gained during this course of study, may it be used in some tangible way in the service of humanity.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS . . . vi

ABSTRACT . . . vii

CHAPTER 1. OVERVIEW . . . 1
1.1 Previous Work . . . 2
1.1.1 ISPD Gate Sizing Contests . . . 3
1.1.2 Major Drawbacks With Existing LR Gate Sizers . . . 4
1.2 Contributions . . . 5
1.3 Thesis Outline . . . 6

CHAPTER 2. FAST LAGRANGIAN RELAXATION BASED GATE SIZING USING MULTI-THREADING . . . 7
2.1 Introduction . . . 7
2.2 Problem Formulation . . . 10
2.3 Sequential Approach . . . 12
2.3.1 Initialization . . . 12
2.3.2 LDP solver . . . 12
2.3.3 Greedy post-pass . . . 14
2.4 Multi-threaded Approach . . . 15
2.4.1 Requirements for Multi-threading the LRS . . . 15
2.4.2 A Simple Approach - Clustering and Leveling . . . 16
2.4.3 Mutual Exclusion Edge Assignment - An Alternative to Clustering . . . 17
2.4.4 DAG Based Netlist Traversal - An Alternative to Leveling . . . 18
2.4.5 Modified Approach - An Alternative to the Simple Approach . . . 19
2.4.6 Parallelizing STA and LM update . . . 20
2.5 Enhancements in Sequential Approach . . . 21
2.5.1 Fast-OLR . . . 22
2.5.2 Early Exit Policy . . . 23
2.5.3 Fast-GTR . . . 24
2.6 Experimental Results . . . 26
2.6.1 Comparing Power and Performance Against Previous Works . . . 26
2.6.2 Comparison Against the Simple Approach . . . 27
2.6.3 Impact of Fast-GTR on the Overall Speedup . . . 28
2.6.4 Scalability Analysis . . . 28
2.7 Conclusion . . . 30
References . . . 30

CHAPTER 3. RAPID GATE SIZING WITH FEWER ITERATIONS OF LAGRANGIAN RELAXATION 33
3.1 Introduction 33
3.2 Problem Formulation 36
3.3 Overall Flow 38
3.4 LR Based Gate Sizing 41
3.4.1 Lagrange Multiplier Update 41
3.4.2 Timing Recovery 42
3.4.3 Power Recovery 44
3.4.4 Multi-Gate Sizing (MGS) 44
3.4.5 Critical Path Sizing (CPS) 46
3.5 Experimental Results 47
3.5.1 A Comparison With Previous Works 48
3.5.2 Factors Contributing to the Reduction in LR Iterations 49
3.5.3 Impact of CPS on Timing Convergence 53
3.6 Conclusion 53
References 53

CHAPTER 4. FAST LAGRANGIAN RELAXATION BASED MULTI-THREADED GATE SIZING USING SIMPLE TIMING CALIBRATIONS 55
4.1 Introduction 55
4.2 Previous Work 57
4.3 Problem Formulation 58
4.4 Background on Lagrangian Relaxation Based Gate Sizing 61
4.4.1 Initialization 61
4.4.2 Solve LDP 62
4.4.3 Greedy post-pass 64
4.5 Overall Flow 64
4.6 Timing Models With Resistive Interconnect 66
4.6.1 Modeling Effective Capacitance 68
4.6.2 Modeling Slew of an RC Tree 69
4.6.3 Modeling Delay of an RC Tree 70
4.6.4 Modeling Gate Slew 71
4.6.5 Modeling Gate Delay 71
4.6.6 Calibration Pseudo Code and Runtime 72
4.7 Fast LRS Solver 74
4.8 Lagrange Multiplier Update 75
4.9 Greedy Post-Pass 75
4.9.1 Greedy Timing Recovery 76
4.9.2 Greedy Power Recovery 76
4.10 Experimental Results 77
4.10.1 Results on the ISPD 2012 Designs 77
4.10.2 Results on the ISPD 2013 Designs 81
4.10.3 Breakdown of Results for the ISPD 2013 Designs 82
4.10.4 Impact of Relaxing the Target Clock Period 84

4.10.5 Effectiveness of Calibration 85
4.10.6 Multi-Threading Different Sub-blocks of the Gate Sizer 86
4.11 Conclusion 87
References 87

CHAPTER 5. LAGRANGIAN RELAXATION BASED GATE SIZING WITH CLOCK SKEW SCHEDULING - A FAST AND EFFECTIVE APPROACH 90
5.1 Introduction 90
5.2 Previous Work 93
5.3 Problem Formulation 94
5.4 Min-Cost Network Flow Modeling 98
5.5 EGSS: Effective Gate Sizing With Skew Scheduling 102
5.5.1 Solving LRS_{λ} 102
5.5.2 Skew Update 103
5.5.3 Modified Lagrange Multiplier Update 104
5.6 NetFlo Vs EGSS: Limitations of Optimizing Primal Via Dual Maximization 104
5.7 Greedy Refinements in Timing and Power 106
5.8 Experiments and Results 107
5.9 Conclusion 109
References 110

CHAPTER 6. CONCLUSION AND FUTURE DIRECTIONS 113

REFERENCES 115

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Sri Krishna, the Supreme Personality of Godhead, for He is the intelligence of the intelligent [Bhagavad Gita 7.10] and the ability in living beings [Bhagavad Gita 7.8], and above everything, one thing that appeals the most to me is that He is the dearest well-wishing friend of everyone irrespective of who you are [Bhagavad Gita 5.29]. He is so big-hearted and broad-minded.

I would like to express my heartfelt gratitude to my advisor Dr. Chu for his expert guidance, support and encouragement through out my program of study. I cherish both his constructive criticism and honest appreciations. I would also like to thank my committee members - Dr. Degang Chen, Dr. Akhilesh Tyagi, Dr. Phillip Jones, Dr. Sarah Ryan for taking out their precious time to serve on my POS.

My heartfelt gratitude to Dr. Ganesh Ramakrishnan - my mentor at my undergraduate alma mater Indian Institute of Technology Bombay, for seeing potential in me and inspiring me to take on my graduate journey.

I would like to take this opportunity to sincerely thank my elder brother Ashish for being with me as a friend and advisor through thick and thin. Earlier his advices would often not appeal to me. But over the period of time, I started to see wisdom in his words. As I look back, all I see is his self-less support and encouraging assurance - I feel fortunate to have him. Next I would like to offer my deepest respects and love for my parents. Their continuous shower of blessings and love always felt like cooling shade of large trees under scorching heat of the sun. Trees suffer all the hard-ships and give comfort to those who are under their shade. I would like to thank my wife, Nidhi, who joined me in the last one year of my studies. I must confess she was instrumental in getting me finish my studies fast :-) Jokes apart, thank you Nidhi for your support while you felt homesick and put through my busy schedules.

I thank my lab mates - Lin Tao, Yixiao Ding and Gang Wu - in all of whom I appreciated the quality of focus. During this course of study, I made several friends whom I fondly remember and would like to mention their names here for they became an integral part of my journey and I feel gratitude for them. They are Anand Jagannath, Ravikiran, Ishana, Ankit, Pratik, Akshit, Rakesh, Abhinav, Rishi, Pranav, Paavan.

ABSTRACT

In integrated circuit design gate sizing is one of the key optimization techniques which is repeatedly invoked to trade-off delays for area and/or power of the gates during logic design and physical design stages. With increasing design sizes of a million gates and larger, discrete gate sizes and non-convex delay models the gate sizing algorithms that were designed for continuous sizes and convex delay models are slow and timing inaccurate.

Of the several published discrete gate sizing algorithms, recent works have shown that Lagrangian relaxation based gate sizers have produced designs with the lowest power on average with high timing accuracy. But they are also very slow due to a large number of expensive timing updates spread across hundreds of iterations of solving the Lagrangian sub-problem.

In this thesis we present a Lagrangian relaxation based multi-threaded discrete gate sizer for fast timing and power reduction by swapping the gate sizes and the threshold voltages. We developed two parallelization enabling techniques to reduce the runtime of Lagrangian sub-problem solver, namely, mutual exclusion edge (MEE) assignment and directed acyclic graph (DAG) based netlist traversal. MEEs are dummy edges assigned to reduce computational dependencies among gates sharing one or more common fan-ins. DAG based netlist traversal facilitates simultaneous resizing of gates belonging to different topological levels.

We designed a Lagrange multiplier update framework that enables rapid convergence of the timing recovery and power recovery algorithms. To reduce the runtime of timing updates, we proposed a simple and fast-to-compute effective capacitance model and several mechanisms to calibrate the timing models to improve their accuracy. Compared to the state-of-the-art gate sizer, our proposed gate sizer is on average 15x faster and the optimized designs have only 1.7% higher power.

In digital synchronous designs simultaneous gate sizing and clock skew scheduling provides significantly more power saving. We extend the gate sizer to simultaneously schedule the clock skew. It can achieve an average of 18.8% more reduction in power with only 20% increase in the runtime.

CHAPTER 1. OVERVIEW

Power consumption of integrated circuits has increased substantially with much larger circuits integrated on a single chip with shrinking technology dimensions. Circuit performance is now limited by power due to higher power densities and device limits. Reducing power consumption is a high priority for circuit designers to allow higher performance, to reduce cooling and packaging costs, and to extend battery life in mobile devices. With increasing design sizes of a million gates and larger, optimization tools must be very fast while not sacrificing the quality of results.

In VLSI physical design, gate sizing is one of the most frequently used circuit optimizations. Each logic gate has several possible implementations in terms of size and threshold voltage (Vt) of cell alternatives in a standard cell library. Different implementations (cells) trade off typically power and/or area for delay. The task of a gate-sizer is to choose a suitable cell for every gate to minimize the objective cost while meeting the design timing constraints.

An example of discrete gate sizing is shown in Figure 1.1. The figure shows a standard cell library consisting of two types of logic gates (identified by their shapes). Each gate type has four different cell alternatives - two sizes (1x and 2x) times two Vt (high Vt shown in blue color and low Vt shown in red color). Compared to the smaller size cells, larger size cells have smaller delay through them but leak more power and have larger area. On the other hand, compared to higher Vt cells, lower Vt cells have smaller delay and leak more power. The figure also shows an example original netlist (used interchangeably with

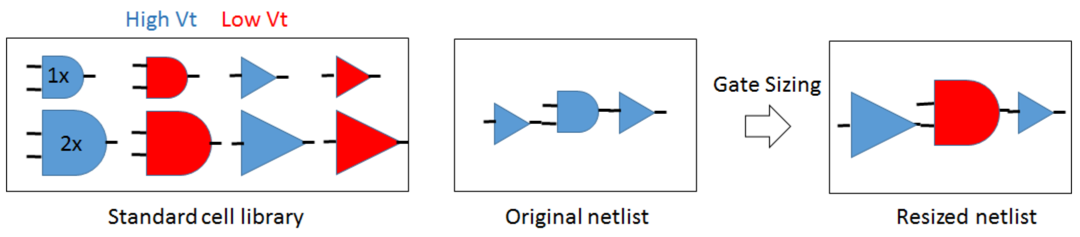

Figure 1.1: A pictorial depiction of discrete gate sizing.

the term *design*) which is composed of three gates of which two gates are of the same type. The task of the gate sizer is to select a suitable cell alternative for each gate type so that, say, total power of the design is minimized while satisfying the delay constraint that the total delay of the slowest path in the design (there is only one path in this example) is less than a pre-determined delay budget.

Apart from the huge number of gates in a design, discreteness of the cell alternatives and non-convexity of the delay models have made the gate sizing problem quite difficult. Industry has mostly been using the table lookup based non-linear delay models that do not have nice properties like convexity, but are fairly accurate. Therefore, a high quality gate sizer must be able to effectively handle the discreteness and the non-convexity. Of the several different techniques that researchers have been using to attack the gate sizing problem like dynamic programming, convex programming and network flow, to name a few, Lagrangian relaxation (LR) based gate sizers have produced high quality design solutions in a competitive runtime compared to other kinds of gate sizer.

1.1 Previous Work

Gate sizing has been studied for several decades. As the constraints changed and newer timing models emerged, researchers applied different techniques. One standard technique is greedy iterative sensitivity-based heuristics Fishburn (1985); Hu *et al.* (2012); Kahng *et al.* (2013). These heuristics greedily resize the gate that promise maximum benefit in terms of some sensitivity score. If the solution space is convex and has no local minimum, then greedy heuristics can yield the optimal solution, but they can be very slow. In the presence of local minima, such heuristics can easily get stuck. For global optimization, gate sizing problem has been formulated as a linear programming problem Nguyen *et al.* (2003); Chinnery *et al.* (2005), or, a more general, convex programming Kasamsetty *et al.* (2000); Roy *et al.* (2007), problem. Using standard optimization techniques, these techniques can converge to the optimal or a near-optimal solution. However, when cell alternatives are discrete, then optimality cannot be guaranteed and the approach is too slow. Even continuous sizing followed by mapping to the discrete space is not that effective, especially if the discrete gate sizes are of coarse granularity Hu *et al.* (2007).

With non-convex delay models, like the table lookup based models, timing accuracy is often significantly compromised. For example, researchers often assume simplified delay models like the Elmore delay model Chen *et. al.* (1999), or an input-slew independent delay-model Nguyen *et al.* (2003). Alternatively, they approximate with a convex delay model Roy *et al.* (2007), such as a posynomial function Kasamsetty *et al.* (2000). Such delay models can be quite inaccurate versus lookup tables, as reported in Chinnery *et al.* (2005); Rahman *et al.* (2013).

With discrete sizes, the gate sizing problem can be formulated as a combinatorial optimization problem which has been shown an NP-hard problem Ning (1994). Several works have applied dynamic programming (DP) Hu *et al.* (2007); Liu *et al.* (2010, 2011), but those strategies while optimal for tree topologies, cannot optimize well on graphs with reconvergent paths.

Another global optimization technique which has been empirically shown to be quite effective in the discrete domain with non-convex models is based on Lagrangian relaxation. The earliest work from Chen *et. al.* (1999) proved convergence for a continuous and convex problem. The authors proposed to solve it iteratively by alternating between solving an LR subproblem and updating the Lagrange multipliers using a sub-gradient method. Since then several works have been published that improved the LR subproblem solver and/or strategy to update the Lagrange multipliers. While Tennakoon *et al.* (2002); Huang *et al.* (2011); Wang *et al.* (2009) proposed strategies to update Lagrange multipliers that were quite different from the sub-gradient approach; the LR subproblem was formulated as a graph problem in Ozdal *et al.* (2011). Reimann *et al.* (2016) proposed a strategy to initialize Lagrange multipliers for a pre-optimized design.

1.1.1 ISPD Gate Sizing Contests

In International Symposium on Physical Design (ISPD) 2012 and 2013, Intel organized discrete gate sizing contests. Organizers provided a dummy discrete standard cell library containing 30 cell alternatives (10 sizes times 3 Vt) for each gate type; a suite of 7 designs with total gate count ranging from few hundreds of gates to an order of a million gates. The contest used Synopsys PrimeTime - *de facto* industry standard sign-off timer for timing verification. A public platform for comparing different gate-sizing approaches under more realistic constraints greatly advanced the state-of-the-art in gate sizing. We also use these contest benchmark suites to evaluate our gate sizer.

Since the contests, several works have been published that used the contest framework to benchmark their gate sizers Li *et al.* (2012); Hu *et al.* (2012); Reimann *et al.* (2013); Kahng *et al.* (2013); Livramento *et al.* (2014); Flach *et al.* (2014); Yella *et al.* (2017); Daboul *et al.* (2018). While authors of Kahng *et al.* (2013); Flach *et al.* (2014) employed sensitivity guided greedy meta-heuristic, gate sizers from the authors of Li *et al.* (2012); Livramento *et al.* (2014); Flach *et al.* (2014); Yella *et al.* (2017) solved an LR formulation of the gate sizing problem. The state-of-the-art gate sizer of Flach *et al.* (2014) demonstrated minimum leakage power on most of the designs as well as on average with competitive runtime. Daboul *et al.* (2018) recently published their gate sizer where they formulated the gate sizing problem as a resource sharing problem. They used the IBM's EinsTimer for timing verification instead of the Synopsys PrimeTime which was used in the contests. They reported shorter runtime using several tens of threads and slightly more leakage power compared to Flach *et al.* (2014). However, timing was not closed when checking their final optimized netlists in PrimeTime.

1.1.2 Major Drawbacks With Existing LR Gate Sizers

Although LR gate sizers have yielded designs with the least power and their runtime is lesser than the other gate sizing approaches, in absolute terms they are still quite slow especially on the larger designs. For an instance, the state-of-the-art gate sizer Flach *et al.* (2014) spends more than 6.5 hours to optimize designs with a million gates. Considering that gate sizing needs to be invoked several times in the design cycle, the design process would greatly benefit if gate sizers are much faster while achieving the similar power. LR gate sizers in general have following major drawbacks:

- LR gate sizers require hundreds of iterations to converge to a good quality of solution. Each iteration necessitates a huge number of timing updates which can significantly slow down the sizer, especially when interconnects are resistive. With resistive interconnects, accurate timing updates are quite CPU intensive.

- The typical heuristics applied to solve the discrete LR subproblem are parallelizable, but there has been no systematic effort to utilize the parallelism.
- The commonly used Lagrange multiplier update strategies are all heuristics with one or more parameters to tune. Although the multiplier update is very crucial to convergence rate and quality of solution, it is not clear how to tune the parameters.
- Since discrete LR gate sizing is done in a heuristic manner, often a post-pass is necessary for finer-grained timing and power reduction. The post-pass strategies proposed in the existing LR gate sizing literature are very slow. In some cases, the post-pass can account for the majority of the runtime.

1.2 Contributions

In this thesis we develop a multi-threaded discrete gate sizer using the LR formulation for fast timing and power reduction without loosing the solution quality. In the first half of this thesis, we assume that the interconnects are purely capacitive. In the second half, we extend our gate sizer to optimize designs with resistive interconnects. We further extend it to simultaneously schedule the clock skew which significantly reduces the power with a small runtime penalty. Our gate sizer has been benchmarked using the ISPD 2012 and the ISPD 2013 gate sizing contest suites. Our major contributions are listed below:

- We develop two parallelization enabling techniques for a multi-threaded LR subproblem solver. Combined with other enhancements to speed up the sequential portion of the code, our multi-threaded gate sizer could achieve 5.4x speedup with 8 threads without compromising the solution quality. Our gate sizer's runtime with single thread is on average similar to the state-of-the-art gate sizer runtime.
- We design a tunable and effective Lagrange multiplier update framework. We analyze the impact of different tuning parameters on the convergence rate and the final solution quality. We further propose two strategies for finer-grained timing and power reduction. They allow early termination of the runtime expensive iterations and the final solution quality at the end actually improves (power reduces). Faster convergence due to our proposed multiplier update framework along with early termination

cuts down the number of iterations by 5 times, and the total runtime reduces by 2.5x compared to our multi-threaded gate sizer using 8 threads.

- To realize similar speedups in runtime on designs with resistive interconnect, we propose a new gate sizing flow based on simple though inaccurate timing models. In addition to developing a new, fast-to-compute, closed form expression for modeling effective capacitance, we propose several calibration mechanisms to improve the accuracy of the timing models. Compared to the state-of-the-art gate sizer Flach *et al.* (2014), our gate sizer is, on average, more than 15x faster and leakage power is only 2.5% higher.
- We extend our LR gate sizer to simultaneously schedule the clock skew. For that, we incorporate skew into the LR subproblem and propose a simple skew update strategy. With our modified flow to simultaneously size the gates and update the skews, our tool reduces 19.7% more power, on average.

1.3 Thesis Outline

The rest of the thesis is organized as follows. In Chapter 2, we present our proposed multi-threaded gate sizer. In Chapter 3, we discuss our tunable Lagrange multiplier update framework. In Chapter 4, we extend our gate sizer to optimize designs with resistive interconnect. In Chapter 5, we discuss simultaneous gate sizing and clock skew scheduling problem. Lastly, we conclude in Chapter 6.

CHAPTER 2. FAST LAGRANGIAN RELAXATION BASED GATE SIZING USING MULTI-THREADING

Published in International Conference on Computer Aided Design 2015

Abstract - We propose techniques to achieve very fast multi-threaded gate-sizing and threshold-voltage swap for leakage power minimization. We focus on multi-threading Lagrangian Relaxation (LR) based gate sizing which has shown both better power savings and better runtime compared to other gate sizing approaches. Our techniques, mutual exclusion edge assignment and directed graph-based netlist traversal, maximize thread execution efficiency to take full advantage of the inherent parallelism when solving the LR subproblem, without compromising the leakage power savings.

With 8 threads, our multi-threading techniques achieve on average 5.23x speedup versus our single-threaded (sequential) implementation. This compares well to the maximum achievable speedup of 5.93x by Amdahl's law due to 5% of the execution not being parallelizable. To highlight the problems with load imbalance and poor scheduling, we also propose a simpler approach based on clustering and topological level-by-level netlist traversal, which can achieve only 3.55x speedup.

We also propose three simple yet effective enhancements - fast optimal local resizing, early exit policy, and fast greedy timing recovery - to speed up single-threaded LR-based gate-sizing without degrading the leakage power. We test our gate sizer using the ISPD 2012 gate sizing contest benchmarks and guidelines. Compared to other researchers' state-of-the-art LR-based gate sizer, our approach is 1.03x (with 1-thread) and 5.40x (with 8-threads) faster and only 2.2% worse in leakage power.

2.1 Introduction

Power consumption of integrated circuits has increased substantially with much larger circuits integrated on a single chip with shrinking technology dimensions. Circuit performance is now limited by power due to higher power densities and device limits. Reducing power consumption is a high priority for circuit designers to allow higher performance, to reduce cooling and packaging costs, and to extend battery life in mobile devices.

In VLSI physical design, gate sizing is one of the most frequently used circuit optimizations. Each logic gate has several possible implementations in terms of size and threshold voltage (Vth) of cell alternatives in a standard cell library. Different implementations (cells) trade off area or power for delay. The task of a gate-sizer is to choose a suitable cell for every gate to minimize power while meeting the design timing constraints. With increasing design sizes of a million gates and larger, optimization tools must be very fast while not sacrificing the quality of results.

The discrete gate-sizing problem is NP-hard Ning (1994). Also, cell delays do not vary in a convex manner with area or power, because for example internal capacitances vary with cell layout due to multiple transistor fingers or fins to implement greater drive strengths. This makes the gate-sizing problem very difficult to solve optimally. Researchers have applied various techniques such as greedy iterative sensitivity-based heuristics Fishburn (1985); Hu *et al.* (2012), linear programming Nguyen *et al.* (2003); Chinnery *et al.* (2005), convex programming Kasamsetty *et al.* (2000) Roy *et al.* (2007), Lagrangian Relaxation (LR) Chen *et. al.* (1999); Tennakoon *et al.* (2002); Wang *et al.* (2009); Ozdal *et al.* (2011); Huang *et al.* (2011); Li *et al.* (2012); Livramento *et al.* (2013); Flach *et al.* (2014), network flow Sundararajan et al. (2002) Ren *et al.* (2008), dynamic programming (DP) Hu *et al.* (2007); Liu *et al.* (2010, 2011), and logical effort Rahman *et al.* (2013) Reimann *et al.* (2013).

Two major drawbacks of most of these works are inaccurate delay models and the assumption that the gate sizes are continuous. In academia, researchers often assume simplified delay models like the Elmore delay model Chen *et. al.* (1999), or an input-slew independent delay-model Nguyen *et al.* (2003). Alternatively, they approximate with a convex delay model Roy *et al.* (2007), such as a posynomial function Kasamsetty *et al.* (2000). Such delay models can be quite inaccurate versus SPICE models or library lookup tables, as reported in Chinnery *et al.* (2005), Rahman *et al.* (2013). Whereas, industry has mostly been using the table-lookup based non-linear delay models that do not have nice properties like convexity, but are fairly accurate. Continuous sizing followed by mapping to the discrete space may not be able to satisfy the timing constraints especially if the discrete gate sizes are of coarse granularity in size Hu *et al.* (2007). Moreover, in most of the previous works, the benchmark suites for evaluation either contained only small designs or were proprietary, making a fair comparison difficult.

Some of these concerns were addressed with the ISPD 2012 Ozdal *et al.* (2012) and ISPD 2013 Discrete Gate Sizing Contests Ozdal *et al.* (2013), which provided a common platform for fairly comparing different gate-sizing approaches. The objective is to minimize leakage power while meeting the timing constraints. Since then, several works have utilized the contest framework to compare their gate sizers Li *et al.* (2012); Hu *et al.* (2012); Livramento *et al.* (2013); Reimann *et al.* (2013); Flach *et al.* (2014). We observe that some of the fastest approaches with competitive solution quality[1] use LR as the main technique Li *et al.* (2012); Livramento *et al.* (2013); Flach *et al.* (2014). LR achieves excellent results as it provides a global optimization avoiding local minima, and Karush-Kuhn-Tucker (KKT) optimality conditions Chen *et. al.* (1999) greatly prune the search space.

Significant further speedups in LR based gate-sizers can be achieved by smartly multi-threading different blocks of the LR framework. Liu *et al.* (2011) use a GPU to accelerate their DP-based gate-sizer. DP-based gate sizing is optimal for tree topologies but it is known to exhibit suboptimal behavior due to path reconvergence. Li *et al.* (2012) very briefly talk about multi-threading their iterative LR framework. Their multi-threading strategy in each iteration is to simultaneously resize the gates that are either in the same topological level or three levels apart. (We shall refer to topological levels as levels in the rest of the chapter.) While three level separation might avoid inaccuracies in slew and capacitance computation, on the downside, only one-third of the gates are resized in each iteration which results in slow convergence and/or higher leakage. We propose techniques that consider all the gates in each iteration without compromising accuracy, and yet achieve high thread utilization. Compared to Li *et al.* (2012), on ISPD2012 contest benchmarks, our sizer is 7.8x faster and, on average, saves 12% more leakage power.

In this chapter, we focus on developing techniques that enable efficient multi-threading to realize a very fast gate sizer. With 8 threads, our multi-threading techniques achieve on average 5.23x speedup versus our sequential implementation, without degrading the leakage power. This compares well to the maximum achievable speedup of 5.93x by Amdahl's law due to only 5% of the execution not being parallelizable.

[1]Up until now, Flach *et al.* (2014) presents the best results for both leakage and runtime for all of the ISPD 2012 gate-sizing benchmarks.

To highlight the problems with load imbalance and poor scheduling, we also propose a simpler set of approaches based on clustering and topological level-by-level netlist traversal. Our major contributions are summarized below:

- We propose mutual-exclusion edge (MEE) assignment to avoid inaccuracies in capacitance computation. In contrast with clustering, MEE greatly improves the load balancing.
- We use directed acyclic graph (DAG) netlist traversal (DNT) to systematically propagate the slew. In contrast with leveling, DNT does not require threads to synchronize at each level.
- We provide three enhancements to the sequential approach: a fast optimal local resizing (Fast-OLR) and an early exit policy to speedup the parallelized sequential runtime, and a fast greedy timing recovery (Fast-GTR) to reduce the non-parallelized sequential runtime. Due to Fast-GTR we are able to reduce the unparallelized sequential runtime to a mere 5%.

This chapter is organized as follows. In Section 2.2, we formulate the problem. Section 2.3 presents our sequential LR approach. There we describe the overall flow of our gate-sizer. In Section 2.4, we present MEE assignment and DAG-based netlist traversal. Section 2.5 details our enhancements to reduce sequential runtime. We discuss the experimental results in Section 2.6 and conclude in Section 2.7.

2.2 Problem Formulation

Following the ISPD 2012 contest guidelines, we assume that 1) only combinational gates can be resized, whereas sequential gates have a fixed size, and 2) a lumped capacitance model is used for modeling interconnect capacitance. Consequently, all nodes of a net (one driver and one or more sinks) share the same timing information (slew, arrival and required times).

Throughout this work, T is the clock period; node i is the driver node of the net i; $i \to j$ denotes the timing arc from node i to node j; a_i and q_i are the actual and required arrival times (AAT and RAT) at node i, respectively; and $d_{i\to j}$ is the delay of the timing arc $i \to j$.

The objective is to choose suitable cells for every gate so that the total leakage power (sum of leakage powers of individual gates) is minimized under three types of timing constraints: 1) worst path delay $\leq T$;

2) maximum output capacitance $\leq maxcap$; and 3) maximum output slew $\leq maxslew$. This minimization is over all the available cells for every gate in the library. In our approach, we guarantee to satisfy the second and the third constraints throughout, so we do not formalize them mathematically below. We do model rise and fall timing constraints separately, but we have omitted them here for clear presentation. The original problem is formulated as follows:

$$\begin{aligned} \underset{cell,a}{\text{minimize}} \quad & \sum_{gates} leakage \\ \text{subject to} \quad & a_i + d_{i\to j} \leq a_j, \text{ for each timing arc } i \to j \\ & a_k \leq T \text{ for each primary output } k \end{aligned} \tag{2.1}$$

We apply the Lagrangian Relaxation technique to Equation (2.1) to include all the constraints into the objective. To avoid constraint violations, a positive penalty term is introduced per constraint, called the Lagrangian multiplier (LM, λ). This gives the Lagrangian Relaxation Subproblem (LRS):

$$\begin{aligned} \underset{cell,a}{\text{minimize}} \quad & \sum_{gates} leakage + \sum_{i\to j} \lambda_{i\to j}\left(a_i + d_{i\to j} - a_j\right) \\ & + \sum_{k} \lambda_k \left(a_k - T\right) \end{aligned} \tag{2.2}$$

By applying KKT conditions Chen *et. al.* (1999), the LRS can be simplified to

$$\underset{cell}{\text{minimize}} \quad \sum_{gates} leakage + \sum_{i\to j} \lambda_{i\to j} d_{i\to j} \tag{2.3}$$

The Lagrangian Dual Problem (LDP) is shown in (2.4).

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \quad & \left(\underset{cell}{\text{minimize}} \sum_{gates} leakage + \sum_{i\to j} \lambda_{i\to j} d_{i\to j} \right) \\ \text{subject to} \quad & \sum_{u\in fanin(i)} \lambda_{u\to i} = \sum_{v\in fanout(i)} \lambda_{i\to v}, \text{ for each node } i \end{aligned} \tag{2.4}$$

The term $\sum \lambda_{i\to j} d_{i\to j}$ is referred to as the *lambda-delay* Flach *et al.* (2014).

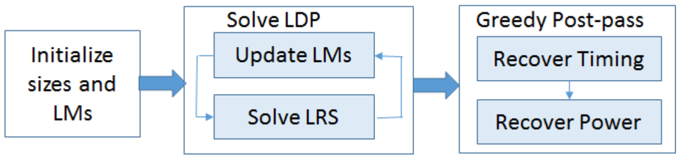

Figure 2.1: Gate-sizing algorithm flowchart

2.3 Sequential Approach

In this section we present a brief overview of our sequential gate-sizer. As shown in Figure 2.1, the algorithm has three stages: the initialization; the LDP solver; and the final greedy stage.

2.3.1 Initialization

In the initialization stage, each gate is assigned its minimum leakage library cell. However, that might violate all three types of timing constraints. Gates are minimally upsized to satisfy the maximum load capacitance and maximum slew constraints. This is done by traversing the netlist in reverse topological order and upsizing the gates that have capacitance violations; followed by a forward topological traversal to upsize the gates to satisfy the slew constraints, where necessary. The LMs are all initialized to 1.

2.3.2 LDP solver

This is an iterative stage to approximately solve (2.4). In each iteration, the LM update and the LRS solver alternate to update the LMs for the given cells and update the cells for the given LMs, respectively.

2.3.2.1 LM update

The LDP solver begins with the static timing analysis (STA) and updating of the LMs according to the criticality of the corresponding timing arc. We developed our own static timer and verified its accuracy against Synopsys PrimeTime, as per the contest guidelines. For updating the LMs, a common strategy is to increase the LMs in proportion to their timing criticality Tennakoon *et al.* (2002). We use an exponential factor *cexp* to emphasize the LMs (see the pseudo code in Figure 2.2). Although the use of *cexp* was originally presented in Flach *et al.* (2014), details were not provided therein on how to update it, except

```
Algorithm: LM-update
// WPD: Worst path delay
$cexp = 1$
if $(WPD > r * T)$    // $r = 1.01$
    $cexp \mathrel{*}= \frac{WPD}{T}$
else
    $cexp \mathrel{*}= \left(1 + k * \frac{WPD - r*T}{r*T}\right)$  // $k = 10$
endif
foreach timing arc $i \rightarrow j$
    $\lambda_{i\rightarrow j} \mathrel{*}= \left(1 + \frac{a_j - q_j}{T}\right)^{cexp}$
endfor
KKT projection
```

Figure 2.2: Pseudo code for LM update

for a few hints. In our experience, the update method for *cexp* is crucial to the final solution quality, so we explicitly show it in the pseudo code. As long as the design violates the relaxed delay target ($r * T$), we increase *cexp*. By the time the design satisfies the relaxed delay target, *cexp* is usually very large because we started out with the minimum power solution instead of a minimum delay solution. Therefore, in order to accelerate the power recovery, we rapidly reduce *cexp* by using the factor k. KKT projection ensures that LMs satisfy the KKT conditions Tennakoon *et al.* (2002). Both STA and LM update are highly parallelizable sub-blocks (discussed in Section 2.4), though they contribute only a small fraction of the total runtime.

2.3.2.2 LRS solver

The LRS solver approximately solves (2.3) by optimal local resizing (OLR) of one gate at a time, assuming all the other gates are fixed. Gates are traversed in forward topological order from the primary inputs (PIs) to the primary outputs (POs), i.e., OLR of a gate begins after all its fanin gates have been processed Livramento *et al.* (2013). Such an order can be precomputed and stored. To optimally resize a gate, the *lambda-delay-cost* is computed for all the valid cells[2] of that gate and the cell with the lowest cost (*lambda-delay-cost* + *leakage*) is chosen. The *lambda-delay-cost* approximates *lambda-delay*. To exactly compute *lambda-delay*, an incremental timing analysis is needed which becomes prohibitively expensive when done for several cells of every gate. To limit the runtime overhead when analyzing alternate cells, timing of only the local arcs is recomputed Li *et al.* (2012). To improve the accuracy, Flach *et al.* (2014)

[2]A cell is invalid if it causes capacitance or slew constraint violations.

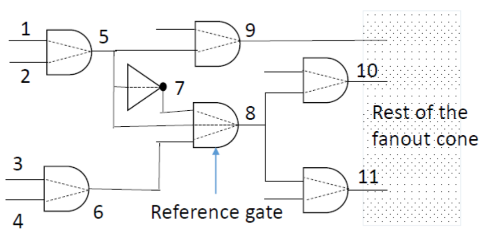

Figure 2.3: Local arcs include fanin-arcs: 1-5, 2-5, 5-7, 3-6, 4-6; gate-arcs: 7-8, 5-8, 6-8; fanout-arcs: 8-10, 8-11; side-arcs:5-9. Drain-nets: 10, 11; side-nets: 9.

proposed global delay/slew sensitivity functions to approximate the change in *lambda-delay* for the rest of the fanout cone (the second term in eq.(2.5)). Therefore, the *lambda-delay-cost* for a cell c can be computed as follows:

$$\textit{lambda-delay-cost(c)} = \sum_{i \to j \in local-arcs(c)} \lambda_{i \to j} d_{i \to j} + \sum_{n \in drain-nets(c) \cup side-nets(c)} \triangle D_n^\lambda \tag{2.5}$$

where $\triangle D_n^\lambda$ is the lambda-delay change in the net n due to the change in output slew of the gate driving the net Flach *et al.* (2014). Local-arcs constitute fanin-arcs, gate-arcs, side-arcs and fanout-arcs, as shown in Figure 2.3 along with the drain-nets and the side-nets. The LRS solver is the most expensive sub-block. It also offers great parallelism since multiple gates can be processed simultaneously.

2.3.3 Greedy post-pass

The last stage combines two greedy heuristics: greedy timing recovery (GTR), and greedy power recovery (GPR). The least power solution obtained from LDP is the starting point for the greedy post-pass. If the starting solution satisfies the timing, GTR is skipped then GPR tries to squeeze out as much power as possible by greedily downsizing the most sensitive gates without violating any constraint Hu *et al.* (2012). GTR very effectively complements LR, as GTR allows the final solution at the end of LDP to have small timing violations. Owing to its global view, LR is not very efficient in recovering timing exactly. To resolve the remaining timing violations, LR ends up expending more power than a more localized greedy approach like GTR. The GTR algorithm will be discussed in Section 2.5.3. Both GPR and GTR lack parallelism, hence they are left sequential in this work.

2.4 Multi-threaded Approach

We propose techniques to parallelize the following sub-blocks: LRS solver, STA, and LM update. Parallelization of the LRS is our key focus since its the most runtime expensive of all, so we discuss it first. We shall refer to multi-threaded LRS as MT-LRS.

2.4.1 Requirements for Multi-threading the LRS

As mentioned above, LRS involves OLR of all the gates and several gates can be processed simultaneously. We shall use OLR and "processing" interchangeably. Simultaneous OLR of two or more gates requires all of them to satisfy two properties that we have identified as follows:

Property 1 None of them should have a fanin in common. Otherwise, when two or more gates sharing a fanin are being resized, some of them might witness an unexpected change in the fanin's load while they are in the middle of OLR. Even worse, the fanin *maxcap* constraint might be violated if gates commit their new sizes simultaneously. This would require one or more gates to redo OLR, which can become very expensive and therefore should be avoided.

Property 2 None of them should lie in the fanout cone of a gate undergoing OLR. Otherwise, the gate in the fanout cone might be using the stale values for input slew. This can be easily avoided by following a forward topological order as used in the sequential approach (see Section 2.3.2.2). However, unlike the sequential execution, precomputing an order with multiple threads is undesirable because processing time of the gates is not known *a priori*. Therefore, instead of a precomputed order, a dynamic structure like a queue of *ready* gates whose fanins have already been resized needs to be maintained.

To ensure both of the above properties, we first propose a simple approach, followed by the MEE assignment and the DNT.

```
Algorithm: MT-LRS via Clustering and Leveling
foreach level $l$
    $readyQ = cluster(l)$
    // Each thread executes in parallel
    while (1)
        # critical section
            if ($readyQ.empty()$) break
            $z = readyQ.next()$
        # endcritical
        $process\_cluster(z)$
    endwhile
    $barrier()$
endfor
```

Figure 2.4: Pseudo code for MT-LRS by the simple approach.

2.4.2 A Simple Approach - Clustering and Leveling

To ensure the first property, we propose to cluster all the gates that share fanins and process them by a single thread. If gate A shares a fanin with gate B, and B shares a fanin with gate C, then all three gates should be clustered, even if A and C do not share any fanin. Note that the clusters are always disjoint. The advantage of clustering is that it is simple to implement and the clusters can be precomputed, but the disadvantage is that it can cause heavy load imbalancing if the cluster sizes are highly non-uniform.

To ensure the second property, we propose to group all the clusters by their topological level (PIs being at level 0) and process one level at a time. We refer to this as *leveling*. If a cluster spans multiple levels, it can be safely broken down into that many sub-clusters. When all the clusters at a level have been processed, all the clusters in the next level at once become ready. The advantage of leveling is that it does not require any book-keeping to determine when a cluster becomes ready, whereas the disadvantage is that threads need to wait for every other thread to finish before moving onto the next level.

Figure 2.4 shows the pseudo code for MT-LRS with this approach. At each level, a queue (*readyQ*) of ready clusters is initialized with the clusters at that level. Then, each thread enters a *critical section* to retrieve a cluster from the *readyQ*. In a critical section a mutually exclusive (mutex) lock is used to ensure that no other thread writes or reads the data that is being updated. Other threads wanting to read/write that data stall until the first thread exits the critical section. Threads process their respective clusters and re-enter

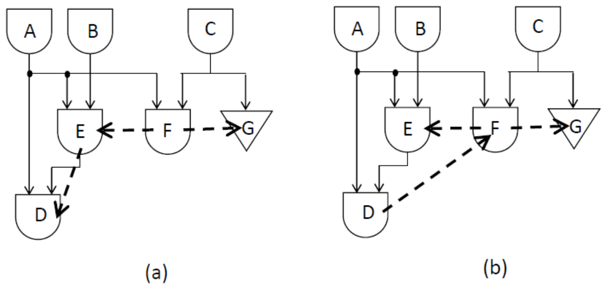

Figure 2.5: (a) One possible MEE assignment: Fanouts of A - {D,E,F} are chained by MEEs (dashed lines). Similarly, fanouts of C - {F,G} are chained. With MEEs, the fanouts of A will be processed in the following order: F then E then D. (b) An incorrect MEE assignment as it forms the cycle D-F-E. Edges D-F and F-E are MEEs, and the edge E-D is a netlist edge - due to D being a fanout of E.

the critical section. When the *readyQ* is empty, threads exit and wait at the *barrier* for the other threads to finish. In multi-threaded programming, barrier is used to synchronize all the threads at that point.

2.4.3 Mutual Exclusion Edge Assignment - An Alternative to Clustering

As noted above, clustering can cause heavy load imbalancing due to non-uniform cluster sizes. If we want to avoid clustering, we need an alternative mechanism to ensure the first property. We propose to *chain* the fanouts of every gate by additional edges, referred to as mutual exclusion edges (MEE), thereby ensuring that the fanouts of the same gate are not processed simultaneously. An example is shown in the Figure 2.5(a).

If the fanouts are arbitrarily chained then either cycles (Figure **??**(b)) or very long chains might get created. While cycles would lead to deadlock, long chains would adversely affect the performance.

We propose a randomized algorithm for MEE assignment to avoid cycles and probabilistically reduce the maximum chain length. Its pseudo code is shown in Figure 2.6. It consists of two stages: (1) assignment of random IDs to each gate, and (2) assignment of MEEs. Random IDs are assigned such that the gates at the higher topological level have larger IDs. Then in the second stage, for every gate, its fanouts are sorted in ascending order of their IDs and an MEE is assigned between every consecutive pair of sorted fanouts - from the lower ID fanout to the higher ID fanout. Thus, the fanouts are chained. MEEs are assigned once and it has linear time complexity. In reference to the pseudo code, $s[i]$ is referred to as a *pseudofanin* of

```
Algorithm: MEE Assignment
// Assign random ID
$maxID = 0$
for level $l = 0$ $to$ $maxlevel$
    $x = maxID$
    foreach gate $g$ in level $l$
        $g.ID = x$
        while ($g.ID$ is not unique)
            $g.ID = x + rand()$
        endwhile
        if ($g.ID > maxID$)
            $maxID = g.ID$
        endif
    endfor
endfor
// Assign MEE
foreach gate $g$
    $s$ = sort fanouts of $g$ by ascending IDs
    for $i = 0$ $to$ $(s.size() - 2)$
        $MEE(s[i] \rightarrow s[i+1])$
    endfor
endfor
```

Figure 2.6: Pseudo code for MEE assignment.

$s[i+1]$ and $s[i+1]$ is a *pseudofanout* of $s[i]$. In our scheme of MEE assignment, this strategy guarantees cycle-free assignment.

If during the ID assignment stage we do not ensure larger IDs for gates at higher levels (PIs are at level 0), then an MEE may get assigned from a higher to a lower level fanout. This may create cycles involving the netlist edges that are always from the lower to the higher levels.

While MEE can achieve much better load balancing than clustering, it creates additional edges which means more precedence constraints that limit the parallelism. We empirically show that the thread idling time is actually very small for larger designs.

2.4.4 DAG Based Netlist Traversal - An Alternative to Leveling

Though leveling is a simple idea, it has two disadvantages: 1) a barrier at the end of each level causes thread idling, and 2) within a level, parallelism is limited by MEEs. The repercussions of barrier are more visible with the clustering where loads can be highly imbalanced. With an increasing number of threads, this barrier and the limited parallelism worsen the thread utilization. An alternative approach to leveling is

```
Algorithm: init_precedence_count (gate g)
$g.preCount = |g.fanins| + |g.pseudofanins|$

Algorithm: identify_ready_gates (gate g)
$readyG = \emptyset$        // Ready gates
foreach $x \in (g.fanout \cup g.pseudofanout)$
    $x.preCount = x.preCount - 1$
    if ($x.preCount == 0$)
        $readyG.push\_back(x)$
    endif
endfor
return $readyG$
```

Figure 2.7: Pseudo codes for two book-keeping functions of DNT.

DAG based netlist traversal (DNT). By some book-keeping, we can track the gates as they become ready, and keep pushing them into the *readyQ*. As long as the *readyQ* is non-empty, threads need not wait.

Pseudo code for two book-keeping functions needed to implement the DNT are shown in Figure 2.7. The first function, *init_precedence_count(gate g)*, initializes the precedence count (*preCount*) of the gate *g*. *preCount* is the number of predecessors which can be either fanins or *pseudofanins*. The second function, *identify_ready_gates(gate g)*, is invoked when the gate *g* has been processed. It decrements the *preCount* of each one of its fanouts as well as *pseudofanouts*. Those fanouts or pseudofanouts whose *preCount* reaches zero are returned as ready gates.

The disadvantage of DNT over leveling is that it requires book-keeping to track the ready gates. This needs to be done inside a critical section because multiple threads might want to simultaneously read/write the *preCount* of the same gate. However, a critical section is in any case required to update the *readyQ*. Therefore, by merging the two critical sections, we can optimize away the additional thread idling.

Next, we present a better approach for MT-LRS with MEE assignment and DNT.

2.4.5 Modified Approach - An Alternative to the Simple Approach

This modified approach replaces clustering and leveling by MEE assignment and DNT, respectively. Like clustering, the MEE assignment is also required only once in the beginning. On the other hand, the DNT must perform book-keeping every time MT-LRS is invoked. Figure 2.8 shows the pseudo code of MT-LRS via the modified approach. Firstly, the *preCount* of each gate is initialized. The gates with zero

```
Algorithm: MT-LRS via MEE Assignment and DNT
foreach gate g
    init_precendence_count(g)
endfor
init_readyQ()
// Each thread executes in parallel
# critical section
    if (all_gates_done()) return
    g = readyQ.next()
# endcritical
while (1)
    process_gate(g)
    # critical section
        x = identify_ready_gates(g)
        update_readyQ(x)
        if (all_gates_done()) return
        g = readyQ.next()
    # endcritical
endwhile
```

Figure 2.8: Pseudo code for the MT-LRS via the modified approach.

preCount are the ready gates and they form the *readyQ*. Then, each thread retrieves a ready gate (more gates can also be retrieved) from the *readyQ* and processes it. Processed gates identify new ready gates, if any, and update the *readyQ*. If all the gates have been processed, the thread exits, otherwise it retrieves the next ready gate and processes it. Note that unlike in the leveling, an empty *readyQ* does not imply all the gates are processed, rather it means that some threads are still working and they might soon generate new ready gates.

2.4.6 Parallelizing STA and LM update

Parallelizing the STA and the LM update is much more straightforward than parallelizing the LRS. In STA, gate timings are updated in the forward topological order; whereas in LM update, gate LMs are updated in the reverse topological order. We apply the leveling idea for topological traversal. Note that the clustering is not needed in either STA or LM update, because gates are not being resized. Therefore, at each topological level, there is total freedom to update any number of gates simultaneously and in any order. We form groups of ten gates at each level and feed them to the threads whenever threads become available. When all the gates at a level are updated, we go to the next level. Note that the STA and the LM update do not involve much computation and the time spent updating a group of gates is more or less the same across

```
1   Algorithm: Fast-OLR
2   $candidates = \emptyset$
3   $currw$ = current width
4   for current Vth, one Vth higher and one Vth lower
5       $c = cell(currw, Vth)$
6       $bestcost = cost(c)$
7       $bestcell = c$
8       for $w = currw: maxw$
9           $c = cell(w, Vth)$
10          Ensure $c$ is valid
11          if $cost(c) < bestcost$
12              $bestcost = cost(c)$
13              $bestcell = c$
14          else break
15      endfor
16      $candidates.insert(bestcell)$
17      // Repeat lines 5-16 and
18      // replace $maxw$ by $minw$
19  endfor
20  Apply $\min_{c \in candidates} cost(c)$ such that
21  local slack does not worsen
```

Figure 2.9: Pseudo-code for Fast-OLR.

all the groups. Therefore, an explicit barrier at the end of each level does not degrade the thread utilization much.

2.5 Enhancements in Sequential Approach

To complement the multi-threading performance improvements, the sequential approach should be free from sub-optimality as far as possible. Although the recent LR-based gate sizers have been demonstrated to be the fastest on the ISPD 2012 benchmarks, at several places in the general strategy we observe sub-optimal performance. For example: during the OLR of a gate, Li *et al.* (2012); Livramento *et al.* (2013); Flach *et al.* (2014) suggest evaluating all the cells, but this may not be necessary in all the iterations. While Livramento *et al.* (2013) executes a fixed number of LDP iterations, Flach *et al.* (2014) does not define convergence criteria for when LDP can be terminated. To address these issues and more, we propose the following three enhancements.

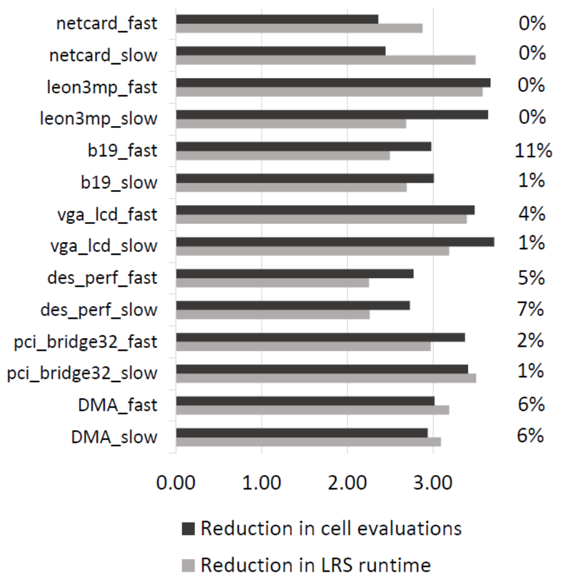

Figure 2.10: The bar chart shows the reductions in the cell evaluations and the LRS runtime due to Fast-OLR. The numbers on the right of each bar denote the power savings after the LDP stage, relative to OLR.

2.5.1 Fast-OLR

During the OLR, a gate is resized to the cell that has the lowest cost (*lambda-delay-cost* + *leakage*). To determine such a cell, Li *et al.* (2012); Livramento *et al.* (2013); Flach *et al.* (2014) suggest evaluating all the valid alternatives. However in practice, we observed that the cost function almost always has a single local minimum considering a fixed Vth and varying sizes. Therefore, by searching for the optimal cell locally, we can reach the globally optimal cell most of the time. Although this may be an artifact of the ISPD 2012 contest library, there is another advantage of local searching which is library independent. Local searching induces incremental changes to the current solution, whereas jumping to the globally optimal cell may significantly perturb the current solution. Large perturbations during the last few iterations tend to destabilize the solution, preventing convergence and degrading the results.

Figure 2.9 shows the pseudo code of our proposed algorithm, Fast-OLR. A cell is characterized by its size (or width) and Vth. Since we do local searching, we restrict our search to the current Vth, the next higher Vth and the next lower Vth cells. For each of the three Vth, we iterate over the cells with increasing sizes. We continue as long as the cost is reducing, and store the least cost cell as a suitable candidate. The

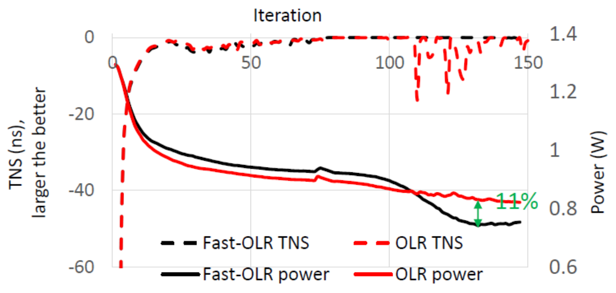

Figure 2.11: Comparing TNS and power profiles due to Fast-OLR and OLR, on b19_fast. After 100 iterations, TNS destabilizes due to OLR. Consequently, OLR cannot focus on power recovery, and ends up with 11% higher power.

same procedure is repeated for the decreasing sizes. At the end, the least cost candidate that does not worsen the local slack much is applied.

Before choosing the least cost cell, Flach *et al.* (2014) suggested computing the change in the slack of the driver and the sink nets because the locally optimal cell, whether found by local searching or otherwise, might significantly worsen the TNS. We apply this check as we recover power, after the design timing is within 1% of the target delay.

In Figure 2.10, we compare the Fast-OLR against OLR for the following three metrics: the number of cell evaluations, the LRS runtime, and the power after LDP. Results are from single-threaded execution. On average, the cell evaluations reduce by 3.3x and the LRS runtime reduces by 3.0x. We also observe an average 3% reduction in the power due to the better solution stability offered by local searching, as discussed above. In Figure 2.11, we demonstrate destabilization of TNS due to OLR after 100 iterations for the b19_fast benchmark. If the local slack worsens significantly that further adds to the instability.

2.5.2 Early Exit Policy

When the timing constraint has almost been met, the early-exit policy determines if it is likely that power will reduce in future iterations or not. It can be used as a rule of thumb to terminate the LDP iterations before the maximum number of iterations are reached. *The LDP solver can be terminated if neither the average power, nor the minimum power solution found thus far, improve during two consecutive sets of iterations.*

The early exit policy is derived based on the following empirical observations: power averaged over a few iterations (5, in this work) reduces before stabilizing; in general, power is not a monotonic function of the number of iterations; and, in some cases, power may oscillate around a value. If the power is oscillating, we may never observe power degradation for two consecutive sets of iterations. So there must be a reduction in the minimum power found thus far to justify continuing. Our experiments showed that some designs (large, as well as small) terminate LDP after 35 iterations, whereas some required up to 160 iterations. The benchmarks required on average 95 iterations to converge, with a standard deviation of 49.

2.5.3 Fast-GTR

The GTR applied in the last phase of the optimization serves to fix small timing violations without expending much power. At this stage timing degradation is not allowed. Although different researchers refer to it by different names like Slack Legalization Hu *et al.* (2012) and Timing Recovery Flach *et al.* (2014), the basic algorithm is the same. It's an iterative algorithm: in each iteration, the critical gates are sorted by some criterion like slack, or change in the delay after upsizing; then the most critical gate is upsized; followed by an incremental STA. If the TNS degrades, the change is undone. GTR terminates when all timing violations have been fixed.

The sub-optimality here is that the incremental STA can be a significant waste of CPU cycles if the TNS degrades. One of the solutions is to have a metric that can predict if the timing is going to degrade. Hu *et al.* (2012) developed one such metric for their approach. However, it may generate false negatives, thereby causing a wasteful incremental STA nonetheless. We propose a very simple heuristic, referred to as Fast-GTR. It is based on the empirical observation that the gates that fail to improve the timing in the current iteration, are unlikely to improve the timing in the future iterations as well, unless one of its side gates (a gate that shares a fanin) is successfully upsized. Therefore, we simply skip such gates until then.

In general, Fast-GTR improves the parallelized fraction of the total runtime by speeding up the greedy post-pass stage which directly benefits the multi-threaded speedup (discussed in Section 2.6.3). In particular, we observed that Timing Recovery Flach *et al.* (2014) can consume up to 50% of the total runtime on the benchmarks like des_perf, whereas Fast-GTR consumes less than 5% of runtime without degrading the results or causing any timing constraint violations.

Table 2.1: Comparing quality and performance of our single-threaded (FF1) and 8-threaded (FF8) against results in Li *et al.* (2012) - referred as [1] in the table below, and Flach *et al.* (2014) - referred as [2].

Benchmark	**Total**	**Leakage Power (W)**						**Total Runtime (min)**					
	Cells	**[1]**	**[2]**	**FF1**	**FF8**	**FF1/[2]**	**FF8/FF1**	**[1]**	**[2]**	**FF1**	**FF8**	**[2]/FF1**	**FF1/FF8**
DMA_slow	25301	0.153	0.132	0.136	0.135	1.027	0.995	0.60	0.79	0.64	0.15	1.23	4.28
DMA_fast	25301	0.281	0.238	0.250	0.250	1.050	0.999	0.60	0.92	1.48	0.34	0.62	4.37
pci_bridge32_slow	33203	0.111	0.096	0.099	0.100	1.036	1.003	1.20	0.87	1.78	0.34	0.49	5.29
pci_bridge32_fast	33203	0.167	0.136	0.143	0.143	1.053	1.000	1.20	0.92	1.95	0.37	0.47	5.21
des_perf_slow	111229	0.671	0.570	0.597	0.595	1.048	0.996	6.00	25.31	1.79	0.51	14.11	3.53
des_perf_fast	111229	1.930	1.395	1.457	1.457	1.045	1.000	6.60	16.37	5.82	1.38	2.81	4.23
vga_lcd_slow	164891	0.375	0.328	0.330	0.330	1.007	1.000	7.80	5.67	9.08	1.61	0.62	5.64
vga_lcd_fast	164891	0.460	0.413	0.432	0.432	1.046	1.000	10.20	8.37	11.70	1.94	0.72	6.04
b19_slow	219268	0.604	0.564	0.568	0.569	1.007	1.001	10.20	9.15	20.55	3.45	0.45	5.95
b19_fast	219268	0.784	0.717	0.734	0.734	1.024	0.999	12.00	11.75	21.88	3.60	0.54	6.08
leon3mp_slow	649191	1.400	1.334	1.333	1.334	0.999	1.000	43.80	38.98	26.75	4.53	1.46	5.91
leon3mp_fast	649191	1.640	1.443	1.445	1.442	1.002	0.998	54.60	46.62	33.55	5.94	1.39	5.65
netcard_slow	958780	1.780	1.763	1.763	1.763	1.000	1.000	48.00	34.39	37.07	5.86	0.93	6.32
netcard_fast	958780	2.180	1.841	1.849	1.850	1.004	1.001	88.80	47.41	41.82	7.38	1.13	5.67
Average		**0.895**	**0.784**	**0.796**	**0.795**	**1.025**	**0.999**	20.83	17.68	15.42	2.67	1.03†	5.23†

†Geometric mean. The geometric mean has been used to compare speedups due to the wider range in values.

2.6 Experimental Results

Our gate-sizer is implemented in C++. Experiments are performed on a server with two 2.67GHz Intel(R) Xeon(R) X5650 CPUs. Each CPU has six cores and each core has two hyperthreads. Aggregate memory is 48GB. For multi-threading, OpenMP Dagum *et al.* (1998) is used. We use ISPD 2012 gate-sizing contest benchmarks to test our sizer. All of our reported results are averaged over 5 runs and all the final optimized designs satisfy the timing constraints.

We compare our results against works of the other researchers, namely Flach *et al.* (2014) and Li *et al.* (2012). Flach *et al.* (2014) used a single thread on a faster machine, a 3.40GHz Intel(R) Core(TM) i7-3770 CPU. Li *et al.* (2012) employed 8 threads on a server like ours with two 2.67GHz CPUs with 6 cores and 72GB memory. Since we do not have access to their source code, results are cited from their respective works.

In this section we shall discuss results pertaining to different types of executions. Their nomenclature is defined as follows: the modified approach + Fast-GTR is referred to as Fast-Fast (FF); the modified approach + Timing Recovery Flach *et al.* (2014) is Fast-Slow (FS); and, the simple approach + Fast-GTR is Slow-Fast (SF). When x threads are used, they are respectively referred to as FFx, FSx and SFx. All of them are equipped with the Fast-OLR (we switch from OLR to Fast-OLR at the fifth iteration) as well as the Early Exit policy.

2.6.1 Comparing Power and Performance Against Previous Works

Referring to Table 2.1, we first compare FF1 against Flach *et al.* (2014). FF1 is at par with Flach *et al.* (2014), averaging 3% faster and 2.5% higher leakage power. On benchmarks with more than 500K gates, FF1 is 23% faster, primarily due to the early exit policy. Consequently, FF1 provides an excellent baseline to showcase the multi-threaded speedup that can be achieved. Compared to Flach *et al.* (2014), FF8 is on average 5.40x faster (multiply the last two columns) with only 2.2% higher power. Runtime improvement is likely to be even more than 5.40x as Flach *et al.* (2014) use a faster machine.

Compared to Li *et al.* (2012), which also used 8 threads, FF8 spends 7.80x lesser runtime ($= 20.83/2.67$) to execute all the benchmarks and saves 12% more power. With FF8 we demonstrate an average speedup

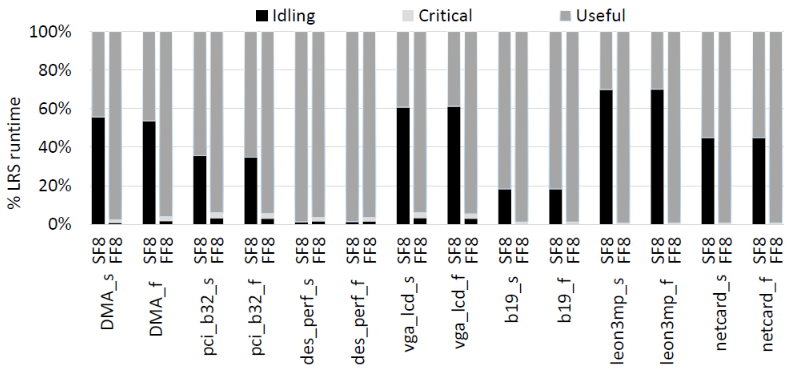

Figure 2.12: Comparing the MT-LRS runtime breakdown for the modified approach (FF8) and the simple approach (SF8).

of 5.23x over FF1 without degrading results. In comparison, Li *et al.* (2012) reported an overall speedup of only 2.2x with 8 threads, mainly because 40% of their algorithm's runtime is unparallelized.

2.6.2 Comparison Against the Simple Approach

To compare the modified (FF8) and the simple approach (SF8), we analyze the runtime of MT-LRS for both of them in Figure 2.12. A thread executing LRS would either be idling, executing a critical section, or doing useful work, i.e., resizing. For SF8, we observed that on 10 out of 14 benchmarks, threads could be idling for $>$35% of the LRS runtime. This is mainly the waiting time at the barrier caused by heavy load imbalancing, which is caused by the clustering approach. The worst cluster sizes at a given level can have up to 46% of the gates at that level. On des_perf benchmark where SF8's idling time is only 1%, the worst cluster sizes were no bigger than 1%. In the worst case for FF8, threads idle for only $<$ 3% of the LRS runtime.

The time spent in the critical section by SF8 threads is negligible. On the other hand, FF8 threads can expend up to 3% of the LRS runtime in the critical section. This is due to book-keeping to track the ready gates and updating of the *readyQ*. Across all the benchmarks, the average thread utilization for FF8 threads is 97%, and for SF8 threads it is only 59%. As a result, the FF8 MT-LRS is on average 1.73x faster than SF8 MT-LRS.

Table 2.2: Overall runtime speedup with Fast-GTR (FF8) versus Timing Recovery (FS8). Speedups are roughly correlated with the parallelized runtime fractions. Major improvements are highlighted in bold.

Benchmark	Speedup		Parallel Fraction	
	FS1/FS8	FF1/FF8	FS1	FF1
DMA_slow	4.62	4.28	0.93	0.91
DMA_fast	4.86	4.37	0.96	0.96
pci_bridge32_slow	5.72	5.29	0.98	0.97
pci_bridge32_fast	5.50	5.21	0.97	0.97
des_perf_slow	**1.83**	**3.53**	**0.27**	**0.90**
des_perf_fast	**2.38**	**4.23**	**0.65**	**0.94**
vga_lcd_slow	5.83	5.64	0.98	0.97
vga_lcd_fast	5.86	6.04	0.97	0.97
b19_slow	5.82	5.95	0.98	0.98
b19_fast	6.17	6.08	0.98	0.98
leon3mp_slow	5.82	5.91	0.92	0.94
leon3mp_fast	5.09	5.65	**0.88**	**0.95**
netcard_slow	6.43	6.32	0.97	0.97
netcard_fast	4.40	5.67	0.90	0.88
Geometric Mean	**4.77**	**5.23**	**0.84**	**0.95**

2.6.3 Impact of Fast-GTR on the Overall Speedup

In Table 2.2 we compare the speedup in the total runtime achieved by FF8 (with Fast-GTR) and FS8 (without Fast-GTR) with respect to their corresponding single-threaded versions. On average, FF8 is 5.23x faster than FF1, whereas FS8 is 4.77x faster than FS1. FF is faster due to the improvement in the parallel fraction of the sequential runtime, from 0.84 for FS8 to 0.95 for FF8. On des_perf and leon3mp, Fast-GTR significantly reduces the time spent in the greedy post-pass as shown in the same table. We observed larger speedup with FF in netcard_fast despite slightly lower parallel fraction. This is supposedly due to the runtime noise caused by server loading.

2.6.4 Scalability Analysis

In this subsection, we analyze how the speedup scales as the number of threads grow and what factors contribute to the loss of the speedup. In Table 2.3, we show the speedups obtained from FF2, FF4, FF8, FF12, FF14 and FF16 with respect to FF1.

Table 2.3: Speedup for different threads with respect to FF1.

Benchmark	FF1	FF2	FF4	FF8	FF12	FF14	FF16
DMA_s	1.00	1.71	2.77	4.28	5.36	4.62	4.40
DMA_f	1.00	1.64	2.75	4.37	5.89	5.52	4.86
pci_bridge32_s	1.00	1.95	3.13	5.29	6.18	6.00	5.56
pci_bridge32_f	1.00	1.96	3.19	5.21	6.58	6.16	5.63
des_perf_s	1.00	1.57	2.37	3.53	4.15	4.39	3.93
des_perf_f	1.00	1.66	2.73	4.23	3.06	4.90	4.72
vga_lcd_s	1.00	1.88	3.10	5.64	7.85	7.48	6.75
vga_lcd_f	1.00	1.76	2.89	6.04	6.25	6.12	6.63
b19_s	1.00	1.83	3.11	5.95	7.57	7.63	7.65
b19_f	1.00	1.66	3.23	6.08	7.96	7.75	7.69
leon3mp_s	1.00	1.70	3.14	5.91	7.75	7.46	7.34
leon3mp_f	1.00	1.85	2.86	5.65	7.03	7.18	7.08
netcard_s	1.00	1.95	2.97	6.32	7.97	8.13	8.05
netcard_f	1.00	1.97	2.86	5.67	5.21	6.08	6.37
Geom Mean	**1.00**	**1.79**	**2.93**	**5.23**	**6.14**	**6.27**	**6.04**

We see a performance saturation from 12 to 14 threads, and slight performance degradation from 14 to 16 threads. The average speed-up with 16 threads is 6.04x which is significantly smaller than the theoretical upper bound of 9x predicted by Amdahl's law. There are two primary factors for this gap: 1) limitations of the hardware architecture, and 2) increase in overhead. The server has 2 CPUs each with 6 cores with 2 hyper threads. The two hyperthreads per core do not provide much additional speedup due to hardware resource contention between threads Valles (2009). As a result speedup begins saturating as threads exceed the physical cores. Our experiments on a synthetic completely parallel code showed that we could only achieve a 12x speed-up using 16-threads. In other words, we effectively have only 12 threads. Considering this, we would expect the upper-bound of the achievable speed-up on our multi-threaded application to be 7.74x, which is closer to our achieved result. With an increase in the number of threads, 1) thread idling increases, and 2) critical section overhead increases. Additionally, we suspect runtime overhead due to increased communication to keep memories synchronized.

2.7 Conclusion

Today's designs with millions of gates require very fast gate-sizing and threshold voltage assignment, as it is a crucial circuit optimization that is performed at multiple steps in the design flow. We have shown the effectiveness of our proposed techniques to speed up multi-threading of Lagrangian relaxation-based gate sizing; mutual exclusion edge assignment and DAG-based netlist traversal help achieve 97% thread utilization with 8 threads. In contrast, the simpler multi-threading strategies - clustering and topological level-based traversal, allow only 59% thread utilization. To complement our multi-threading techniques, we also propose fast optimal local resizing, early-exit policy, and fast greedy timing recovery, all of which are simple yet highly performance effective enhancements to the sequential LR-based gate sizing approach. We combine all these to realize a very fast and high quality gate sizer. Compared to the state-of-the-art (both in runtime as well as power) algorithm Flach *et al.* (2014), our gate sizer using 8 threads is 5.40x faster and has only 2.2% higher power, without any timing constraint or other violations, on the ISPD 2012 Disrete Gate Sizing Contest benchmarks.

References

Chen, C.-P., Chu, C., and Wong, D. (1999). Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(7):1014–1025.

Chinnery, D. and Keutzer, K. (2005). Linear programming for sizing, Vth and Vdd assignment. In *Proceedings of the 2005 international symposium on Low power electronics and design*, pages 149–154. ACM.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Fishburn, J. (1985). TILOS: A posynomial programming approach to transistor sizing. In *Proc. of IEEE International Conference of Computer-Aided Design, Nov. 1985.*

Flach *et. al.*, G. (2014). Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 33(4):546–557.

Hu, S., Ketkar, M., and Hu, J. (2007). Gate sizing for cell library-based designs. In *Proceedings of the 44th annual Design Automation Conference*, pages 847–852. ACM.

Hu *et al.*, J. (2012). Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 233–239. IEEE.

Huang, Y.-L., Hu, J., and Shi, W. (2011). Lagrangian relaxation for gate implementation selection. In *Proceedings of the 2011 international symposium on Physical design*, pages 167–174. ACM.

Kasamsetty, K., Ketkar, M., and Sapatnekar, S. (2000). A new class of convex functions for delay modeling and its application to the transistor sizing problem [CMOS gates]. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 19(7):779–788.

Li *et al.*, L. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on*, pages 226–232. IEEE.

Liu, Y. and Hu, J. (2010). A new algorithm for simultaneous gate sizing and threshold voltage assignment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 29(2):223–234.

Liu, Y. and Hu, J. (2011). GPU-based parallelization for fast circuit optimization. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 16(3):24.

Livramento *et al.*, V. (2013). Fast and efficient lagrangian relaxation-based discrete gate sizing. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1855–1860. EDA Consortium.

Nguyen *et al.*, D. (2003). Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization. In *Proceedings of the 2003 international symposium on Low power electronics and design*, pages 158–163. ACM.

Ning, W. (1994). Strongly NP-hard discrete gate-sizing problems. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(8):1045–1051.

Ozdal *et al.*, M. (2011). Gate sizing and device technology selection algorithms for high-performance industrial designs. In *Proceedings of the International Conference on Computer-Aided Design*, pages 724–731. IEEE Press.

Ozdal *et al.*, M. (2012). The ISPD-2012 discrete cell sizing contest and benchmark suite. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 161–164. ACM.

Ozdal *et al.*, M. (2013). An improved benchmark suite for the ISPD-2013 discrete cell sizing contest. In *Proceedings of the 2013 ACM international symposium on International symposium on physical design*, pages 168–170. ACM.

Rahman, M., Tennakoon, H., and Sechen, C. (2013). Library-Based Cell-Size Selection Using Extended Logical Effort. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(7):1086–1099.

Reimann *et al.*, T. (2013). Simultaneous gate sizing and vt assignment using fanin/fanout ratio and simulated annealing. In *Circuits and Systems (ISCAS), 2013 IEEE International Symposium on*, pages 2549–2552. IEEE.

Ren, H. and Dutt, S. (2008). A Network-Flow Based Cell Sizing Algorithm. In *The International Workshop on Logic Synthesis*.

Roy, S., Chen, C.-P., and Hu, Y. (2007). Numerically convex forms and their application in gate sizing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(9):1637–1647.

Sundararajan, V., Sapatnekar, S., and Parhi, K. (2002). Fast and exact transistor sizing based on iterative relaxation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(5):568–581.

Tennakoon, H. and Sechen, C. (2002). Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 395–402. ACM.

Valles, A. (2009). Performance Insights to Intel(r) Hyper-Threading Technology. Technical report, http://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology.

Wang, J., Das, D., and Zhou, H. (2009). Gate sizing by Lagrangian relaxation revisited. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(7):1071–1084.

CHAPTER 3. RAPID GATE SIZING WITH FEWER ITERATIONS OF LAGRANGIAN RELAXATION

Published in International Conference on Computer Aided Design 2017

Abstract - Existing Lagrangian Relaxation (LR) based gate sizers take many iterations to converge to a competitive solution. In this chapter, we propose a novel LR based gate sizer which dramatically reduces the number of iterations while achieving a similar reduction in leakage power and meeting the timing constraints. The decrease in the iteration count is enabled by an elegant Lagrange multiplier update strategy for rapid coarse-grained optimization as well as finer-grained timing and power recovery techniques, which allow the coarse-grained optimization to terminate early without compromising the solution quality. Since LR iterations dominate the total runtime, our gate sizer achieves an average speedup of 2.5x in runtime and saves 1% more power compared to the previous fastest work.

3.1 Introduction

In modern chip design methodologies, circuit optimization via gate sizing is regarded as one of the key techniques that needs to be invoked at several design stages to trade off various metrics such as timing, area, and power. Due to the large number of gates in a design, gate sizing can be very time consuming. In a standard cell based design, each gate can be implemented by many different options which are characterized by a size and a threshold voltage (Vt). Each option trades off power, area, and delay. The task of gate sizing is to assign a suitable option to each gate such that the desired objective is optimized under the given design constraints. In this work, we focus on timing constrained leakage power (hereafter referred to as power) minimization.

The problem of gate sizing has been studied for over three decades. Earlier the gate sizes were assumed to be continuous and the timing models were either derived from RC Elmore delay models, which can be transformed into a convex function of sizes, or approximated by a convex function. Under such scenarios, an optimal solution could be obtained by applying techniques like Lagrangian Relaxation (LR) Chen *et. al.* (1999). However, such delay models are too inaccurate to achieve a good solution with modern process technologies - non-convex lookup table-based delay models have been industry standard for more than a decade. With discrete size and Vt options, the gate sizing problem is NP hard Ning (1994) and thus no polynomial time optimal algorithm is known. With millions of gates in designs and tens to hundreds of discrete options for each gate size, it can take a day or more of runtime to deliver acceptable solution quality.

For discrete gate sizing, researchers have presented several heuristics based on dynamic programming Liu *et al.* (2010), sensitivity guided greedy frameworks Hu *et al.* (2012), network flow Li *et al.* (2012), and LR based techniques like Livramento *et al.* (2014); Flach *et al.* (2014); Sharma *et al.* (2015); Reimann *et al.* (2016). After the ISPD 2012 gate sizing contest Ozdal *et al.* (2012), several publications established the superiority of the LR based gate sizers, showing both faster runtime and lower power. However, the proposed LR based gate sizers take many LR iterations, even more than 100 on some benchmarks. The high number of iterations can be very detrimental for runtime, especially with expensive timing updates to account for the RC parasitics on large designs.

A reason for the high iteration count of the LR based gate sizers is that they are effective only for coarse-grained timing and power recovery. As the total negative slack (TNS) and the total leakage power of the design reduce, the efficiency of each iteration also degrades. Although LR based gate sizers are usually equipped with greedy post-pass heuristics for finer-grained timing and power recovery, they cannot be invoked too early as they are very time consuming and get stuck in a local minimum. The LR iterations need to go on until TNS and power are sufficiently small. Otherwise, the outstanding timing violations and the remaining potential power savings would be too large to be effectively handled by those greedy techniques. Therefore, we need strategies that can reduce the runtime of coarse-grained optimization by reducing LR iterations, and techniques for finer-grained optimization.

In this chapter, we develop an LR based rapid gate sizer (RGS). We propose an elegant Lagrange multiplier update strategy that makes the coarse-grained LR based sizing converge very rapidly to a solution with sufficiently low TNS and power. We propose two LR-based techniques, one for finer-grained timing recov-

Table 3.1: Acronyms and their meanings.

Acronym	Meaning
LR	*L*agrangian *R*elaxation
LRS	*L*agrangian *R*elaxation *S*ubproblem
LDP	*L*agrangian *D*ual *P*roblem
RGS	*R*apid *G*ate *S*izer
CPS	*C*ritical *P*ath *S*izing
MGS	*M*ulti-*G*ate *S*izing
SGS	*S*ingle *G*ate *S*izing
TNS	*T*otal *N*egative *S*lack
STA	*S*tatic *T*iming *A*nalysis

ery, and the other for finer-grained power recovery. For timing recovery, our proposed technique is called critical path sizing (CPS), which reduces the delay along critical paths. For power recovery, our proposed technique is called multi-gate sizing (MGS), which sizes several gates simultaneously, unlike typical sizing heuristics employed by LR based sizers which size one gate at a time. While CPS is able to efficiently fix the timing violations that may occur during power recovery, MGS allows coarse-grained optimization to terminate early, thereby reducing the expensive LR iterations without compromising on the final solution quality. MGS can also potentially take the design out of a local minimum, thus creating opportunities for further power recovery. With these three techniques, the number of LR iterations is significantly lower than those in previous works. Since LR iterations dominate the total runtime, RGS achieves an average speedup of 3x compared to the previously fastest work Sharma *et al.* (2015).

Our major contributions are summarized as follows:

- We propose an elegant Lagrange multiplier update strategy.
- We propose two LR-based techniques, MGS and CPS, for fine-grained power and timing recovery, respectively.
- We develop a rapid gate sizing flow, and empirically verify its effectiveness.

This chapter is organized as follows: Section 3.2 formulates the problem. Section 3.3 presents the overall flow of RGS and briefly discusses some of its components. Section 3.4 describes the core solver of

RGS in detail. There we discuss our proposed techniques: Lagrange multiplier update strategy, MGS, and CPS. Section 3.5 discusses the empirical results, and we conclude in Section 3.6.

3.2 Problem Formulation

With our gate sizing algorithm, we solve the following constrained optimization problem:

Given a gate-level netlist, a standard cell library with discrete choices for cell size and threshold voltage (Vt), timing constraints, and lumped parasitics, compute the size and Vt combination (hereafter referred to as 'option') for each combinational gate in the netlist such that the leakage power is minimized without violating the timing constraints.

This is the same formulation as used in the ISPD 2012 gate sizing contest. For our experiments, we use the same setup including the set of benchmarks as provided in the contest. Per the contest guidelines, there are two types of electrical constraints: load at the output of a gate cannot exceed a maximum value (max load constraint), and slew at the input of a gate cannot exceed a maximum value (max slew constraint). Since computing the max slew violations is computationally more expensive than computing the max load violations, we translate max slew constraints into max load constraints - this can be done because a lumped wire capacitance model is used for the ISPD 2012 gate sizing contest. Thus ensuring that there is zero max load violation guarantees zero max slew violation.

Before formally defining the problem, we detail notations commonly used in this work. Table 3.1 lists acronyms that are commonly used throughout this chapter. T is the target clock period. For gate i, x_i denotes the size/Vt option, and a_i denotes the arrival time at its output. $d_{i \to j}$ is the delay of the timing arc $i \to j$ which is defined from the output of the gate i to the output of the gate j. Endpoint of a timing path can be either a primary output of the design or input of a sequential element (e.g. flip-flop). Note that sequential elements have a fixed size as per the contest.

Mathematically, the above problem is commonly formulated as a non-convex, discrete mathematical program as:

$$\begin{aligned} \underset{x,a}{\text{minimize}} \quad & \sum_i leakage_i \\ \text{subject to} \quad & a_i + d_{i\to j} \le a_j, \; \forall\, i \to j \\ & a_k \le T \text{ , } \forall \text{ endpoints } k \end{aligned} \tag{3.1}$$

We refer to Equation (3.1) as the *Primal Problem* (PP). To solve this NP-hard problem, as in previous work Chen *et. al.* (1999), we relax the constraints and derive its Lagrangian dual. Each constraint is associated with a non-negative Lagrange multiplier, λ, that acts as a penalty for violating the respective constraint. The Lagrangian function, $L(x, a, \lambda)$ is:

$$\begin{aligned} L(x,a,\lambda) = & \sum_i leakage_i + \sum_{i\to j} \lambda_{i\to j}\left(a_i + d_{i\to j} - a_j\right) \\ & + \sum_{k\in endpoints} \lambda_k \left(a_k - T\right) \end{aligned} \tag{3.2}$$

For a given set of Lagrange multipliers, the Lagrangian relaxation subproblem $LRS(\lambda)$ is:

$$LRS(\lambda) : \underset{x,a}{\text{minimize}}\; L(x,a,\lambda) \tag{3.3}$$

By applying the Karush-Kuhn-Tucker (KKT) conditions for optimality, and omitting the term $\sum_k \lambda_k T$ since it is a constant for a given set of λ_k, the LRS(λ) can be simplified to:

$$LRS^*(\lambda) : \underset{x}{\text{minimize}} \quad \sum_i leakage_i + \sum_{i\to j} \lambda_{i\to j} d_{i\to j} \tag{3.4}$$

where Lagrange multipliers must satisfy the ‘flow constraints’:

$$\sum_{u\in fanin(i)} \lambda_{u\to i} = \sum_{v\in fanout(i)} \lambda_{i\to v}, \forall i \tag{3.5}$$

The objective in Equation (3.4) is referred to as the *LRS cost*. Flow constraints in Equation (3.5) are obtained by setting the partial derivatives over the a variables to 0.

The Lagrangian dual problem (LDP) is then defined as:

$$\begin{aligned} \underset{\lambda}{\text{maximize}} \quad & LRS^*(\lambda) \\ \text{subject to} \quad & KKT\ flow\ constraints \text{ and } \lambda \ge 0 \end{aligned} \tag{3.6}$$

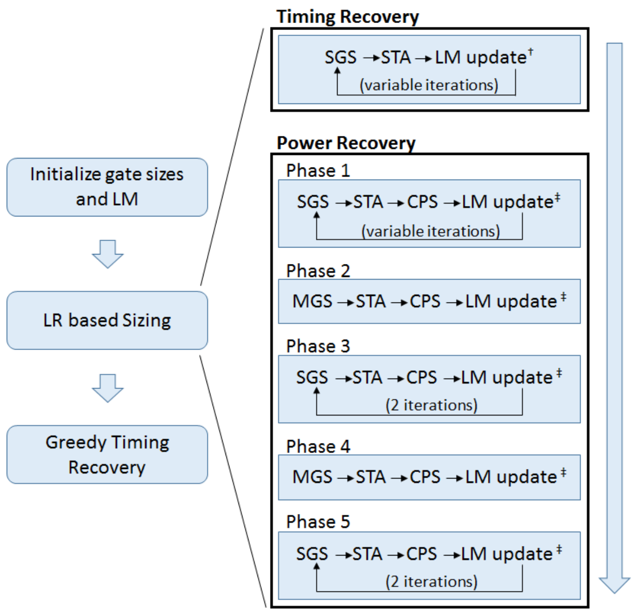

Figure 3.1: RGS flow chart. Each iteration involving SGS and Lagrange multiplier (LM) update is referred to as an LR iteration. †: LM are updated to rapidly recover timing. ‡: LM are updated to rapidly recover power. During SGS of phase three and five, no gate is allowed to degrade power.

Solving LRS*(λ) for a given set of λ gives a lower bound on the PP. And, solving LDP maximizes that lower bound. Optimal values of LDP and PP would match, if the duality gap is zero. A similar Lagrangian relaxation based formulation can be derived for other objective functions like area and dynamic power.

3.3 Overall Flow

Like several other LR-based gate sizers, our gate sizer RGS is also composed of initialization, LR based sizing which is equivalent to solving the LDP in Equation (3.6), and a greedy post-pass. The Overall flow chart for RGS is shown in Figure 3.1.

During the initialization, RGS initializes all gates to the least leakage power option (lowest size and highest Vt), followed by a reverse topological scan to remove the load violations. Lagrange multiplier values for all the arcs are initialized to 1. Initial Lagrange multiplier values would depend upon the leakage power of a typical gate and its timing arc delay. Therefore, for a different library, one-time tuning of the initial Lagrange multiplier value might improve the convergence. To satisfy the KKT flow constraints, Lagrange multipliers are then updated using the projection technique Tennakoon *et al.* (2002). Reimann *et al.* (2016)

proposed a strategy to estimate Lagrange multipliers to speedup the convergence on a pre-optimized design. Since the ISPD 2012 gate sizing contest provided unoptimized designs, their strategy is not needed for these benchmarks.

After initialization, LR based sizing begins. LR based sizing can be broadly divided into two stages: timing recovery, and power recovery. The timing recovery stage is focused on fixing most of the setup timing violations. It is composed of iterations between single gate sizing (SGS) which is a typical heuristic for solving LRS* (described later in this section), static timing analysis (STA) and Lagrange multiplier update. Each such iteration is referred to as an LR iteration. The Lagrange multipliers are updated to facilitate quick timing recovery. The Lagrange multiplier update strategy will be discussed in Section 3.4.1. When TNS is below a threshold, tsh_{TNS}, power recovery can begin.

The second stage of LR based sizing is power recovery. It is composed of five phases. While phases one, three, and five use SGS to solve LRS*; phases two and four use MGS. Like timing recovery, phase one of power recovery is coarse-grained optimization. It iterates between SGS, STA, CPS, and Lagrange multiplier update. In each phase of power recovery, CPS is optionally invoked if timing violations exceed the threshold tsh_{TNS} to keep timing violations under control so that power recovery can continue unimpeded. Lagrange multipliers are updated to facilitate quick power recovery. Phase one achieves the bulk of the power recovery and is usually the most runtime expensive phase. It is terminated as soon as the improvement in power is less than a threshold, tsh_{pow}, compared to the previous iteration.

Phases two through five perform finer-grained power recovery. Phase two performs a single iteration of MGS followed by STA, CPS, and multiplier update. Since MGS is time consuming we perform only one iteration. In addition to power recovery, MGS can potentially perturb the design out of a local minimum, creating opportunities for more power reduction. Therefore, we invoke SGS based power recovery iterations in phase three. Empirically, we determined that beyond two iterations in this phase, power recovery diminishes significantly. Unlike the SGS during coarse-grained optimization, i.e., during timing recovery and phase one of power recovery, SGS in phase three does not allow any gate to increase its power. To recover more power, we repeat phases two and three again in phases four and five, respectively.

After LR based sizing, small timing violations can still remain. Therefore, we invoke greedy timing recovery to eliminate all the violations. In greedy timing recovery, gates with a larger number of critical paths passing through them are processed first. Each gate is upsized, and timing is propagated through its entire fanout cone. If timing degrades, the sizing is undone and the next gate is processed. This is a common greedy heuristic to recover small timing violations.

There are two main differences between our flow and previous flows. Firstly, we have an explicit timing recovery stage followed by a power recovery stage. This provides finer grained control on runtime and solution quality. Previous works did not make such distinction. Secondly, instead of a greedy power recovery, we employ MGS which is parallelizable and is LR compliant.

Solving LRS*(λ) In the context of LR based gate sizing, single gate sizing (SGS) is a common heuristic for solving LRS* in Equation (3.4). SGS is briefly described as follows. For a given set of Lagrange multipliers, assuming no other gate can change its option, SGS near-optimally minimizes the objective in Equation (3.4) for each gate separately. Gates are visited in the forward topological order. For each gate, several options are evaluated to compute their contribution to the LRS cost, and the option that minimizes the cost is assigned to that gate. Changing the option of a gate can potentially affect the delays in the entire fanout cone of that gate. However, in the interest of runtime, and without much loss of accuracy, delay changes only in the local neighborhood of the gate are computed and applied towards the LRS cost computation. We use the multi-threaded version of SGS as proposed in Sharma *et al.* (2015).

Flach *et al.* (2014) suggested that while solving the LRS, in addition to minimizing the LRS cost, its important to prevent timing degradation to ensure solution stability for faster convergence. They proposed an approximate way of ensuring this by restricting the local slack degradation. We call it *local slack check* and apply it in SGS. Through our experiments, we verify that the local slack check indeed facilitates faster timing convergence and thereby, reduce the LR iterations, especially during the timing recovery stage.

3.4 LR Based Gate Sizing

In this section we discuss various components of LR based gate sizing. We present our proposed Lagrange multiplier update strategy and describe how it differs in timing recovery and power recovery. Then, we discuss our proposed MGS and CPS techniques.

3.4.1 Lagrange Multiplier Update

The Lagrange multiplier update strategy is very crucial in reducing the number of iterations for faster convergence. The general strategy is to increase (decrease) the multipliers across timing critical (non-critical) arcs. The key is how much to change. Most of the previous works use a non-linear expression to direct the optimization. The strategies presented in previous works Flach *et al.* (2014), Sharma *et al.* (2015) are reproduced in Algorithms 1 and 2, respectively. Both of these strategies use an exponent to adjust the multipliers and then project them to satisfy Equation (3.5). While the projection heuristic has not changed since it was proposed in Tennakoon *et al.* (2002), the main difference is how the exponent is tuned. Flach *et al.* (2014) do not provide much detail on how to tune the exponent. Moreover, they use different expressions for critical and non-critical arcs (refer Algorithm 1) without any insight. On the other hand, Sharma *et al.* (2015) complicate the tuning of the exponent by introducing two other parameters, namely r and k (refer Algorithm 2). Both of the strategies have slow convergence.

Algorithm 1 Lagrange multiplier update algorithm from Flach *et al.* (2014)

```
// $q_x$: required time at the output of gate $x$
for timing arc $i \rightarrow j$ do
    if $a_j \geq q_j$ then
        $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(1 + \frac{a_j - q_j}{T}\right)^{1/k}$
    else
        $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(1 + \frac{q_j - a_j}{T}\right)^{-k}$
Projection to satisfy Equation (3.5).
```

We propose a single expression for the Lagrange multiplier update (refer Algorithm 3) along with a much simpler strategy to tune the exponent. $D_{i\rightarrow j}$, in Algorithm 7, is the **worst path delay through the arc** $i \rightarrow j$, and K is the 'acceleration' factor. The ratio $D_{i\rightarrow j}/T$ indicates the timing criticality of the

Algorithm 2 Lagrange multiplier update algorithm from Sharma *et al.* (2015)

```
// $WPD$: Worst path delay
$cexp = 1$
$T' = r \times T$
if $WPD > T'$ then
    $cexp = cexp \times \frac{WPD}{T}$
else
    $cexp = cexp \times \left(1 + k \times \frac{WPD-T'}{T'}\right)$
for timing arc $i \rightarrow j$ do
    $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(1 + \frac{a_j - q_j}{T}\right)^{cexp}$
Projection to satisfy Equation (3.5).
```

arc $i \rightarrow j$. The ratio is more than one for an arc with a timing violation, so the Lagrange multiplier for such an arc is increased. For a non-critical arc, the ratio is less than one, therefore its multiplier is decreased. The acceleration factor determines how quickly the Lagrange multipliers increase or decrease. Larger acceleration factors can speedup the convergence but can also cause the solution to get stuck in a worse local minimum.

Algorithm 3 Our proposed Lagrange multiplier update algorithm

```
for timing arc $i \rightarrow j$ do
    $\lambda_{i\rightarrow j} = \lambda_{i\rightarrow j} \times \left(\frac{D_{i\rightarrow j}}{T}\right)^{K}$
Projection to satisfy Equation (3.5). Refer Tennakoon et al. (2002)
```

3.4.2 Timing Recovery

Since timing is a hard constraint that must be met, we first focus on fixing setup timing violations. To enable fast timing recovery, delay on the timing arcs with timing violations needs to be emphasized, so Lagrange multipliers for critical timing arcs need to increase faster. Therefore, acceleration factors of more than one for critical arcs during timing recovery improves the convergence of this phase. However, the larger the acceleration factor, the more the overshoot in the power. Figure 3.2 shows the TNS (solid lines) and power (dashed lines) profiles for different values of K for critical timing arcs. For non-critical timing arcs, we set $K = 1$. As K is increased, we observe faster convergence in TNS and larger overshoots in

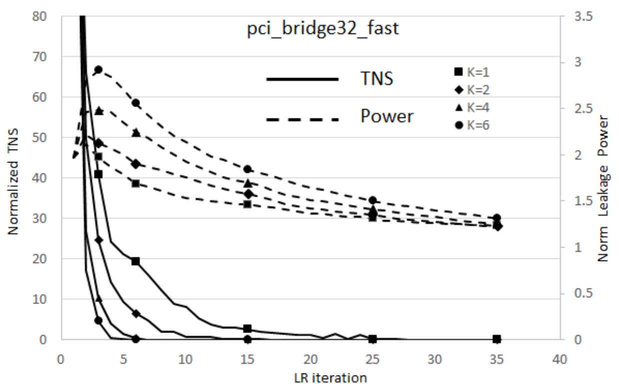

Figure 3.2: TNS and power profiles for different values of K for critical arcs during the timing recovery stage are shown. TNS has been normalized with respect to T. As K increases, TNS reduces faster.

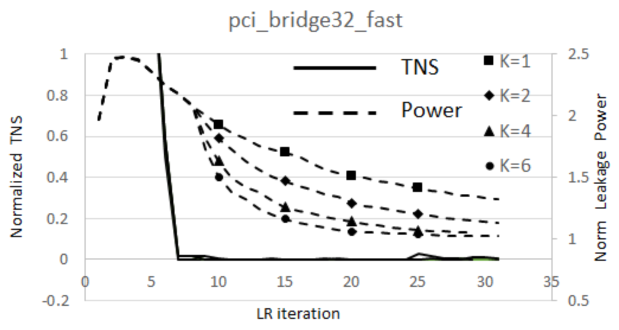

Figure 3.3: TNS and power profiles for different values of K for non-critical arcs are shown. Around iteration 10, timing recovery ends and power recovery phase one begins. Note: TNS profiles for all K indistinguishably overlap. Therefore, markers are not used for them.

power. Note that, from $K = 4$ to $K = 6$, improvement in the TNS convergence is marginal but overshoot in the power is significant, as it may not always be recoverable due to the likelihood of the algorithm getting stuck in some bad local minimum. Also, due to the design being oversized for power, it is possible to simultaneously improve power and timing in the later iterations.

We use $K = 4$ for the critical arcs, and $K = 1$ for the non-critical arcs, during the timing recovery stage of LR based sizing. The small value of K for non-critical arcs means we gradually reduce their Lagrange multipliers and recover some power even during the timing recovery phase. In our case, timing recovery is said to converge when timing violations are less than a threshold tsh_{TNS}. With this setting, timing recovery on our experimental benchmark suite converges in four iterations on average.

3.4.3 Power Recovery

To quickly reduce power, Lagrange multipliers need to rapidly reduce along the non-critical timing arcs. Therefore, a large acceleration factor for the non-critical timing arcs is crucial. Figure **??** shows TNS and power profiles for different values of K applied to such arcs. For critical timing arcs, we set $K = 1$. We can observe that with larger K, power reduces more rapidly. We note that the gain diminishes from $K = 4$ to $K = 6$. On the other hand, TNS does not seem to be affected by K. This is because timing degradation is discouraged by the local slack check. Even if, on account of reduced Lagrange multipliers, LRS cost favors a smaller size (or larger Vt) for a gate, that size is not applied if local slack would degrade. Additionally, we invoke CPS for sizing critical paths whose timing violations exceed the threshold.

During all phases of power recovery, we set K to 1 and 6 for critical and non-critical timing arcs, respectively. If we use a larger K for non-critical arcs, we notice that the final power is 1 to 2% worse, because the design gets stuck in a worse local minimum. In order to terminate power recovery phase one, we propose an aggressive early exit strategy. We do not want to wait until phase one yields the best power that it can, because it can be very runtime inefficient in recovering power at later LR iterations. Therefore, we terminate this phase as soon as reduction in power is less than a threshold, tsh_{pow}, compared to the previous iteration, and invoke MGS which can size multiple gates simultaneously to recover power in larger chunks. We empirically verify that our aggressive early exit strategy can significantly reduce the number of iterations without compromising on the final power.

3.4.4 Multi-Gate Sizing (MGS)

As briefly discussed in Section 3.3, a typical way of solving LRS is SGS which processes one gate at a time assuming that the other gates do not change their options. Such an approach restricts the solution space exploration, and increases the likelihood of the sizing solution getting stuck in a local minimum. We propose MGS to alleviate this drawback by allowing multiple gates to simultaneously change their sizes. In favor of runtime, we do not change Vt and thereby, restrict the number of sizing combinations.

We'll briefly describe the MGS algorithm in reference to its pseudo code shown in Algorithm 4. MGS processes gates in forward topological order. Gate g is downsized and, if the new size of g is valid (lines

Algorithm 4 Multi-gate sizing (MGS)

```
1:  for each gate $g \in S$ in forward topological order do
2:      $status = false$
3:      $resize_g = g.downsize()$
4:      $resizes = \{resize_g\}$
5:      if $resize_g.valid()$ then
6:          for each $fo \in g.fanout()$ do
7:              $resize_{fo} = single\_gate\_sizing(fo)$
8:              $resizes = resizes \cup \{resize_{fo}\}$
9:          if $change\_in\_power(resizes) < 0$ then
10:             $\triangle negSlack = local\_neg\_slack\_change(resizes)$
11:             if $\triangle negSlack \leq 0$ then
12:                 $status = true$
13:     if $status \neq true$ then
14:         $undo(resizes)$
15:     STA every four topological levels
```

3-5) - in other words, no new load violations are created - then all the fanouts of g are sized in the same way as in SGS (lines 6-8). However unlike SGS, in favor of runtime and also since we do not anticipate large perturbations in the size of a gate at this stage, for each fanout only three options are evaluated: the current option, the option with the next bigger size, and the option with the next smaller size. The least LRS cost option is assigned to each fanout. New options for the gate g and all its fanouts are referred to as a set of *resizes*. Lines 9 and 11 describe the conditions to accept the resizes. The first condition is that the total power must decrease. If the first condition holds true, then the change in negative slack of the neighboring gates is computed (line 11). If the negative slack has not degraded then the resizes are accepted. If either condition fails, the resizes are undone (line 14). Since local slack degradation is only a rough indicator of the impact of the resizes on circuit timing, to prevent large timing violations from accumulating, we update timing after every four topological levels (line 15).

Since MGS can be more time consuming than SGS, we refrain from applying it during phase one of power recovery. We apply it only twice, during phases two and four of power recovery.

Algorithm 5 Critical Path Sizing (CPS)

```
1: $tsh = tsh_{TNS}/vend$                          ▷ $vend$: #endpoints that violate timing
2: $S = \phi$
3: for each timing endpoint, $end$ do
4:     if $end$.slack $< -tsh$ then
5:         $S = S \cup \{end\}$
6: Sort elements of $S$ in the ascending order of slack
7: for each $end \in S$ do
8:     $P = critical\_path(end)$
9:     $minDeltaLM =$ some arbitrarily large value
10:    for each arc $i \to j \in P$ do
11:        $\triangle\lambda_{ij} = \lambda_{ij}\left[\left(\frac{D_{ij}}{T}\right)^K - 1\right]$
12:        if $\triangle\lambda_{ij} < minDeltaLM$ then
13:            $minDeltaLM = \triangle\lambda_{ij}$
14:    for each arc $i \to j \in P$ do              ▷ Update Lagrange multiplier along the path
15:        $\lambda_{ij} = \lambda_{ij} + minDeltaLM$
16:    for each gate $g \in P$ do                   ▷ LRS along the path
17:        $single\_gate\_sizing(g)$
18:    increamental_STA()
```

3.4.5 Critical Path Sizing (CPS)

The timing recovery stage can reduce the bulk of the timing violations in a few LR iterations. During various phases of power recovery, despite the local slack check, a few paths may become timing critical and TNS may exceed the threshold, tsh_{TNS}. In such cases, it is an overkill to run LR iterations to recover timing, because each LR iteration scans the entire design several times and is quite expensive. Moreover, LR iterations are not as effective in recovering finer-grained timing violations as they are during the coarse-grained optimization.

To reduce the delay of a timing critical path, usually either a gate along the path is upsized or its load is reduced. (Reducing the Vt at this stage of the algorithm is generally avoided due to the large increase in the leakage power.) Typically, to upsize a gate via SGS, Lagrange multipliers of the gate's timing arcs need to be large enough to justify trading power for reduced delay. However, at this stage in the algorithm when timing violations are not very big, for most of the timing arcs $i \to j$, $D_{i\to j}$ would be either close to T or significantly smaller than T. Consequently, it may require several LR iterations before the gate will be upsized by SGS.

The delay of a timing critical gate can also be reduced by reducing its load, but that may induce timing violations on near-critical paths. Thus, critical and near-critical paths may compete with each other, thereby slowing down the convergence. Another strategy to reduce the small timing violations is to uniformly scale up all the Lagrange multipliers. This strategy is similar to applying the 'power weighting factor' of Livramento *et al.* (2014). Although it can eliminate all the timing violations in one to two iterations, it tends to upsize even the non-critical gates, which causes unnecessary increase in the leakage power.

The CPS is designed to reduce the timing violations of the critical paths, while minimally affecting the total design power, and it is also very fast as it works on only a small sub-circuit. We describe the CPS algorithm with pseudo-code shown in Algorithm 5. In line 1, we compute a threshold, tsh, to identify timing critical endpoints. tsh is derived from tsh_{TNS} which is the allowed total timing violation during the LR based sizing stage. tsh is simply the average violation allowed per endpoint. Critical endpoints are then sorted by their slack (line 6). More timing critical endpoints, with more negative slack, are processed first. For each endpoint, its critical path, P, is computed (line 8). Then, lines 9 through 13 compute how much to increase the Lagrange multiplier along P. We use the Lagrange multiplier update expression from Algorithm 3 to compute the potential change in the Lagrange multiplier of each timing arc along the path (line 11), and track the minimum value, $minDeltaLM$. In order to emphasize the delay along the critical path, we would want substantial increase in the Lagrange multipliers. Therefore, we set $K = 10$ during the CPS. Then, the Lagrange multipliers of all the arcs along P are increased by $minDeltaLM$ (lines 14-15), followed by resizing all the gates along P. Lastly, the timing is incrementally updated in line 18 before processing the next primary output, so that the critical path of the next endpoint is computed based on the updated timing.

3.5 Experimental Results

We implemented our gate-sizer in C++. Experiments are performed on an 8-node cluster made up of two quad-core Intel(R) Xeon(R) E3-1240 v5 CPU @ 3.67GHz with an aggregate memory of 16GB. For multi-threading, OpenMP Dagum *et al.* (1998) is used. We use 8 threads. All the results reported in this work are averaged over 10 runs to minimize the bias due to non-determinism caused by multi-threading.

Table 3.2: Comparison of overall runtime and power of RGS versus the baseline (Sharma *et al.* (2015) [1]). Slow refers to the loose timing constraints, and fast refers to the tighter timing constraint. Benchmarks are listed in order of ascending number of combinational cells. DMA through netcard approximate combinational cell count is 23K, 30K, 102K, 148K, 213K, 540K and 861K, respectively.

Benchmark	Total runtime (min)			Leakage Power (W)		
	[1]	RGS	[1]/RGS	[1]	RGS	RGS/[1]
DMA_slow	0.11	0.07	1.47	0.135	0.135	0.997
pci_b32_slow	0.27	0.09	3.02	0.099	0.098	0.995
des_perf_slow	0.43	0.32	1.33	0.597	0.583	0.977
vga_lcd_slow	1.43	0.44	3.27	0.331	0.329	0.995
b19_slow	3.03	0.83	3.66	0.568	0.569	1.001
leon3mp_slow	3.91	2.52	1.55	1.335	1.335	1.000
netcard_slow	5.48	2.35	2.33	1.763	1.763	1.000
DMA_fast	0.26	0.08	3.05	0.251	0.245	0.979
pci_b32_fast	0.31	0.10	2.95	0.142	0.141	0.993
des_perf_fast	1.16	0.40	2.91	1.455	1.436	0.987
vga_lcd_fast	1.82	0.56	3.28	0.433	0.417	0.963
b19_fast	3.19	1.13	2.82	0.733	0.729	0.995
leon3mp_fast	4.88	3.13	1.56	1.443	1.449	1.004
netcard_fast	7.05	3.33	2.12	1.848	1.846	0.999
Average			**2.52**			**0.992**

The experimental set up including the benchmark suite is identical with the ISPD 2012 gate sizing contest. In our flow, we set $tsh_{TNS} = 0.1 \times T$ and $tsh_{pow} = 0.1\%$.

3.5.1 A Comparison With Previous Works

We use the algorithm proposed by Sharma *et al.* (2015) as our baseline. To the best of our knowledge, among all the published results so far, they have reported the best runtime on the ISPD 2012 contest benchmarks with 2.5% degradation in the average leakage power compared to the best quality published results Flach *et al.* (2014). For fair comparison against Sharma et. al., we executed their binary with 8 threads on our cluster, and we are using those results as the baseline in Table 3.2. Since our cluster uses faster CPUs, the runtimes shown in column two of Table 3.2 are on average 18% smaller than the runtimes published by Sharma *et al.* (2015). Powers in column five are on average 0.001% more than the published results. We also compare against Flach *et al.* (2014). Their results are obtained from single threaded runs executed on 3.40GHz Intel(R) Core(TM) i7-3770 CPU.

Table 3.2 compares the total runtime and the power between RGS and the baseline. Both the flows yield timing violation free designs on all the benchmarks. On average, RGS is 2.52x faster than the baseline with 0.8% extra power savings. Compared with Flach *et al.* (2014), which is single threaded, RGS is 19x faster and 1.5% worse in power. Authors believe that the 1.5% degradation in power can be attributed to the differences in tuning of the acceleration factor, K. We are able to optimize the biggest design in the suite, *netcard* which has around 861K combinational gates, in 3.33 min for the 'fast' and 2.35 min for the 'slow' timing constraints. The main contributor towards the speedup is significant reduction in LR iteration count. In Section 3.5.2, we analyze various factors that contributed towards reducing the LR iteration count.

Average runtime breakdown of our flow is as follows: LR iterations dominate the runtime by accounting for 78% of the total runtime; followed by MGS (5%); greedy timing recovery (3%); and lastly, the CPS (1%). 14% of the total runtime is consumed in parsing the input verilog, spef, library files, and pre-processing. Note that although MGS can be parallelized like SGS, currently it is sequential.

Compared to the power reported by the baseline (fifth column in Table 3.2), power reported by RGS after the coarse grained optimization, i.e., after phase one of power recovery, is the same as the baseline. Then phases two through five perform finer-grained power recovery and achieve a further 1.2% reduction in power, followed by the greedy timing recovery which increases power by 0.2%.

3.5.2 Factors Contributing to the Reduction in LR Iterations

Table 3.3: Catalog of flows referred for different analysis.

Name	Flow
v1	RGS with the early exit policy adapted from the baseline Sharma *et al.* (2015)
v2	v1 with Lagrange multiplier update strategy adapted from the baseline
v3	v2 without the local slack check in the timing recovery phase
v4	v3 with CPS in the timing recovery phase

Most of the speedup in RGS is due to the reduction in the LR iterations. In this section, we empirically analyze the contribution of different factors towards the reduction in the total LR iteration count. We identify three main factors, namely, (1) the aggressive early exit from the first phase of power recovery, (2)

Table 3.4: We compare the following for v3 and RGS: LR iteration count for the timing recovery stage (TR); LR iteration count for power recovery phase one (PR); and power after the power recovery phase one (LRpow). Power numbers are normalized with respect to the baseline (fifth column of Table 3.2). At the bottom we also append the average results for v1 and v2.

Benchmark	v3			RGS		
	TR	PR	LRpow	TR	PR	LRpow
DMA_slow	10	42	0.999	3	14	1.007
pci_bridge32_slow	64	68	0.987	4	14	0.990
des_perf_slow	9	54	0.984	3	16	0.992
vga_lcd_slow	53	61	0.999	4	10	1.003
b19_slow	17	111	0.996	4	16	1.007
leon3mp_slow	8	34	1.001	3	10	1.003
netcard_slow	1	45	1.000	1	5	1.000
DMA_fast	72	57	0.982	5	15	1.000
pci_bridge32_fast	66	76	0.956	6	17	0.996
des_perf_fast	71	54	1.004	5	24	0.999
vga_lcd_fast	61	69	0.990	8	13	0.988
b19_fast	71	59	0.998	6	30	1.004
leon3mp_fast	10	37	1.003	3	18	1.012
netcard_fast	5	30	0.999	2	11	1.000
Average	**37**	**57**	**0.993**	**4**	**15**	**1.000**
v2 **Average**	**20**	**48**	**0.993**			
v1 **Average**	**4**	**44**	**1.002**			

the proposed Lagrange multiplier update strategy, and (3) restricting the timing degradation in the timing recovery phase via local slack check. To evaluate the impact of each one of these factors individually we derive three different versions of gate sizers from RGS: v1, v2 and v3. They are summarized in Table 3.3. In v1, we replace our aggressive early exit strategy by the early exit policy of the baseline. Baseline runs LR iterations as long as power is improving, whereas RGS terminates phase one of power recovery as soon as improvement in power is less than tsh_{pow}. v2 is built on top of v1 by replacing our proposed Lagrange multiplier update strategy with the corresponding strategy from the baseline. Lastly, v3 is built on top of v2 by disabling the local slack check in the timing recovery phase. On average, v3 has similar runtime as the baseline and it yields designs with 1% better power. So it is a good comparison point for our further analysis.

Table 3.4 shows the LR iteration count during timing recovery stage, LR iteration count during phase one of power recovery stage, and power after phase one of power recovery for v3 and the RGS flows. For the sake of comparison, average results for the same metrics are shown for v1 and v2 as well. Comparison of v1 and RGS shows that, by exiting early from phase one of power recovery, RGS could save 29 more iterations with only 0.2% higher power after phase one. There are slight fluctuations due to randomness on account of multi-threading. Compared to v1, v2 which uses the Lagrange multiplier update strategy of the baseline, takes 5x as many iterations to recover timing. v2's power after power recovery is around 1% lower than v1's power. Our Lagrange multiplier update uses large acceleration factors for quick timing and power recovery. That may cause the solution to get stuck in a worse local minimum, so power is slightly worse after phase one of power recovery. However, later power recovery phases involving MGS are able to recover it. Compared to v2, v3 which disables the local slack check during its timing recovery stage, spends on average 17 more iterations to converge the timing, and still yields the same power.

In summary, while our Lagrange multiplier update strategy is extremely effective in improving the convergence of timing recovery, the local slack check also helps to a smaller extent. The Lagrange multiplier update strategy also enables fast power recovery during phase one. Additionally, by exiting early from phase one, and relying on the later phases for finer-grained power recovery, the iteration count in phase one significantly reduced. Figure 3.4 compares TNS and power profiles for v3 and RGS runs on pci_bridge32_fast. As seen in the TNS profile of v3, timing convergence is initially quite slow due to poor Lagrange multiplier updates, and after iteration 22 due to lack of local slack check. As seen in the power profile of v3 between iterations 60 and 110, the power recovery in phase one is slow. This is due to the poor multiplier update strategy. After iteration 110, in an attempt to recover finer-grained power, phase one does not exit. Consequently, v3 takes about 140 iterations in total to complete timing recovery and phase one of power recovery, whereas RGS completes in only about 25 iterations. Overall, compared to v3, RGS reduces the total LR iterations of timing recovery and phase one of power recovery by 80%. The average power of RGS after phase one of power recovery is only 0.8% worse.

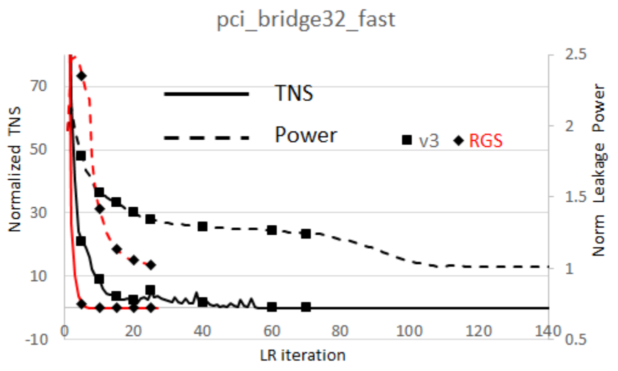

Figure 3.4: Comparison of TNS and power profiles for v3 and RGS runs on the pci_bridge32_fast. LR iterations shown on the x-axis are from the timing recovery stage and phase one of power recovery. For RGS, by iteration 6, timing recovery ends and power recovery phase one begins. For v3, timing recovery extends until around iteration 60, followed by the power recovery phase one until iteration 140.

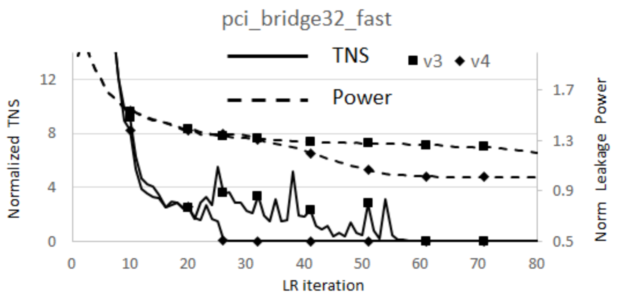

Figure 3.5: TNS and power profiles for v3 and v4 runs on the pci_bridge32_fast are plotted. Only 80 LR iterations are shown. v4 invokes CPS around iteration 22 when it is still in its timing recovery stage, and TNS starts to degrade. Within 3 calls, CPS recovers the timing from the critical paths, and power recovery phase one begins at around iteration 26. On the other hand, v3 could not converge until 60 iterations.

3.5.3 Impact of CPS on Timing Convergence

In order to evaluate the impact of CPS on timing convergence, we build a flow version, v4, by adding CPS in the timing recovery stage of v3. CPS has the ability to quickly recover timing from the critical paths. Therefore, once the TNS falls below a threshold ($2T$, in v4), and then if it degrades, we invoke CPS **instead of** SGS to converge the timing. Figure 3.5 shows pci_bridge32_fast as an example. In the figure, around iteration 20, TNS falls below $2T$. Then, TNS starts to degrade (increase) around iteration 22 at which point the CPS is invoked. CPS is able to recover the timing in just 3 calls. On the other hand, v3 is not able to converge the timing until iteration 60.

Overall results show that v4 on average, can cut down the number of LR iterations in timing recovery from 37 to 25. Moreover, each iteration of CPS is around 10x faster than an iteration of LRS. In fixing critical path timing violations, CPS causes only marginal increase in the total design power.

3.6 Conclusion

In modern VLSI physical design flows, gate sizing is a time consuming optimization. LR-based gate sizers provide good quality results, but can take significant runtime due to the need to update timing after each iteration, as they can take many LR iterations to converge. In this work, we propose several techniques to enable rapid gate sizing by reducing the number of LR iterations. We utilize an elegant Lagrange multiplier update strategy to speed up the coarse-grained timing and power recovery. We also propose two LR-based techniques, MGS and CPS, for finer-grained power and timing refinement. These techniques allow the coarse-grained optimization to terminate early, further cutting down the number of iterations. Since LR iterations dominate the total runtime, our proposed gate sizer, RGS, is 3x faster than the previous fastest LR-based gate sizer, while still achieving state-of-the-art reduction in leakage power.

References

Chen *et. al.*, C.-P. (1999). Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):1014–1025.

Dagum *et al.*, L. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55.

Flach *et al.*, G. (2014). Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557.

Hu *et al.*, J. (2012). Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 233–239.

Li *et al.*, L. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–232. IEEE.

Liu *et al.*, Y. (2010). A new algorithm for simultaneous gate sizing and threshold voltage assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(2):223–234.

Livramento *et al.*, V. S. (2014). A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4):40.

Ning, W. (1994). Strongly NP-hard discrete gate-sizing problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1045–1051.

Ozdal *et al.*, M. (2012). The ISPD-2012 discrete cell sizing contest and benchmark suite. In *ACM International Symposium on Physical Design*, pages 161–164. ACM.

Reimann *et al.*, T. J. (2016). Cell selection for high-performance designs in an industrial design flow. In *ACM International Symposium on Physical Design*, pages 65–72. ACM.

Sharma *et al.*, A. (2015). Fast Lagrangian relaxation based gate sizing using multi-threading. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 426–433. IEEE.

Tennakoon *et al.*, H. (2002). Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 395–402. ACM.

CHAPTER 4. FAST LAGRANGIAN RELAXATION BASED MULTI-THREADED GATE SIZING USING SIMPLE TIMING CALIBRATIONS

Abstract - Accurate delay analysis with distributed RC delay can be computationally expensive, and can contribute the majority of the total runtime for gate sizers. Recent works have shown that Lagrangian relaxation based gate sizers have produced designs with the lowest power on average. But they are also very slow due to a large number of expensive timing updates spread across several tens of iterations. In this chapter we develop a Lagrangian relaxation based discrete gate sizer for fast timing and power reduction.

Our gate sizer is multi-threaded and is equipped with parallelization enabling techniques, namely, mutual exclusion edge (MEE) assignment and directed acyclic graph (DAG) based netlist traversal. MEE are dummy edges assigned to improve load sharing among different threads. DAG based netlist traversal facilitates simultaneous resizing of gates belonging to different topological levels.

Our Lagrange multiplier update strategy enables rapid convergence of our timing and power recovery algorithms. To reduce the runtime of timing updates, we propose a simple and fast-to-compute effective capacitance model. We further propose mechanisms to calibrate timing models to improve their accuracy. By calibrating the internal timing models only twice, our proposed gate sizing flow facilitates extremely fast design optimization.

We benchmark our gate sizer using the ISPD 2012 and 2013 gate sizing contest benchmark suites. Compared to the state-of-the-art gate sizer, our proposed gate sizer is on average 15x faster and the optimized designs have 2.5% higher leakage power. Since we trade-off timing accuracy for larger runtime speedup, our optimized designs have small timing violations.

4.1 Introduction

One of the most time consuming aspects of gate sizing in general, and LR gate sizing in particular, is the timing update. To respect the timing constraint, timing needs to be updated very frequently - either fully, incrementally, or locally. Since LR gate sizing necessitates evaluation of several cell alternatives for power and delay trade-off for every gate in every iteration of the algorithm, timing updates tend to become the runtime bottleneck. Resistive interconnects not only require modeling the interconnect delay and transition time (or slew) degradation, which by themselves are complicated and CPU intensive when done accurately, but also make the accurate gate delay and gate slew modeling complicated. Typically, an external industrial timer needs to be invoked for accurate timing updates which in general is very slow.

In this work we extend our previous gate sizer Sharma *et al.* (2015) to quickly recover the bulk of the timing violations and the leakage power even from designs where we need to more accurately model the interconnect. We use several calibration mechanisms to greatly improve the internal timer accuracy, while limiting interactions with the external timer.

We use benchmark suites from the ISPD 2012 and the ISPD 2013 gate sizing contests for experimentation. On the ISPD 2012 designs, compared to Flach *et al.* (2014), on average, we achieve 15.3x speedup in the total runtime and leakage power is 1.0% more. On the ISPD 2013 designs, compared to Flach *et al.* (2014), on average, we achieve 15.5x speedup and leakage power is 2.5% more. Our major contributions in this work include:

- a new, fast-to-compute model for effective capacitance that is used in computing the gate delay;
- mechanisms for calibrating the internal delay and slew models for both interconnect and gate to closely track the external timer;
- a gate sizing flow for minimal interaction with the external timer without significant loss of accuracy;
- implementation of a gate sizer that achieves large speedup with little sacrifice of the solution quality.

The rest of the chapter is organized as follows. In Section 4.2 we summarize various relevant works on gate sizing and timing models. We present the LR formulation for the gate sizing problem in Section 4.3. In Section 4.4 we discuss a typical methodology for LR gate sizing. In Section 4.5 we present the overall flow of our gate sizer, and then discuss our proposed calibration mechanisms in Section 4.6. Sections 4.7 and 4.8 review the multi-threaded LR subproblem solver and the Lagrange multiplier update strategy which together form the core of our sizer. We discuss our greedy post-pass strategies for solution refinement in Section 4.9. Various experimental results are discussed in Section 4.10. We lastly conclude in Section 4.11.

4.2 Previous Work

Timing models are crucial factors in determining the final solution quality, rate of convergence to that solution and the total runtime of the gate sizer. While it is a standard practice to use table lookup for computing the gate delays and the gate slews, several different models are used for modeling the delay and the slew degradation in an RC interconnect. Various interconnect models have a trade-off between computational complexity and accuracy. Yella *et al.* (2017); Daboul *et al.* (2018) use an external sign-off timer for all the timing updates; whereas Kahng *et al.* (2013); Flach *et al.* (2014), in the interest of runtime, postpone external timer invocation to the later stages of their respective algorithms. During the early stages, Kahng *et al.* (2013) use the D2M delay model Alpert *et al.* (2000) delay model and the PERI model Kashyap *et al.* (2003) for modeling the slew. Flach *et al.* (2014) use the Elmore delay model Elmore (1948) and PERI. In this work, we use the Elmore delay model and the PERI model since they are computationally less expensive. We discuss calibration mechanisms, to improve their accuracy, in Section 4.6.

Empirical formula for gate delay estimation requires input slew and *effective capacitance* as parameters for look up in the 2D table. Effective capacitance denotes the effective loading at the output pin of a gate. Several algorithms for computing the effective capacitance have been published, like Qian *et al.* (1994); Kahng *et al.* (1999); Abbaspour *et al.* (2003). Most of these algorithms are iterative in nature due to lacking a closed form explicit expression. Even if there is a closed form expression, it depends on the π-RC model parameters of the interconnect being driven by the gate, which needs to be derived whenever a loading gate's input pin capacitance changes. LR gate sizing, where effective capacitance computation would need to sit in the inner most loop of the gate sizing algorithm, all of these effective capacitance models are too slow. Therefore, we use our our proposed simple model along with the calibration mechanism to improve its accuracy.

Table 4.1: Notations used in section 4.3.

Notation	Meaning
T	Target clock period
$\mathcal{G}$	Set of gates in the design
$\mathcal{X}_g$	Discrete set of cells for gate g
$x_g \in \mathcal{X}_g$	Current cell assigned to gate g
$leak_x$	Leakage power of cell x
max_load_x	Maximum load capacity of cell x
$\mathcal{N}$	Set of nodes in the timing graph
$\mathcal{N}_{in}$	Set of nodes that are either input pins of gates or output ports
$\mathcal{N}_{out}$	Set of nodes that are either output pins of gates or input ports
$\mathcal{N}_{end}$	Set of timing end point nodes, i.e., data input pin of a sequential gate or an output port
$\mathcal{E}$	Set of timing arcs in the timing graph
a_i	Arrival time at node i
g_i	Gate or port associated with node i
λ_{ij}	Lagrange multiplier for the timing arc (i, j)
$\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{\lambda}$	Respective set of variables x, a and λ. For example, $\boldsymbol{x} = \{x_g \mid g \in \mathcal{G}\}$
$d_{ij}(\boldsymbol{x})$	Delay function of the timing arc from node i to node j
$slew_i(\boldsymbol{x})$	Function to compute slew at node i
$ceff_i(\boldsymbol{x})$	Function to compute effective capacitance at node $i \in \mathcal{N}_{out}$
S_{max}	Maximum slew defined in the cell library

4.3 Problem Formulation

We solve the same problem as presented in the gate sizing contests. The objective is to minimize the total leakage power (referred hereafter as power) of the design while satisfying the timing constraints. In addition, the effective capacitance load at the output pin of a gate must be less than the maximum load handling capacity of that gate which depends on the cell implementation of that gate. Also, at each input pin of every gate and each output port, the slew has to be less than a maximum value. In the library provided for the contest, there is a global maximum input slew of 300ps. In the contest library, sequential gates have a single cell implementation - they cannot be resized. We are allowed to alter the size and Vth of only the combinational gates. We model rise and fall timing constraints separately, but we omit their separate mention throughout this work for clear presentation. Using the notations shown in Table 4.1, the primal problem is stated as follows:

$$\begin{aligned}
&\underset{\boldsymbol{x},\boldsymbol{a}}{\text{minimize}} && \sum_{g\in\mathcal{G}} \text{leak}_{x_g} && \\
&\text{subject to} && a_i + d_{ij}(\boldsymbol{x}) \le a_j && \forall (i,j)\in\mathcal{E} \\
& && a_k \le T && \forall k \in \mathcal{N}_{end} \\
& && \text{ceff}_i(\boldsymbol{x}) \le max\_load_{x_{g_i}} && \forall i \in \mathcal{N}_{out} \\
& && \text{slew}_i(\boldsymbol{x}) \le S_{max} && \forall i \in \mathcal{N}_{in} \\
& && x_g \in \mathcal{X}_g && \forall g \in \mathcal{G}
\end{aligned} \tag{4.1}$$

where, minimization is over the set of discrete cell variables $\boldsymbol{x}$ and the continuous arrival time variables. While leak$_x$ and max_load$_x$ are specified in the standard cell library for each cell x, computation models for $d_{ij}(\boldsymbol{x})$, slew$_i(\boldsymbol{x})$ and ceff$_i(\boldsymbol{x})$ are discussed later. By definition of arrival times, $a_i + d_{ij}(\boldsymbol{x}) \le a_j$ are always satisfied for all timing arcs. Hence, violations in the timing constraints which are quantified by the metric called total negative slack (TNS), are computed as $TNS = \sum_{k\in\mathcal{E}} (a_k - T)$. Similarly, total load violations and total slew violations can be computed. For computing the load violations, the maximum of the rise ceff$_i(\boldsymbol{x})$ and the fall ceff$_i(\boldsymbol{x})$ is compared against $max\_load_{x_{g_i}}$, for each node $i \in \mathcal{N}_{out}$.

The primal problem (4.1) has non-convex timing constraints. We relax the problem by including those constraints into the objective function. To penalize any constraint violations, each one of them is associated with a non-negative penalty term. These penalty terms are called the Lagrange multipliers and denoted by λ_{ij} (or λ_k, depending upon the constraint it gets associated with). Thus, we get a new objective function for a given set of Lagrange multipliers $\boldsymbol{\lambda}$, called the Lagrangian function, $L_{\boldsymbol{\lambda}}(\boldsymbol{x},\boldsymbol{a})$. It is shown below:

$$L_{\boldsymbol{\lambda}}(\boldsymbol{x},\boldsymbol{a}) : \quad \sum_{g\in\mathcal{G}} \text{leak}_{x_g} + \sum_{(i,j)\in\mathcal{E}} \lambda_{ij} \times (a_i + d_{ij}(\boldsymbol{x}) - a_j) + \sum_{k\in\mathcal{N}_{end}} \lambda_k \times (a_k - T) \tag{4.2}$$

We do not relax the load and slew constraints because they are easy to track during the optimization. We can now define the Lagrangian relaxation subproblem for a given set of Lagrange multipliers, $LRS/\boldsymbol{\lambda}$, as follows:

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{a}}{\text{minimize}} \quad & L_{\boldsymbol{\lambda}}(\boldsymbol{x},\boldsymbol{a}) \\ \text{subject to} \quad & \text{ceff}_i(\boldsymbol{x}) \leq max\_load_{x_{g_i}} \quad && \forall i \in \mathcal{N}_{out} \\ & \text{slew}_i(\boldsymbol{x}) \leq S_{max} && \forall i \in \mathcal{N}_{in} \\ & x_g \in \mathcal{X}_g && \forall g \in \mathcal{G} \end{aligned} \tag{4.3}$$

Chen *et. al.* (1999) derived the Karush-Kuhn-Tucker (KKT) optimality conditions for the subproblem $LRS/\boldsymbol{\lambda}$, and in the process they simplified the Lagrangian function. Wang *et al.* (2009) later noted that the KKT conditions are sufficient but not necessary. However, they derived the same conditions using a different reasoning. For completeness, we reproduce the main steps from their derivation. The Lagrangian function can be rewritten as follows:

$$\begin{aligned} L_{\boldsymbol{\lambda}}(\boldsymbol{x},\boldsymbol{a}) = & \sum_{g\in\mathcal{G}} \text{leak}_{x_g} + \sum_{(i,j)\in\mathcal{E}} (\lambda_{ij} \times d_{ij}(\boldsymbol{x})) - T \times \sum_{k\in\mathcal{N}_{end}} \lambda_k + \\ & \sum_{i\in\mathcal{N}\setminus\mathcal{N}_{end}} a_i \times \left(\sum_{\{v|(i,v)\in\mathcal{E}\}} \lambda_{iv} - \sum_{\{u|(u,i)\in\mathcal{E}\}} \lambda_{ui} \right) + \\ & \sum_{k\in\mathcal{N}_{end}} a_k \times \left(\lambda_k - \sum_{\{u|(u,k)\in\mathcal{E}\}} \lambda_{uk} \right) \end{aligned} \tag{4.4}$$

The Lagrangian function is unbounded below if the following conditions are not satisfied:

$$\begin{aligned} \sum_{\{v|(i,v)\in\mathcal{E}\}} \lambda_{iv} &= \sum_{\{u|(u,i)\in\mathcal{E}\}} \lambda_{ui} \quad && \forall i \in \mathcal{N}\setminus\mathcal{N}_{end} \\ \lambda_k &= \sum_{\{u|(u,k)\in\mathcal{E}\}} \lambda_{uk} && \forall k \in \mathcal{N}_{end} \end{aligned} \tag{4.5}$$

We refer to the constraints in (4.5) as the 'flow constraints' on the Lagrange multipliers. The flow constraints can be stated as follows: at each node, the sum of the incoming Lagrange multipliers should be equal to the sum of the outgoing Lagrange multipliers. Upon applying the flow constraints (4.5) to the Lagrangian function (4.4), coefficients of a_i and a_k become zero, and additionally, by ignoring the term $T \times \sum_{k \in \mathcal{N}_{end}} \lambda_k$ which is constant for a given $\boldsymbol{\lambda}$, we get the simplified subproblem, $sLRS/\boldsymbol{\lambda}$, as shown below:

$$\begin{aligned}
\underset{\boldsymbol{x}}{\text{minimize}} \quad & \sum_{g \in \mathcal{G}} \text{leak}_{x_g} + \sum_{(i,j) \in \mathcal{E}} (\lambda_{ij} \times d_{ij}(\boldsymbol{x})) \\
\text{subject to} \quad & \text{ceff}_i(\boldsymbol{x}) \leq max\_load_{x_{g_i}} \qquad \forall i \in \mathcal{N}_{out} \\
& \text{slew}_i(\boldsymbol{x}) \leq S_{max} \qquad \forall i \in \mathcal{N}_{in} \\
& x_g \in \mathcal{X}_g \qquad \forall g \in \mathcal{G}
\end{aligned} \tag{4.6}$$

where $\boldsymbol{\lambda}$ satisfy the flow constraints (4.5). We refer to $\sum_{(i,j) \in \mathcal{E}} (\lambda_{ij} \times d_{ij}(\boldsymbol{x}))$ as the *lambda-delay sum*, and the objective function of (4.6) as the *LRS cost*. Let $L^*_{\boldsymbol{\lambda}}$ denote the optimal value of the $sLRS/\boldsymbol{\lambda}$. Then, it can be shown that $L^*_{\boldsymbol{\lambda}}$ is a lower bound on the optimal value of (4.1). The Lagrange dual problem is to find an optimal set of Lagrange multipliers that maximize this lower bound. The Lagrangian dual problem (LDP) is thus formulated as follows:

$$\begin{aligned}
\underset{\boldsymbol{\lambda}}{\text{maximize}} \quad & L^*_{\boldsymbol{\lambda}} - T \times \sum_{k \in \mathcal{N}_{end}} \lambda_k \\
\text{subject to} \quad & \sum_{\{v | (i,v) \in \mathcal{E}\}} \lambda_{iv} = \sum_{\{u | (u,i) \in \mathcal{E}\}} \lambda_{ui} \qquad \forall i \in \mathcal{N} \backslash \mathcal{N}_{end} \\
& \lambda_k = \sum_{\{u | (u,k) \in \mathcal{E}\}} \lambda_{uk} \qquad \forall k \in \mathcal{N}_{end}
\end{aligned} \tag{4.7}$$

4.4 Background on Lagrangian Relaxation Based Gate Sizing

In this section we discuss a common strategy to do LR gate-sizing. It has three stages: the initialization; the LDP solver; and the final greedy post-pass. It is assumed that the sizer has an accurate internal timer, otherwise additional stages involving interaction with an accurate external timer must be considered.

4.4.1 Initialization

In the initialization stage, each gate is initialized to its minimum leakage power library cell alternative (smallest size and highest Vth). This usually creates the maximum load and maximum slew violations. To fix load violations the design is scanned in reverse topological order and those gates whose loads exceed the respective gate's maximum load handling capacity are sufficiently upsized and/or, less preferably, a lower Vth cell alternative is used. Afterwards, to fix the slew violations, design may need to be scanned in forward topological order and those gates whose output slew is too large would need to use a higher drive strength cell alternative such that no new load violations are created for the upstream gates.

Initial Lagrange multiplier values would depend upon the leakage power of a typical gate and its timing arc delay. Therefore, for a different library, one-time tuning of the initial Lagrange multiplier value might improve the convergence. To satisfy the flow constraints, Lagrange multipliers are updated using the projection heuristic Tennakoon *et al.* (2002).

4.4.2 Solve LDP

This is an iterative stage to approximately solve the LDP (4.7). In each iteration, the Lagrange multipliers are updated for the given design and the $sLRS/\lambda$ (4.6) is solved by an LRS solver for the updated set of multipliers. Iterations can terminate, for example, when the LRS cost converges.

4.4.2.1 LRS solver

The LRS solver heuristically solves (4.6), for the given set of Lagrange multipliers. In other words, the LRS solver tries to find an optimal cell for each gate such that the weighted sum of delay and power for the entire design is minimized subject to the maximum load and the maximum slew constraints. Note that the weights are the Lagrange multipliers. Since sizes are discrete and delay as well as power are non-convex functions of the gate sizes, (4.6) is a tough problem.

A commonly used heuristic to solve it is as follows. Each gate is resized to locally minimize the objective while all the other gates in the neighborhood and in the fanout cone are assumed to have a fixed size. Gates are traversed in the forward topological order from the timing start points to the timing end points. A gate is resized only after all its fanin gates have been resized. While resizing a gate, the LRS cost is estimated for all the valid cell alternatives[1] of that gate and the gate is resized to the cell with the lowest estimated LRS cost.

[1]A cell alternative is invalid if and only if it causes a violation of the maximum load or the maximum slew.

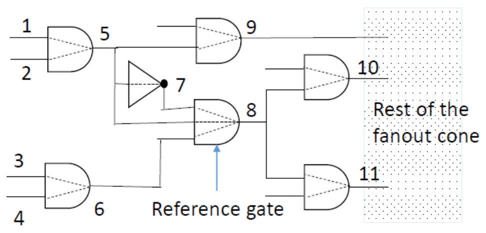

Figure 4.1: Local arcs include fanin-arcs: 1-5, 2-5, 5-7, 3-6, 4-6; gate-arcs: 7-8, 5-8, 6-8; fanout-arcs: 8-10, 8-11; and, side-arcs:5-9.

The exact computation of the LRS cost necessitates exact computation of the lambda-delay sum for which an incremental timing analysis is needed which becomes prohibitively expensive because it has to be done for several cell alternatives of every gate in the design, in each iteration. To limit the runtime overhead when analyzing the cell alternatives, timing of only the *local arcs* is recomputed. Therefore, the lambda-delay sum for a cell x_g of gate g is estimated as follows:

$$\textit{lambda-delay-sum}(x_g) \approx \sum_{(i,j)\in local\_arcs(g)} (\lambda_{ij} \times d_{ij}(\boldsymbol{x})) \tag{4.8}$$

where local-arcs constitute fanin-arcs, gate-arcs, side-arcs and fanout-arcs as shown in Figure 4.1 for an example reference gate.

To improve the lambda-delay sum estimate, Flach et al. Flach *et al.* (2014) computed global delay by slew sensitivity functions to approximate the change in the lambda-delay sum for the rest of the fanout cone. We implemented the sensitivity functions and found that with the cell library provided for the ISPD gate sizing contests it had a negligible impact on the final solution quality. Mostly, the sensitivity function contributed less than 0.1% of the LRS cost. The LRS solver is the most expensive sub-block in the entire algorithm. However, owing to the local nature of the cost computation, several gates can be simultaneously resized.

4.4.2.2 Lagrange Multiplier Update Strategy

The Lagrange multiplier indicates the timing criticality of the corresponding timing arc. A timing arc is timing critical if and only if it's slack is negative. Therefore, a general strategy for updating the multipliers has been to increase (decrease) the multiplier for a timing critical (non-critical) arc in such a way that in the next solving of the LRS, delays of the critical arc may decrease and timing slack of the non-critical arc may be traded off for lower power. The key is how much to change the multiplier. It has a major impact on the convergence rate, as explored in our previous work Sharma *et al.* (2017). Afterwards, the multipliers are projected to satisfy the flow constraints (4.5). To do that, at each node i, the sum of the multipliers on the outgoing timing arcs is distributed among the incoming timing arcs in proportion of their respective multiplier.

4.4.3 Greedy post-pass

Due to discreteness of the problem and the heuristic nature of the LDP solver, LR gate sizers are not able to recover power in a finely grained manner. Note that solving the LRS does not guarantee a zero TNS design. When the LDP solver terminates, there is usually a small TNS. Therefore, a post-pass is needed to recover finer grained timing violations and power. Typically, such a post-pass invokes variants of sensitivity guided greedy heuristics. Ideally, the greedy post-pass should not consume much runtime but on certain benchmarks it can be very slow either due to the size of the benchmark or difficulty in recovering the timing violations.

4.5 Overall Flow

In this section, we discuss various stages constituting our proposed gate sizer. Figure 4.2 shows the flow chart of our sizer. In the first stage, it initializes the sizes and Lagrange multipliers, as discussed in Section 4.4.1. Once load and slew violations are fixed during initialization, they are not allowed afterwards.

The next stage is calibration of effective capacitance. We observed that total capacitance over-estimates the delay to the extent that on some benchmarks the timing does not converge. In this stage, the sizer invokes the external timer and updates the parameters of our proposed effective capacitance model for each interconnect (discussed in Section 4.6). It is skipped for the designs where interconnect is purely capacitive on the ISPD 2012 benchmark designs.

In the third stage, the LDP is solved in two sub-stages: LR timing recovery and LR power recovery. LR timing recovery is responsible for recovering the bulk of the timing violations. It iterates between the LRS

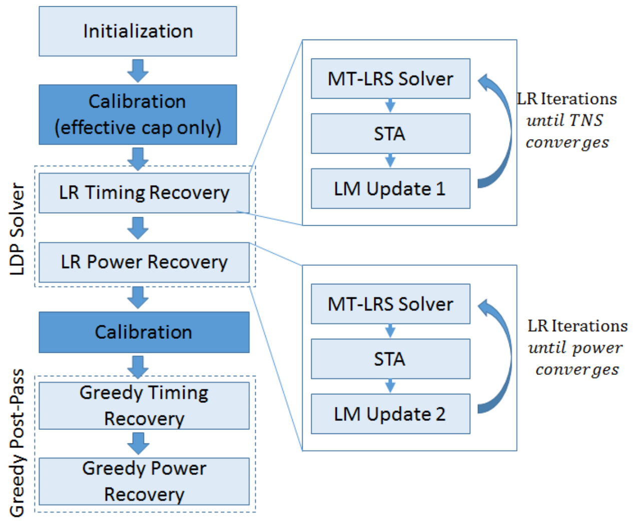

Figure 4.2: Flowchart of our proposed gate sizer. Calibration is invoked for distributed RC interconnect model. Our LRS solver is multithreaded (MT). There are two differences between LR timing and power recovery: (a) convergence criterion, and (b) parameters used for updating Lagrange multipliers (LM).

solver, timing update via static timing analysis (referred as STA) and Lagrange multiplier update. Iterates continue until TNS converges. LR power recovery follows the same iterative structure. However, a different set of parameters is used for the Lagrange multiplier update to drive the optimization towards lower power design. Iterations terminate when power has converged. Iterations inside the LR timing and power recovery are referred to as LR iterations. We set a maximum limit of 200 LR iterations for each of the sub-stages.

Livramento et al. Livramento *et al.* (2014) had shown larger reduction in leakage on the ISPD 2012 benchmark designs by slightly increasing (in other words, relaxing) the target clock period during the LR iterations. Therefore, to enable timing violation and power trade-off during the LR iterations, we introduce a parameter R called the target clock relaxation factor which is indicated in %. For example, if $R = 0.2\%$ and $T = 300ps$, then the relaxed target clock period is 300.6ps. R is reset to 0 after the LDP solver ends.

Our LRS solver differs from a typical LRS solver that was discussed in Section 4.4.2.1 in the following ways. Firstly, while resizing a gate we restrict the negative slack degradation around that gate. This idea and technique was proposed by Flach *et al.* (2014) who suggested that in addition to minimizing the LRS cost, its important to prevent timing degradation to ensure solution stability for faster convergence. The degradation in local negative slack was used as a proxy for the actual degradation in the TNS. We refer to this technique as *local slack check*. We found the local slack check to be quite effective in keeping TNS under control, especially during the LR power recovery. Secondly, our LRS solver is multi-threaded. Thirdly, our LRS solver evaluates a reduced number of valid cells. The last two differences significantly speedup our LRS solver without compromising the solution quality.

After the LR power recovery ends, the sizer can immediately start the greedy post-pass if it has an accurate internal timer. However, with resistive interconnect, it is CPU intensive to accurately model the timing and quite difficult to correlate well with an industrial sign-off timer. Hence, for such designs, as in the ISPD 2013 benchmark suite, we propose to do a full calibration of our internal timing models. In this stage, the parameters for our effective capacitance, delay, and slew models are updated. Thus, equipped with a better timing model, the sizer can then invoke the greedy post-pass. In this last stage, the sizer greedily recovers the timing by upsizing the gates, and then recovers power by increasing the Vth and downsizing the gates.

4.6 Timing Models With Resistive Interconnect

Unlike the ISPD 2012 gate sizing contest where the interconnect of each benchmark was modeled by a single lumped capacitance, in ISPD 2013, the interconnect was modeled by a fixed distributed RC tree. Due to resistive interconnect, not only the interconnect delay and slew degradation need to be modeled, even gate delay and gate slew cannot be computed from the table look-up based on the total capacitance loading the gate output. Gate delay is a function of effective capacitance which is the same as total capacitance only when interconnect is purely capacitive (lumped). To accurately model the gate slew under different loads and input slews, timer must be able to account for the exponential tail in the output voltage waveform.

For purely capacitive interconnects, our internal timer correlates well with Synopsys PrimeTime which is de facto standard timing analyzer used in industry, within 0.001ps. For resistive interconnects, implementing a timer of this accuracy is beyond the scope of this work, and calling an external timer frequently is prohibitively slow. For this reason, Flach *et al.* (2014) invoked the external timer only during the greedy post-pass. During LR iterations they only model the effective capacitance.

To speedup the timing updates we propose a simple model to estimate the effective capacitance and propose mechanisms to calibrate our effective capacitance model along with the other timing models - the delays and slews for gates as well as interconnects. For calibrating a basic timer against a sign-off timer Moon *et al.* (2010) had proposed to additively adjust the endpoint slacks. As shown by Kahng *et al.* (2013), this mechanism requires very frequent calibrations to keep the average slack errors low. Therefore, for better accuracy we propose to calibrate each timing model separately. We use Synopsys PrimeTime for timing calibration as it was also used in the gate sizing contests.

Notations used in the subsequent subsections are as follows. An interconnect (also referred as net) is an RC tree composed of taps and wire segments connecting two adjacent taps. The tree of a net n is denoted by T_n. Each tap, except the root tap, has exactly one parent. The root tap has no parent. Figure 4.3 shows an example net modeled by an RC tree.

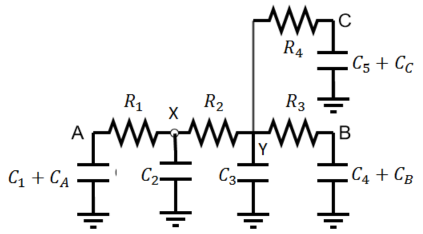

Figure 4.3: An example net modeled by an RC tree. Pin A is the driver and is also the root tap. Pins B and C are loads and are also the leaf taps.

Capacitance associated with each tap i is denoted by cap_i. For those taps that are connected to an input pin of a gate, referred to as leaf taps (or, leaves), cap_i includes the input pin's capacitance as well. The subtree rooted at tap i is denoted as $subtree(i)$. The sum of all capacitances in $subtree(i)$ is referred to as downstream capacitance at tap i, and it is denoted as c_i. It is defined as follows:

$$c_i = \sum_{j \in tap(subtree(i))} cap_j$$

where $tap(.)$ returns all the taps. For an example, in reference to Figure 4.3, $cap_Y = C_3$ and $c_Y = C_3 + C_4 + C_5 + C_B + C_C$.

Resistance of the wire segment connecting the parent of tap i, $parent(i)$, to tap i is referred to as the upstream resistance at tap i, and it is denoted by r_i. The path connecting the root of the tree to tap i is denoted by $path(i)$. The sum of all the resistances on $path(i)$ is referred to as the path resistance to tap i, and it is denoted by r_{oi}. It is formally defined as follows:

$$r_{oi} = \sum_{j \in tap(path(i))} r_j$$

In reference to Figure 4.3, $r_B = R_3$ and $r_{oB} = R_1 + R_2 + R_3$.

4.6.1 Modeling Effective Capacitance

Resistances in the interconnect shield the capacitances due to which the output of the gate gets charged faster. Hence, the effective capacitance seen by a driver is usually smaller than the total capacitance. The larger the resistance the more shielding there is, and the smaller is the effective capacitance. With this intuition, we propose the following model for estimating effective capacitance of net n,

$$\mathit{ceff}(n) = \sum_{i \in tap(T_n)} \frac{cap_i}{1 + \alpha_n \times r_{oi}/R_d} \tag{4.9}$$

where α_n is a net-specific non-negative parameter and R_d is the drive resistance of the arc that drives the net n. If o_n is the node corresponding to the root tap of the net, then *ceff*(n) is same as the effective capacitive load at node o_n that was used in section 4.3. The default value of α_n is 0 for all nets, in which case, effective capacitance is same as the total capacitance. During calibration it is updated so that the estimated effective capacitance matches the effective capacitance obtained from the industrial timer for the same input conditions, namely, the input slew and the input pin capacitance at all the leaves of the net. We refer to $1/(1 + \alpha_n \times r_{oi}/R_d)$ as the shielding factor at the tap i. As per our model, the larger the path resistance to tap i (r_{oi}), the more the shielding effect on the capacitance at that tap (cap_i), and the smaller the contribution of that capacitance towards the effective capacitance (*ceff*(n)). We compute the drive resistance R_d from the delay lookup table as the ratio of the change in delay to the change in load, averaged over all input transition times and all loads. For every delay table, we derive a single R_d value.

To test the accuracy of our proposed model, we compared the error in effective capacitance before calibration and after calibration. After calibration, we randomly perturbed the design. We either upsized or downsized a pre-determined percentage of gates, P, chosen uniformly at random. Before calibration, we observed that the average error was up to 4.4% and the standard deviation was up to 17%, across all the

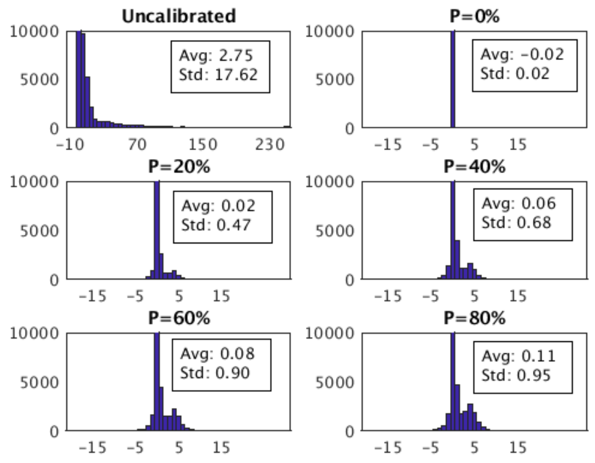

Figure 4.4: Histograms of effective capacitance error for the design *matrix_mult* before calibration (Uncalibrated) and after calibration for different amounts of random perturbations, P, measured as percentage of total number of combinational gates. X-axis is the % error and Y-axis is the count of interconnects. Positive error implies over-estimation.

designs. The maximum error was more than 500%. Immediately after calibration, as expected, we observed near-zero error across all designs. As perturbations increased up to $P = 80\%$, the average error, across all the designs, was less than 0.15% and the standard deviation was less than 1.6%. Maximum error observed was 137%. Figure 4.4 shows histograms of errors in effective capacitance for the design *matrix_mult* before calibration and after calibration for five different values of P.

4.6.2 Modeling Slew of an RC Tree

Kashyap *et al.* (2003) had proposed that the output slew of an RC tree can be approximated as the root-mean square of the input slew and the step response slew. Using the single dominant pole approximation, the step response slew at a tap i can be expressed as the Elmore slew metric Bakoglu (1990) i.e., $(ln4) \times d_i$ for 20-80% transition, where d_i is the Elmore delay from the root of the RC tree to the tap i. Thus, we can write the 20-80% ramp response slew at tap i as follows,

$$s_i = \sqrt{s_o^2 + 1.92 \times d_i^2} \tag{4.10}$$

where s_o is the slew at the root of the net and $(ln4)^2 = 1.92$. The same model was used by Flach *et al.* (2014) as well.

We propose an alternative slew model for computing slew at the leaves. We replace the constant term (1.92) by a leaf specific *slew correction factor*, denoted by scf_i for leaf i. Thus, we can estimate the slew at leaf i as follows,

$$\hat{s}_i = \sqrt{s_o^2 + scf_i \times d_i^2} \tag{4.11}$$

Note that since square rooting is a CPU intensive operation, it suffices to compute slew only at the leaves. By default $scf_i = 1.92$ for each leaf i. During calibration it is updated for each leaf such that the estimated slew at a leaf exactly matches the corresponding slew from the PrimeTime under the same input conditions. Although, theoretically, we can define a correction factor for every tap of the tree, since PrimeTime does not report their slews, it is not straightforward to compute the corresponding correction factors.

4.6.3 Modeling Delay of an RC Tree

Elmore delay has been a popular delay metric for RC tree due to simplicity of its computation as a function of circuit parameters. Gupta had established Elmore delay as an upper bound on 50% delay of an RC tree response for a wide category of input signals including the ramp and the step input Gupta (1995). The Elmore delay at the tap i can be written as,

$$d_i = \sum_{j \in tap(path(i))} (r_j \times c_j) \tag{4.12}$$

where r_j and c_j are upstream resistance and downstream capacitance at tap j, respectively. Flach *et al.* (2014) used the same model.

We propose an alternative delay model for computing delay at the leaves. We add a leaf-specific *delay correction factor*, denoted by dcf_i for leaf i. Thus, we propose to estimate the delay at leaf i as follows,

$$\hat{d}_i = dcf_i \times \sum_{j \in tap(path(i))} (r_j \times c_j) \tag{4.13}$$

The default value of dcf_i is 1 for each leaf in every net. During calibration it is updated such that the estimated delay exactly matches the corresponding delay from PrimeTime under the same input conditions. Note that even though we do not compute correction factors for the taps of the tree (for the same above-mentioned reason), we can still compute delay at the leaves using (4.13).

4.6.4 Modeling Gate Slew

For a purely capacitive interconnect, output slew of a gate's timing arc $i \to j$ is a function of the input slew (s_i), total capacitive load at the output (l_j) and the library cell for that gate. It can be computed by linear 2-D interpolation using the appropriate four boundary points in the slew lookup table. Formally, slew at node j due to the timing arc (i, j) can be represented as follows,

$$s_j = SlewTable(s_i, l_j, cell) \tag{4.14}$$

If multiple timing arcs are incident at a node j then, as per the contest guidelines, slew at node j is the maximum of all the slews.

With resistances in the interconnect, the slew may increase or decrease. There are two opposite factors in play. On one hand, lower effective capacitance causes the output to more quickly charge (or, discharge). On the other hand, charge leaking through the resistances, slows down the transition to 20/80 threshold resulting in a long exponential tail in the output voltage waveform. Depending upon the ratio of interconnect impedance to the ground versus the driver resistance, one of these two factors can dominate. Qian *et al.* (1994) had proposed to capture the output waveform using a two-piece approximation - one piece was obtained from the effective capacitance model and the other (the tail portion) was obtained from a resistance model. Since it is quite complicated and CPU intensive to accurately capture both the effects, we propose to use a 'slew correction factor' denoted by scf_j to improve the accuracy of the original model.

$$\hat{s}_j = scf_j \times s_j \tag{4.15}$$

By default, $scf_j = 1$. During calibration it is updated for each node so that the estimated slew exactly matches PrimeTime under the same input conditions.

4.6.5 Modeling Gate Delay

Delay of a gate's timing arc (i, j) can be computed by linear 2-D interpolation in the delay lookup table, as a function of input slew (s_i), effective capacitance of the net being driven by the output node j, denoted as $(\mathit{ceff}(net(j)))$ and the library cell for that gate. Formally, delay of a gate arc (i, j) is defined as follows,

$$d_{ij} = DelayTable(s_i, \mathit{ceff}(net(j)), cell) \tag{4.16}$$

Although d_{ij} is able to accurately model the gate delay, in some cases there are slight deviations. One such case that we observed while working with PrimeTime is when net's impedance to ground is much larger than the drive resistance of the simplified driver model that PrimeTime builds internally for the gate timing arc. In such situations, as per the PrimeTime (2012) the output waveform is not very smooth which potentially reduces the accuracy of delay calculation in the RC tree being driven by the gate. Hence, PrimeTime adjusts the drive resistance to improve accuracy.

Like in gate slew, we propose to use a delay correction factor, dcf_{ij}, for every gate arc (i, j), to account for the above-mentioned deviations. Thus, we estimate the delay as follows,

$$\hat{d}_{ij} = dcf_{ij} \times d_{ij} \tag{4.17}$$

The default value of dcf_{ij} is 1. During calibration it is updated so that the estimated delay exactly matches the PrimeTime under the same input conditions.

4.6.6 Calibration Pseudo Code and Runtime

Calibration is the process of updating α_n for all nets, slew correction factors, and delay correction factors for all leaves as well as gate arcs. Its pseudo code is shown in Algorithm 6. Algorithm writes out a Verilog file *out.v* for the current design (line 1) and then, makes a system call to PrimeTime (line 2) which sources a TCL script. The TCL script reads in *out.v* along with the contest provided timing constraints (.sdc) file, parasitics (.spef) file, and the liberty (.lib) file. It outputs a file containing effective capacitance for every gate output pin and input port, delay for every timing arc, and slew at every pin and port. Since all of these are obtained from PrimeTime, we refer to it as *golden* timing data.

Then, algorithm scans the design in forward topological order, and updates parameters for each gate and its output net. The algorithm queries the golden effective capacitance value on line 6. On line 7, it solves (4.9) for α_n using the bisection method, such that the estimated effective capacitance matches the golden value within a threshold.

The algorithm, then computes delay and slew correction factors for each timing arc of the gate. On line 12, it queries the golden slew at the input pin of the arc. It is used for computing the lookup table delay and slew on lines 13 and 16. For delay computation it re-uses the golden value for the effective capacitance that was queried on line 6. Then, it queries the golden delay for the same arc on line 14, and updates the delay correction factor as the ratio of golden delay and the lookup table delay, as shown on line 15. For computing slew on line 16, total capacitive load is used because effective capacitance severely under-estimates the slew for the given contest Liberty file. It is not clear what PrimeTime uses. On line 17, maximum slew at the output pin is calculated, which is then used to compute the slew correction factor at that output pin, as shown on lines 18-19.

Algorithm 6 Pseudo code for calibration

```
1:  $out.v \leftarrow write\_verilog()$
2:  $golden \leftarrow PrimeTime(out.v, ...)$
3:  for each gate $g$ in forward topological order do
4:      $o = output\_pin(g)$
5:      $n = net(o)$                                        ▷ net driven by pin $o$
6:      $ceff_{PT} = golden.get\_ceff(o)$
7:      Setting $ceff(n) = ceff_{PT}$, solve (4.9) for $\alpha_n$
8:      $x_g = cell(g)$
9:      $l = total\_cap(o)$                                 ▷ total capacitive load at pin $o$
10:     $s_o = 0$                                           ▷ slew at pin $o$
11:     for each input pin $i$ of gate $g$ do
12:         $s_i = golden.get\_slew(i)$
13:         Plug $s_i$, $ceff_{PT}$ and $x_g$ into (4.16); compute $d_{ij}$
14:         $d_{PT} = golden.get\_delay(i, o)$
15:         $dcf_{io} = d_{PT}/d_{ij}$
16:         Plug $s_i$, $l$ and $x_g$ into (4.14); compute $s_j$
17:         $s_o = max(s_o, s_j)$
18:     $s_{PT} = golden.get\_slew(o)$
19:     $scf_o = s_{PT}/s_o$
20:     for each leaf $k$ of net $n$ do
21:         compute $d_k$ using (4.12)
22:         $d_{PT} = golden.get\_delay(o, k)$
23:         $dcf_k = d_{PT}/d_k$
24:         $s_k = golden.get\_slew(k)$
25:         Set $s_o = s_{PT}$, $\hat{s}_i = s_k$, $d_i = d_k$ in (4.11); solve for $scf_i$
26:         $scf_k = scf_i$
```

The algorithm, then iterates over each leaf of the output net. For each leaf, it computes Elmore delay using (4.12) on line 21; queries golden delay for the timing arc from the root tap of net to the leaf (line 22); and updates the delay correction factor at the leaf on line 23. On line 24, it queries the golden slew at the leaf. Then, it solves (4.11) for the correction factor such that the estimated slew at the leaf matches the golden value obtained on line 24. Finally, scf_i is assigned to the slew correction factor at the leaf.

The PrimeTime call is the single most runtime expensive operation of not only calibration but the entire gate sizer. We found that the slow TCL interface between our C++ code and PrimeTime is the runtime bottleneck. There are two factors in the interface that slow us down. Firstly, we use TCL commands which do many function calls for every timing query in PrimeTime. If the user has access to the native tool, then timing queries can be much faster. Secondly, we ask PrimeTime to dump out a file with the timing information which incurs significant I/O overhead.

4.7 Fast LRS Solver

Next to calibration, the LRS solver is the most time consuming sub-block across most of the benchmarks. Inside the LRS solver, every gate is resized which requires evaluation of various valid cell alternatives for the LRS cost and the local slack check. As noted in the Section 4.4.2.1, multiple gates can be resized simultaneously and thus, the LRS solver can be multi-threaded. However, not any two gates in any order can be simultaneously resized. In section 2.4 of chapter 2 we had discussed the requirements for simultaneous resizing, and presented two algorithms to enable effective multi-threading.

Reduced Option Evaluation We discussed in section 2.5.1 of chapter 2 that while resizing a gate inside the LRS solver, it is sufficient to evaluate and locally search only among those valid cells that have a similar drive strength as the current cell, rather than globally searching among all the tens of cells provided in the standard cell library. The ISPD contest library has 30 cells for each gate. For the first five iterations of the LR timing recovery we evaluate all the valid cells for each gate. Afterwards, we restrict the search space. However, if the search space is too much restricted, say Vth is not allowed to change, then the algorithm is not able to recover much power. It was empirically observed that evaluating four adjacent sizes - two on each side, in the current Vth cells and in the adjacent Vth cells was sufficient to converge to the lowest power. Across the ISPD 2013 designs, we observed that during the first five iterations, on average 17 valid cells were evaluated per gate per iteration, which reduced to 7 in the later iterations. This sped up the MT-LRS solver by 1.77 times without compromising the quality.

4.8 Lagrange Multiplier Update

The Lagrange multiplier update is very crucial in driving the optimization. Depending upon how quickly Lagrange multipliers change, that determines the rate of TNS convergence in the LR timing recovery and the rate of power reduction in the LR power recovery. In our previous work Sharma *et al.* (2017), we analyzed the impact of the Lagrange multiplier update on the convergence rate. We presented a concise and intuitive expression to update the Lagrange multipliers which decouples the treatment of timing critical and timing non-critical arcs. Our Lagrange multiplier update strategy is shown in Algorithm 7, where D_{ij} is the worst path delay through the arc $i \rightarrow j$, and K is the 'acceleration' factor. The ratio D_{ij}/T indicates the timing criticality of the arc $i \rightarrow j$. Computationally, this ratio is same as $1 + (a_i + d_{ij} - q_j)/T$ when arrival times at the timing start points are all zero. Here, a_i is the arrival time at node i and q_j is the required arrival time at node j. The ratio is more than one for a timing critical arc, so the Lagrange multiplier for such an arc is increased. For a non-critical arc, the ratio is less than one, therefore its multiplier is decreased. The acceleration factor determines how quickly the Lagrange multipliers change. Larger acceleration factors can speedup the convergence but can also cause the solution to get stuck in a worse local minimum. Sections 3.4.2 and 3.4.3 of chapter 3 may be referred for more details.

Algorithm 7 Our proposed Lagrange multiplier update algorithm

```
for timing arc $i \rightarrow j$ do
    $\lambda_{ij} = \lambda_{ij} \times \left(\frac{D_{ij}}{T}\right)^K$
Projection to satisfy KKT constraints. Refer Tennakoon et al. (2002)
```

4.9 Greedy Post-Pass

The design obtained from the LDP solver usually has some timing violations and some more power to be recovered. A greedy post-pass is therefore necessary for finer-grained refinement. It is composed of two algorithms: Greedy Timing Recovery, and Greedy Power Recovery. Unlike previous works Flach *et al.*

(2014); Daboul *et al.* (2018), we use our calibrated internal timing models in this stage. Since on average only 5% gates change size or Vth during this stage, our calibrated internal timing models tend to be quite accurate, and they are much faster to compute than a single invocation of the PrimeTime.

4.9.1 Greedy Timing Recovery

The greedy timing recovery (GTR) is meant to recover the few remaining timing violations. So our algorithm using the internal timing models must guarantee improvement in TNS. Moreover, power expenditure during this stage must be minimized. Consequently, compared to LR timing recovery, it has several differences. Firstly, the gate that has the maximum number of critical paths passing through it is upsized first. Secondly, gates are upsized only to the next available size. Vth swap is not allowed at this stage to prevent excessive power increase. Thirdly, after every upsize, timing of the entire fan-out cone is incrementally updated. This is to ensure that the TNS does not worsen. And fourthly, the new size of the gate is committed if the TNS improves. Otherwise, the sizing and the timing updates are undone.

4.9.2 Greedy Power Recovery

The greedy power recovery (GPR) is meant to recover the power in a finer-grained manner without degrading the timing, as per the internal timing models. Like Flach *et al.* (2014), we also consider all the gates for Vth swap as well as downsizing in forward topological order, and make sure to commit a new lower power cell implementation for a gate if and only if the TNS does not degrade. To compute the change in TNS, an incremental timing update is required. Flach et. al. continued to scan all the gates in the same order as long as even one gate would successfully commit a new size. Since most of the time TNS degrades and the timing updates are undone, this strategy wastes a lot of CPU cycles.

We propose to use the local slack check as a way to predict TNS degradation without having to do expensive incremental timing analysis. If the new cell implementation degrades either the total negative slack or the worst negative slack at the fan-outs then, it is not committed. This simple trick cuts down the average GPR runtime by 13 times without affecting the quality. We also propose *selective downsizing*, which means that not all gates are considered for power recovery in every iteration. Only those gates (and their neighbors) that successfully committed a lower power cell in the previous iteration are considered for power recovery in the next iteration.

To speedup the incremental timing updates we parallelize the incremental STA. The incremental STA, like full STA, computes the timing of each gate and the interconnect driven by that gate in the forward topological order. Gates belonging to the same topological level can be simultaneously processed by multiple threads. However, the parallelism can be very limited depending upon the fan-out cone structure.

4.10 Experimental Results

We implemented our gate-sizer in C++. Experiments are performed on two quad-core Intel(R) Xeon(R) E3-1240 v5 @ 3.50GHz CPUs with an aggregate memory of 16GB. For multi-threading, OpenMP Dagum *et al.* (1998) was used. We use 8 threads for solving LRS and for incremental STA. For calibration, we use 4 threads and invoke a separate PrimeTime license on each one of them to distribute the timing queries. All the results reported in this work are averaged over 10 runs to minimize the bias due to non-determinism caused by multi-threading and random MEE assignment. We used the PrimeTime version E-2010.12 for amd64. For benchmarking, we use the suites provided for the ISPD 2012 and the ISPD 2013 gate sizing contests. However, the C++ application programming interface (API) and the TCL scripts to interface with the PrimeTime are not available. On the ISPD 2013 designs, these interfacing scripts and APIs, and also the PrimeTime version, might make a difference in the gate sizer runtime as well as the final solution quality. Since our internal timer correlates perfectly with PrimeTime on the ISPD 2012 designs, we do not need to communicate with PrimeTime to calibrate for those designs.

4.10.1 Results on the ISPD 2012 Designs

In this subsection, we compare our gate sizing results on the ISPD 2012 contest designs against Li *et al.* (2012) and Flach *et al.* (2014). While Flach et. al. have reported the least leakage power on most of the designs; Li et al. have reported slightly better runtime. Flach et. al. ran their single threaded gate sizer on a 3.40GHz Intel(R) Core(TM) i7-3770 CPU, and Li et. al. ran on six 2.67 GHz two-socket cores with 72 GB memory using 8 threads. Unlike the LR gate sizer of Flach et. al. which does timing as well as power

Table 4.2: Results summary on the ISPD 2012 benchmarks suite Ozdal *et al.* (2012). There are seven designs and per design there are two different timing constraints - ‘slow’ and ‘fast’. The slow constraint has larger target clock period than the fast constraint. Results for Li *et al.* (2012) - referred as [1] in this table, and Flach *et al.* (2014) - referred as [2], are cited from their respective works.

Benchmark	**Comb.**	**Clock**	**Leakage Power (W)**			**Power saved (%)**		**Total Runtime (min)**			**Speedup (X)**	
	Gates	**T, (ps)**	**[1]**	**[2]**	**Our**	**Vs [1]**	**Vs [2]**	**[1]**	**[2]**	**Our**	**Vs [1]**	**Vs [2]**
DMA_slow	23109	900	0.153	0.132	0.135	13.4	-2.2	0.60	0.79	0.07	9.0	11.8
pci_bridge32_slow	29844	720	0.111	0.096	0.098	13.7	-1.7	1.20	0.87	0.09	13.0	9.4
des_perf_slow	102427	900	0.671	0.57	0.583	15.0	-2.3	6.00	25.31	0.24	24.6	103.6
vga_lcd_slow	147812	700	0.375	0.328	0.329	13.8	-0.4	7.80	5.67	0.44	17.9	13.0
b19_slow	212674	2500	0.604	0.564	0.564	7.0	-0.1	10.20	9.15	0.87	11.7	10.5
leon3mp_slow	540352	1800	1.400	1.334	1.332	5.1	0.2	43.80	38.98	2.56	17.1	15.2
netcard_slow	860949	1900	1.780	1.763	1.763	1.0	0.0	48.00	34.39	2.32	20.7	14.8
DMA_fast	23109	770	0.281	0.238	0.247	13.7	-3.7	0.60	0.92	0.09	6.8	10.4
pci_bridge32_fast	29844	660	0.167	0.136	0.135	23.3	0.4	1.20	0.92	0.12	9.8	7.5
des_perf_fast	102427	735	1.930	1.395	1.446	33.5	-3.5	6.60	16.37	0.29	22.6	56.0
vga_lcd_fast	147812	610	0.460	0.413	0.418	10.1	-1.1	10.20	8.37	0.62	16.5	13.5
b19_fast	212674	2100	0.784	0.717	0.715	9.7	0.3	12.00	11.75	1.08	11.1	10.9
leon3mp_fast	540352	1500	1.640	1.443	1.444	13.6	0.0	54.60	46.62	3.70	14.8	12.6
netcard_fast	860949	1200	2.180	1.841	1.842	18.3	-0.1	88.80	47.41	3.56	24.9	13.3
Average			**0.895**	**0.784**	**0.789**	**13.7**	**-1.0**	**20.83**	**17.68**	**1.15**	**14.7**†	**15.3**†

† Geometric mean. The geometric mean has been used to compare speedups due to the wider range in values.

recovery, the gate sizer of Li et. al. uses the LR formulation for unconstrained delay minimization and then uses a network flow based approach for recovering the leakage power.

During the LR timing recovery, we set K to 4 and 1 for critical and non-critical arcs, respectively and during the LR power recovery, we set it to 1 and 6 for critical and non-critical arcs, respectively. We do not relax the target clock, i.e, $R = 0$. These parameters are uniformly applied to all the ISPD 2012 designs. The LR timing recovery iterations are terminated when the TNS falls below 20% of the relaxed target clock period. The LR power recovery iterations are terminated when the improvement in power is less than 0.1% compared to the previous iteration. Our executable for the ISPD 2012 designs does not include data structures for RC tree interconnect and it skips the function calls for computing the effective capacitance and for timing propagation in the interconnects.

Table 4.2 summarizes the power and runtime results from the above-mentioned three gate sizers. All final designs from all three gate sizers satisfy the timing constraints. Compared to Li et. al., our designs have 13.7% lower leakage power on average and our gate sizer is 14.7x faster. Compared to the gate sizer of Flach et. al., our gate sizer is 15.3x faster and yields designs that are slightly more leaky, 1.0% on average. As seen from the table, compared to Flach et. al., the three largest designs have very little power difference, in fact we achieve lower power in several cases.

The main reasons for the runtime speedup are 1) multi-threading of the LRS solver, which is the single most time consuming sub-block in the gate sizer; 2) rapid convergence of timing and power recovery due to our Lagrange multiplier update strategy. On average over all the designs, LR timing recovery takes 3 iterations and LR power recovery takes 13 iterations to converge. In our previous work Sharma *et al.* (2015), we did detailed analysis of the speedup due to multi-threading along with the thread overhead, scalability, etc. Compared to our previous work, in this work we have a better Lagrange multiplier update strategy and a more exhaustive GPR as we now allow Vth swap. Consequently, compared to Sharma *et al.* (2015), our current gate sizer is faster and yields designs with 1.4% lower power.

Table 4.3: Results summary on the ISPD 2013 benchmarks suite Ozdal *et al.* (2013). Results for Flach *et al.* (2014) (experiments with hard runtime limit) - referred as [1] in this table, and Kahng *et al.* (2013) (experiments with fast mode) - referred as [2], are cited from their respective works. Dashes indicate that the results were not reported.

Benchmark	**Comb.**	**Clock**	**Leakage Power (W)**			**Power diff (%)**		**Our**	**Total Runtime (min)**			**Speedup (X)**	
	Gates	**T, (ps)**	**[1]**	**[2]**	**Our**	**Vs [1]**	**Vs [2]**	**TNS (ps)**	**[1]**	**[2]**	**Our**	**Vs [1]**	**Vs [2]**
usb_phy_slow	510	450	0.001	0.001	0.001	-0.2	0.4	0	0.49	0.13	0.22	2.2	0.6
pci_bridge32_slow	27244	1000	0.057	0.060	0.058	-1.0	4.6	12	10.53	1.80	0.97	10.9	1.9
fft_slow	30782	1800	0.087	0.096	0.088	-1.9	9.1	2	25.71	3.48	1.37	18.8	2.5
cordic_slow	41673	3000	0.271	0.395	0.293	-7.6	34.8	128	69.04	25.70	2.29	30.2	11.2
des_perf_slow	104310	1300	0.330	0.392	0.332	-0.6	17.9	56	132.27	19.23	7.27	18.2	2.6
edit_dist_slow	121004	3600	0.425	0.488	0.440	-3.3	10.9	562	123.9	31.60	4.92	25.2	6.4
matrix_mult_slow	153542	2800	0.444	0.562	0.448	-0.9	25.4	95	226.13	77.30	8.80	25.7	8.8
netcard_slow	884427	2400	5.155	5.184	5.170	-0.3	0.3	17	483.55	148.60	24.67	19.6	6.0
usb_phy_fast	510	300	0.002	0.002	0.002	-1.8	1.1	1	0.42	0.14	0.23	1.8	0.6
pci_bridge32_fast	27244	750	0.085	0.105	0.090	-5.0	17.0	64	22.62	2.40	1.54	14.7	1.6
fft_fast	30782	1400	0.194	0.342	0.213	-8.7	60.8	14	40.43	5.50	1.64	24.7	3.4
cordic_fast	41673	2626	1.001	–	1.080	-7.3	–	254	117.08	–	5.66	20.7	–
des_perf_fast	104310	1140	0.649	–	0.639	1.5	–	434	347.87	–	26.16	13.3	–
edit_dist_fast	121004	3000	0.540	0.690	0.549	-1.7	25.7	1313	352.96	42.20	6.66	53.0	6.3
matrix_mult_fast	153542	2200	1.611	–	1.633	-1.3	–	499	395.96	–	13.94	28.4	–
netcard_fast	884427	2000	5.200	5.277	5.205	-0.1	1.4	143	400.89	190.77	30.60	13.1	6.2
Average			**1.003**	–	**1.015**	**-2.5**	**16.1**	**225**	**171.87**	–	**8.56**	**15.5**[†]	**3.2**[†]

[†] Geometric mean. The geometric mean has been used to compare speedups due to the wider range in values.

4.10.2 Results on the ISPD 2013 Designs

Compared to the ISPD 2012 gate sizing contest, the major difference in the ISPD 2013 contest was that the interconnects were modeled by distributed RC trees instead of a lumped capacitance. We compare our results against Flach *et al.* (2014) and Kahng *et al.* (2013). While Flach et. al. reported the least leakage power on average over the ISPD 2013 designs as well, Kahng et. al. reported lower runtimes and larger power. Li *et al.* (2012) did not have results on the ISPD 2013 designs. The gate sizer from Kahng et. al. performs randomized multi-starts and uses several greedy sensitivity based heuristics for timing and power recovery. They use up to 16 threads.

For the ISPD 2013 designs, during the LR timing recovery, we set K to 1 and 0.25 for critical and non-critical arcs, respectively and during the LR power recovery, we set it to 1 and 4. Above mentioned values of K yielded netlists with the least leakage power on average. Using larger K during LR timing recovery although we observed faster convergence, the final netlists had slightly more leakage at the end. We relax the target clock period during the LR iterations by 0.5% ($R = 0.5$). On the ISPD 2013 designs, a non-zero relaxation in the target clock helps improve the power at the expense of some runtime. We later analyze these trade-offs. These parameters are uniformly applied to all the ISPD 2013 designs. The LR timing recovery iterations are terminated when the TNS falls below 20% of the relaxed target clock period. The LR power recovery iterations are terminated when the improvement in average power is less than 0.1%. Averaging is done over three consecutive iterations. Our executable for the ISPD 2013 designs has the necessary data structures and built-in functionalities to support gate sizing with interconnects modeled as fixed distributed RC trees.

Table 4.3 summarizes the results on the ISPD 2013 gate sizing contest designs. Compared to Flach et. al., our gate sizer is 15.5x faster and the final optimized netlists have 2.5% more leakage power, on average. Compared to the leakage power reported after the LR iterations by Flach *et al.* (2014), on average, our designs have a lower power after LR iterations even with $R = 0$. However, we do not have the PrimeTime TNS after LR iterations for the gate sizer of Flach et. al. The differences in the leakage power might be because our greedy post-pass algorithms use the less accurate timing models. We invoke PrimeTime only once before the greedy post-pass to calibrate our internal timing models. Whereas, Flach et. al. use accurate timing from PrimeTime to optimize their designs during the post-pass. Other than the timing models, the difference in PrimeTime versions might also cause timing mis-correlation which in turn may prevent further leakage power optimization. As we trade-off some timing accuracy for large runtime speedups, our final optimized netlists may have small constraint violations. As shown in Table 4.3, the average TNS in the final optimized netlists is 225ps which, on average, is less than 10% of their respective target clock periods. While there are no maximum load violations on any design, we observed total slew violations of 31ps only on *edit_dist_**. These minor TNS and slew violations can be recovered by other optimization techniques like buffer insertion, sequential gate optimization, etc. Flach et. al. reported zero violations.

As shown in Table 4.3, compared to the fast mode results of Kahng *et al.* (2013), on average our gate sizer achieves 16.1% lower leakage power and is 3.2x faster. Kahng et. al. do not report their results for all the designs with the fast timing constraint. Recently Daboul *et al.* (2018) published results of their gate sizer on the ISPD 2013 designs. Using IBM's EinsTimer, they showed significant power improvements on several larger design compared to Flach et. al. Since different timers tend to have timing mis-correlations (as seen by significant TNS violation on their final optimized netlists when we measured in PrimeTime), we cannot make a fair comparison. However, for the sake of completeness, compared to their results, our average runtime is 6.8 times smaller but power is 1.4% larger.

Apart from the multi-threaded LRS solver and Lagrange multiplier update strategy, the main reasons for the runtime speedup on the ISPD 2013 designs are: 1) we use computationally much simpler model for the effective capacitance; 2) we call PrimeTime less frequently; and 3) GPR is quite run-time efficient.

4.10.3 Breakdown of Results for the ISPD 2013 Designs

Table 4.4 shows the breakdown of the total runtime for all the ISPD 2013 designs. On average, 18% of the total runtime is consumed in solving the LDP. Inside the LDP solver, the MT-LRS solver dominates the LDP solver runtime, and even more so on the ISPD 2013 designs because of the additional runtime to compute the interconnect delay and slew degradation whenever a cell is evaluated for the LRS cost.

Both the calibration stages combined together account for 46% of the total runtime, on average. This includes the total runtime of two PrimeTime calls. Most of the calibration runtime is spent in outputting the

Table 4.4: Fraction of the total runtime consumed by different blocks of our gate sizer on the ISPD 2013 designs. 'Calibrations' include both the calibration stages. Maximum in each row is in **bold** font.

Benchmark	**LDP Solver**	**Calibrations**	**GTR**	**GPR**
usb_phy_slow	0.00	**0.94**	0.00	0.00
pci_bridge32_slow	0.13	**0.73**	0.01	0.00
fft_slow	0.26	**0.58**	0.04	0.00
cordic_slow	0.33	**0.40**	0.18	0.01
des_perf_slow	0.12	0.25	**0.56**	0.00
edit_dist_slow	0.15	**0.45**	0.28	0.00
matrix_mult_slow	**0.31**	0.29	**0.31**	0.01
netcard_slow	0.14	**0.62**	0.05	0.00
usb_phy_fast	0.02	**0.91**	0.00	0.00
pci_bridge32_fast	0.20	**0.46**	0.25	0.03
fft_fast	0.26	**0.48**	0.15	0.01
cordic_fast	0.31	0.16	**0.40**	0.13
des_perf_fast	0.08	0.07	**0.85**	0.02
edit_dist_fast	0.21	0.33	**0.37**	0.01
matrix_mult_fast	0.24	0.19	**0.48**	0.05
netcard_fast	0.18	**0.50**	0.17	0.00
Average	0.18	**0.46**	0.26	0.02

timing data into a file through a slow TCL interface. We observed that on bigger designs, PrimeTime spends only 15-17% of the calibration runtime to update the timing, the rest is spent in the file I/O. When there is an efficient way to interface with the PrimeTime, say if the user has access to the source code APIs which is possible with the commercial licenses, then the calibration can be made roughly 5 times faster.

GTR and GPR account for 26% and 2% of the total runtime, respectively. On des_perf and matrix_mult designs, as shown in table 4.4, GTR takes relatively more time to fix the timing violations. It may happen if several upsizing attempts need to be undone due to TNS degradation. On cordic and matrix_mult, GPR spends a significant runtime and recovers a lot of leakage power, especially for the fast timing constraints.

Table 4.5 shows the leakage power and the PrimeTime TNS after different stages of our gate sizer. After the LR iterations terminate, the average TNS is 8698ps. The GTR using our internal timing models reduces it to 43ps and increases the leakage power by 1.8%. Then, the GPR reduces the leakage power by 3.2%. In the process, the actual TNS increases to 225ps because our timing models slightly under-estimate the delay along the paths from where the power was recovered.

Table 4.5: Power and TNS after different stages of our gate sizer for the ISPD 2013 contest designs. The absolute value of power after LR iterations and subsequent incremental changes by GTR and GPR algorithms are shown below, along with the PrimeTime TNS after each stage.

Benchmark	**After LR**		**After GTR**		**After GPR**	
	Power (W)	**TNS (ps)**	**△ Pow (%)**	**TNS (ps)**	**△ Pow (%)**	**TNS (ps)**
usb_phy_slow	0.001	11	0.7	0	-0.4	0
pci_bridge32_slow	0.057	799	0.5	8	-0.2	12
fft_slow	0.088	2593	1.4	0	-0.9	2
cordic_slow	0.296	7846	3.4	34	-4.2	128
des_perf_slow	0.328	10859	2.6	33	-1.1	56
edit_dist_slow	0.439	21596	0.5	30	-0.3	562
matrix_mult_slow	0.454	7643	1.6	16	-2.8	95
netcard_slow	5.169	3861	0.0	7	0.0	17
usb_phy_fast	0.002	11	2.5	0	-0.5	1
pci_bridge32_fast	0.088	2570	3.3	64	-1.4	64
fft_fast	0.209	3471	4.9	8	-2.9	14
cordic_fast	1.273	9297	2.0	109	-16.8	254
des_perf_fast	0.648	8743	2.5	216	-3.8	434
edit_dist_fast	0.551	25664	1.0	25	-1.3	1313
matrix_mult_fast	1.859	8163	2.2	81	-14.1	499
netcard_fast	5.195	26044	0.2	54	0.0	143
Average	**1.041**	**8698**	**1.8**	**43**	**-3.2**	**225**

4.10.4 Impact of Relaxing the Target Clock Period

The idea behind relaxing the target clock period during the LR iterations is based on the observation that the greedy timing recovery is able to effectively recover large timing violations without expending much power. Hence, we experimented with different values of relaxation factor, R. Table 4.6 shows various metrics of interest as R is increased from 0 to 0.2 to 0.5 to 0.8. Results have been averaged over all the ISPD 2013 designs. As R increases, the power after LR iterations improves by up to 4.3% at the cost of 7x more TNS which in turn increases power expenditure during GTR as well as the runtime of GTR. We observed that $R = 0.5$ resulted in the minimum average power with a 6x slowdown in the greedy timing recovery which resulted in an overall 1.4x slowdown in the gate sizer. Although the average power with $R = 0.5$ is only 0.5% better compared to $R = 0.0$, on cordic_fast power reduced by 5% for a similar TNS as the end. As R increases, runtime of the greedy timing recovery increases a lot on the larger designs.

Table 4.6: Impact of different R values on various metrics of interest. Results have been averaged over all the ISPD 2013 designs, and then normalized. Reference values for normalization are highlighted below.

R	**After LR**		**After GPR**		**Runtime**	
	Power	**TNS**	**Power**	**TNS**	**GTR**	**Total**
0.0	**1.000**	**1.000**	0.966	0.140	**1.0**	11.5
0.2	0.988	1.969	0.963	0.117	2.0	12.3
0.5	0.971	4.295	0.961	0.097	6.3	15.6
0.8	0.957	7.512	0.962	0.189	19.5	29.3

4.10.5 Effectiveness of Calibration

In our proposed flow, we calibrate our timing models twice. The first time, before starting the LR iterations, we only calibrate the effective capacitance; and the second time, after LR iterations terminate, we calibrate all the timing models. In this subsection, we analyze the impact of both of these calibration steps on the final solution quality. We consider two separate flows. The first flow is without any calibration step. We refer to it as ‘NoCalb’. The second flow calibrates only effective capacitance both times. We refer to it as ‘CeffCalb2’. Table 4.7 reports the final power and PrimeTime TNS for NoCalb as well as CeffCalb2. Power numbers have been normalized with respect to the power obtained from our proposed flow (column six in table 4.3).

Compared to our proposed flow, we observe that NoCalb produces designs that are up to 222% and on average 16% more leaky but have smaller TNS. Without calibration, effective capacitance is over-estimated due to which the LR iterations have a hard time fixing the TNS and therefore, gates get over-sized. Therefore, calibration of effective capacitance before LR iterations is necessary, but not sufficient. We do not calibrate delay and slew model parameters in the first calibration because those parameters are not robust (in terms of model accuracy) to the many changes in the design during LR iterations. We observed that calibrating those parameters misleads the optimization during LR iterations. Flach et. al. also, used a more accurate model only for effective capacitance during LR iterations.

With CeffCalb2 we observe that it produces designs that have better power but much worse TNS compared to our proposed flow. For example, as shown in Table 4.7, for *matrix_mult_fast* power is 5% lower but TNS is more than 10ns. This is due to delay under-estimation. Table lookup mostly under-estimates the gate slew which in turn under-estimates the fan-out gate delays. The default parameter values of our interconnect delay and slew model tend to under-estimate the respective metric. Hence, a complete calibration is necessary to improve the accuracy of our internal timer before starting the greedy post-pass.

Table 4.7: Final power and PrimeTime TNS for 'NoCalb' flow and 'CeffCalb2' flow. NoCalb refers to the flow without calibration, and CeffCalb2 refers to the flow where only effective capacitance is calibrated both times. The Power is normalized with respect to 'Our' results reported in column six of Table 4.3.

Benchmark	**NoCalb**		**CeffCalb2**	
	Power	**TNS (ps)**	**Power**	**TNS (ps)**
usb_phy_slow	1.001	0	1.004	0
pci_bridge32_slow	1.004	2	1.018	539
fft_slow	0.997	339	1.032	1061
cordic_slow	1.013	232	1.123	10531
des_perf_slow	1.197	0	0.979	28063
edit_dist_slow	1.031	0	1.066	22139
matrix_mult_slow	1.048	0	1.000	6947
netcard_slow	1.000	0	1.005	3207
usb_phy_fast	0.988	2	1.030	5
pci_bridge32_fast	1.059	13	1.075	2489
fft_fast	1.001	333	1.165	3314
cordic_fast	1.079	377	1.093	16837
des_perf_fast	2.222	297	0.902	38362
edit_dist_fast	1.145	0	1.030	42567
matrix_mult_fast	1.732	97	0.950	10773
netcard_fast	1.001	5	1.000	29824
Average	**1.157**	**106**	**1.029**	**13541**

4.10.6 Multi-Threading Different Sub-blocks of the Gate Sizer

In our gate sizer, we have implemented a multi-threaded version of LRS solver, LM update, STA, calibration, and incremental STA. While each one of them improves the runtime to a smaller or larger extent, none of them affect the quality of results. We ran the LDP solver with different number of multiple threads ranging from 2 to 12. Speedups in LDP solver runtime versus the number of threads for various ISPD 2013 designs are shown in Figure 4.5. With 2 threads we observed a near ideal speedup in LDP solver runtime. As the number of threads increased from 4 to 8 to 12, average speedup increased from 3.4x to 6.4x to 9.5x. Due to the randomness in our gate sizer - on account of multi-threading and random mutual exclusion edge assignment, and variability in the server load - sometimes super-linear speedups are also observed.

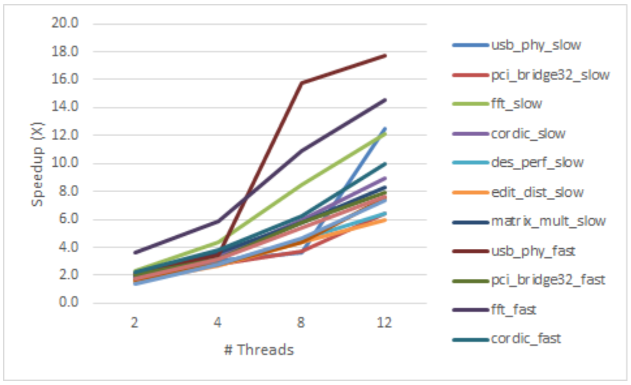

Figure 4.5: Speedup in LDP Solver runtime on ISPD 2013 benchmarks with different number of threads with respect to single thread.

4.11 Conclusion

Today's designs with millions of gates require very fast gate-sizing and threshold voltage assignment, as it is a crucial circuit optimization that is performed at multiple steps in the design flow. Due to the overheads for timing accurate modeling of resistive interconnect, timing updates are the runtime bottleneck. We extended our multi-threaded Lagrangian relaxation based gate sizer to include support for very fast gate sizing of the designs with resistive interconnect. We proposed a simple model for quickly computing the effective capacitance and several calibration mechanisms to improve the accuracy of our internal timing models. We have shown that our gate sizer quickly recovers the timing violations and the leakage power while minimally interacting with the external timer and demonstrated the effectiveness of each calibration step in improving the timer accuracy. Compared to the state-of-the-art (both in runtime as well as power) gate sizer Flach *et al.* (2014), on the ISPD 2013 discrete gate sizing contest benchmark designs, our gate sizer is, on average, 15.5x faster. The optimized designs have only 2.5% higher leakage power and small timing violations.

References

Abbaspour *et al.*, S. (2003). Calculating the effective capacitance for the rc interconnect in vdsm technologies. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, pages 43–48. ACM.

Alpert *et al.*, C. J. (2000). A two moment rc delay metric for performance optimization. In *Proceedings of the 2000 international symposium on Physical design*, pages 69–74. ACM.

Bakoglu, H. B. (1990). Circuits, interconnections, and packaging for vlsi.

Chen *et. al.*, C.-P. (1999). Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):1014–1025.

Daboul *et al.*, S. (2018). Provably fast and near-optimum gate sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Dagum *et al.*, L. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55.

Elmore, W. C. (1948). The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of applied physics*, 19(1):55–63.

Flach *et al.*, G. (2014). Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557.

Gupta, R. (1995). The elmore delay as a bound for rc trees with generalized input signals. In *Design Automation, 1995. DAC'95. 32nd Conference on*, pages 364–369. IEEE.

Kahng *et al.*, A. B. (1999). Improved effective capacitance computations for use in logic and layout optimization. In *VLSI Design, 1999. Proceedings. Twelfth International Conference On*, pages 578–582. IEEE.

Kahng *et al.*, A. B. (2013). High-performance gate sizing with a signoff timer. In *Proceedings of the International Conference on Computer-Aided Design*, pages 450–457. IEEE Press.

Kashyap *et al.*, C. V. (2003). Closed form expressions for extending step delay and slew metrics to ramp inputs. In *Proceedings of the 2003 international symposium on Physical design*, pages 24–31. ACM.

Li *et al.*, L. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–232. IEEE.

Livramento *et al.*, V. S. (2014). A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4):40.

Moon *et al.*, C. W. (2010). Method of designing a digital circuit by correlating different static timing analyzers. US Patent 7,823,098.

Ozdal *et al.*, M. (2012). The ISPD-2012 discrete cell sizing contest and benchmark suite. In *ACM International Symposium on Physical Design*, pages 161–164. ACM.

Ozdal *et al.*, M. (2013). An improved benchmark suite for the ISPD-2013 discrete cell sizing contest. In *ACM International Symposium on Physical Design*, pages 168–170. ACM.

PrimeTime (2012). Synopsys PrimeTime and PrimeTime SI User Guide. Version H-2012.12, December 2012. www.synopsys.com

Qian *et al.*, J. (1994). Modeling the" effective capacitance" for the rc interconnect of cmos gates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(12):1526–1535.

Sharma *et al.*, A. (2017). Rapid gate sizing with fewer iterations of lagrangian relaxation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 337-343. IEEE.

Sharma *et al.*, A. (2015). Fast Lagrangian relaxation based gate sizing using multi-threading. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 426–433. IEEE.

Tennakoon *et al.*, H. (2002). Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 395–402. ACM.

Wang *et al.*, J. (2009). Gate sizing by Lagrangian relaxation revisited. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):1071–1084.

Yella *et al.*, A. K. (2017). Improved lagrangian relaxation-based gate size and v t assignment for very large circuits. In *PhD Research in Microelectronics and Electronics Latin America (PRIME-LA)*, pages 1–4. IEEE.

CHAPTER 5. LAGRANGIAN RELAXATION BASED GATE SIZING WITH CLOCK SKEW SCHEDULING - A FAST AND EFFECTIVE APPROACH

Abstract - Recent works have established Lagrangian relaxation (LR) based gate sizing as the state-of-the-art gate sizing approach in terms of power reduction and runtime. Gate sizing has limited potential to reduce the power when the timing constraints are tight. In sequential designs, clock skew scheduling can help relax the timing constraints to facilitate more power reduction.

Since variable clock skew introduces loops in the timing graph, it becomes a challenge for the LR based gate sizing as it relies on a projection based heuristic Lagrange multiplier update for faster convergence and better solution quality. This heuristic requires the timing graph to be directed and acyclic. To address this challenge, we develop a fast and effective LR based tool for simultaneous gate sizing and clock skew scheduling.

We derive an LR formulation for gate sizing combined with useful skew assignment. We propose a novel flow for simultaneously updating the cell sizes, updating the skew, and updating the Lagrange multipliers in an iterative fashion. Compared to the state-of-the-art LR gate sizing for a fixed skew of zero, our tool achieves an average of 19.7% additional power reduction overall, and 26.5% power reduction for designs with tighter timing constraints, on the ISPD 2012 gate sizing contest benchmark designs. Our tool is only 1.1 times slower vs. LR gate sizing alone. We also explore a previously proposed more formal approach based on a network flow formulation for updating Lagrange multipliers in the presence of timing loops.

5.1 Introduction

In modern VLSI (very large system integration) circuit design, power consumption has increased substantially as larger circuits are being integrated on a single chip while the technology continues to shrink. That results in high power density causing reliability challenges, large cooling cost in data centers, quicker

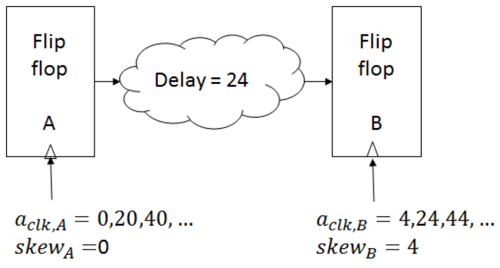

Figure 5.1: An example showing how clock skew scheduling can relax the timing constraints. The clock rising edge arrives at the clock pin of flip-flop A at an interval of 20 time units starting at t=0. The path delay is 24. In order to satisfy the timing constraints, the clock arriving at flip-flop B can be delayed by 4. Flip-flop B is said to have a clock skew of 4.

discharge of the batteries in mobile devices. Circuit performance also gets limited by power due to higher power densities. Thus, reducing the power has become a major concern.

In VLSI physical design, gate sizing is one of the most frequently used circuit optimizations. Each gate can be implemented by several possible cell options - having different sizes and threshold voltages (Vth). Different cell options trade off area or power for delay. The task of a gate-sizer is to choose a suitable cell for every gate to minimize the objective cost while meeting the design timing constraints.

In synchronous designs, appropriate scheduling of the clock skew can relax the timing constraints. Figure 5.1 illustrates this using a simple example. Relaxed timing constraints in turn, can be used to reduce the minimum feasible clock period, power and/or area. As shown in the previous works Chuang *et al.* (1995); Wang *et al.* (2009); Shklover *et al.* (2012), a gate sizer with the ability to schedule skew has a much larger potential to reduce power compared to the gate sizer that assumes a fixed skew at all clock pins.

In this chapter, we investigate two different approaches for simultaneous skew scheduling and gate sizing using an LR formulation. The first approach was originally proposed by Wang *et al.* (2009). Wang et. al. transformed the original timing graph to eliminate the skew variables and in the process introduced loops in the graph. The Lagrangian dual problem on the transformed graph was modeled as a min-cost network flow problem. Assuming continuity in the cell sizes and convexity of the delay models, their algorithm maximized the dual cost to realize primal optimality, if the primal problem was feasible. We work out the details to apply their algorithm with more realistic constraints - discrete cell sizes and non-convex delay models - and discuss several limitations of this approach in the presence of these realistic constraints. We refer to this approach as *NetFlo*.

The second approach for simultaneous gate sizing and skew scheduling is built upon the state-of-the-art LR gate sizing algorithm. We derive a modified LR formulation with skew as the variables while keeping the timing graph directed acyclic. The timing graph must be acyclic in order to apply the projection based Lagrange multiplier update heuristic which is crucial to the performance of gate sizing tools. We propose a new flow to simultaneously update skew along with the cell sizes and the Lagrange multipliers. We discuss our skew update strategy and propose modifications to the Lagrange multiplier update strategy in order to reflect the modified timing constraints caused by skewed clocks. We refer to this approach as *EGSS* which stands for *E*ffective *G*ate sizing with *S*kew *S*cheduling. For benchmarking, we use the ISPD 2012 gate sizing contest benchmark suite Ozdal *et al.* (2012). Compared to the state-of-the-art LR gate sizer Sharma *et al.* (2017) that assumes skew is fixed, EGSS achieves an average of 18.8% additional power reduction overall, and 25.5% power reduction for designs with tighter timing constraints.

Our main contributions are as follows:

- We derive an LR formulation for simultaneous gate sizing and clock skew scheduling without loops in the timing graph.
- In the context of discrete gate sizing with non-convex delay models, we discuss several limitations of achieving primal optimality via dual maximization.
- We propose a new algorithm to solve the Lagrangian subproblem with skew and cell size as the variables.
- We propose a modified Lagrange multiplier update heuristic accounting for the skew.

The rest of the chapter is organized as follows. Section 5.2 summarizes the previous works on LR gate sizing and combined sizing plus skew scheduling. Section 5.3 formulates the problem. Sections 5.4 and 5.5 present NetFlo and EGSS, respectively. Section 5.6 discusses several limitations of trying to achieve primal optimality via dual optimality. Section 5.7 briefly describes the greedy refinement strategies that we use in this work. We present the experimental results in Section 5.8 and conclude in Section 5.9.

5.2 Previous Work

Gate sizing problem has been researched for several decades. Earlier, most of the approaches assumed continuity in the gate sizes, convex delay models and experimented on relatively smaller designs. The gate sizing contests organized by Intel in ISPD 2012 Ozdal *et al.* (2012) and ISPD 2013 Ozdal *et al.* (2013) gave a fresh momentum to research in this area. The objective in the contests was to minimize the leakage power under the delay constraints. Contests were based on realistic constraints - discrete cell options and table lookup based non-linear delay models, and provided a suite of small to large designs having up to a million gates for benchmarking. Most of the post-contest publications in gate sizing utilized the contest framework for benchmarking and thus, greatly pushed ahead the state-of-the-art.

Some of the post-contest publications like Hu *et al.* (2012) used several sensitivity guided greedy meta-heuristics to reduce timing violations and then, reduce the power. Daboul *et al.* (2018) modeled the gate sizing problem as a resource sharing problem. However, most of the gate sizers that were published post-contest like Li *et al.* (2012); Livramento *et al.* (2014); Flach *et al.* (2014); Sharma *et al.* (2015, 2017); Yella *et al.* (2017) used LR formulation. Li *et al.* (2012) first achieved minimum clock design and then recovered power using the min-cost network flow formulation. Ren *et al.* (2008) also used network flow for discrete cell sizing but neither of their formulations considered skew. Flach *et al.* (2014) improvised on the projection based Lagrange multiplier update heuristic that was originally proposed in Tennakoon *et al.* (2002). They demonstrated the least power results on the ISPD 2012 gate sizing contest designs. Sharma *et al.* (2015) proposed a multi-threaded LR gate sizer and reported the least runtime which they further improved in Sharma *et al.* (2017). They proposed a simple and tunable framework for projection based Lagrange multiplier update which significantly improved the convergence. LR gate sizing using the projection heuristic has been demonstrated to yield designs with lower power and much smaller runtime compared to the other approaches. LR gate sizing idea is credited to Chen *et. al.* (1999).

Some of the previous works on simultaneous gate sizing and skew scheduling include Chuang *et al.* (1995); Wang *et al.* (2009); Shklover *et al.* (2012); Sathyamurthy *et al.* (1998); Roy *et al.* (2008). Chuang *et al.* (1995) directly solved the primal problem by formulating it as a linear programming problem using the piecewise-linear (PWL) approximations of the convex delays. Roy *et al.* (2008) assumed continuous sizes and convex delays, and thus, were able to minimize the Lagrangian subproblem simultaneously over size and skew variables using a bound constrained optimization solver. Without giving much details, they claim to use the projection heuristic of Tennakoon *et al.* (2002) for updating Lagrange multipliers. Wang *et al.* (2009) eliminated the skew variables from the primal problem in order for the Hessian of the primal objective to be positive definite so that optimality of their algorithm can be guaranteed. As a result, timing graph could no longer be acyclic. Wang et. al. maximized the dual cost by solving the min-cost network flow formulation of the Lagrangian dual problem. While they prove primal optimality under the assumptions that sizes are continuous and delay models are convex, these assumptions are not valid in modern design methodologies which would limit the effectiveness of their approach. Shklover *et al.* (2012) accounted for the cost of implementing the clock skew via clock tree. Although they formulate a simultaneous discrete gate sizing and skew scheduling problem using LR, they mainly focus on clock tree optimization via dynamic programming. For the gate sizing part they simply refer to the previous works Chen *et. al.* (1999); Roy *et al.* (2008); Wang *et al.* (2009). They do not discuss their multiplier update strategy.

5.3 Problem Formulation

In order to formally define the problem, we are using the notations tabulated in Table 5.1. Following the ISPD gate sizing contest objective, we are minimizing the leakage power subject to the delay constraints, maximum load constraints and maximum slew (or, transition time) constraints. Skew variables are bounded. We are allowed to change only combinational gates. Sequential gates are fixed. The primal problem is formally defined as follows:

Table 5.1: Commonly used notations.

Notation	Meaning
T	Target clock period
$\mathcal{G}$	Set of gates in the design
$\mathcal{X}_g$	Discrete set of cells for gate g
$x_g \in \mathcal{X}_g$	Current cell for gate g
$\mathcal{FF}$	Set of flip-flops in the design
$\mathcal{PO}$	Set of primary outputs in the design
w_k	Skew at flip-flop $k \in \mathcal{FF}$
$\mathcal{N}$	Set of nodes in the timing graph
$\mathcal{E}$	Set of timing arcs in the timing graph
$d_{ij}(\boldsymbol{x})$	Delay function of the timing arc (i, j)
λ_{ij}	Lagrange multiplier for the timing arc (i, j)
a_i	Arrival time at node i
a_{d_k}, a_{q_k}	Arrival times at D and Q pins of flip-flop k
$setup_k$, $d_{clk2q_k}(\boldsymbol{x})$	Setup delay and clock to Q delay of flip-flop k
λ_{d_k}, λ_{q_k}	Lagrange multipliers associated with the setup and the clock to Q delay timing arcs of flip-flop k
$gate\_power(x)$	Power of cell x
$skew\_power(w_k)$	Power cost for realizing a skew of w_k at flip-flop k
$p(\boldsymbol{x}, \boldsymbol{w})$	Total power of the design
$max\_load(x)$	Maximum load capacity of cell x
$\boldsymbol{x}, \boldsymbol{w}, \boldsymbol{a}, \boldsymbol{\lambda}$	Respective set of variables x, w, a and λ
$load_g(\boldsymbol{x})$	Capacitive load at the output of gate g
$slew_i(\boldsymbol{x})$	Slew at node i
max_slew	Maximum slew defined in the cell library

$$\begin{aligned}
\underset{\boldsymbol{x},\boldsymbol{a},\boldsymbol{w}}{\text{minimize}} \quad & p(\boldsymbol{x},\boldsymbol{w}) & \\
\text{subject to} \quad & a_i + d_{ij}(\boldsymbol{x}) \le a_j & \forall (i,j) \in \mathcal{E} \\
& a_{po} \le T & \forall po \in \mathcal{PO} \\
& a_{d_k} \le T + w_k - setup_k & \forall k \in \mathcal{FF} \\
& w_k + d_{clk2q_k}(\boldsymbol{x}) \le a_{q_k} & \forall k \in \mathcal{FF} \\
& load_g(\boldsymbol{x}) \le max\_load(x_g) & \forall g \in \mathcal{G} \\
& slew_g(\boldsymbol{x}) \le max\_slew & \forall g \in \mathcal{G} \\
& x_g \in \mathcal{X}_g & \forall g \in \mathcal{G} \\
& w_{min} \le w_k \le w_{max} & \forall k \in \mathcal{FF}
\end{aligned} \tag{5.1}$$

where, minimization is over the set of discrete cell variables $\boldsymbol{x}$, continuous arrival time variables $\boldsymbol{a}$ and continuous skew variables $\boldsymbol{w}$. p(**x**,**w**) is the total power cost of the design defined as the sum of power over all the gates and the power cost of implementing the desired skew $\boldsymbol{w}$,

$$p(\boldsymbol{x},\boldsymbol{w}) = \sum_{g \in \mathcal{G}} gate\_power(x_g) + skew\_power(\boldsymbol{w})$$

We use table lookup based non-linear (and, non-convex) delay models for modeling cell arcs. In accordance with the ISPD 2012 contest framework, interconnects are modeled by a lumped capacitance without any resistance. Consequently, the net timing arcs have zero delay. Even if interconnects are modeled by distributed RC trees as in the ISPD 2013 contest, the problem formulation and our approach would not change. With RC interconnects, the main challenge is the interconnect and the cell delay modeling which is beyond the scope of this work. We model rise and fall timing constraints separately, but we omit their separate mention throughout this work for clear presentation.

The timing graph for the primal problem is shown in Figure 5.2. The graph has two dummy nodes - a global input I and a global output O. There are weighted edges from I to Q pins; from I to primary inputs; from D pins to O; and, from primary outputs to O. In addition to accounting for the skew at each flip-flop, the weights have been adjusted so that the arrival times at I and O satisfy following: $a_I = 0$ and $a_O \le 0$, without having to add an edge from O to I which would create loops as presented in Wang *et al.* (2009). In

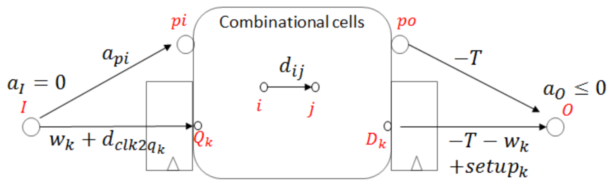

Figure 5.2: Directed acyclic timing graph with skew variables.

presence of clock trees, we can extend this timing graph to include timing arcs from the clock generator to the clock pins without creating loops.

We relax the primal problem by including the non-convex constraints into the objective function. To penalize any constraint violations, each one of them is associated with a non-negative Lagrange multiplier that acts as a penalty for constraint violation. Multipliers indicate the timing criticality of the corresponding arc. For a given set of Lagrange multipliers $\boldsymbol{\lambda} \geq 0$ the Lagrange function can be defined as follows:

$$\begin{aligned} L_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{w}) : \quad & p(\boldsymbol{x}, \boldsymbol{w}) + \sum_{(i,j) \in \mathcal{E}} \lambda_{ij} \times (a_i + d_{ij}(\boldsymbol{x}) - a_j) + \\ & + \sum_{po \in \mathcal{PO}} \lambda_{po} \times (a_{po} - T) \\ & + \sum_{k \in \mathcal{FF}} \lambda_{d_k} \times (a_{d_k} - T - w_k + setup_k) \\ & + \sum_{k \in \mathcal{FF}} \lambda_{q_k} \times \left(w_k + d_{clk2q_k}(\boldsymbol{x}) - a_{q_k}\right) \end{aligned} \tag{5.2}$$

We do not relax the constraints that are easy to track in our proposed approach. The Lagrangian dual function is defined as the minimum value of the Lagrangian function over $\boldsymbol{x}$, $\boldsymbol{a}$ and $\boldsymbol{w}$, for given $\boldsymbol{\lambda}$,

$$\begin{aligned} g(\boldsymbol{\lambda}) = & \underset{\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{w}}{\text{minimize}} \quad L_{\boldsymbol{\lambda}}(\boldsymbol{x}, \boldsymbol{a}, \boldsymbol{w}) \\ & \text{subject to} \quad load_g(\boldsymbol{x}) \leq max\_load(x_g) \qquad \forall g \in \mathcal{G} \\ & \qquad slew_g(\boldsymbol{x}) \leq max\_slew \qquad \forall g \in \mathcal{G} \\ & \qquad x_g \in \mathcal{X}_g \qquad \forall g \in \mathcal{G} \\ & \qquad w_{min} \leq w_k \leq w_{max} \qquad \forall k \in \mathcal{FF} \end{aligned} \tag{5.3}$$

Since the Lagrangian function is affine in arrival times, in order for the dual function to take a finite value, the set of multipliers must satisfy the following, so called, *flow constraints*.

$$\sum_{\{v|(i,v)\in\mathcal{E}\}} \lambda_{iv} = \sum_{\{u|(u,i)\in\mathcal{E}\}} \lambda_{ui} \quad \forall i \in \mathcal{N}\backslash\{I,O\} \tag{5.4}$$

Let Ω denote the space of Lagrange multipliers defined by the flow constraints (5.4),

$$\Omega = \{\boldsymbol{\lambda}|\boldsymbol{\lambda} \text{ satisfies (5.4) and } \boldsymbol{\lambda} \geq 0\}$$

Upon applying the flow constraints (5.4) to the Lagrangian function (5.2), arrival time variables can be eliminated and additionally, by ignoring the constant terms involving T and $setup_k$, we get a simplified Lagrangian function,

$$\begin{aligned} P_{\boldsymbol{\lambda}\in\Omega}(\boldsymbol{x},\boldsymbol{w}): \quad & p(\boldsymbol{x},\boldsymbol{w}) + \sum_{(i,j)\in\mathcal{E}} (\lambda_{ij} \times d_{ij}(\boldsymbol{x})) + \\ & \sum_{k\in\mathcal{FF}} \lambda_{q_k} \times d_{clk2q_k} + \sum_{k\in\mathcal{FF}} (\lambda_{q_k} - \lambda_{d_k}) \times w_k \end{aligned} \tag{5.5}$$

For $\boldsymbol{\lambda} \in \Omega$, the dual function can be re-written as the following minimization problem, which we refer as the simplified Lagrangian relaxation subproblem or *sLRS*$_\lambda$,

$$\begin{aligned} g(\boldsymbol{\lambda} \in \Omega) = & \underset{\boldsymbol{x},\boldsymbol{w}}{\text{minimize}} \quad P_{\boldsymbol{\lambda}}(\boldsymbol{x},\boldsymbol{w}) \\ & \text{subject to} \quad \text{constraints in (5.3)} \end{aligned} \tag{5.6}$$

The Lagrange dual problem (*LDP*) maximizes the dual function,

$$\underset{\boldsymbol{\lambda}\in\Omega}{\text{maximize}} \quad g(\boldsymbol{\lambda}) - \sum_{po\in\mathcal{PO}} \lambda_{po} \times T + \sum_{k\in\mathcal{FF}} \lambda_{d_k} \times (setup_k - T) \tag{5.7}$$

5.4 Min-Cost Network Flow Modeling

In this section we discuss the first approach for simultaneous gate sizing and skew scheduling that we investigate in this work. We refer to it as *NetFlo*. It is based on the work of Wang *et al.* (2009). Authors assumed continuous sizes and convex delay models. We extend their algorithm to make it applicable for discrete sizes and table lookup based delay models.

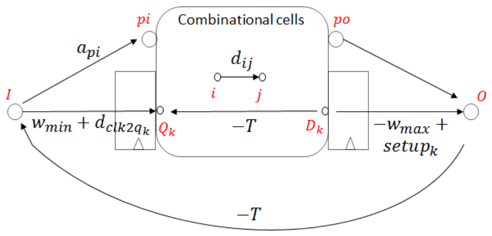

Figure 5.3: Timing graph for NetFlo approach: Original timing graph of Figure 5.2 was transformed in order to eliminate skew variables. In the process, the graph has directed cycles.

Wang et. al. transformed the timing graph to eliminate the skew from the set of primal variables which resulted in directed cycles. An example is shown in Figure 5.3. Their Lagrangian function $L_{\boldsymbol{\lambda}}^{NF}(\boldsymbol{x})$ is, therefore, different than ours (5.2). See Wang *et al.* (2009) for the complete expression of $L_{\boldsymbol{\lambda}}^{NF}(\boldsymbol{x})$. We define the Lagrangian subproblem for NetFlo ($LRS_{\boldsymbol{\lambda}}^{NF}$) as,

$$\begin{aligned} g^{NF}(\boldsymbol{\lambda} \in \Omega) = & \underset{\boldsymbol{x}}{\text{minimize}} \quad L_{\boldsymbol{\lambda}}^{NF}(\boldsymbol{x}) \\ & \text{subject to} \quad \text{constraints in (5.3)} \\ & \qquad\qquad\quad \text{except } w \text{ bounds} \end{aligned} \tag{5.8}$$

g^{NF} is the dual function. Then, LDP^{NF} is defined as,

$$\underset{\boldsymbol{\lambda}\in\Omega}{\text{maximize}} \quad g^{NF}(\boldsymbol{\lambda}) \tag{5.9}$$

In the neighborhood of current $\boldsymbol{\lambda}$, the LDP^{NF} can be formulated as a min-cost network flow problem, $MCNF_{\boldsymbol{\lambda}}$, defined as,

$$\begin{aligned} \underset{\triangle\boldsymbol{\lambda}}{\text{minimize}} \quad & \langle -\nabla g^{NF}(\boldsymbol{\lambda}), \triangle\boldsymbol{\lambda} \rangle \\ \text{subject to} \quad & \triangle\boldsymbol{\lambda}_{lb} \leq \triangle\boldsymbol{\lambda} \leq \triangle\boldsymbol{\lambda}_{ub} \\ & \triangle\boldsymbol{\lambda} + \boldsymbol{\lambda} \in \Omega \end{aligned} \tag{5.10}$$

where, $\nabla g^{NF}(\boldsymbol{\lambda})$ is the gradient of the dual function g^{NF} at $\boldsymbol{\lambda}$; and, $\langle . \rangle$ denotes the dot product operation.

In order to solve the gate sizing problem, the dual function is maximized over $\boldsymbol{\lambda} \in \Omega$. The intuition behind the overall algorithm is to iteratively improve the dual function $g^{NF}(\boldsymbol{\lambda})$ by maximizing its first-order approximation in a close neighborhood. Pseudo-code of the algorithm is shown in Algorithm 8. All

Lagrange multipliers are initialized to 0 since that is a trivial dual feasible solution. $LRS^{NF}_{\boldsymbol{\lambda}}$ is solved to initialize the sizes. Since multipliers are all zeros, the optimal solution to $LRS^{NF}_{\boldsymbol{\lambda}}$ is the minimum power design subject to maximum load and slew constraints. Then, we initialize the skews to their minimum values and update the timing. The timing information is needed to compute the bounds on $\triangle\boldsymbol{\lambda}$. Our proposed bound computation strategy is discussed later in this section, and our proposed skew update strategy is discussed in Section 5.5. After the initialization, LDP^{NF} is iteratively solved. In each iteration, firstly the lower and the upper bounds are computed. Then, $MCNF_{\boldsymbol{\lambda}}$ (5.10) is solved using the computed bounds. Optimal solution $\triangle\boldsymbol{\lambda}^*$ gives the steepest ascent direction of $g^{NF}(\boldsymbol{\lambda})$. Then, a line search is performed along $\triangle\boldsymbol{\lambda}^*$ in order to improve g^{NF}. For each step-size during the line search, $LRS^{NF}_{\boldsymbol{\lambda}}$ is solved. Based on the step-size that yielded the maximum g^{NF}, the multipliers are updated. Then, skews and timing are updated, again only for the purpose of computing the bounds. Iterations continue until the change in g^{NF} is below a threshold or a maximum number of iterations are reached. Since the problem is discrete and non-convex, this approach has several limitations which are discussed in Section 5.6. Consequently, even though dual function converges, often there are some timing violations and scope for further power reduction. Therefore, we add a greedy refinement step at the end to try to recover any remaining timing violations and reduce power.

Algorithm 8 Pseudo code for NetFlo

```
1: $\boldsymbol{\lambda} = 0$. Solve $LRS^{NF}_{\boldsymbol{\lambda}}$ (5.8) for $\boldsymbol{x}$
2: Initialize skew to $w_{min}$. Update timing.
3: **while** $g^{NF}(\boldsymbol{\lambda})$ has not converged and iterations $< N$ **do**
4:     $(\triangle\boldsymbol{\lambda}_{lb}, \triangle\boldsymbol{\lambda}_{ub}) \leftarrow$ compute bounds on $\triangle\boldsymbol{\lambda}$
5:     Solve $MCNF_{\boldsymbol{\lambda}}$ (5.10) for optimal $\triangle\boldsymbol{\lambda}^*$
6:     Perform line search on $g^{NF}(\boldsymbol{\lambda} + step \times \triangle\boldsymbol{\lambda}^*)$, $0 < step \leq 1$ for an increase in $g^{NF}$.
7:     Update $\boldsymbol{\lambda}$
8:     Update skew. Update timing (for bound computation)
9: Greedy refinements
```

Bound Computation Wang *et al.* (2009) did not give details on how to set the bounds. Based on the insights from the state-of-the-art LR gate sizing works, we propose to set the upper and lower bounds on $\triangle\lambda_{ij}$ for each combinational cell timing arc (i, j), as follows:

$$\begin{aligned}\triangle\lambda_{ij,ub} &= \max\left\{\frac{\lambda_{ij} \times |(q_j - a_i - d_{ij})|}{T}, \bar{\boldsymbol{\lambda}} \times M\right\} \\ \triangle\lambda_{ij,lb} &= \max\left\{-\lambda_{ij}, -\triangle\lambda_{ij,ub}\right\}\end{aligned} \tag{5.11}$$

where, q_j is the required time at node j; $\bar{\boldsymbol{\lambda}}$ is the average value of multipliers over all the arcs; M is a tuning parameter. We make the upper bound directly proportional to the current value of multiplier and the slack along the arc. If either of them is large, that indicates the need for large changes in the multiplier. For near-critical arcs, upper bound can be very small which may slow down the convergence. Hence, we ensure that the upper bounds are at least of the order of the average multiplier value across the design. For lower bounds, we need to ensure that the updated multiplier remains non-negative. Note that for some arcs, including the arc from O to I, it is not necessary to set explicit bounds as their multiplier values are implied by multiplier values of all the other incident arcs, being constrained by the Ω space.

Solving $LRS_{\boldsymbol{\lambda}}^{NF}$ The optimal strategy used in Wang *et al.* (2009) for solving the subproblem (5.8) assumes continuity in sizes and convexity of delay models. In presence of discrete sizes and non-convex delay models, it becomes a difficult combinatorial problem. We propose to use the same strategy that is commonly used in discrete LR gate sizing Li *et al.* (2012) to solve (5.8). We applied the multi-threading techniques Sharma *et al.* (2015) to parallelize the LRS solver. All gates in the design are traversed in the forward topological order. For each gate, assuming other gates are fixed, all the cell sizes are evaluated and that cell is chosen which contributes minimally to the objective. In order to define a topological order in a cyclic timing graph, we cut it at the flip-flops. One major limitation of this strategy is that it ignores the interactions between multiple gates that can be resized simultaneously. Sharma *et al.* (2017) proposed multi-gate sizing as an alternative but it did not show significant improvement in practice and had much larger runtime.

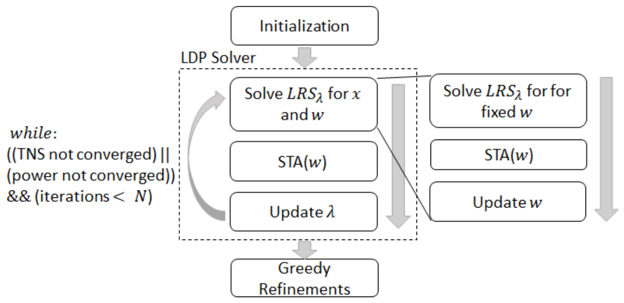
Initialization
LDP Solver
Solve LRS_λ for x and w
STA(w)
Update λ
Solve LRS_λ for for fixed w
STA(w)
Update w
while: ((TNS not converged) \|\| (power not converged)) && (iterations < N)
Greedy Refinements

Figure 5.4: Overall flow of our proposed EGSS. STA refers to the static timing analysis.

5.5 EGSS: Effective Gate Sizing With Skew Scheduling

In this section we describe our proposed approach for simultaneous gate sizing and skew scheduling. We call it *EGSS*. Figure 5.4 shows the overall flow of our algorithm. It has three stages: initialization, solving LDP and greedy refinements.

During initialization gates are initialized to minimum power cell sizes subject to the maximum load and slew constraints; skews are initialized to the minimum values; timing is updated; and, Lagrange multipliers are initialized to a non-zero constant and projected on to the Ω space. Unlike NetFlo, owing to the multiplicative nature of the Lagrange multiplier update heuristic which is discussed later in this section, multipliers cannot be initialized to zero.

The LDP solver is an iterative stage to recover bulk of the timing violations and reduce power. It iterates between solving the $LRS_{\boldsymbol{\lambda}}$ (5.3) over size and skew variables, timing update and Lagrange multiplier update. During the first few iterations, multipliers are updated to reduce the timing violations, measured as TNS (total negative slack). Once the timing violations are reduced, multipliers are updated to drive the design towards lower power. Extra checks are incorporated inside the LRS solver to discourage timing violations.

Like NetFlo, greedy refinements are performed in the last stage.

5.5.1 Solving $LRS_{\boldsymbol{\lambda}}$

In LR gate sizing, LRS needs to be solved only over the sizes. But with skew scheduling, optimization is done over skew variables as well. We propose to solve $LRS_{\boldsymbol{\lambda}}$ in three steps, as shown in Figure 5.4. The first step is to solve LRS only for the size variables assuming that skews are fixed. The second step is to update the timing using the old skew values, and the third step is to update the skews. Our proposed skew update strategy is discussed later in this section.

For the first step, wherein LRS is solved over the size variables only, we use the same algorithm as discussed for the NetFlo approach in Section 5.4, except that we add a check to discourage local slack degradation, as was proposed earlier Flach *et al.* (2014). This check is necessary initially for TNS convergence, and later to keep the timing violations under control, while power reduction iterations are in progress. This check was not applied while solving LRS inside the NetFlo because the NetFlo objective was dual maximization, and reduction in TNS and power (primal objective) was expected as a consequence.

5.5.2 Skew Update

Let us consider the skews terms in the simplified Lagrangian function (5.5),

$$h(\boldsymbol{w}) = skew\_power(\boldsymbol{w}) + \sum_{k \in \mathcal{FF}} w_k \times (\lambda_{q_k} - \lambda_{d_k})$$

where, h(**w**) is referred as the skew cost. Skews should be updated subject to the bounds such that the skew cost is minimized. For a given clock tree, Shklover *et al.* (2012) proposed a dynamic programming approach to solve this optimization problem. In this work we ignore the skew power cost. Then, the expression for $h(\boldsymbol{w})$ suggests that each skew variable can be separately optimized. For each flip-flop k, if $\lambda_{q_k} > \lambda_{d_k}$ then, $w_k = w_{min}$, else $w_k = w_{max}$. Intuitively, if Q pin is more timing critical than the D pin, then reduce the skew to reduce timing violations (or, increase the timing slack) at Q pin. If D pin is more timing critical, then increase the skew to increase the slack at D pin. Note that always setting skew to either of the extreme values can cause oscillations. Hence, with the insight developed above, we propose following skew update strategy,

$$\begin{aligned} \triangle w_k &= \frac{slack_q - slack_d}{2} \\ w_k &= \text{max } \{w_{min}, \text{ min } \{w_{max}, w_k + \triangle w_k\}\} \end{aligned} \tag{5.12}$$

This is an intuitive choice for skew at other places as well, for example, to improve the reliability of the data path Kourtev *et al.* (2008).

5.5.3 Modified Lagrange Multiplier Update

Projection based Lagrange multiplier update strategy is very crucial for fast convergence and final solution quality. A simple and tunable framework for projection based multiplier update was proposed in Sharma *et al.* (2017). We extend it to account for the skew impact. If skew at a flip-flop is positive then, the timing paths ending at the D pin of that flip-flop have a larger required time which reduces the rate at which multipliers increase. Pseudo code for the Lagrange multiplier update is shown in Algorithm 9.

Algorithm 9 Lagrange multiplier update algorithm

```
for each flip-flop $k$ do
    $\lambda_{d_k} = \lambda_{d_k} \times \left(1 + \frac{a_{d_k} + setup_k - T - w_k}{T}\right)^K$
for each primary output $po$ do
    $\lambda_{po} = \lambda_{po} \times \left(1 + \frac{a_{po} - T}{T}\right)^K$
for each timing arc $(i, j)$ do
    $\lambda_{ij} = \lambda_{ij} \times \left(1 + \frac{a_i + d_{ij} - q_j}{T}\right)^K$        ▷ $q_j$: required time at $j$
Projection to satisfy flow constraints. Refer Tennakoon et al. (2002)
```

5.6 NetFlo Vs EGSS: Limitations of Optimizing Primal Via Dual Maximization

The NetFlo approach is based on results from the well-known Lagrangian duality theory Boyd *et al.* (2004), that under certain conditions which generally hold for the convex and continuous primal problems it is possible to attain the primal optimality by maximizing the dual function. However, for non-convex discrete gate sizing which has been proven to be NP-hard Ning (1994), this approach has following limitations:

- non-zero duality gap;
- minimizer $\boldsymbol{x}^*$ of the Lagrangian function while solving LRS for the optimal set of dual variables $\boldsymbol{\lambda}^*$ may not be primal feasible;
- discreteness tends to cause oscillations.

We explain these limitations with the help of a simple illustration that explains the process of dual maximization in the primal space.

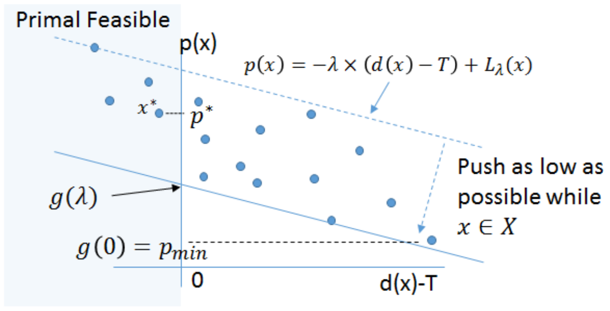
Primal Feasible
p(x)
$p(x) = -\lambda \times (d(x) - T) + L_\lambda(x)$
x^*
p^*
Push as low as possible while $x \in X$
$g(\lambda)$
$g(0) = p_{min}$
0
d(x)-T

Figure 5.5: Minimization of Lagrangian function $L_\lambda(x)$ for a single gate circuit is shown in the power-delay space parameterized by the discrete cell size x. X-axis is delay shifted to the right by T. Y-axis is the power. Each dot corresponds to a unique cell size x. Left of the power axis ($d(x) - T \leq 0$) is primal feasible. Minimum feasible power p^* and minimum possible power p_{min} are indicated above.

Consider a single inverter circuit with cell size x as the only variable. Let power of the circuit be $p(x)$ and delay be $d(x)$. Then, all the various discrete sizes can be plotted on a power-delay space as shown in Figure 5.5. Each dot (referred as a design point) in the plot corresponds to a distinct size. Let $\lambda \geq 0$ be the Lagrange multiplier associated with timing arc of the inverter. Then, Lagrange function can be written as $L_\lambda(x) = p(x) + \lambda \times (d(x) - T)$. In the power-delay space, this is a line with slope $-\lambda$ and intercept on the $p(x)$ axis is the value of the Lagrangian function. Solving LRS or computing the dual function is equivalent to minimizing $L_\lambda(x)$ which is equivalent to pushing the line as low as possible as long as it passes through at least one design point. This process is illustrated in Figure 5.5.

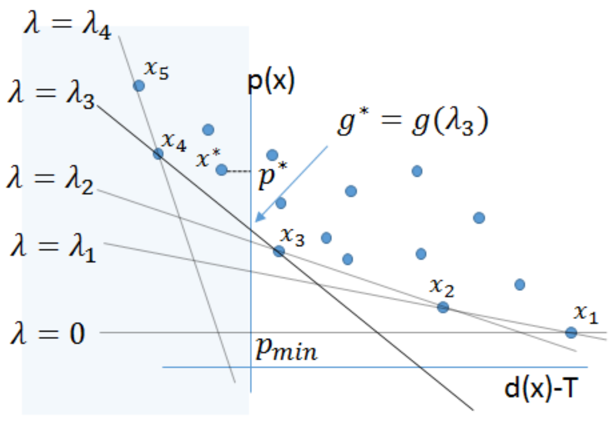

Figure 5.6: Maximization of dual function $g(\lambda)$ is shown graphically for a single gate circuit in (a) primal space and, (b) dual space. g^* is the dual optimal attained at $\lambda = \lambda_3$. Due to non-convexity and discreteness there is a non-zero duality gap, $p^* - g^*$. For the same reasons, $L_{\lambda_3}(x)$ is minimized at x_4 and x_3. While x_4 is primal feasible, x_3 is not.

Now we explain the process of dual maximization using Figure 5.6. Initially, when $\lambda = 0$, we know that $g(0) = p_{min}$, where p_{min} is the lowest possible power of the design. In the power-delay space, $\lambda = 0$ corresponds to the horizontal line and $L_0(x)$ is minimized at x_1 as shown in the Figure 5.6. As λ increases, the slope of the line increases and the dual cost also increases. For $\lambda < \lambda_1$, $x1$ minimizes $L_\lambda(x)$. At $\lambda = \lambda_1$, both x_1 and x_2 minimize the Lagrangian function. Dual function continues to increase with λ as long as $\lambda \leq \lambda_3$ and attains a maximum of g^* at $\lambda = \lambda_3$. At $\lambda = \lambda_3$, Lagrangian function is minimized at x_3 and x_4 out of which only x_4 is primal feasible. So even if the dual optimal is attained, there is no guarantee that primal feasible solution can be derived. It can also be observed that, the primal optimal value $p^* = p(x^*)$ is strictly more than the dual optimal g^*. The gap $p^* - g^*$ refers to the duality gap.

Note that these limitations are due to both - non-convexity as well as discreteness. While NetFlo tries to maximize the dual cost and has the above-mentioned limitations, EGSS uses Lagrange multipliers to help attain primal feasibility and then reduce power. EGSS rapidly increases the Lagrange multipliers initially to attain feasibility. That causes very low dual cost and high power. Then, using local slack check while solving LRS, the design to forced to stay in the feasible region while Lagrange multipliers reduce to recover power. It is important to note that while NetFlo requires an optimal LRS solver, which it is not, to maximize the dual cost, EGSS deliberately sacrifices the optimality of the LRS solver to maintain primal feasibility.

5.7 Greedy Refinements in Timing and Power

On account of sub-optimality of the LDP solver, designs obtained from the LDP solver have timing violations and additional power that can be recovered. In case of NetFlo, due to the limitations discussed in Section 5.6, there are large timing violations as well as a lot of power to be recovered. Hence, greedy refinements are all the more important for NetFlo type of approaches. In either case, it is a common practice to apply a timing recovery followed by a power recovery algorithm - both are greedy and local in nature.

One of the approaches for recovering timing violations which we implemented is to upsize those gates that lead to the maximum number of timing critical end-points. In order to compute the change in TNS, timing is updated in an incremental fashion. If the TNS improved (reduced) then the new size is committed, otherwise the upsizing is undone and the next gate in the order is upsized. This process is continued as long as the TNS is non-zero and it is reducing. During timing recovery we avoid reducing Vth because that significantly worsens the power.

For reducing power, we first try to increase Vth for each gate - one at a time, in the design. If Vth cannot be increased without worsening the TNS, then each gate is considered for downsizing. Like Flach *et al.* (2014) we traverse the gates in forward topological order.

5.8 Experiments and Results

We implemented our tool in C++. We performed our experiments on two quad-core Intel(R) Xeon(R) 3.50GHz CPUs. For solving the min-cost flow problem we used Gurobi. For line search we used a step size of 0.2 and evaluated 5 steps at uniform spacing. We used $M = 10$ - the tuning parameter for computing bounds. Our codes are multi-threaded using OpenMP Dagum *et al.* (1998). and we use 8 threads for solving LRS. We used PrimeTime version E-2010.12 for amd64, to verify the timing of our final designs obtained from NetFlo and EGSS. All of them satisfy all the constraints. For benchmarking we used the ISPD 2012 gate sizing contest benchmark suite.

We use the sizing results from Sharma *et al.* (2017) who also used 8 threads as the baseline since they are the fastest and have competitive power results. The baseline is gate sizing alone. Our NetFlo and EGSS flows have been provided with a minimum skew bound w_{min} of 0 and a maximum skew bound of 165ps. Table 5.2 summarizes the results from all three flows. Compared to the baseline, EGSS on account of being able to schedule skew simultaneously with sizing the gates, saves on average 19.7% more power. On designs with tighter timing constraints (‘fast’) average power saved is 26.5%. EGSS has only 1.1X slow down in the total runtime. Compared to NetFlo, EGSS saves 5.3% more power and is 70X faster. While in NetFlo, greedy refinement stage accounts for an average of 5.4% power reduction, in EGSS, it accounts for only 0.4% power reduction. This shows that the core idea behind EGSS is more effective than the core idea of NetFlo.

Main reasons for larger runtime of NetFlo are as follows: 1) Solving the min-cost flow problem is orders of magnitude more runtime expensive compared to the projection based Lagrange multiplier update shown in Algorithm 9. Latter has a linear time complexity in the number of gates. Using a network flow solver

Table 5.2: Results summary on ISPD 2012 benchmarks suite Ozdal *et al.* (2012) is shown for three algorithms: Sharma *et al.* (2017) - referred as [1] in this table, which is the baseline for comparison - it assumes a fixed skew; NetFlo and EGSS. For NetFlo as well as EGSS minimum and maximum skew bounds are 0 and 165ps.

Benchmark	**Comb.**	**Clock**	**Leakage Power (W)**			**Power saved (%)**		**Total Runtime (min)**			**Speedup (X)**	
	Gates	**T, (ps)**	**[1]**	**NetFlo**	**EGSS**	**Vs [1]**	**Vs NetFlo**	**[1]**	**NetFlo**	**EGSS**	**Vs [1]**	**Vs NetFlo**
DMA_slow	23109	900	0.135	0.111	0.104	23.1	6.6	0.07	7.90	0.08	0.9	94.0
pci_bridge32_slow	29844	720	0.098	0.073	0.072	26.9	2.2	0.09	8.70	0.10	0.9	88.0
des_perf_slow	102427	900	0.583	0.420	0.404	30.6	3.7	0.32	21.09	0.30	1.1	69.2
vga_lcd_slow	147812	700	0.329	0.310	0.310	5.9	0.2	0.44	28.35	0.44	1.0	64.0
b19_slow	212674	2500	0.569	0.577	0.556	2.2	3.7	0.83	45.75	1.24	0.7	37.0
leon3mp_slow	540352	1800	1.335	1.326	1.321	1.0	0.4	2.52	194.90	2.91	0.9	67.0
netcard_slow	860949	1900	1.763	1.762	1.762	0.1	0.0	2.35	343.90	2.82	0.8	122.0
DMA_fast	23109	770	0.245	0.173	0.137	44.3	20.8	0.08	9.20	0.10	0.8	92.0
pci_bridge32_fast	29844	660	0.141	0.083	0.078	44.7	6.2	0.10	9.20	0.11	0.9	84.0
des_perf_fast	102427	735	1.436	0.686	0.615	57.2	10.3	0.40	23.39	0.34	1.2	69.1
vga_lcd_fast	147812	610	0.417	0.318	0.316	24.3	0.8	0.56	27.90	0.50	1.1	55.3
b19_fast	212674	2100	0.729	0.823	0.682	6.5	17.1	1.13	19.58	1.61	0.7	12.2
leon3mp_fast	540352	1500	1.449	1.393	1.360	6.1	2.4	3.13	233.10	3.56	0.9	65.5
netcard_fast	860949	1200	1.846	1.804	1.800	2.5	0.2	3.33	237.05	3.98	0.8	59.5
Average			**0.791**	**0.704**	**0.680**	**19.7**	**5.3**	**1.10**	**86.43**	**1.29**	**0.9**	**69.9**

instead of Gurobi is likely to improve the runtime. 2) NetFlo has a slower convergence than EGSS. So it takes more iterations. 3) Each NetFlo iteration is more expensive due to solving LRS several times during the line search.

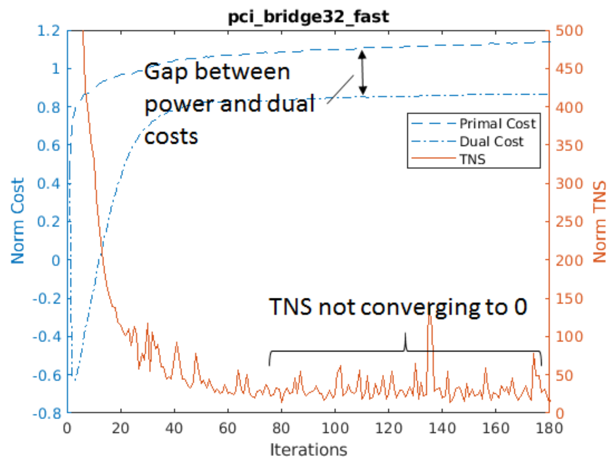

Figure 5.7: Primal cost (power), dual cost and TNS profiles obtained from NetFlo are shown for pci_bridge32_fast. Left Y-axis is the normalized dual or primal cost and the right Y-axis is the TNS normalized with respect to the target clock period which is 660ps. For these runs maximum skew bound is set to 0 to highlight the limitations of the NetFlo approach.

Figure 5.7 shows dual cost, power and TNS profiles from an execution of NetFlo on the design pci_bridge32_fast. We can see that although the dual cost increases and converges to an apparently maximum value, TNS is not able to converge down to 0. We also observe that there is a distinct gap between the dual cost and the primal cost, which is nothing but the total gate power of the design. This seems to indicate the non-zero duality gap.

Figure 5.8 compares the TNS and power profiles from EGSS and NetFlo. EGSS starts with high overshoots in power as TNS rapidly reduces. But very quickly it recovers the power as well. It converges in less than 40 iterations with near-zero TNS and better power than NetFlo. Across all the benchmarks, EGSS takes on average only 2 iterations to converge the TNS and an additional 18 iterations to reduce the power.

5.9 Conclusion

Gate sizing is a crucial circuit optimization technique for trading off delay for area and power. The potential of gate sizing can be enhanced by allowing variable skew. In this work we investigate two approaches

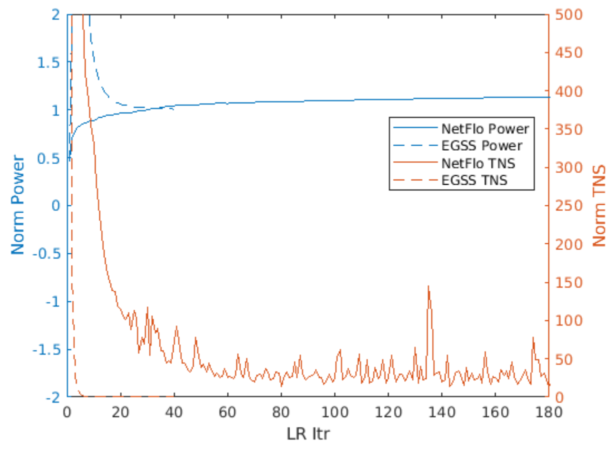

Figure 5.8: Comparing power and TNS profiles for pci_bridge32_fast design from NetFlo and EGSS. Maximum skew bound is set to 0 for both approaches.

for simultaneous gate sizing and skew scheduling. One approach derives a Lagrangian dual problem from the primal problem and tries to maximize the dual objective. We investigated several limitations arising from the non-convexity and the discreteness of the primal space, due to which dual maximization cannot guarantee primal feasibility and consequently, primal optimality, as also indicated by our experimental results. In the second approach, we extend the state-of-the-art high performance Lagrangian relaxation based gate sizing. We propose a new flow in which we first make use of the variable skew to recover bulk of the timing violations in just two iterations, on average. Then, we iteratively reduce power. In each iteration skew is updated to redistribute the slack between each side of the flip-flop. Compared to the state-of-the-art gate sizer which treats skew as fixed, our proposed flow for simultaneous gate sizing and clock skew scheduling reduces an average of 19.7% more power on the ISPD 2012 gate sizing contest designs while consuming slightly more runtime.

References

Boyd *et al.*, S. (2004). *Convex optimization*. Cambridge university press.

Chen *et. al.*, C.-P. (1999). Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):1014–1025.

Chuang *et al.*, W. (1995). Timing and area optimization for standard-cell vlsi circuit design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(3):308–320.

Daboul *et al.*, S. (2018). Provably fast and near-optimum gate sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Dagum *et al.*, L. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55.

Flach *et al.*, G. (2014). Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557.

Hu *et al.*, J. (2012). Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 233–239.

Kourtev *et al.*, I. (2008). *Timing Optimization Through Clock Skew Scheduling*, volume 166. Springer Science & Business Media.

Li *et al.*, L. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–232. IEEE.

Livramento *et al.*, V. S. (2014). A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4):40.

Ning, W. (1994). Strongly NP-hard discrete gate-sizing problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1045–1051.

Ozdal *et al.*, M. (2012). The ISPD-2012 discrete cell sizing contest and benchmark suite. In *ACM International Symposium on Physical Design*, pages 161–164. ACM.

Ozdal *et al.*, M. (2013). An improved benchmark suite for the ISPD-2013 discrete cell sizing contest. In *ACM International Symposium on Physical Design*, pages 168–170. ACM.

Ren *et al.*, H. (2008). A Network-Flow Based Cell Sizing Algorithm. In *The International Workshop on Logic Synthesis*.

Roy *et al.*, S. (2008). An optimal algorithm for sizing sequential circuits for industrial library based designs. In *Proceedings of the 2008 Asia and South Pacific Design Automation Conference*, pages 148–151. IEEE Computer Society Press.

Sathyamurthy *et al.*, H. (1998). Speeding up pipelined circuits through a combination of gate sizing and clock skew optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(2):173–182.

Sharma *et al.*, A. (2017). Rapid gate sizing with fewer iterations of lagrangian relaxation. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 337-343. IEEE.

Sharma *et al.*, A. (2015). Fast Lagrangian relaxation based gate sizing using multi-threading. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 426–433. IEEE.

Shklover *et al.*, G. (2012). Simultaneous clock and data gate sizing algorithm with common global objective. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 145–152. ACM.

Tennakoon *et al.*, H. (2002). Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 395–402. ACM.

Wang *et al.*, J. (2009). Gate sizing by Lagrangian relaxation revisited. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):1071–1084.

Yella *et al.*, A. K. (2017). Improved lagrangian relaxation-based gate size and v t assignment for very large circuits. In *PhD Research in Microelectronics and Electronics Latin America (PRIME-LA)*, pages 1–4. IEEE.

CHAPTER 6. CONCLUSION AND FUTURE DIRECTIONS

Today's designs with millions of gates require very fast gate-sizing and threshold voltage assignment, as it is a crucial circuit optimization that is performed at multiple steps in the design flow. With non-convex delay models and discrete sizes, traditional methods for gate sizing that assume continuity in sizes and/or convexity, are either not accurate enough or too slow. Recent works have demonstrated that Lagragian relaxation (LR) based gate sizing achieves least power on most of the designs in a competitive runtime compared to other types of sizing methodologies. However, on account of large number of Lagrangian iterations, slow timing updates, sequential execution and slow post-pass even the LR gate sizing can be very slow, especially on the larger designs. To alleviate these problems, we develop a multi-threaded discrete gate sizer that solves the LR formulation of the original problem in much lesser number of iterations and each iteration is very fast.

Our unique innovations include two parallelization enabling techniques that facilitate effective multi-threading; a tunable and effective Lagrange multiplier update framework to speed up the convergence and thereby, reduce the number of iterations; two strategies for finer-grained timing and power recovery to allow early termination of the iterations; a new, fast-to-compute model for effective capacitance and several timing calibration mechanisms to improve accuracy of simple timing models while optimizing designs with resistive interconnect; and, several enhancements to speed up the post-pass algorithms. Our gate sizer combines all these innovations to achieve an average speedup of more than 15x compared to the state-of-the-art gate sizer, on the ISPD 2012 as well as the ISPD 2013 gate sizing contest designs, while the leakage power is higher by not more than 2.5%, on average.

We further improve the potential of our gate sizer by adding the capability to simultaneously size the gates and schedule the clock skew. We incorporated the skew variables into the LR subproblem objective; developed a new flow to solve the subproblem; proposed an intuitive skew update strategy; and, modified the Lagrange multiplier update to account for skew at the sequential timing end-points. Our simultaneous gate sizing and skew scheduling tool can reduce 19.7% more power compared to the gate sizing alone at the expense of only 10% increase in the total runtime.

Some of the future research problems in the domain of LR gate sizing that as follows:

- How to initialize Lagrange multipliers for an optimized initial design? There has not been much work on the 'right way' to initialize the multipliers. That research is bound attract a lot of attention from the industry wherein, for example, a timing optimized design is passed to the gate sizer to reduce area or power without increasing the timing violations is not an un-common scenario. In other words, this is the problem of 'incremental gate sizing' which is still an open-ended problem under LR framework.
- The Lagrange multiplier update strategy is a heuristic and has a few drawbacks. One of them is that only timing criticality is used to update the multipliers with no regards to the worst slew propagation which may contribute to the timing criticality and thus, indirectly inhibit power reduction. Another non-intuitive aspect of the typical multiplier update strategy is that even though an arc is non-critical, it is possible that its multiplier value increases (intuitively, it should reduce) due to one or more critical end-points in its fan-out cone.
- How to schedule the clock skew with gate sizing in a runtime efficient manner while accounting for the cost of clock tree that would be needed to implement the desired skew? Previous work Shklover *et al.* (2012) does not discuss the multiplier update strategy. It merely refers to the runtime expensive strategies like min-cost network flow based formulation.

REFERENCES

Chen *et. al.*, C.-P. (1999). Fast and exact simultaneous gate and wire sizing by Lagrangian relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(7):1014–1025.

Chinnery *et al.*, D. (2005). Linear programming for sizing, Vth and Vdd assignment. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 149–154. ACM.

Daboul *et al.*, S. (2018). Provably fast and near-optimum gate sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

Fishburn, J. (1985). TILOS: A posynomial programming approach to transistor sizing. In *IEEE International Conference of Computer-Aided Design*.

Flach *et al.*, G. (2014). Effective Method for Simultaneous Gate Sizing and V-th Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557.

Hu *et al.*, J. (2012). Sensitivity-guided metaheuristics for accurate discrete gate sizing. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 233–239.

Hu *et al.*, S. (2007). Gate sizing for cell library-based designs. In *ACM/IEEE Design Automation Conference*, pages 847–852. ACM.

Huang *et al.*, Y.-L. (2011). Lagrangian relaxation for gate implementation selection. In *ACM International Symposium on Physical Design*, pages 167–174. ACM.

Kahng *et al.*, A. B. (2013). High-performance gate sizing with a signoff timer. In *Proceedings of the International Conference on Computer-Aided Design*, pages 450–457. IEEE Press.

Kasamsetty *et al.*, K. (2000). A new class of convex functions for delay modeling and its application to the transistor sizing problem [CMOS gates]. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):779–788.

Li *et al.*, L. (2012). An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 226–232. IEEE.

Liu *et al.*, Y. (2010). A new algorithm for simultaneous gate sizing and threshold voltage assignment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(2):223–234.

Liu *et al.*, Y. (2011). GPU-based parallelization for fast circuit optimization. *ACM Transactions on Design Automation of Electronic Systems*, 16(3):24.

Livramento *et al.*, V. S. (2014). A hybrid technique for discrete gate sizing based on lagrangian relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4):40.

Nguyen *et al.*, D. (2003). Minimization of dynamic and static power through joint assignment of threshold voltages and sizing optimization. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 158–163. ACM.

Ning, W. (1994). Strongly NP-hard discrete gate-sizing problems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(8):1045–1051.

Ozdal *et al.*, M. (2011). Gate sizing and device technology selection algorithms for high-performance industrial designs. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 724–731. IEEE Press.

Rahman *et al.*, M. (2013). Library-Based Cell-Size Selection Using Extended Logical Effort. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(7):1086–1099.

Reimann *et al.*, T. (2013). Simultaneous gate sizing and Vt assignment using fanin/fanout ratio and simulated annealing. In *IEEE International Symposium on Circuits and Systems*, pages 2549–2552. IEEE.

Reimann *et al.*, T. J. (2016). Cell selection for high-performance designs in an industrial design flow. In *ACM International Symposium on Physical Design*, pages 65–72. ACM.

Roy *et al.*, S. (2007). Numerically convex forms and their application in gate sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(9):1637–1647.

Shklover *et al.*, G. (2012). Simultaneous clock and data gate sizing algorithm with common global objective. In *Proceedings of the 2012 ACM international symposium on International Symposium on Physical Design*, pages 145–152. ACM.

Tennakoon *et al.*, H. (2002). Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 395–402. ACM.

Wang *et al.*, J. (2009). Gate sizing by Lagrangian relaxation revisited. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):1071–1084.

Yella *et al.*, A. K. (2017). Improved lagrangian relaxation-based gate size and v t assignment for very large circuits. In *PhD Research in Microelectronics and Electronics Latin America (PRIME-LA)*, pages 1–4. IEEE.