Timing Optimization Techniques for the Scalable Physical Synthesis of Digital Integrated Circuits

PhD Thesis

Dimitrios Mangiras

December 13, 2022

Advisor: Associate Prof. Giorgos Dimitrakopoulos
Department of Electrical and Computer Engineering
Democritus University of Thrace, Xanthi, Greece

Abstract

Physical synthesis is a fundamental part of the design flow of the modern VLSI since it transforms automatically the designer's RTL models to an integrated circuit ready for fabrication. As the technology continues to scale and the number of transistors per chip grows, the complexity of designing an integrated circuit increases steadily. The burden of delivering efficient designs passes through the physical synthesis tools that should satisfy two contradictory goals. First comes Quality-of-Results (QoR), i.e., to place and route a design that satisfies the required timing, area and power constraints. Then, comes efficiency that allows executing physical synthesis algorithms in the least amount of time even for very large designs. This thesis tackles exactly those two contradicting goals focusing on timing closure of complex digital designs.

Timing closure is a complex process that involves many iterative optimization steps applied in various phases of the physical design flow. The scaling of the size of the designs, the examination of multiple modes of operations and multiple design corners including also On-Chip Variations (OCV), are major critical challenges that timing optimization should face effectively. To this end, we propose four timing optimization techniques that tackle efficiently such challenges. The proposed approaches can be used both for global timing optimization at the first steps of the physical synthesis flow or close to the end where repairing timing violations requires incremental operations that are nondisruptive and execute as fast as possible. In every case, the proposed methods are tuned for runtime scalability that allows their application to very large designs without sacrificing QoR.

The first approach focuses on incremental timing-driven placement, with the goal to fix the placement of timing-critical cells and improve overall timing. As opposed to previous methods that independently move combinational gates, flip-flops, and/or LCBs using loosely-connected algorithms, we propose, an Lagrangian Relaxation-based (LR) timing-driven placement algorithm that handles the relocation of all types of cells in a unified manner. Cells are allowed to move within an appropriately positioned search window, the location of which is decided by force-like timing vectors covering both late and early timing violations. The magnitude of these timing vectors is determined by the value of the corresponding Lagrange Multipliers. The introduced placement optimization is applied in conjunction with a newly proposed flip-flop clustering algorithm that (re)assigns flip-flops to local clock buffers, to separate flip-flops with incompatible timing profiles and to facilitate the subsequent timing-optimization steps.

The other two approaches focus on LR-based gate sizing and voltage threshold assignment techniques. Firstly, we present a way to transform a robust gate sizer, used as global optimizer, into an incremental optimizer that can successfully improve the timing, power and area of the design really fast even when considering multiple corners.

The proposed methodology relies on different initialization of the LMs and therefore the solution is orthogonal to the core of the optimizer. This means that it is easy to be generalized and adopted by other similar timing optimizations to be transformed in an incremental context.

Physical synthesis engines need to embrace all available parallelism to cope with the increasing complexity of modern designs and still offer high quality of results. To achieve this goal, the involved algorithms need to be expressed in a way that facilitates fast execution time across a range of computing platforms. Motivated by this target, in this thesis, we introduce a task-based parallel programming template that can be used for speeding up timing and power optimization. This approach utilizes all available parallelism and enables significant speedup relative to custom multithreaded approaches. Task-based parallelism is applied to all parts of the optimization engine covering also parts that are traditionally executed serially for preserving maximum timing accuracy. Additionally, this result was supported by two dynamic heuristics that restrict the number of examined gate sizes and simplify local timing updates. Both heuristics trade off additional runtime reduction with marginal leakage power increases.

Timing optimization methods are completed by a novel methodology to reduce the timing impact of the clock-induced OCV. To reduce the magnitude of the clock-induced OCV, we incrementally relocate the flip-flops and the clock gaters in a bottom-up manner to implicitly guide the clock tree synthesis engine to produce clock trees with increased common clock tree paths. The relocation of the clock elements is performed using a soft clustering approach that is orthogonal to the clock tree synthesis method used. The clock elements are repeatedly relocated and incrementally re-clustered, thus gradually forming better clusters and settling to more appropriate positions to increase the common paths of the clock tree. This behavior is verified by applying the proposed method in industrial designs, resulting in clock trees which are more resilient to process variations, while exhibiting improved overall timing.

Acknowledgements

I would like to thank a couple of people, without them I would not have been able to complete my dissertation.

First and foremost, I would like to express my deep gratitude and sincere appreciation to my advisor, Giorgos Dimitrakopoulos, for the continuous support, his patience and his invaluable guidance during my Ph.D, contributing to my academic self-growth with his technical expertise. Under his supervision, I gained valuable knowledge and developed important skills that significantly helped me to become not only a better researcher, but also a better person. Furthermore, I am grateful to Giorgos for all the professional and research opportunities he offered me.

I would also like to thank the members of my advisory and defense committee, Ioannis Andreadis, Chrysostomos Nicopoulos, Spyridon Nikolaidis and Georgios Sirakoulis for evaluating and reviewing my work, and for providing insightful comments that helped to improve this thesis. I am very thankful to Chrysovalantis Kavousianos for his useful technical feedback to refine this work. Also, my sincere thanks goes to Ioannis Karafyllidis for providing me with his helpful advise.

Moreover, I need to thank my colleagues, Pavlos Mattheakis, Laurent Masse-Navete, Pierre-Olivier Ribet, Nikita Nikitin, Javier de San Pedro Martin and Joseph Shinnerl for the fruitful collaboration we had together in real industrial environment, as well as for transferring to me crucial knowledge.

I am very grateful to the funding received through the Onassis Foundation and its Program of Scholarships for Hellenes.

Many thanks go to my friends and labmates Dimitris Konstantinou, Apostolis Stefanidis, Ioannis Seitanidis, Tasos Martidis, Zacharias Takakis, Dionisis Filippas and Christos Gkantidis for the support, the many interesting discussions and exchanges of ideas as well as the productive working environment when we shared a common office. Nonetheless, the time spent together at night outs helped me a lot to overcome the difficulties of this journey.

Last but not least, I would like to thank my parents, Moschos and Vasiliki, as well as, my sister, Dimitra, for their continuous support, encouragement and for feeling empathy with my concerns. This journey would not have been possible without them, and therefore, I dedicate this thesis to them.

Contents

Acknowledgements iii

1 Introduction 1
- 1.1 Physical Synthesis . . . 2
- 1.2 Towards timing closure . . . 3
  - 1.2.1 Timing optimization during Logic synthesis . . . 5
  - 1.2.2 Timing-driven Placement . . . 6
  - 1.2.3 Useful clock skew . . . 8
  - 1.2.4 Interconnect delay optimization . . . 11
  - 1.2.5 Logic Restructuring . . . 15
  - 1.2.6 Integrated optimizations . . . 19
- 1.3 Thesis Contribution . . . 20
- 1.4 Thesis Organization . . . 22

2 Lagrangian-Relaxation based Timing-driven Placement 25
- 2.1 Introduction . . . 25
- 2.2 Timing Compatibility Flip-Flop Clustering . . . 27
  - 2.2.1 Assign a Timing Profile to each Flip-Flop . . . 29
  - 2.2.2 Initialize Clusters and Prioritize Flip-Flops . . . 29
  - 2.2.3 Flip-Flop Clustering . . . 30
  - 2.2.4 Update Cluster Centers and Timing Profiles . . . 32
  - 2.2.5 Clustering Behavior . . . 32
- 2.3 LR-Based Timing Optimization . . . 33
- 2.4 Overall Flow and LR-based Cell Relocation . . . 39
  - 2.4.1 Local Cost Function . . . 42
  - 2.4.2 Lagrange Multiplier Update . . . 43
  - 2.4.3 Timing Recovery with Flip-Flop-to-LCB Re-assignment . . . 44
- 2.5 Placement of the Search Window . . . 44
- 2.6 Experimental Results . . . 47
  - 2.6.1 Comparison with winner of the ICCAD 2015 contest . . . 48
  - 2.6.2 Comparison with recent state-of-the-art . . . 50

2.6.3 Runtime comparisons 54
2.7 Conclusions 55

3 Incremental Lagrangian-Relaxation based Discrete Gate Sizing and Threshold Voltage Assignment 57
3.1 Introduction 57
3.2 Basics of LR-based gate sizing 59
3.3 Incremental LR-based gate sizing 63
3.3.1 What is the problem? 64
3.3.2 What can we do about it? 65
3.4 Experimental Results 67
3.4.1 Quality-of-Results and Runtime comparisons 67
3.4.2 Exploring in depth the proposed LM initialization 71
3.4.3 Optimization with a restricted number of available gate sizes . . 74
3.5 Conclusions 75

4 Task-based Parallel Programming for Gate Sizing 77
4.1 Introduction 77
4.2 Related Work 79
4.3 Generic Gate Sizing Template 80
4.4 Initial sizing 81
4.5 Main gate sizing optimization 84
4.5.1 Forward Pass 84
4.5.2 Backward pass 90
4.6 Timing and Power Recovery 93
4.7 Experimental Results 96
4.7.1 The characteristics of the tasks graphs 96
4.7.2 Comparison with state-of-the-art 96
4.7.3 Highlighting the contribution of RTS and FLTU 102
4.7.4 The contribution of final timing and power recovery 105
4.7.5 Recovery with Composite Tasks 108
4.8 Conclusions 109

5 Flip-flop Placement Targeting Clock-induced OCV 111
5.1 Introduction 111
5.2 Motivation–Problem formulation 112
5.3 Soft Clustering-based Placement 116
5.3.1 Initialize cluster centers 117
5.3.2 Compute membership function 118

5.3.3 Update cluster center 120
5.3.4 Relocate Cells 121
5.3.5 Algorithm complexity 123
5.4 Experimental Results 124
5.4.1 Timing comparisons 124
5.4.2 Clock-induced OCV redistribution 126
5.4.3 Clock tree complexity 127
5.5 Conclusions 128

6 Conclusions **131**
6.1 Summary 131
6.2 Future Work 132

Bibliography **135**

1 Introduction

Semiconductor and electronics industries have enjoyed tremendous growth and innovations over the past 50 years thanks to Moore's law that guided the rate of scaling of transistor dimensions over time. Historically, we were able to harness all of the available transistors to deliver exponential increases in computational power by capitalizing on technological improvements, architectural innovation as well as *evolution of integrated circuit (IC) design tools* that allowed us to manage extremely complex designs in a timely manner.

Digital IC design tools, broadly categorized as electronic design automation (EDA), has always been the connection between technological improvements and the designer. Simulation and verification tools support early architecture exploration and microarchitecture design phases where the design is verified that it correctly performs the desired functions and that it achieves the needed performance in terms of computational efficiency. Physical synthesis tools, which is the focus of this thesis, transform the verified microarchitecture-level designs to ready-to-be-fabricated chips. The logic synthesizer maps the design into a gate-level netlist in the targeted technology of the chip. Placement and routing engines complete the chip's floorplan, place the cells of the design, synthesize the clock tree and connect the cells together with appropriate wiring. Incremental timing and power optimizers bind together the gross placement and routing steps thus simplifying timing closure (e.g., satisfying designer's timing constraints) and achieving significant power and area reductions.

As fabrication processes shrink in dimensions, the ICs become commensurately more complex, and, as any design engineer appreciates, complexity increases design turnaround time (TAT) and makes harder to improve results in power, performance and area (PPA) above a threshold that would justify a new product in a new technology node. In EDA, results are everything and to remain competitive, you can't afford to make any tradeoffs to either PPA or TAT, even if optimizing under multiple operating modes and multiple PVT (process, voltage, temperature) corners. A scalable physical synthesis engine should enable deeper solution-space exploration for increasing the quality-of-results (QoR) in terms of PPA, and execute with reasonable runtime and memory utilization on blocks of constantly increasing complexity. At 7nm technology nodes, 10 million cell blocks are becoming the norm, and ICs can easily integrate 100 plus such blocks. Keeping pace with this escalation necessitates higher design efficiency and,

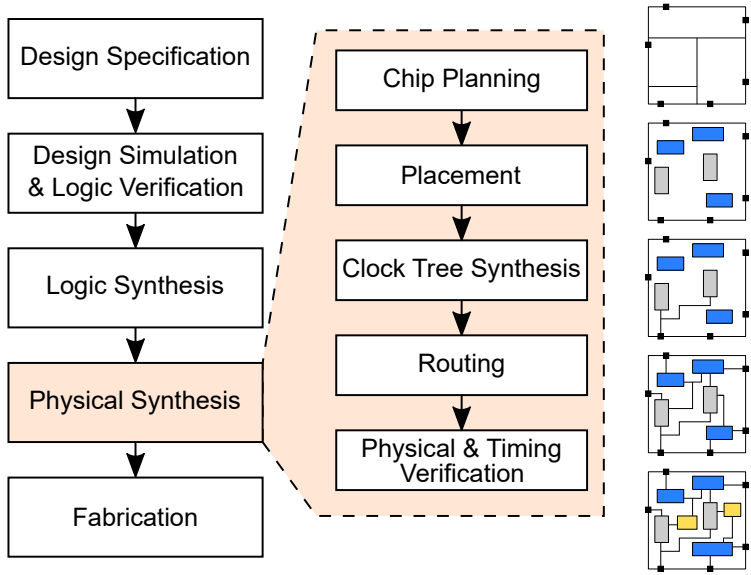


Figure 1.1: The process of VLSI design and the main steps of the physical flow.

today, that can be most effectively addressed by runtime-efficient physical synthesis optimization algorithms.

1.1 Physical Synthesis

Physical synthesis refers to the process of placing and routing the logic netlist of a design while concurrently optimizing for multiple objectives like timing, power, area and routability constraints. To achieve this, physical synthesis consists of multiple iterative techniques which are shown in the right part of Figure 1.1.

Chip Planning: The IC is partitioned into smaller individual subcircuits that can be designed in parallel and independently. At the same time, the identified blocks should be floorplanned selecting for each one its shape and size. Some of these blocks are mapped to rectangular shapes with changeable dimensions while some others can have fixed dimensions with possibly fixed locations. Also, in this step, the locations of the I/O ports are determined, and the structure of the power delivery network is determined.

Placement should find the physical location of every logic cell of the design. To

perform this complex task, placement works in two steps: global and detailed placement. During global placement the movable cells of each block are distributed within the block's boundaries determined during chip planning. The locations of the cells are roughly specified allowing overlaps between them. The detailed placement (*aka* legalization) step that follows specifies the final location of the cells so that each cell is aligned to the designs rows and sites without overlaps. During this step, the cells are moved to nearby locations in order to improve the circuit delay and utilization.

Clock Tree Synthesis: The purpose of clock tree synthesis is to automatically generate a clock network of buffers and wires that connects all clocked storage elements of the design with the source of the clock. Besides mere connectivity, clock tree synthesis ensures that the clock signal exhibit fast transition times and gets delivered with the appropriate latency. In addition, the clock tree is equipped with clock gating logic that significantly reduces the design power by switching off parts of the circuit that are not in use.

Routing: In global routing the routing resources are allocated in a more roughly way to connect the design elements. Then, detail routing specifies for every wire connection to which specific metal layers is assigned and which routing tracks are used. Routing besides identifying the appropriate wiring paths for all nets of the design, should ensure that timing is not negatively affected by the RC parasitics of wires, and also that there aren't any wire-congested regions.

Physical and Timing Verification: Before IC fabrication, it is important to ensure its proper timing with the given clock target. In other words, all the timing constraints including the setup and hold analysis have to be met. For this reason, timing is also part of the objective in each step of the physical design flow. However, if timing is not closed at the end, other incremental timing optimization methods are invoked to eliminate the timing violations. Timing verification is performed by sign-off static timing analyzers that check the timing behavior of all the timing paths of the the design again multiple timing constraints.

1.2 Towards timing closure

Timing closure is a complex process that involves multiple iterative optimization steps that are applied multiple times during physical synthesis or are integrated inside global physical synthesis algorithms such as placement and routing. Given timing constraints and the characteristics of the technology as reflected to the delay of the gates, the wires and the sequential elements, the goal of timing closure is to *synthesize automatically* the physical structure of the design having all timing constraints satisfied. Removing any possible timing violation is a hard requirement that enables design to operate correctly

after fabrication and cannot be skipped without performance degradation.

The challenges that timing closure faces nowadays can be summarizes as follows:

- *Design's size:* In the designs with millions or even billions of cells, the optimization algorithms need to scale well in order to complete in reasonable time. The speedup of the optimizations becomes more essential because the turnaround time is specified to only 12 hours per million of cells for the whole physical synthesis flow [120].

- *Multiple modes and corners:* The chip designs operate under many different operating conditions with different electrical properties and thus there are multiple active modes and corners of which the timing constraints have to be satisfied simultaneously [94, 135]. However, trying to remove a timing violation from one timing scenario could easily create a new violation in another making the physical process even more challenging.

- *On-Chip Variations:* The On-Chip Variations (OCV) effect refers to the intrinsic variability involved in semiconductor manufacturing processes and the fluctuation of operating conditions, such as voltage and temperature, and how they impact a circuit's timing [34]. The OCV increases the delay of the cells in the launch path and concurrently decreases the delay of the cells in the capture path tightening the timing constraints.

- *Low power designs:* Reducing power consumption has become a key design challenge at advanced technology nodes. For many IC designs, optimizing for power is as important as timing, due to the need to extend battery lifetime, reduce packaging and cooling costs, and permit higher device performance.

- *Higher interconnect RC:* Even though, transistors successfully followed the technology scaling, the wires did not scale accordingly. This resulted the wires to have increased resistance and the interconnection delay not only to have by far the lion's share of total delay, but also its variation across the stack has reached over one order of magnitude between the lowest and the highest metal layers with dramatically increase of the vias resistance.

To meet all the timing constraints, there is a wide variety of optimization techniques that are applied on every step of design flow. Even though the first stages of design flow, such as logic synthesis and placement, are not targeting explicitly timing closure, they include algorithms to minimize the path delays and remove a big number of the initial violations. As the design passes through the next steps of the design flow, timing becomes more accurate that enables finer grained optimizations. For instance, during

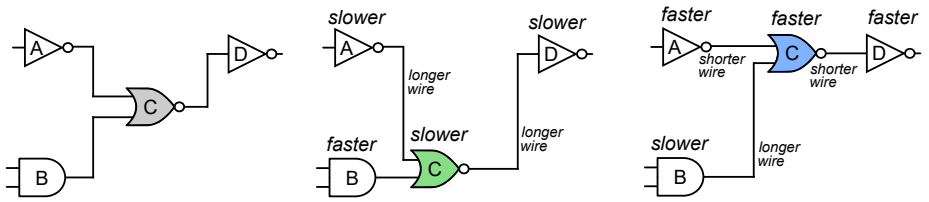
clock tree construction the useful clock skew applies specific target clock delays on each register that will improve the timing in overall, while routing inserts extra cells to cut long wires into short wire segments and finally reduce the interconnection delay. There are also many timing optimizations techniques, such as gate sizing and buffering, that are applied incrementally between (or even inside) the main steps of the design flow. The recent years integrated approaches have also emerged that interleave in a tightly-coupled manner various timing optimization methods that were traditionally applied separately with a predetermined serial order.

1.2.1 Timing optimization during Logic synthesis

Logic synthesis usually involves two separate optimizations phases. Firstly, technology-independent optimizations are applied with the goal to minimize the area and the delay of the design assuming that the netlist consists of generic gates. Later, during the technology-dependent optimizations the area and the delay of the standard cells in the library of the specific technology are used in order to meet the timing requirements. Both stages try to minimize the propagation delay so that a netlist with less timing violations is given to the rest part of the physical synthesis flow.

Reduction of the logic depth is a well-known technique to improve timing. The intuition of this approach is that reducing the logic depth less gates are involved in the critical path and therefore the propagation delay of the path is improved. Initially, Singh *et al.* [147] proposed a methodology that re-synthesizes a subset of the critical paths to reduce the delay without significant area increase. The resynthesis first decides to collapse some logic in the critical paths and then decomposes the respective nodes decreasing the longest path delay. The identification of the subsets is done using a weighted min-cut algorithm. Fishburn [41] proposed a greedy heuristic that decreases the initial logic depth N to $\log N$. The heuristic iteratively selects a subcircuit of the critical path, tries different transformations and finally applies the solution that improves the timing. Another successful technique to reduce the logic depth introduced by Cortadella [28] that interleaves the simplification of boolean functions using different functions with simpler components and tree-height decrease of boolean expressions. As the need to achieve higher performance increased even the technology mapping step started to consider the delay. One of the first attempts was to change the originally used tree-based min-area mapping algorithm so that the initial netlist has the minimum number of logic levels and the objective targets the arrival time minimization instead of the area [137]. Later proposed attempts are trying to reduce the path delays lower than a threshold, that is usually the clock cycle, to guarantee correct functionality [19].

Retiming has been also proved to be an effective timing optimization during logic synthesis. More specifically, changing the position of the sequential elements without



(a): Initial circuit (b): Gate C approaches B (c): Gate C closer to A and D

Figure 1.2: Placement changes the wire length around the moved cell and consequently alters the wire delay. In (b) the gate C is placed closer to gate B resulting to shorten the wire that connects B and C and increase the rest wires. In (c) the gate C is moved closer to both A and D, accelerating them and increasing the delay of gate B.

affecting the design behavior can alter the propagation delay. There are two types of retiming; in forward retiming a flip-flop is moved forward and from fanin element of a gate it becomes fanout of this gate, while the opposite happens in backward retiming [145]. However, the challenging part is to identify which flip-flops need to be retimed and in which direction in order to achieve the highest performance gains. Combined with the retiming, the latest industrial tools also use clock scheduling algorithms that change the clock latencies in order to achieve optimal slack balance of the sequential elements [1].

1.2.2 Timing-driven Placement

Timing-driven placement is a significant step to the performance of the design because it determines the physical location of the cells and thus directly affects the length of the wires as well as the wire delay. A placement without considering timing, may cause long interconnections with increased resistance and capacitance that would slow down the propagation of the signal. As shown in Figure 1.2, the different location of gate C can favor different gates that are directly connected to it. For instance, in Figure 1.2(b), gate C is placed closer to gate B shortening their in between wire. This results to faster propagation delay of both gate B and their interconnect delay but to slower delay of all the other interconnections and gates. If the gate C is moved closer to both A and D (Figure 1.2(c)), the timing of both A and D is improved at the expense of the timing of gate B. It is clear that timing-driven placement is crucial to obtain a solution with significant reduced timing violations. To achieve this all the timing violations have to be considered in order to properly relocate the gates. However, the global placement usually does not consider timing because in the early stages of the physical flow the

locations are not available yet and therefore the estimation of the wire and gate delay becomes very inaccurate. Instead, the timing is preferred to be optimized during detailed placement where the locations are known and the timing estimations are more accurate.

Timing-driven placement approaches can be divided into three different categories; net based, path based and hybrids. The net based methods usually assign net weights to each net that reflect the timing criticality and they try to minimize a weighted wirelength cost function iteratively. The idea is the weights to be proportional to the timing criticality of the nets. In other words, nets with negative slack get weights with higher values than the non-critical nets, so that the critical nets contribute more to the cost function and therefore guide the placer to decrease the length of these nets.

The assigned weights can either remain static during the optimization iterations or they can change dynamically i.e. in each iteration a timing analysis is performed and the weights are updated to reflect the new criticality of the design. For instance, in [84] the authors assign static nets weights according to the number of paths. To do this, they also propose a new algorithm that counts accurately the number of paths. The later work of [132], introduced a new net weighting method that targets also to minimize the total negative slack instead of only worst negative slack.

Although static weights can improve timing, the dynamic weights are more promising because they are updated in each iteration and thus they never become stale as it can happen with the static net weights. A new force directed placer proposed in [35] that was targeting timing closure. In order to avoid oscillations, the net weights are updated taking into account the values of the previous iterations. To increase the contribution of the timing critical nets in the cost function, the authors of [80] used an exponential function to increase the weights of the nets with negative slack. For more accurate wire delay computation, the authors of [133] used the Elmore delay model accompanied by the star model for the wirelength estimation. Without limitation, Lagrangian Relaxation, that has been widely used in gate sizing, is also applied in placement to minimize the total number of violating endpoints [53, 96, 165]. In this case, Lagrange Multipliers are net weights and they are used to solve the Lagrangian Dual Problem of the Lagrangian Relaxation Subproblem.

Some alternative approaches of net based techniques assign wire length or delay constraints to nets instead of weights [48,49,154]. For example, wire length constraints can be assigned into nets using a linear programming which are later met by analytic placer engines [55] or using a force directed placer [125]. Typically, for the placer engines is more complicated to meet the net or delay constraints because it is really difficult to know the exact effect of a net change to the delay. On top of this, the assigned constraints can further limit the solution space.

In the path based techniques, the minimization of the total or worst negative slack is typically achieved using linear programming. In the formulation, all or part of the paths

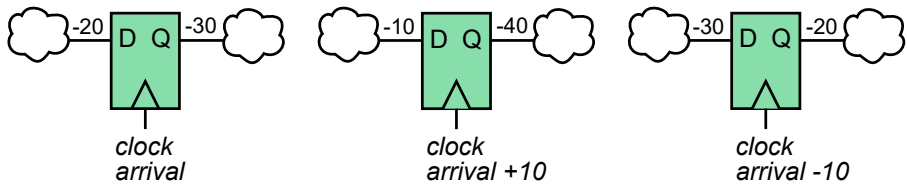
with negative slack are converted to form the constraints of the problem. For example, Srinivasan *et al.* [149] used the Lagrangian Relaxation to relax the linear programming problem and smooth the critical paths. The authors of [163] used simulated annealing to decrease the path delay of the paths with negative slack. A different algorithm, enumerates the possible candidate locations for each gate with the corresponding changes in the delay in order to solve timing violations and uses the branch-and-bound method to avoid exhaustive search across all possible combinations [111]. The authors of [25] determine the locations of the cells solving a linear programming problem in which the changes in timing are computed using a differential timing model. Later, pin-based constraints were added in the linear programming of the timing driven placement engine [131] to significantly reduce the timing violations and the wirelength at the same time. The main drawback of the path based approaches is the prohibitive runtime as the design size increases because the number of paths grow exponentially and the same does the number of variables in the linear programming.

Regarding the hybrid timing-driven methods, they combine features of both net and path based techniques. As an example, the authors of [103] proposed a linear programming that takes into account the critical gates as well as the non-critical gates near to the critical. Even though the formulation uses net weights it also takes advantage of the path based delay sensitivity. In [46] a quadratic placement algorithm from global placement is used to operate solely on the critical paths after assigning them appropriate weights in order to improve their timing. The ITOP method [158] uses an accurate timing analyzer instead of simplified timing models to perform critical path optimization and slack histogram compression and it finally achieves significant timing improvements without degrading the routability of the design.

There are some other alternative methods that have been also studied and they do not clearly belong to any of the previous categories. OWARU [77] improves the timing of the critical paths by smoothing their physical curve using the Bézier curves. While most of the techniques target the setup optimization, there are approaches which improve the hold slacks too. [44] proposed a set of algorithms to decrease hold violation such as moving apart the registers without combinational logic in between them or exchanging the physical locations of the appropriate registers. Similarly, [67] proposed re-assignment of the flip-flops to new local clock buffers as well as flip-flop and local clock buffer relocation that in total can improve significantly the hold slacks at no cost of setup slacks.

1.2.3 Useful clock skew

Despite the data paths, the clock tree paths have also a significant impact on the timing violations. The clock tree connects all the sequential elements of the design in order



(a): Initial circuit (b): Clock arrives later (c): Clock arrives earlier

Figure 1.3: Changing the clock arrival affects the timing at both side of the flip-flops. (b): A later clock arrival improves the timing at the D side and degrades the Q side. (c): When the clock signal arrives earlier, the slack of the starting paths is improved and the ending paths become worse.

to distribute them the clock signal. Even though there is the assumption that the clock signal arrives at all the sequential elements at the same time, this is not possible considering the different propagation delays of the clock tree branches. However, determining appropriately different clock arrivals at each clock element can improve the slack of the data paths. For instance, delaying the clock arrival of a flip-flop, as shown in Figure 1.3(b), improves the paths of the D side because the setup required arrival time is increased, and degrades the paths that start from the Q side. In contrast, an earlier clock arrival favors the timing of the Q side at the expense of the timing at the D side (Figure 1.3(c)). Obviously, altering the clock arrival can improve the timing violations without even touching the combinational paths. But the main target of this optimization is to determine which registers should receive a later and which an earlier clock signal, as well as by how much in order the overall timing to be improved. This approach is known as useful clock skew optimization.

Firstly, Fishburn [42] proposed a linear program that determines the clock tree delays of the flip-flops in order to achieve higher performance under both setup and hold timing constraints. The clock period minimization problem extended to include also uncertainty factors and a different solution proposed using a graph-based approach [33]. Later, a linear program was introduced that targets the potential slack budgeting by changing the clock skew [159]. The UST/DME [153] engine applies skews to the clock tree elements and at the same time determines the locations of the clock tree routes using deferred-merge embedding algorithm with minimal impact on the wirelength. In addition, to achieve higher global timing improvements in critical and non-critical paths, the useful skew can be formulated as a maximum mean weight cycle (MMWC) [2] problem. Wang *et al.* [160] proposed a clock skew scheduler with almost linear runtime, while Wei *et al.* [162] achieved significant speedup of the MMWC using a fast

negative cycle detection method. The predictive useful skew methodology NOLO [18], that chooses the skews before placement phase and applies them with only one pass – without iteration, achieves better total negative slack. Considering multiple modes and multiple corners, Lung *et al.* [102] used a linear programming model to optimize skew, while in [136], positive and negative offsets are applied on each clock pin of an already constructed clock tree in order to improve the timing in multiple modes and corners.

Moreover, as the technology scales further, the process, voltage and temperature (PVT) variations introduce additional uncertainties which highly impact the timing. In other words, some cells become faster while some other cells slower than expecting making even more challenging to meet the timing constraints of the paths. For this reason, the authors of [37] presented the idea of constructing useful skew trees with large safe margins so that the skew constraints can be met even when there are variations which degrade the timing, while later in [39] they proposed the usage of a latency constraint graph based on skew constraints and latencies to finally build a robust latency-bounded clock tree that can increase the design's performance. [127] and [17] reduce the impact of variations by minimizing the clock divergence, while the authors of [156] reconstruct the clock tree using a predicted leaf buffer slack graph to reconnect buffers and achieve higher improvements of timing metrics like total negative slack.

The useful skew optimizations are vital to achieve timing closure. However, the fact that the majority of the skews are computed theoretically even before the clock tree by using a linear programming is their main drawback. The reason is that the solver determines the values of the variables that minimize the selected function without knowing what is the cost to implement the corresponding delays in practice. For example, some solutions may be infeasible just because they require a significant number of clock buffers in order to implement them which degrade the clock tree latency, the clock power consumption, etc. In other words, it is not easy to predict what are the consequences of the theoretically computed skews in the clock tree features such as the number of clock buffers, the power and the area.

In addition, traditional skew optimization approaches assume the combinational paths have constant delays and thus try to improve the timing changing only the clock delays. At the same time, every data path optimizer assumes fixed clock tree delays and focuses only on how to improve the slack of the gates in the data path. But there is no argument that every attempt to integrate different physical optimizations has resulted to much better timing. For instance, [1, 3] integrate the useful clock skew with the RTL synthesis stage for higher performance, while in [73] the placement approach uses additionally useful skew to improve the increased delay of the timing critical paths achieving significant improvements. Since both data path and useful skew approaches have the same timing target, much effort has to be made on how these methods can be integrated which mostly are applied as different optimization phases until now.

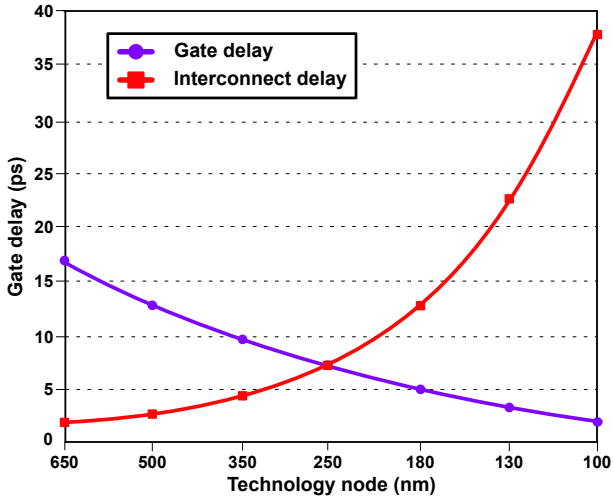
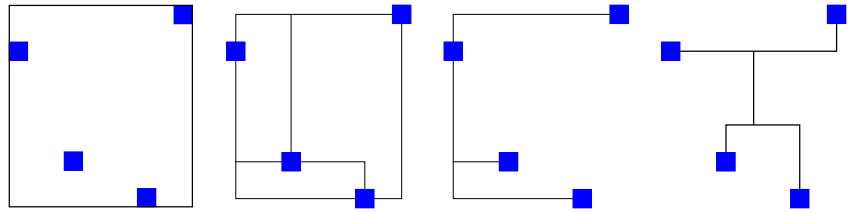


Figure 1.4: Gate delay and interconnect delay as the technology scales. Initially, the delay of the gates was dominating the wire delay in the computation of the path delay but in newer technologies the interconnect delay has become the bottleneck of the timing.

1.2.4 Interconnect delay optimization

With the continuous scaling of the technology, interconnect delay started to contribute more and more to the path delay. Figure 1.4 shows the interconnect and gate delay for different technologies. Even though gates could follow the technology scaling, interconnect was scaling slower leading to wires with high resistance. As a result, the interconnect delay became the bottleneck of the path delays in the newer technologies and new timing optimizations had to be investigated.

Since the need to take into account the interconnection delay arose, different models have been used in order to estimate the wire-length of the net and then to compute the wire delay. Initially, the Elmore delay model was used that computes with precision the wire delay of a net with only one sink. However, for nets with multiple sinks, the wire delay calculation wasn't accurate enough and thus the moment matching [122] was proposed. Besides these, other models were also tried as illustrated in Figure 1.5. For example, in half perimeter wirelength (Figure 1.5(a)) a rough estimation of the wirelength is done computing the half perimeter of the minimum bounding box that encloses all the pins of the net. But even this approach, becomes inaccurate for the nets with more



(a): HPWL (b): Clique (c): Star (d): Steiner tree

Figure 1.5: Wirelength estimation models for nets with multiple pins.

than 4 pins. In clique model (Figure 1.5(b)) each pin is directly connected to each other pin using only horizontal and vertical wires, while in the star model (Figure 1.5(c)) there are wires connecting the source pin to each sink independently. The drawback with the clique is that overestimates the wirelength. The most accurate model is the steiner tree depicted in Figure 1.5(d). For the construction, additional points (Steiner points) are inserted but the problem of finding the optimal tree is NP-hard. Thanks to FLUTE, accurate and optimal steiner trees can be constructed very fast using precomputed lookup tables [26].

Routing

Typically, a net with increased length affects the designs performance in two ways. First of all, more time is needed for the signal to propagate from the driver cell to the sinks, and secondly the increased capacitance of the interconnection slows down the driver whose delay is a function of the input transition and the output load capacitance. Usually, the routing methods determine the wire routes so that congested areas are avoided in the design. However, as the interconnection delay started to be substantial, even routing approaches have become timing-driven.

Wire topology optimization

Some of the timing-driven routers target to decrease the maximum wire delay from the driver to a specific sink. Boese *et al.* [14] introduced an algorithm that alters the traditional routing in order to build routing trees for multi-sink nets with reduced Elmore wire delay towards specific timing critical sinks. In a similar way, in [83] a heuristic approach is proposed to build a steiner tree that minimizes the delay of the sink with the most negative slack. To achieve this, first a minimum-cost steiner tree is constructed that connects all the sinks except of the most critical. For the unconnected sink a set of

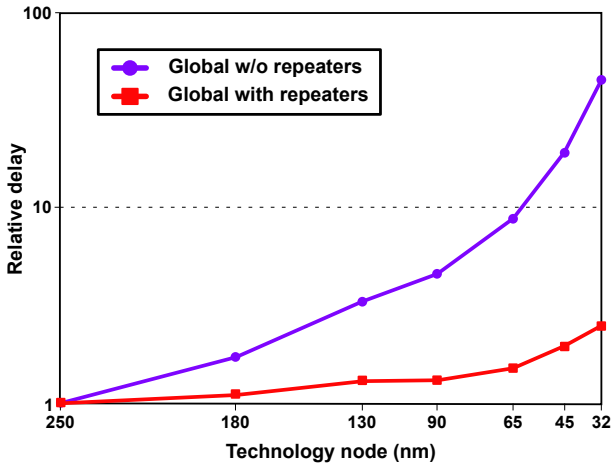
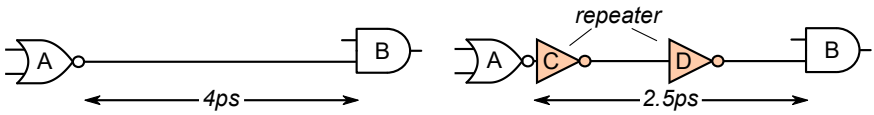


Figure 1.6: As the technology scales, becomes more and more meaningful to add repeaters on the nets which can significantly reduce the interconnection delay.

different heuristic methods is tried and finally the most critical sink is connected to the steiner tree using the solution that minimizes its interconnection delay.

Some other routers, try to route the design but avoid creating long wires for the critical nets that could degrade further the timing. For example, a timing-driven algorithm uses bounded-radius minimum routing tree formulation and achieves reduction of the total net length and at the same time minimizes the interconnection length of the longest wire that connects source and sink [27]. Also, the [6] work enhanced the algorithm presented in [5] introducing different costs for critical and non-critical nets so that the net construction determine shorter wire routes for the nets with negative slack. Other newer approaches, propose algorithms that reduce the usage of the vias [168] which have large process variation and thus can compromise the timing. In the modern technologies there are available for routing up to twelve metal layers, each of them with different characteristics [8]. Typically, the upper metal layers are more suitable to be used for timing critical wires because they are thicker with significant lower resistance. Hu *et al.* [65] proposed a layer assignment methodology that improves the timing of the design and achieves significant speedups compared to the older approaches. Also, Lagrangian Relaxation has been also used to minimize the designs total negative slack by re-assigning the net segments to the most suitable metal layers while satisfying the routing capacity constraints [95].



(a): Initial circuit (b): Repeaters C and D inserted

Figure 1.7: Repeaters are inserted to cut long wires to shorter equal segments reducing the interconnection delay. In (a) the initial long wire resulted to 4ps wire delay, while in (b) after adding two repeaters the propagation delay from gate A to B reduced to 2.5ps even though two extra cell delays are including in the path delay.

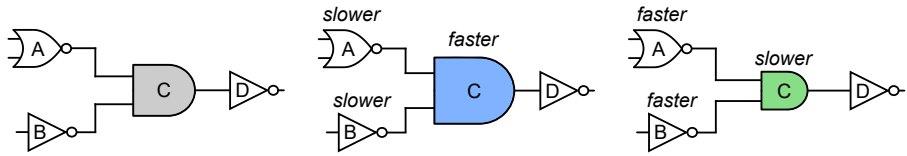
Adding repeaters

Another effective technique to alleviate the propagation delay of long wires is to partition them using repeaters. Figure 1.6 depicts the relative delay of the wires over different technologies. In the early technologies, even without adding repeaters the interconnection delay remains low. However, after 130nm it becomes clear that inserting repeaters is essential for timing closure because in this way the propagation delay is reduced significantly.

The repeaters are added to cut the long wire to a number of equal shorter wire segments so that the total propagation is lower than previously. As illustrated in Figure 1.7(b) after adding the repeaters C and D in the long wire that connects the gates A and B, the propagation delay has decreased by 1.5ps compared to its initial delay even though the new delay involves the cell delays of the two repeaters. However, the number of repeaters is crucial in order to decrease the wire delay.

To do that, there is the assumption that the initial long wire will be cut to equal wire segments and the repeaters will have the same size. This means that every wire segment has the same propagation delay and the new total propagation delay is the summation of the propagation delays of each of the segments. Therefore, by just differentiating the formula of the new delay by the length of the wire segment, the optimal length of each segment is computed that minimizes the new propagation delay.

However, inserting repeaters has two main drawbacks. First of all, since the repeaters are usually inverters, always an even number of repeaters needs to be inserted in order to not change the polarity of the propagated signal. This means that in cases where the optimal number of repeaters is odd, a suboptimal number of repeaters is finally inserted. Also, each repeater contributes to the total power and area of the design. In other words, even though they reduce the propagation delay, they should be used frugally to keep the area and the power under control.



(a): Initial circuit (b): Gate C is upsized (c): Gate C is downsized

Figure 1.8: Gate sizing changes the timing characteristics of the sized gate and consequently changes the delay of the gates connected to it. In (b) the gate C is upsized resulting to accelerate itself while slowing down its fanin compared to their initial delays in (a). In (c) the gate C is downsized resulting to slower gate C but faster fanin gates.

1.2.5 Logic Restructuring

Gate sizing

Gate sizing refers to the process of optimizing the sizes of the transistor in order to satisfy timing constraints and reduce gate's area and leakage power. In the past, the sizes of the gates' transistor could have been selected arbitrarily. Nowadays, individual transistors are not separately optimized since the design is mapped to a pre-specified standard-cell library during logic synthesis. Cell libraries contain multiple equivalent options for each gate and flip-flop that has the same logic functionality but offers different area, leakage power, and timing characteristics.

Upsizing a logic gate decreases its delay since the wider transistor can (dis)charge faster its output capacitance. However, at the same time, gate upsizing increases its input capacitance that acts as output capacitance for its fanin gates, as highlighted in Figure 1.8(b). The opposite happens when downsizing a gate. For instance, in the example shown in Figure 1.8(c) down sizing gate C slows down itself and accelerates its fanin (gates A and B) due to the lower input pin capacitances.

Gate sizing needs to select which cell fits best in each occasion by trading off area (and power) with delay. However, the decision to upsize or downsize a gate can be very challenging when considering all the critical paths that pass through a gate. Therefore, the target is to properly determine the most suitable size for each logic gate that improves the timing in overall.

Gate sizing does not change only the size of the corresponding cells but it also selects the appropriate threshold voltage for each cell too. Choosing a transistor with lower threshold voltage decreases the needed time to charge the output load reducing the propagation delay. However, the lower threshold voltage increases the leakage cur-

rent that flows in the transistors and consequently increases its leakage power. Deep sub micron libraries usually provide three types of threshold voltage for their transistors, i.e low, standard and high threshold voltage. Thus, gate sizing has to also select the appropriate threshold voltage transistors that improve the designs performance without significant power increase.

During the eighties and nineties gate sizing was modeled as convex problem assuming that each gate could take any size in a range of transistor sizes. Posynomial functions were used for estimating the delay of each gate [20, 43]. According to [138], similar problems were solved with geometric programming. The drawback was the high memory and the runtime, so its application was limited to only small designs [15]. As technologies continued to scale, delay modeling with posynomial functions became highly inaccurate. Also, continuous transistor sizing was abandoned and each design was mapped to a library of discretely-sized cells. To adjust to this changing environment, gate sizing algorithms "rounded" the continuous solution to the closest discrete cell. [64].

From this point forward, all gate sizing algorithms worked directly on the discrete cells [29] Assuming only discrete sizes, gate sizing becomes an NP-hard problem [115]. Pseudo-polynomial dynamic programming has been used to solve such problem but their application was limited for netlists with tree structure [16]. In addition, gate sizing has been solved using linear programming (LP). Slack can be distributed to gates using LP to maximize the power reduction [113]. This work was further enhanced by proposing an LP that takes into consideration the wire loads and the impact of slew and input capacitance changes on delay and thus obtains better QoR [23]. However, these approaches aren't suitable for large designs due to the prohibitive runtime as the number of variables increases.

Alternative methods, choose the new size using a sensitivity function. In these cases, the gates are resized to new sizes that maximize the selected gain under constraints. The optimization usually involves two phases in which the power and the timing are improved separately. For instance, Hu *et al.* [63] proposed a sensitivity-guided metaheuristic method that optimizes power and timing and he revised the sensitivity functions later in [78] to include the impact of slew on delay increasing the accuracy of the delay computation. Sensitivity guided methods have wide application range and they have been applied even on design under multiple corners. More specifically, multi-corner multi-mode sensitivity functions were presented in [40] to effectively resize the gates in order to meet the timing constraints and save power. The main drawback with the sensitivity heuristics is that they may converge to local optimal point and thus the best solution can not be guaranteed.

From all the available gate sizing approaches, those that are based on Lagrangian Relaxation (LR) obtain final solutions with higher timing improvements. In these ap-

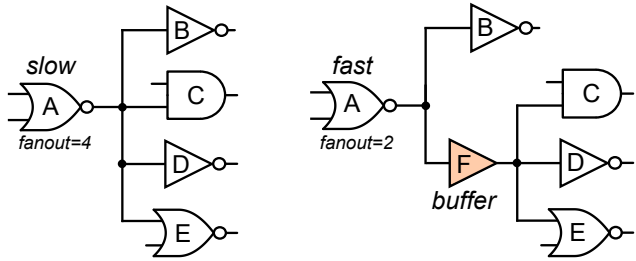
proaches usually the cost function is the total power minimization while respecting the timing constraints. Then, the timing constraints, which are hard to be respected, are inserted in the initial cost function multiplied by non-negative weights forming the relaxed version of the initial cost function. These weights, which are called Lagrange Multipliers (LMs), act as penalty factors and thus they have high values only if the corresponding timing constraints get violated. The relaxed cost function is minimized iteratively and each iteration involves two separate steps; the solution of the Lagrangian Dual Problem (LDP) and the solution of the Lagrangian Relaxation Subproblem (LRS). In the LDP phase, the values of the LMs are updated in order to reflect the criticality of the design and this is usually done using a subgradient method [20]. The target of LRS is to determine the sizes for each gate that minimize the relaxed cost function assuming constant values for the LMs.

There is a wide variety of gate sizing techniques using LR. For example, LR has been used to size gates and wires simultaneously [20]. Ozdal *et al.* [118] uses a graph model in which each available equivalent discrete gate size is represented by a node and the edge weights capture the delay costs. The most efficient sizes of the LR-based gate sizing problem are chosen by dynamic programming. Contrary to previous LR-based works, the selection of gate sizes that minimize the local LR cost can obtain better timing results with further reduced leakage power, is introduced in [47]. But due to the single threaded implementation, the work reported high run times. Later, Sharma *et al.* [143] proposed new equations for the update of the Lagrange Multipliers (LMs) and how the LR-based gate sizing can be implemented with multiple threads to obtain faster results. Since most of the designs operate under different conditions, the LR cost can be extended to effectively perform power and timing optimization under multiple scenarios [134]. For better final results, clock-related formulations were also included in the LR method leading to resize gates in both data and clock paths [146]. The LR can be also extended to adjust clock arrival times resulting to higher power savings as evaluated in [141].

Buffering

As the technology continued to scale further more, the interconnection delay had more and more high impact on the timing leading the engineers to find new and more effective techniques to handle it. Even though placement and routing could improve the wire delay, for some specific nets these techniques were insufficient. For example, nets with high degree of fanouts needed a special treatment such as adding buffers.

Typically, a buffer consists of two serially-connected inverters. Usually, the buffers are inserted to increase the drive strength or to accelerate the drivers by shielding them from a portion of the load they drive mostly in the nets with multiple fanouts or fanouts



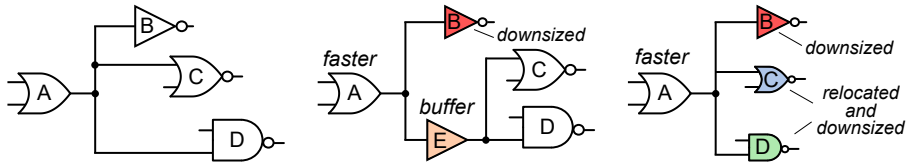
(a): Initial circuit (b): Buffer F inserted to shield gate A

Figure 1.9: (a): Gate A drives 4 gates which slow down its propagation delay. In (b) a buffer is inserted that drives the 3 of the original fanouts. The buffer shields the gate A accelerating its delay.

with increased input pin capacitances. For instance, in Figure 1.9(a), gate A has increased propagation delay due to the high load it drives. However, inserting buffer F to drive the gates C–E (Figure 1.9(b)), gate's A load capacitance is reduced and therefore its propagation delay is decreased.

The most known buffering algorithm, proposed by van Ginneken [157], improves the design performance by inserting buffers in a known steiner tree topology using the Elmore delay model. Later, the signal slew included in the buffer delay model achieving higher improvements [88]. Buffering is typically an NP-complete problem with high runtime ans complexity due to the multiple buffer sizes, number of sinks and the number of candidate positions for buffers in the nets. Using a new pruning rule and the predictive merging technique, Wang *et al.* [161] succeeded to accelerate the buffering, while in [66] a fully polynomial time approximation proposed approaching the optimal solution with $4\times$ speedup. Lagrange Relaxation has not been used only for gate sizing or placement but also for adding buffers trying to minimize the power consumption and cell density under timing constraints [61, 91].

Buffers are also used to improve the hold slacks. In this case, the buffers add extra delay in the very fast paths. Linear programming formulation has been used to find solutions for the early slacks with the minimum number of inserted buffers [68]. In [167], the proposed LP formulation models the setup and hold constraints and uses graph reduction to decrease the number of variables and thus the time complexity. The timing improvement can become more challenging when there are more than one corners and modes. For example, to remove the hold violations across multiple corners an integer LP formulated proposed in [57], while in [155] a technique to solve the early violations across multiple power modes introduced.



(a): Initial circuit (b): Buffer insertion and gate sizing (c): Placement and gate sizing

Figure 1.10: (a) An example circuit in which the timing of gate A needs to be improved and (b), (c) different changes that can be applied at once from the integrated optimizations to choose the best one.

The main disadvantage of adding buffers in the design to either improve setup or hold timing is that each buffer contributes to the total power consumption and area of the design. Especially, in the large designs where the number of buffers increases significantly e.g. up to 44% of the total cell instances, the impact on both area and power can't be ignored.

1.2.6 Integrated optimizations

Typically, each timing optimization method is sufficient in improving the timing of the design independently. But the order of applying these techniques is also crucial for the final Quality-of-Results. For instance, when buffering is used before any other optimization and the design suffers from multiple violations, a significant number of buffers are added degrading the power consumption and the area of the final design. However, when buffering is applied as the last mile technique to solve any timing violations the other timing-driven methods didn't succeed to fix, less buffers are needed. As the exact optimal order of the optimizations is not known and due to the increased demand for extremely effective methods that maximize timing improvements, there are some approaches which have integrated different optimization algorithms into one to achieve even higher performance results.

Figure 1.10(a) depicts an example of what changes an integrated optimizer can try to improve the delay of gate A. An integrated optimizer, applies different netlist modifications simultaneously to finally select the best one. Therefore, as shown in Figure 1.10(b), the first option is to downsize gate B and insert a buffer that shields the load seen by A improving in this way its gate delay. Another option is to downsize all gates B–D and also relocate the gates C and D closer to their driver A in order to further reduce the load driven by gate A.

Similar optimizers have been proposed in the past. OWARU [77] is an efficient timing-driven placement method that integrates also gate sizing as well as layer assignment and improves both the worst negative slack and the total negative slack of the design in higher degree compared to the timing results of applying each method separately. RUMBLE [121] is another engine that achieves remarkable savings in timing by exploring solutions that combine the relocation of the pipeline latches with buffer insertion resulting to substantial decrease of the worst path delay. ITOP [158] performs an iterative timing optimization considering multiple optimizations. In each iteration the gates of the critical paths are relocated to smooth the path, the slow gates are resized to faster alternatives and some buffers are inserted. All the three transformations combined together achieve a globally better timing solution. In addition, Moffitt *et al.* [111] proposed the critical path delay improvement considering simultaneously the effects of a set of different discrete gate changes such as gate relocation and gate sizing. The changes which are finally applied to each gate are resulted solving a disjunctive timing graph. Jiang *et al.* [76] applied interleaved gate sizing with buffering and achieved remarkably lower power consumption with higher performance, while Lillis *et al.* [88] used an efficient method that combines wire sizing and buffering to further reduce the dynamic power dissipation.

The main drawback of the integrated optimizations is the runtime. For each gate, the number of alternative transformations that have to be considered increases substantially. Typically, to overcome this problem, the integrated optimizations either use pruning methods to avoid the exhaustive exploration of the solution space or they are applied only on a subset of the total circuit. The latter means that the subcircuit selection is an important factor for the effectiveness of the integrated optimizations because it tradeoffs the quality of the results and the overall runtime. For example, when small subcircuits are selected, the number of combined optimizations is reduced leading to fast but not so timing effective solutions. On the other hand, in large subcircuits higher timing improvements are achieved but also more time is needed to evaluate them.

1.3 Thesis Contribution

Timing closure is a complex process that involves many iterative optimization steps applied in various phases of the physical design flow. The scaling of the size of the designs, the examination of multiple modes of operations and multiple design corners including also On-Chip Variations (OCV), are major critical challenges that timing optimization should face effectively. To this end, we propose four timing optimization techniques that tackle efficiently such challenges. The proposed approaches can be used both for global timing optimization at the first steps of the physical synthesis flow or close to the end

where repairing timing violations requires incremental operations that are nondisruptive and execute as fast as possible. In every case, the proposed methods are tuned for runtime scalability that allows their application to very large designs without sacrificing QoR.

1. A generalized approach for Lagrangian-Relaxation-based timing optimization is presented that is used to iteratively relocate gates, flip-flops, and local clock buffers, with the goal being to reduce timing violations [109]. In the proposed approach, the cells are allowed to move within an appropriately positioned search window, the location of which is decided by force-like timing vectors covering both late and early timing violations. The magnitude of these timing vectors is determined by the value of the corresponding Lagrange Multipliers. The introduced placement optimization is applied in conjunction with a newly proposed flip-flop clustering algorithm that (re)assigns flip-flops to local clock buffers, to separate flip-flops with incompatible timing profiles and to facilitate the subsequent timing-optimization steps. The efficiency of the proposed methodology has been proven by achieving the best overall results when compared to state-of-the-art timing-driven placement techniques.

2. Even if timing is almost closed at the end of the flow, last-mile placement and routing congestion optimizations may introduce new timing violations. These violations require minimally disruptive solutions such as threshold voltage reassignment and gate sizing that affect only marginally the placement and routing of the almost finalized design. To this end, a new way is presented about the transformation of a powerful Lagrangian-Relaxation-based gate sizer, used for global timing optimization early in the design flow, into a practical incremental timing optimizer suitable for the final stages of the flow [106]. The new incremental optimizer corrects small timing violations with fast runtime and without increasing the area/power of the design in both single corner and multimode multi-corner designs [107].

3. To accelerate timing and power optimization a task-based parallel programming template is proposed [105]. This approach utilizes all available parallelism and enables significant speedup relative to custom multithreaded approaches. Task-based parallelism is applied to all parts of the optimization engine covering also parts that are traditionally executed serially for preserving maximum timing accuracy. Using Taskflow as the parallel programming and execution engine, we achieved a speedup of 1.7× to 2.8× for gate sizing optimizations with marginal extra leakage power relative to state-of-the-art multithreaded gate sizers. This result was supported by two dynamic heuristics that restrict the number of examined

gate sizes and simplify local timing updates. Both heuristics trade off additional runtime reduction with marginal leakage power increases.

4. On-Chip Variations (OCV) introduce delay uncertainties which may cause timing violations. This problem drastically affects the clock tree that, besides the growing design complexity, needs to be appropriately synthesized to tackle the increased variability effects. To reduce the magnitude of the clock-induced OCV, a new incremental approach is introduced that relocates the flip-flops and the clock gaters in a bottom-up manner to implicitly guide the clock tree synthesis engine to produce clock trees with increased common clock tree paths [108]. For the relocation of the clock elements soft clustering approach is used, that is orthogonal to the method of building the clock tree. In the proposed methodology, the clock elements are repeatedly relocated and incrementally re-clustered, thus gradually forming better clusters and settling to more appropriate positions to increase the common paths of the clock tree. This behavior is verified by applying the proposed method in industrial designs, resulting in clock trees which are more resilient to process variations, while exhibiting improved overall timing.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 presents the proposed timing-based placement optimization. Initially, the timing-compatibility clustering method is analyzed which aims in the creation of groups that contain only timing compatible flip-flops. Next the proposed Lagrangian-Relaxation (LR) based formulation for timing optimizations is described in detail. The formulation is common for all types of cells and it can be used in order to relocate without limitation combinational cells, flip-flops and as well as clock buffers. The overall placement optimization flow is presented later on together with the formation of the search window that contains the candidate positions for each movable cell. At the end of the chapter, the experimental results validate the efficacy and the efficiency of the proposed method.

Chapter 3 discusses how a Lagrangian-Relaxation based gate sizer can be transformed to an incremental optimizer that can be applied fast to at the end of the physical synthesis flow. At first, the basics steps of all LR-based gate sizer methods are presented. Then, we highlight the slow convergence of this method when applied to almost finalized designs with a few remaining timing violations. To improve convergence behavior, we propose a smart initialization of the Lagrange multipliers. This chapter concludes with the presentation of the experimental results covering designs with single and multiple corners.

Chapter 4 briefly reviews existing methods on accelerating important steps of the design flow and introduces the transformation of the multi-step timing optimization to a task-based parallel program that favors high parallelism. After the presentation of the overall template for a task-based parallel gate sizer, we describe in detail the formation of the task graphs and the operations of the tasks involved in each step of the optimizer. Additionally, two dynamic heuristics are introduced which can further speedup the execution time trading-off the overall QoR. Finally, the runtime scalability and the competitive behavior of the proposed approach against similar state-of-the-art approaches are derived from the experimental results.

Chapter 5 presents an iterative method to eliminate the timing degradation due to clock-induced on-chip variations. Our approach relocates the flip-flops and the clock gates appropriately in pre-CTS phase to create a better seed for the clock tree synthesis engine to build clock network with less path divergence. Initially, the motivation behind the hierarchical clustering-based flip-flop relocation is presented. In the following, we discuss in details the overall relocation algorithm used for the flip-flops. The chapter concludes with the presentation of a complete set of experimental results on industrial designs.

Finally, a summary of this thesis is given in Chapter 6 including also interesting research questions that remain to be answered as future work.

2 Lagrangian-Relaxation based Timing-driven Placement

2.1 Introduction

Timing closure is a complex process that involves many iterative optimization steps applied in various phases of the physical design flow [85, 104]. Placement is instrumental to the performance of the overall flow, since it determines the length of the wires and their congestion in certain regions of the design. Long wires suffer from increased RC delay (R: Resistance, C: Capacitance), while wire congestion may lead to routing critical nets on non-minimum-distance paths to avoid congested regions. Over the last several years, wire RC delay has not only accounted for the lion's share of the total delay by far, but its variation across the metal stack has also increased dramatically [100]. Such critical factors have significantly increased the importance of timing-driven placement, which is required to reduce timing violations within a reasonable runtime, even for very large designs.

During global placement and cell spreading, the timing is optimized by controlling the wire length of selected nets, or by trying to smooth the physical layout of timing-critical paths [149, 158, 165]. The incremental timing-driven placement steps that follow global placement try to move cells to appropriate locations, to improve timing, with minimal disturbance to the initial placement. In [103], a linear program is utilized to minimize the weighted wire length on critical paths, where the path-delay sensitivities are used as weights. To avoid non-critical paths becoming critical, a novel criticality-adjacency network concept is presented. The work of [132] presents new sensitivity and figure-of-merit functions to guide cell relocation, while, in [84], net weights are computed using a critical-path counting algorithm (the more paths a net affects, the larger its weight). In [25], a differential timing model for moving timing-critical cells is adopted. The validity of the timing model is maintained by constraining the placement changes.

The recently introduced Early Histogram Compression (EHC) [67] technique mitigates hold timing violations through re-assignments of Local Clock Buffers (LCB) to flip-flops and appropriate LCB movements. Better results are achieved in [82], which optimizes the clock arrival at each flip-flop by appropriate re-assignments of LCBs to

flip-flops and flip-flop movements, to improve hold violations while preserving the pre-optimized setup violations. In contrast to such approaches, OWARU [77] utilizes Bézier curves to smoothen the physical curve produced by the placement of the cells participating in timing-critical paths. Other approaches like [46] and [44] rely on analytic formulations for relocating flip-flops, gates, and LCBs, or utilize non-critical cell relocation and cell-swapping to improve Quality-of-Results (QoR).

Other approaches rely on the Lagrange Relaxation (LR)-based formulation of timing optimization, whereby the derived cost function guides cell relocation to reduce timing violations [53, 96, 149, 165]. Using LR, the hard constraints of the optimization are removed and incorporated into the objective function, each one multiplied by a penalty term called a Lagrange Multiplier (LM). During optimization, LMs act as dynamic weights that reflect both the timing criticality of each net and the number of critical endpoints that it affects.

As opposed to previous methods that independently move combinational gates, flip-flops, and/or LCBs using loosely-connected algorithms, we propose an LR-based timing-driven placement algorithm that *handles the relocation of all types of cells in a unified manner*. Each timing-critical cell is relocated iteratively through the selection of an optimized position out of a set of appropriately selected candidate positions. Both the selection of the best position, and the definition of the search window, are based on the value of the LMs in each optimization round.

The proposed LR-based placement optimization methodology is complemented by a pseudo-3D flip-flop clustering algorithm that clusters flip-flops according to their geographical location and the timing slacks of their D and Q pins. The objective is to guarantee that flip-flops of the same cluster share a compatible timing profile (i.e., the flip-flops should benefit in the same way by an increase, or an equivalent decrease, in the clock arrival time). In this way, nearby timing-compatible flip-flops of the same cluster can be driven by the same LCB, while timing-incompatible flip-flops are driven by different LCBs, even if they are placed in the same region. This separation of timing-incompatible flip-flops facilitates the timing optimizations performed later on by LR-based cell relocation.

The proposed approach effectively combines the application of the following processes to yield very promising results:

1. A proposed LR-based formulation for timing optimization that allows the handling of gates, flip-flops, and LCBs in a unified manner.

2. LM-based calibration of the search window to detect candidate placements for each cell.

3. A flip-flop clustering step that clusters flip flops unevenly, based on their timing profile.

The derived results indicate significant improvements in Worst Negative Slack (WNS) and Total Negative Slack (TNS) at a reasonable runtime, as compared to state-of-the-art timing-driven placement-optimization techniques that include either closely related LR-based optimization [53], or other highly-efficient heuristics [67, 77, 82].

2.2 Timing Compatibility Flip-Flop Clustering

The proposed timing-driven placement methodology relocates combinational gates, flip-flops, and LCBs in a unified manner, to improve the circuit's timing. Moving LCBs relative to the group of flip-flops that they drive, increases or decreases the clock arrival time. This change in clock arrival may be beneficial to some flip-flops of the group and harmful to other flip-flops in the same group. For example, delaying clock arrival would benefit flip-flops with negative D/positive Q late slack, and hurt the timing of flip-flops with a positive D/negative Q late slack profile. Therefore, before applying any LCB movement, we need to be sure that each LCB drives flops with *compatible* timing profiles (i.e., all need an increase or reduction in clock arrival time, or are neutral to this choice). To achieve this, we use a pseudo-3D clustering algorithm, where flip-flops are clustered according to their (x, y) position and their timing profile. Flip-flops placed in the same cluster are driven by the same LCB, while timing-incompatible flip-flops are put in different clusters and driven by different LCBs.

The proposed clustering algorithm is given in Algorithm 1. Initially, each flip-flop is given a timing profile that can belong to one of three categories depending on how the clock arrival time would benefit the flip-flops' timing: (a) faster clock arrival (fast), (b) slower clock arrival (slow), and (c) neutral. Any finer-grained categorization is possible. Neutral is considered compatible with both the fast and slow categories, while flip-flops that belong to the fast and slow categories are incompatible. The list of clusters is then initialized, and the iterative loop of flop-to-cluster assignment and cluster updating is executed until convergence is reached (i.e., either all flip-flops remain attached to their previously assigned clusters, or the maximum number of iterations is reached).

The proposed clustering algorithm is a variant of k-means clustering [74, 166], which minimizes the squared distance between each cluster center and its assigned flip-flops, while also taking into account the timing profile of each flip-flop and the size of each cluster. In this way, each cluster contains an appropriate number of timing-compatible flip-flops, thereby facilitating the LCB movement that is subsequently applied.

Algorithm 1: Timing Compatibility FF Clustering

```
1  Assign a timing profile to each flip-flop;
2  InitClusters();
3  FF_PriorityList ← PrioritizeFlipFlops();
4  repeat // FF-to-cluster assignment
5      foreach FF i in FF_PriorityList do
6          if i ∈ fast then
7              C_cand ← {clusters with fast or neutral FFs} ;
8          else if i ∈ slow then
9              C_cand ← {clusters with slow or neutral FFs} ;
10         else
11             C_cand ←{all clusters};
12         end
13         best_cost ← inf ;
14         foreach cluster j in C_cand do
15             if size( j)!=MaxSize & canAssign(i,j) then
16                 cost ← AssignmentCost(i, j) ;
17                 if (cost < best_cost) then
18                     best_cost ← cost ;
19                     assign FF i to cluster j;
20                 end
21             end
22         end
23     end
24     UpdateClusterCenter();
25     UpdateTiming();
26     UpdateFFTimingProfiles();
27 until convergence;
```

To improve timing, the proposed clustering creates clusters of unequal sizes on purpose. The LCBs that drive flip-flops with fast timing profiles are less loaded relative to the LCBs that drive flip-flops belonging to the slow category. This uneven loading decreases, or increases, accordingly the delay of the LCB with the goal being to improve timing. Creating clusters of uneven sizes – to facilitate timing optimization – with k-means is not directly possible. To achieve this goal in a practical manner, we artificially shrink, or expand, the true Euclidean distance, without any further modification to the clustering algorithm.

Table 2.1: Timing profile categorization

Early			Late		
Slack at		Timing	Slack at		Timing
D pin	Q pin	profile	D pin	Q pin	profile
+	+	neutral	+	+	neutral
+	−	slow	+	−	fast
−	+	fast	−	+	slow
−	−−	slow	−	−−	fast
−−	−	fast	−−	−	slow

2.2.1 Assign a Timing Profile to each Flip-Flop

Initially, each flip-flop is assigned to one of the three categories shown in Table 2.1 for late timing, and, separately, for early timing, after taking into account the timing slacks (positive/+ or negative/–) at the D and Q pins of each flip-flop. In the case that the slack is negative at both pins of the flip-flop, the timing profile of the flip-flop is determined by the pin with the most negative slack, depicted as two negative minus signs ($--$) in Table 2.1.

The separate late and early timing profiles initially given to each flip-flop should be merged to one final profile. When both timing profiles are the same, the final timing profile for this flip-flop is the same common profile. If one of the two timing profiles for a flip-flop is neutral (either for late or early timing), the timing profile for this flip-flop is determined by the other non-neutral timing profile. On the contrary, when a flip-flop belongs to contradicting categories in the late and early timing modes, the timing profile for this flip-flop is the one selected for the most critical mode, i.e., the one with the most negative slack in either the D or Q pins.

2.2.2 Initialize Clusters and Prioritize Flip-Flops

The total number of clusters is set equal to the number of available LCBs, and the cluster centers are initialized to the positions of the corresponding LCBs. If no LCBs are present, the number of clusters can be selected with any other density or maximum fanout/capacitance criterion, and the center of each cluster is set to the position of a randomly-selected flip-flop. We should note that the optimal number of clusters, k, for this timing compatibility clustering is not obvious, and it can only be judged by the final timing QoR obtained after executing the entire timing-driven placement flow for various numbers of clusters. Each newly-created cluster is assigned to one flip-flop to

avoid leaving a cluster empty. We try to initialize each cluster with a fast flip-flop (if a fast flip-flop exists nearby), taken from a list of the 20 most closely-placed flip-flops. If this is not possible, the cluster is assigned to a neutral flip-flop to increase the freedom of later assignments. If, however, all nearby flip-flops are slow, the cluster is assigned to a slow flop. In this way, we increase the probability to initialize more fast clusters, implicitly reducing the average size of a fast cluster relative to neutral or slow clusters.

Next, we prioritize the list of flip-flops, so the flip-flops with timing violations select a nearby cluster first. If the number of flip-flops with a fast timing profile is larger than the flip-flops with a slow timing profile, we first examine the fast flip-flops (those flip-flops are put at the top of the *FF_PriorityList*) and then the slow flip-flops. In the opposite case (number of slow flip-flops exceeds the fast flip-flops), we first examine the flip-flops that belong to the slow timing profile. Neutral flip-flops are always examined last.

2.2.3 Flip-Flop Clustering

The assignment of flip-flops to clusters is outlined in lines 4–27 of Algorithm 1. For each flip-flop examined, we identify which clusters are considered as valid assignment candidates. For instance, a flip-flop with a fast timing profile considers only the clusters that contain fast or neutral flip-flops as valid. The flip-flops with a neutral timing profile are compatible with all other flip-flops and can be assigned to any cluster. When examining the assignment of flip-flop i to cluster j, we first check – with the condition in line 15 – that no cluster receives more flip-flops than the maximum allowed, and if this assignment would cause flip-flop i to be incompatible with the flip-flops already assigned to cluster j. The potential incompatibility may arise due to the new wire and LCB delays. These new delays may, in fact, cause marginal, or potentially more notable, alterations to the clock arrival times, which, in turn, may possibly make the timing profiles of the flip-flops outdated. To accurately reflect the new clock arrival times, incremental timing analysis should be performed, but this is prohibitively expensive when checking each independent assignment. Instead, we focus this analysis only on "weak" flops that are most likely to switch to a different timing profile after the changes in the clock arrival times, i.e., flops with timing slacks in their D/Q pins in the range of ± 50 ps. To quickly estimate the timing impact of assigning flip-flop i to cluster j, we use the differential timing model of [25] in a manner similar to [53].

This dynamic checking of the timing profiles can be disabled to significantly reduce the overall runtime. As will be demonstrated in the experimental resuls later on, omission of this step has minimal impact on the resulting timing QoR. The reason is because flip-flop to cluster reassigment rarely affects the timing profile of the flip-flops with *significant* timing slacks; it may only disturb the weak flip-flops. Therefore, the flip-flops

Algorithm 2: AssignmentCost(FlipFlop f, Cluster c)

$dist(f,c) \leftarrow \left((c.x - f.x)^2 + (c.y - f.y)^2\right)^{1/2}$;
$w \leftarrow 1$;
if $f \in$ *neutral and c has slow or only neutral FFs* **then**
  $w \leftarrow \frac{size(c)}{MaxSize}$;
else if $f \in$ *neutral and c has fast FFs* **then**
  $w \leftarrow 1 + \frac{size(c)}{MaxSize}$;
end
return $w \cdot dist(f,c)$;

with significant timing slacks would be correctly separated, even if dynamic compatibility checking is disabled. The weak flops with outdated timing profiles would eventually move to the neutral category via the subsequent timing optimizations, thus becoming compatible with all other categories.

From all valid and compatible clusters, each flip-flop is assigned to the cluster that minimizes the assignment cost computed according to Algorithm 2. The cost is the Euclidean distance between flip-flop f and the center of cluster c, multiplied by a weight w. The role of w is to create clusters with unequal sizes, to further improve timing: clusters with fast/neutral flip-flops should contain fewer flip-flops than clusters with slow/neutral flip-flops. The distance between a neutral flip-flop and a cluster that contains slow, or only-neutral, flip-flops is scaled down by w, thus increasing the probability that the neutral flip-flop is assigned to this cluster. In contrast, the distance of a neutral flip-flop to a cluster with fast flip-flops is artificially increased, thus increasing the cost of this assignment and making the assignment less favorable. In this way, the clusters with fast flip-flops remain with fewer flip-flops in total. Weight w only scales the distance of neutral flip-flops, while w is always equal to one for flip-flops of the fast or slow categories.

When considering flip-flops of different sizes with different clock-pin capacitances, the contribution of each flip-flop to the size of the cluster should be measured relative to its clock-pin capacitance. The larger the clock-pin capacitance, the larger the equivalent count in terms of primitive flip-flops. Hence, the size of each cluster would reflect the total capacitance driven by the LCB of the cluster. The proposed algorithm will end up producing clusters of fast flip-flops with less total capacitance relative to clusters of slow or neutral, flip-flops.

2.2.4 Update Cluster Centers and Timing Profiles

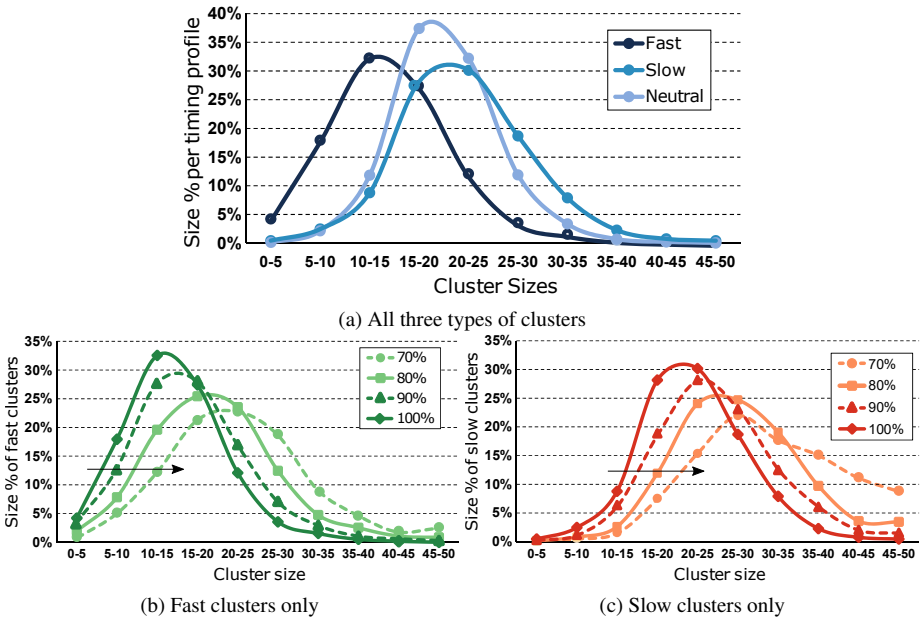
When all flip-flops are assigned to a cluster, we need to update the center of each cluster (line 24 in Algorithm 1) to the center of gravity of the flip-flops belonging to this cluster. Each LCB moves to the center of the cluster to which it belongs. As the centers of the clusters are moved to new updated positions, the LCBs are also moved and instantly legalized to their new positions using the Jezz legalizer [123]. If LCB movement is limited by a maximum displacement constraint, and the updated cluster center lies outside the maximum displacement bounding box of the LCB, the center of the cluster is slid towards the nearest location at the bounds of the displacement bounding box. In this way, in every iteration, the location of the cluster center remains valid in terms of detailed placement constraints.

When the dynamic update of timing profiles is enabled, an incremental static timing analysis is performed after moving the LCBs to their new positions, to update the slacks and the timing profiles of the flip-flops (lines 25–26 of Algorithm 1). Therefore, the new – and accurate – timing profile of each flip-flop is available for the next iteration.

2.2.5 Clustering Behavior

To demonstrate the efficacy of the clustering technique with respect to the generation of clusters of uneven sizes, we analyze the clustering behavior in a representative benchmark of the ICCAD-2015 benchmark set [81]. Specifically, Figure 2.1(a) depicts the percentage of cluster sizes for each of the three flip-flop timing profiles (fast, slow, neutral) for sb10. As shown in the figure, more than 30% of the fast clusters have sizes between 10-15 flip-flops, while an additional cumulative 20% corresponds to even smaller clusters of 0-5 and 5-10 flip-flops. In contrast, the majority of clusters of slow and neutral-only flip-flops have larger sizes; around 30% of all slow clusters have sizes of 20-25 flops. It should be noted that similar behavior has been observed in all examined benchmarks.

More importantly, the behavior of the proposed clustering technique is robust with respect to the total number of clusters. Figures 2.1(b) and (c) illustrate the percentage of cluster sizes for the fast and slow categories, respectively, of the sb10 benchmark, as the total number of clusters, k, decreases. Specifically, the number of clusters decreases from 100% (i.e., the number of clusters in the unmodified benchmark) down to 70% of the initial number. The decrease in clusters is achieved by uniformly removing – while accounting for any disparities in cluster densities – a corresponding number of clusters from the benchmark. As k decreases, both the fast and slow distributions shift to the right, i.e., towards larger cluster sizes. Nevertheless, if we juxtapose the fast and slow distributions, one can clearly see that, even as k decreases, the fast clusters are



(a) All three types of clusters

(b) Fast clusters only

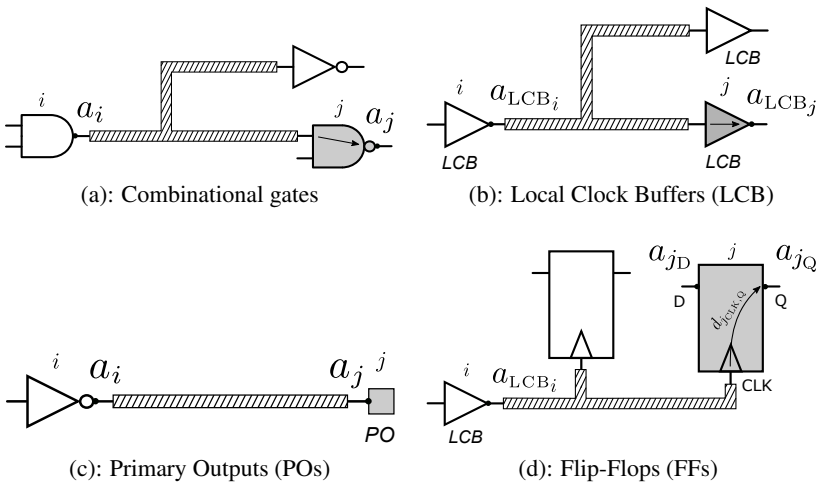
(c) Slow clusters only

Figure 2.1: (a) The percentage of cluster sizes for each timing profile (fast, slow, neutral) for the representative sb10 benchmark of the ICCAD-2015 benchmark set [81]. The percentage of cluster sizes for (b) only fast, and (c) only slow clusters, as the number of cluster centers is artificially decreased.

still smaller, on average, than the corresponding slow clusters. This behavior unequivocally demonstrates that the proposed flip-flop clustering achieves the desired objective, irrespective of the number of available clusters.

2.3 LR-Based Timing Optimization

Our goal is to minimize the sum of early and late TNS, by appropriately relocating the timing-critical cells of the design. Assuming that TNS is computed over the set $\mathcal{E}$ of all the timing endpoints, including primary outputs *POs* and the input-D pins of the



(a): Combinational gates (b): Local Clock Buffers (LCB)

(c): Primary Outputs (POs) (d): Flip-Flops (FFs)

Figure 2.2: The definition of arrival times and delays $d_{i,j}$, including both wire- and arc-delay, for different types of cells.

flip-flops (FFs) [12], the timing optimization problem can be stated as follows:

$$\textbf{min :} \sum_{j\in\mathcal{E}}(-s_j^L)+\sum_{j\in\mathcal{E}}(-s_j^E) \tag{2.1}$$

$$\begin{aligned}
\textbf{s.t.:}\ & s_j^L \le 0, && s_j^E \le 0, && \forall \text{ timing endpoint } j\in\mathcal{E}\\
& s_j^L \le T - a_j^L, && s_j^E \le a_j^E, && \forall \text{ output } j\in POs\\
& s_j^L \le r_{j_\mathrm{D}}^L - a_{j_\mathrm{D}}^L, && s_j^E \le a_{j_\mathrm{D}}^E - r_{j_\mathrm{D}}^E, && \forall \text{ input D pin of flip-flop } j\in FFs\\
& a_i^L + d_{i,j}^L \le a_j^L, && a_i^E + d_{i,j}^E \ge a_j^E, && \forall \text{ gate } j\in Gates \text{ (Fig. 2.2(a))}\\
& a_{\mathrm{LCB}_i}^L + d_{i,j}^L \le a_{j_\mathrm{Q}}^L, && a_{\mathrm{LCB}_i}^E + d_{i,j}^E \ge a_{j_\mathrm{Q}}^E, && \forall \text{ Q pin of flip-flop } j\in FFs \text{ (Fig. 2.2(d))}\\
& a_{\mathrm{LCB}_i}^L + d_{i,j}^L \le a_{\mathrm{LCB}_j}^L, && a_{\mathrm{LCB}_i}^E + d_{i,j}^E \ge a_{\mathrm{LCB}_j}^E, && \forall \text{ LCB } j\in LCBs \text{ (Fig. 2.2(b))}\\
& \text{Cell placement is legalized}\\
& \text{Cell relocation distance} \le \text{maximum displacement}
\end{aligned}$$

Variables s_j^L and s_j^E represent the negative slack at pin $j\in\mathcal{E}$ for late and early timing, while T is the targeted clock period. The timing slack at each timing endpoint is the difference between the required arrival time and the actual arrival time. Parameters a_j and a_{LCBj} are the arrival times (late or early) at the output pins of a gate and an LCB,

respectively, while a_{j_D} and a_{j_Q} represent the arrival times at the D/Q pins of flip-flop j. The required arrival time at the D pin of the same flip-flop is expressed as r_{j_D}. The delay $d_{i,j}$ is the sum of the wire and the cell delay from the output pin of cell i to the output pin of cell j. Figure 2.2 illustrates in detail the timing arcs involved in the computation of $d_{i,j}$ for every type of cell. In the case of gates and LCBs, shown in Figures 2.2(a)–(b), $d_{i,j}$ comprises the wire delay from the output pin of the driving cell of the previous level plus the delay of the jth gate or LCB. When pin j represents a timing endpoint like a primary output (Figure 2.2(c)), $d_{i,j}$ is only the corresponding wire delay. Similarly, as shown in Figure 2.2(d), in the case of a flip-flop, $d_{i,j}$ is the wire delay from the output pin of the LCB that drives the clock pin of this flip-flop plus the clock-to-Q delay of the flip-flop. Representing with I_j the set of fanin cells of cell j, and with λ the LMs, LR can incorporate the timing constraints of (2.1) into the objective as follows:

$$
\begin{aligned}
\textbf{min}: & \sum_{j\in\mathcal{E}}(-s_j^L)+\sum_{j\in\mathcal{E}}(-s_j^E)+\sum_{j\in\mathcal{E}}\lambda_{0j}^L s_j^L+\sum_{j\in\mathcal{E}}\lambda_{0j}^E s_j^E+ \\
& \sum_{j\in POs}\lambda_{\mathrm{PO}_j}^L\left(s_j^L-T+a_j^L\right)+\sum_{j\in POs}\lambda_{\mathrm{PO}_j}^E(s_j^E-a_j^E)+ \\
& \sum_{j\in FFs}\lambda_{\mathrm{D}_j}^L\left(s_j^L-r_{j_\mathrm{D}}^L+a_{j_\mathrm{D}}^L\right)+\sum_{j\in FFs}\lambda_{\mathrm{D}_j}^E(s_j^E-a_{j_\mathrm{D}}^E+r_{j_\mathrm{D}}^E)+ \\
& \sum_{j\in Gates}\left(\sum_{i\in I_j}\lambda_{\mathrm{G}_{i,j}}^L(a_i^L+d_{i,j}^L-a_j^L)+\sum_{i\in I_j}\lambda_{\mathrm{G}_{i,j}}^E(a_j^E-a_i^E-d_{i,j}^E)\right)+ \\
& \sum_{j\in FFs}\left(\lambda_{\mathrm{FF}_{i,j}}^L(a_{\mathrm{LCB}_i}^L+d_{i,j}^L-a_{j_\mathrm{Q}}^L)+\lambda_{\mathrm{FF}_{i,j}}^E(a_{j_\mathrm{Q}}^E-a_{\mathrm{LCB}_i}^E-d_{i,j}^E)\right)+ \\
& \sum_{j\in LCBs}\left(\lambda_{\mathrm{LCB}_{i,j}}^L(a_{\mathrm{LCB}_i}^L+d_{i,j}^L-a_{\mathrm{LCB}_j}^L)+\lambda_{\mathrm{LCB}_{i,j}}^E(a_{\mathrm{LCB}_j}^E-a_{\mathrm{LCB}_i}^E-d_{i,j}^E)\right)
\end{aligned}
\tag{2.2}
$$

The Karush–Kuhn–Tucker (KKT) optimality conditions for the timing endpoints of the design impose that $\lambda_{0j}^L+\lambda_{\mathrm{PO}j}^L=1$, and $\lambda_{0j}^E+\lambda_{\mathrm{PO}j}^E=1$, for each primary output $j\in POs$, and $\lambda_{0j}^L+\lambda_{\mathrm{D}j}^L=1$, and $\lambda_{0j}^E+\lambda_{\mathrm{D}j}^E=1$, for the D pin of flip-flop j. Substituting these equalities into (2.2) causes the slack variables s_j^L and s_j^E to cancel out and simplify

the problem as follows:

$$\begin{aligned}\textbf{min}: & \sum_{j\in POs}\lambda^L_{\mathrm{PO}_j}\big(-T+a^L_j\big)+\sum_{j\in POs}\lambda^E_{\mathrm{PO}_j}(-a^E_j)+ \\ & \sum_{j\in FFs}\lambda^L_{\mathrm{D}_j}\big(-r^L_{j_\mathrm{D}}+a^L_{j_\mathrm{D}}\big)+\sum_{j\in FFs}\lambda^E_{\mathrm{D}_j}(-a^E_{j_\mathrm{D}}+r^E_{j_\mathrm{D}})+ \\ & \sum_{j\in Gates}\Big(\sum_{i\in I_j}\lambda^L_{\mathrm{G}_{i,j}}(a^L_i+d^L_{i,j}-a^L_j)+\sum_{i\in I_j}\lambda^E_{\mathrm{G}_{i,j}}(a^E_j-a^E_i-d^E_{i,j})\Big)+ \\ & \sum_{j\in FFs}\Big(\lambda^L_{\mathrm{FF}_{i,j}}(a^L_{\mathrm{LCB}_i}+d^L_{i,j}-a^L_{j_\mathrm{Q}})+\lambda^E_{\mathrm{FF}_{i,j}}(a^E_{j_\mathrm{Q}}-a^E_{\mathrm{LCB}_i}-d^E_{i,j})\Big)+ \\ & \sum_{j\in LCBs}\Big(\lambda^L_{\mathrm{LCB}_{i,j}}(a^L_{\mathrm{LCB}_i}+d^L_{i,j}-a^L_{\mathrm{LCB}_j})+\lambda^E_{\mathrm{LCB}_{i,j}}(a^E_{\mathrm{LCB}_j}-a^E_{\mathrm{LCB}_i}-d^E_{i,j})\Big)\end{aligned} \tag{2.3}$$

Previous work on LR-based timing-driven placement [53, 96] assumed that flip-flops and LCBs remain locked to their positions and cannot move. This over-simplification allows the removal of the required arrival times from (2.3), since they remain constant. However, this is not allowed in our work. Our goal is to incorporate the movement of all types of cells, gates, flip-flops, and LCBs in the same LR-based formulation, covering both late and early timing constraints. For this reason, we would like to remove the direct contribution of the clock arrival time and keep only the arrival times on the datapath pins in the optimization problem. For the register-to-register paths, we know that

$$r^L_{j_\mathrm{D}}=a^E_{j_\mathrm{CLK}}+T-t_\mathrm{setup}\quad\text{and}\quad r^E_{j_\mathrm{D}}=a^L_{j_\mathrm{CLK}}+t_\mathrm{hold}, \tag{2.4}$$

where $a^L_{j_\mathrm{CLK}}$ and $a^E_{j_\mathrm{CLK}}$ represent the late and early arrival times of the clock on flip-flop j, and t_setup, t_hold are the setup and hold delays of the flip-flop. The clock arrival time a_{j_CLK} and the arrival time at the output Q pin of a flip-flop a_{j_Q} are connected via the clk-to-Q arc delay $d_{j_\mathrm{CLK,Q}}$, i.e., $a^L_{j_\mathrm{CLK}}=a^L_{j_\mathrm{Q}}-d^L_{j_\mathrm{CLK,Q}}$ and $a^E_{j_\mathrm{CLK}}=a^E_{j_\mathrm{Q}}-d^E_{j_\mathrm{CLK,Q}}$. Substituting these equalities into the required arrival times of (2.4) leads to the following equations:

$$r^L_{j_\mathrm{D}}=a^E_{j_\mathrm{Q}}-d^E_{j_\mathrm{CLK,Q}}+T-t_\mathrm{setup} \tag{2.5}$$

$$r^E_{j_\mathrm{D}}=a^L_{j_\mathrm{Q}}-d^L_{j_\mathrm{CLK,Q}}+t_\mathrm{hold}, \tag{2.6}$$

Using (2.5) and (2.6) in the place of the required arrival times r_{j_D} of (2.3), and considering that T, t_hold, t_setup remain unchanged during timing optimization, we end up with

the following formulation:

$$\begin{aligned}\textbf{min}: & \sum_{j\in POs}\lambda^L_{\mathrm{PO}_j}\left(a^L_j\right)+\sum_{j\in POs}\lambda^E_{\mathrm{PO}_j}(-a^E_j)+ \\ & \sum_{j\in FFs}\lambda^L_{\mathrm{D}_j}\left(-a^E_{j_\mathrm{Q}}+d^E_{j_\mathrm{CLK,Q}}+a^L_{j_D}\right)+\sum_{j\in FFs}\lambda^E_{\mathrm{D}_j}\left(-a^E_{j_D}+a^L_{j_\mathrm{Q}}-d^L_{j_\mathrm{CLK,Q}}\right)+ \\ & \sum_{j\in Gates}\left(\sum_{i\in I_j}\lambda^L_{\mathrm{G}_{i,j}}(a^L_i+d^L_{i,j}-a^L_j)+\sum_{i\in I_j}\lambda^E_{\mathrm{G}_{i,j}}(a^E_j-a^E_i-d^E_{i,j})\right)+ \\ & \sum_{j\in FFs}\left(\lambda^L_{\mathrm{FF}_{i,j}}(a^L_{\mathrm{LCB}_i}+d^L_{i,j}-a^L_{j_\mathrm{Q}})+\lambda^E_{\mathrm{FF}_{i,j}}(a^E_{j_\mathrm{Q}}-a^E_{\mathrm{LCB}_i}-d^E_{i,j})\right)+ \\ & \sum_{j\in LCBs}\left(\lambda^L_{\mathrm{LCB}_{i,j}}(a^L_{\mathrm{LCB}_i}+d^L_{i,j}-a^L_{\mathrm{LCB}_j})+\lambda^E_{\mathrm{LCB}_{i,j}}(a^E_{\mathrm{LCB}_j}-a^E_{\mathrm{LCB}_i}-d^E_{i,j})\right)\end{aligned} \tag{2.7}$$

By differentiating (2.7) with respect to the arrival times, according to the KKT optimality conditions, and representing the set of fanin and fanout cells of cell j with I_j and O_j, we end up with the following LM flow conservation rules:

- For each gate $j \in Gates$ connected to other gates and flip-flops at its fanin and fanout cones.

$$\sum_{i\in I_j}\lambda^L_{\mathrm{G}_{i,j}}=\sum_{k\in O_j}\lambda^L_{\mathrm{G}_{j,k}}+\sum_{k\in O_j}\lambda^L_{\mathrm{D}_k}+\sum_{k\in O_j}\lambda^L_{\mathrm{PO}_k} \tag{2.8}$$

$$\sum_{i\in I_j}\lambda^E_{\mathrm{G}_{i,j}}=\sum_{k\in O_j}\lambda^E_{\mathrm{G}_{j,k}}+\sum_{k\in O_j}\lambda^E_{\mathrm{D}_k}+\sum_{k\in O_j}\lambda^E_{\mathrm{PO}_k} \tag{2.9}$$

- For each flip-flop $j \in FFs$ connected to other gates or flip-flops at its input D and output Q pins.

$$\lambda^E_{\mathrm{D}_j}+\sum_{k\in O_j}\lambda^L_{\mathrm{G}_{j,k}}+\sum_{k\in O_j}\lambda^L_{\mathrm{D}_k}+\sum_{k\in O_j}\lambda^L_{\mathrm{PO}_k}=\lambda^L_{\mathrm{FF}_{i,j}} \tag{2.10}$$

$$\lambda^L_{\mathrm{D}_j}+\sum_{k\in O_j}\lambda^E_{\mathrm{G}_{j,k}}+\sum_{k\in O_j}\lambda^E_{\mathrm{D}_k}+\sum_{k\in O_j}\lambda^E_{\mathrm{PO}_k}=\lambda^E_{\mathrm{FF}_{i,j}} \tag{2.11}$$

- For each LCB $j \in LCBs$ driving both clock pins of flip-flops or other LCBs:

$$\lambda^L_{\mathrm{LCB}_{i,j}}=\sum_{k\in O_j}\lambda^L_{\mathrm{FF}_{j,k}}+\sum_{k\in O_j}\lambda^L_{\mathrm{LCB}_{j,k}} \tag{2.12}$$

$$\lambda^E_{\mathrm{LCB}_{i,j}}=\sum_{k\in O_j}\lambda^E_{\mathrm{FF}_{j,k}}+\sum_{k\in O_j}\lambda^E_{\mathrm{LCB}_{j,k}} \tag{2.13}$$

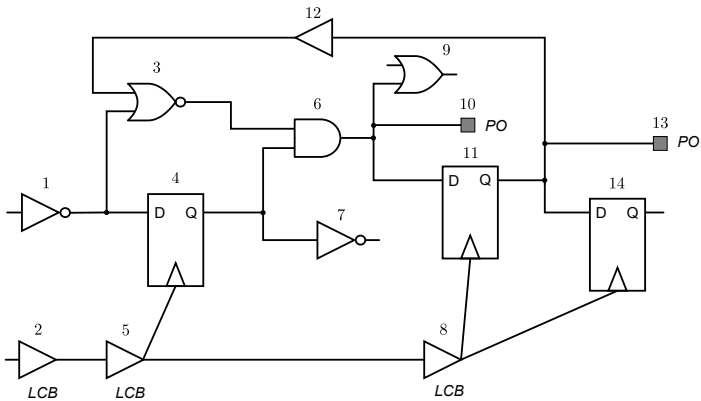


Figure 2.3: An example circuit used to illustrate the LM flow conservation rules to preserve the KKT conditions for different types of cells. This figure also highlights the timing arcs involved in the computation of the local cost for each cell relocation in Section 2.4.1.

- For each timing endpoint $j \in \mathcal{E}$ driven by pin i:

$$\lambda^L_{\mathrm{PO}_j} = \lambda^L_{i,j}, \qquad \lambda^L_{\mathrm{D}_j} = \lambda^L_{i,j} \tag{2.14}$$

$$\lambda^E_{\mathrm{PO}_j} = \lambda^E_{i,j}, \qquad \lambda^E_{\mathrm{D}_j} = \lambda^E_{i,j} \tag{2.15}$$

For gates, the incoming LMs are distributed to outgoing timing arcs that can be other gates, input D pins of flip-flops, or primary outputs. The same holds for the LMs at the output Q pins of flip-flops, while, on the contrary, the LMs of LCBs are spread to other LCBs, or the clock pin of flip-flops. While the equality constraints for combinational gates have been proven in previous work on LR-based optimization, the optimal relation among LMs on the pins of flip-flops and LCBs, including both late and early timing constraints, are introduced in this work. Applying the LM flow conservation equations (2.8)–(2.15) to selected cells of Figure 2.3 gives the following result:

- For gate 6 (using Equations (2.8) and (2.9)),

$$\lambda^L_{\mathrm{G}_{3,6}} + \lambda^L_{\mathrm{G}_{4,6}} = \lambda^L_{\mathrm{G}_{6,9}} + \lambda^L_{\mathrm{PO}_{10}} + \lambda^L_{\mathrm{D}_{11}}$$

$$\lambda^E_{\mathrm{G}_{3,6}} + \lambda^E_{\mathrm{G}_{4,6}} = \lambda^E_{\mathrm{G}_{6,9}} + \lambda^E_{\mathrm{PO}_{10}} + \lambda^E_{\mathrm{D}_{11}}$$

- For flip-flop 11 (using Equations (2.10) and (2.11)),

$$\lambda^E_{\mathrm{D}_{11}} + \lambda^L_{\mathrm{G}_{11,12}} + \lambda^L_{\mathrm{PO}_{13}} + \lambda^L_{\mathrm{D}_{14}} = \lambda^L_{\mathrm{FF}_{8,11}}$$
$$\lambda^L_{\mathrm{D}_{11}} + \lambda^E_{\mathrm{G}_{11,12}} + \lambda^E_{\mathrm{PO}_{13}} + \lambda^E_{\mathrm{D}_{14}} = \lambda^E_{\mathrm{FF}_{8,11}}$$

- For LCB 5 (using Equations (2.12) and (2.13)),

$$\lambda^L_{\mathrm{LCB}_{2,5}} = \lambda^L_{\mathrm{FF}_{5,4}} + \lambda^L_{\mathrm{LCB}_{5,8}}$$
$$\lambda^E_{\mathrm{LCB}_{2,5}} = \lambda^E_{\mathrm{FF}_{5,4}} + \lambda^E_{\mathrm{LCB}_{5,8}}$$

Substituting the LM equality constraints into the optimization problem, we end up with a simplified objective function that combines the contribution of gates, flip-flops, and local clock buffers in a unified cost function, for both late and early timing, while it highlights the independent contribution of the clock-to-Q delays and their associated LMs:

$$\begin{aligned}\textbf{min :}\ & \sum_{j\in FFs} \lambda^L_{\mathrm{D}_j}\left(d^E_{j_{\mathrm{CLK,Q}}}\right) + \sum_{j\in FFs} \lambda^E_{\mathrm{D}_j}(-d^L_{j_{\mathrm{CLK,Q}}}) + \\ & \sum_{j\in Gates}\Big(\sum_{i\in I_j} \lambda^L_{\mathrm{G}_{i,j}}(d^L_{i,j}) + \sum_{i\in I_j} \lambda^E_{\mathrm{G}_{i,j}}(-d^E_{i,j})\Big) + \\ & \sum_{j\in FFs}\Big(\lambda^L_{\mathrm{FF}_{i,j}}(d^L_{i,j}) + \lambda^E_{\mathrm{FF}_{i,j}}(-d^E_{i,j})\Big) + \\ & \sum_{j\in LCBs}\Big(\lambda^L_{\mathrm{LCB}_{i,j}}(d^L_{i,j}) + \lambda^E_{\mathrm{LCB}_{i,j}}(-d^E_{i,j})\Big)\end{aligned} \tag{2.16}$$

Previous LR-based timing optimizations derived similar cost functions, but of a more limited scope, involving only gates for timing-driven placement (as done in [53, 96]), or gate sizing (in [98, 140]), or including gate sizing with local clock skew optimizations for improving the late timing only (in [141, 146]).

2.4 Overall Flow and LR-based Cell Relocation

The overall flow of applying the LR-based cell relocation process is depicted in Figure 2.4. The flip-flop clustering step is executed once at the beginning of the flow, to separate timing-incompatible flip-flops. Then, the iterative LR-based timing-driven placement optimization evolves in two steps. In the first step, assuming constant LMs, we try to move a selected set of cells with the goal of minimizing the cost function (2.16). In

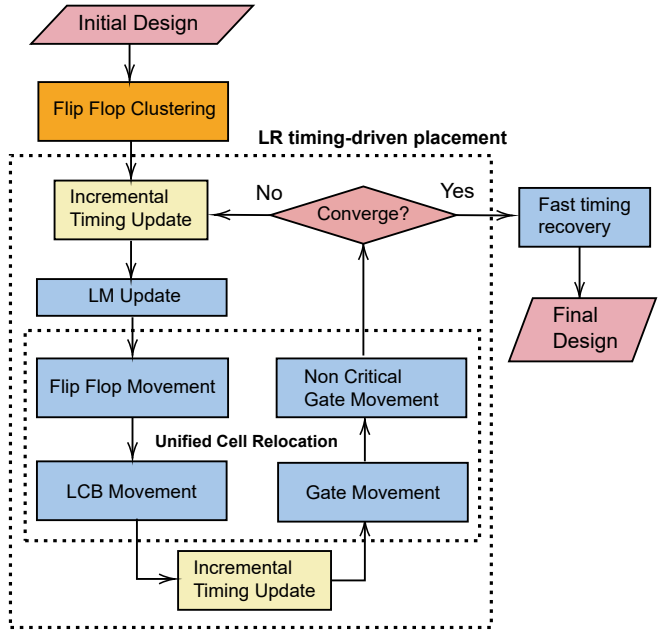


Figure 2.4: The overall cell relocation flow. Timing compatibility flip-flop clustering facilitates the LR-based timing-driven cell relocation that interleaves the LM updates and unified cell relocation until convergence is achieved.

the second step, the LMs are updated to reflect the new criticality of the corresponding timing arcs. On every LM update (and before their initialization), a full incremental timing update takes place.

In each iteration, all cells are relocated using the procedure described in Algorithm 3, which approximately minimizes (2.16) by the Optimal Local Relocation (OLR) of one cell at a time, assuming all the other cells are fixed. Flip-flops and LCBs are relocated first, and then gates are traversed in forward topological order from the inputs to the outputs (POs), i.e., OLR of a cell begins after all its fanin gates have been processed. During OLR, each cell is moved conditionally to several candidate locations. For each candidate position examined, timing is updated locally. This update involves building a new Steiner tree for each fanin and fanout net for the cell under relocation using Flute [26], and recomputing the new delays/slews of the updated nets and all cells connected to those nets, with slew propagation stopping at the immediate fanout of the cell

Algorithm 3: Cell relocation

```
1  sel_cells←Movable cells with timing violations;
2  sel_cells←sel_cells ∪ Non-critical cells at the immediate fanout of sel_cells;
3  foreach cell j ∈ sel_cells in topological order do
4      [best_cost^L,best_cost^E] ← localCost(j) ;
5      best_location←locationOf(j) ;
6      cand_pos[j] ← Candidate slots in Search_Window[j];
7      foreach position (x,y) ∈ cand_pos[j] do
8          move cell j to (x,y);
9          update timing locally;
10         [new_cost^L,new_cost^E] ← localCost(j);
11         if (new_cost^L < best_cost^L ) and (new_cost^E < best_cost^E) then
12             best_cost^L ←new_cost^L ;
13             best_cost^E ←new_cost^E ;
14             best_location← (x,y) ;
15         end
16     end
17     move cell j to best_location ;
18     update timing locally ;
19 end
```

under relocation. Using the new delays computed after the movement, we evaluate the local cost function, as described in the next sub-section. If the new local cost is better than the previous best value, after testing separately the early and the late part of the cost, this candidate location is stored. After trying all the candidate locations, the cell is finally moved to the best stored location.

Every cell movement is made to a legal location using the Jezz legalizer [123]. In this way, any disturbance to the neighborhood around the moved cell is directly taken into account, and, if it degrades timing, it would be handled in the following iterations. Jezz has been used *without* any modifications to prioritize the displacement of timing-critical cells versus non-critical neighbors. Any other legalizer could have readily been used in the place of Jezz.

Contrary to flip-flop clustering, Algorithm 3 does not examine all cells. Only cells with timing violations and non-critical gates (with positive slack) on their fanout are considered. These non-critical gates can change the load of the output net of the critical driver and, therefore, improve its delay. For each cell, based on its current location, new

candidate positions are identified inside an appropriately constructed search window, the size and orientation of which are biased by the local LMs. The formation of this search window is described in Section 2.5.

Before the LM updates and after flip-flop and LCB relocation, the timing is updated incrementally (in a global sense), in order to reflect the new arrival times and the required arrival times that emerge after cell movement.

The iterative optimization stops when a maximum number of iterations is reached, or when TNS stops improving (by more than 1%) for two consecutive iterations. At the end of the main optimization loop, a final brute-force timing recovery step is performed on a list of a few most-critical paths to reduce only early violation.

2.4.1 Local Cost Function

To judge the suitability of each candidate location, we compute a local cost, using Algorithm 4, which reflects the local value of the global cost function (2.16). The local cost involves the summation of the product of the neighbor arc delays and the corresponding LM for late and early timing. The late and early costs are computed separately to enable the cell relocation algorithm to select the best candidate location. The local timing arcs that are included in the calculation of the local cost function involve the timing arcs of the cell under consideration, its immediate fanin and fanout cells, as well as the timing arcs of the cells driven by its fanin cells (side arcs).

Algorithm 4: localCost(Cell c)

```
1  [cost^L, cost^E] ← [0,0] ;
2  foreach arc i → j of local_arcs of c do
3  |   cost^L ← cost^L + λ^L_{i,j} · d^L_{i,j};
4  |   cost^E ← cost^E + λ^E_{i,j} · (−d^E_{i,j});
5  |   if arc i → j is the clock-to-Q arc of a flip flop then
6  |   |   cost^L ← cost^L + λ^E_{Dj} · (−d^L_{jCLK,Q});
7  |   |   cost^E ← cost^E + λ^L_{Dj} · d^E_{jCLK,Q};
8  |   end
9  end
10 return [cost^L, cost^E];
```

For example, in the case of gate 6 shown in Figure 2.3, the local arcs used for evaluating the local cost function consist of the timing arcs of the gate itself, ($3 \rightarrow 6$, $4 \rightarrow 6$), the arcs of the cells driving gate 6 ($12 \rightarrow 3$, $1 \rightarrow 3$, $5 \rightarrow 4$), the arcs of the immediate

fanout of gate 6 ($6 \rightarrow 9$, $6 \rightarrow 10$, $6 \rightarrow 11$), and the arcs of the cells being driven by the fanins of the gate under consideration ($4 \rightarrow 7$). For an LCB, like LCB 8 in Figure 2.3, we consider the timing arcs with respect to other LCBs, or the clock pins of flip-flops: local_arcs $= \{\{5 \rightarrow 8\}, \{2 \rightarrow 5\}, \{8 \rightarrow 11, 8 \rightarrow 14\}, \{5 \rightarrow 4\}\}$. Similarly, in the case of flip-flops, the timing arcs of both D, Q, and clock pins are considered, covering all the local fanin and fanout connections of the flip-flop. For flip-flop 11 in Figure 2.3, the local arcs consist of the following set of arcs: local_arcs $= \{\{8 \rightarrow 11, 6 \rightarrow 11\}, \{5 \rightarrow 8, 4 \rightarrow 6, 3 \rightarrow 6\}, \{11 \rightarrow 12, 11 \rightarrow 13, 11 \rightarrow 14\}, \{6 \rightarrow 9, 6 \rightarrow 10, 8 \rightarrow 14\}\}$.

2.4.2 Lagrange Multiplier Update

Initially, all LMs are initialized to 1. Then, the LMs for each PO and D pin of a flip-flop, for late and early timing, are updated using the modified subgradient optimization proposed in [152], [53], at the beginning of each iteration, as follows:

$$\lambda_j^L = \lambda_j^L \left(\frac{a_j^L}{r_j^L} \right), \quad \lambda_j^E = \lambda_j^E \left(\frac{r_j^E}{a_j^E} \right) \tag{2.17}$$

After updating the output LMs, their values must be distributed to all nets satisfying the flow conservation conditions (2.8)–(2.15). The distribution is performed by traversing the circuit in reverse topological order. At each visited cell, the sum of LMs at the output pins are distributed to the LMs of the input pins. When an LM value needs to be distributed to multiple incoming arcs, this distribution is done based on the ratio of the LMs of the corresponding timing arcs. Such distribution increases the LMs on critical paths, and, therefore, the worst negative slack is also expected to be minimized. Also, since LMs are accumulated at each branching point, the higher the number of violating endpoints affected by an arc, the higher the value of the corresponding LM.

The update of the LMs of all internal timing arcs $i \rightarrow j$, $\lambda_{i,j}$, for late and early timing, is done according to (2.18)–(2.19), following the method presented in [141]:

$$\lambda_{i,j}^L = \lambda_{i,j}^L \left(1 - \frac{r_j^L - (a_i^L + d_{i,j}^L)}{T} \right)^K \tag{2.18}$$

$$\lambda_{i,j}^E = \lambda_{i,j}^E \left(1 - \frac{(a_i^E + d_{i,j}^E) - r_j^E}{T} \right)^K \tag{2.19}$$

The numerator of each fraction is the slack at the output pin of cell j. If the slack is negative, the term in the brackets is greater than 1, thus increasing the corresponding LM. To increase the LM value on timing critical cells in high rate, we empirically set

$K = 4$. Smaller values give slower convergence, while larger values do not show further improvement in timing QoR. On the other hand, when the arc is non-critical, the value in the brackets is less than 1, thereby decreasing the LM value. When the slack is positive, we set $K = 1$ to decrease the LMs slowly. This method prevents the criticality of a path from being forgotten immediately after the timing violation at the endpoint is removed, thus avoiding criticality oscillations, e.g., critical arcs becoming non-critical.

2.4.3 Timing Recovery with Flip-Flop-to-LCB Re-assignment

The efficiency of the timing-driven placement flow depends on the placement utilization of the design, and on how much space is available for moving cells to their appropriately selected positions with respect to timing. In certain cases, cells do not have much freedom to move, due to nearby placement blockages, or macros. Therefore, when cell relocation has converged – either because cells have reached their maximum displacement limit, or there is limited placement freedom nearby – timing can be improved solely by appropriate LCB-to-flip-flop re-assignment. In this context, we examine the 20 most critical flip-flops and test, in a brute-force manner, whether reconnecting each flip-flop to a different nearby LCB would improve timing or not. Ten nearby LCBs per flip-flop are examined. For each flip-flop-to-LCB re-assignment tried, we perform a full incremental timing update, in order to be certain about the expected savings in timing. Each examined flip-flop stays assigned to the LCB that offers the best overall timing. The preferred LCB is the one that reduces TNS on early or late timing, without increasing WNS on the opposite mode, i.e., late or early, respectively.

2.5 Placement of the Search Window

A critical aspect of the timing optimization process is the identification of appropriate new candidate positions for each cell. In this work, cell movement is facilitated through the use of a *search window*. This window encloses the candidate cell, and it includes all positions that the cell could potentially occupy as part of the optimization process. The proposed methodology, dictating how this search window is positioned around the candidate cell, relies on vectors that indicate desirable cell-movement directions. Specifically, each fanin cell $i \in I_j$ and each fanout cell $k \in O_j$ is associated with two independent vectors (one for late and one for early timing violations) on cell j: $v^L_{i,j}$, $v^E_{i,j}$ and $v^L_{j,k}$, $v^E_{j,k}$. These vectors indicate the desired direction of cell movement that would be beneficial in terms of timing. All vectors originate from the center of cell j, and they lay on the imaginary lines connecting the center of cell j to the center of all of its fanin/fanout cells, as depicted in Figure 2.5(a).

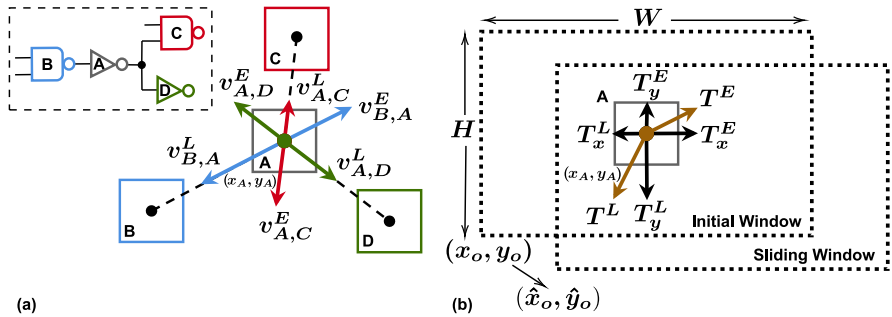


Figure 2.5: (a) Two independent vectors (one for late and one for early timing violations) are associated with each fanin and fanout cell; their magnitude reflects their timing criticality. The vectors indicate the desired directions of cell movement that would be beneficial in terms of timing. (b) All late and early vectors are added together to form resultant late and early vectors T^L and T^E. The directions and magnitudes of these resultant vectors determine the orientation of the search window.

The magnitude of each vector is equal to the value of the LM between these two cells ($|v^L_{i,j}| = \lambda^L_{i,j}$, $|v^E_{i,j}| = \lambda^E_{i,j}$ and $|v^L_{j,k}| = \lambda^L_{j,k}$, $|v^E_{j,k}| = \lambda^E_{j,k}$). The higher the value of the LM, the more timing-critical the net is, which results in a "stronger" (i.e., of larger magnitude) vector. The LMs are considered ideal proxies, because they encompass the timing criticality of each path, in terms of *all* timing arcs passing through that path.

As illustrated in Figure 2.5(a), all the late vectors $v^L_{i,A}$, $v^L_{A,k}$ that act on cell A are always attractive, i.e., they point towards the interconnected nets. By moving the pin in these designated directions, the delay would be reduced, which would decrease the late timing violation. In contrast, all the early vectors $v^E_{i,A}$, $v^E_{A,k}$ are always repulsive, i.e., they point away from the interconnected cells. By moving the pin in these opposite directions, the delay would be increased, which would decrease the early timing violation. All vectors acting on cell j can be viewed as individual "forces" pushing and pulling cell j in their respective directions, towards and away from all interconnected cells. Naturally, these individual "forces" can be added to evaluate the *net* late and early effects on cell j. All late vectors are added together to form one resultant late vector T^L, while a similar process is followed to yield one resultant early vector T^E, as shown in Figure 2.5(b).

The directions of these two resultant vectors must now be used to determine the exact placement location of the search window with respect to cell j. Each of the two resultant vectors T^L and T^E is projected onto the two coordinate axes, in order to extract its

Table 2.2: ICCAD-2015 Contest Benchmark Characteristics

Circuit	# Nodes	# Flops	# LCBs	Density	Max. Disp. (μm)	
					Short	Long
sb1	1209716	144266	7213	0.80	40	400
sb3	1213253	167923	8396	0.87	40	400
sb4	795645	176895	8843	0.90	20	400
sb5	1086888	114103	5704	0.85	30	400
sb7	1931639	270219	13510	0.90	50	500
sb10	1876103	241267	12063	0.87	20	500
sb16	981559	142543	7126	0.85	30	400
sb18	768068	103544	5177	0.8	20	400

x (horizontal) and y (vertical) components. Subsequently, the horizontal and vertical components of the two vectors are added up in the following manner: all the positive horizontal components (pointing to the right) are added to form $|R_j|$; all the negative horizontal components (pointing to the left) are added to form $|L_j|$; all the positive vertical components (pointing upwards) are added to form $|U_j|$; and, finally, all the negative vertical components (pointing downwards) are added to form $|D_j|$.

Our goal is to place the search window of cell j in a position that is proportional to the calculated values of $|R_j|$, $|L_j|$, $|U_j|$, and $|D_j|$. Let us assume that the bottom-left corner of cell j is placed at the (x_j, y_j) position, the search window has width W and height H, and the location of the bottom-left corner of the search window is at (x_o, y_o). Initially, the search window is placed such that the bottom left corner of cell j is at the center of the search window. Therefore, $x_o = x_j - \frac{W}{2}$ and $y_o = y_j - \frac{H}{2}$. The new location of the bottom-left corner of the search window, $(\hat{x}_o, \hat{y}_o)$, is determined by using the following equations:

$$\hat{x}_o = x_j - \frac{|L_j|}{|L_j| + |R_j|} \cdot W, \quad \hat{y}_o = y_j - \frac{|D_j|}{|D_j| + |U_j|} \cdot H$$

Essentially, the ratios of the left-to-right components and the up-to-down components determine the magnitude of the search window's slide in x and y directions.

Having determined the final location of the search window for cell j, we identify a set of N candidate positions uniformly spaced inside the search window. Let us denote the spatial granularity in the x and y dimensions as $step_x$ and $step_y$, respectively. The values of $step_x$ and $step_y$ can be determined from the maximum displacement constraint and the relationship $W/step_x \cdot H/step_y = N$. We iterate from $\hat{y_o}$ to $\hat{y_o} + H$ with granularity $step_y$, and from $\hat{x_o}$ to $\hat{x_o} + W$ with granularity $step_x$. In this manner, a total of N different candidate positions are investigated for cell j, all situated within the search window.

Table 2.3: Timing improvement with short and long displacement limits. Late/early WNS (worst negative slack), late/early TNS (total negative slack) results, as compared to the first-place winner of the ICCAD-2015 contest.

Short Displacement

Circuit	Late						Early						LR iter
	WNS (ns)			TNS (ns)			WNS (ps)			TNS (ps)			
	Init	1st	Ours	Init	1st	Ours	Init	1st	Ours	Init	1st	Ours	
sb1	-4.98	-4.66	-4.60	-459.74	-374.31	-369.19	-9.34	-3.83	0.00	-317.44	-41.56	0.00	6
sb3	-10.15	-9.43	-9.12	-1502.83	-1373.12	-1301.26	-78.36	-65.72	-4.74	-1458.78	-683.51	-17.44	5
sb4	-6.22	-5.94	-5.99	-3476.69	-3195.33	-3065.46	-12.55	-6.08	-8.40	-519.39	-173.92	-50.20	14
sb5	-25.70	-25.07	-25.11	-6965.15	-6779.95	-6639.30	-36.77	-36.77	-12.00	-591.42	-585.78	-111.40	4
sb7	-15.22	-15.21	-15.21	-1857.38	-1703.78	-1579.86	-7.65	-6.75	-6.80	-1985.85	-1943.74	-1821.19	6
sb10	-16.49	-16.18	-16.28	-33152.80	-32514.40	-31649.30	-8.62	-8.62	-2.02	-620.95	-361.06	-13.11	6
sb16	-4.58	-4.36	-4.24	-776.04	-514.25	-418.72	-10.65	-8.38	-2.49	-113.75	-30.67	-2.49	8
sb18	-4.55	-4.12	-4.08	-1034.80	-943.64	-929.43	-19.01	-3.81	-0.03	-283.00	-69.38	-0.03	4
Avg	-10.99	-10.62	-10.58	-6153.18	-5924.85	-5744.06	-22.87	-17.50	-4.56	-736.32	-486.20	-251.98	6.62
Save	-	4.57%	5.35%	-	11.34%	15.66%	-	29.96%	69.81%	-	50.00%	84.29%	-

Long Displacement

Circuit	Late						Early						LR iter
	WNS (ns)			TNS (ns)			WNS (ps)			TNS (ps)			
	Init	1st	Ours	Init	1st	Ours	Init	1st	Ours	Init	1st	Ours	
sb1	-4.98	-4.57	-4.42	-459.74	-351.23	-323.94	-9.34	-16.65	0.00	-317.44	-80.89	0.00	9
sb3	-10.15	-8.70	-8.27	-1502.83	-1160.04	-881.59	-78.36	-13.13	-3.29	-1458.78	-214.03	-16.50	16
sb4	-6.22	-5.76	-5.60	-3476.69	-2464.56	-2309.54	-12.55	-12.28	-3.13	-519.39	-53.84	-13.80	8
sb5	-25.70	-24.29	-24.70	-6965.15	-5842.23	-6327.55	-36.77	-36.77	-12.24	-591.42	-618.27	-54.01	5
sb7	-15.22	-15.21	-15.21	-1857.38	-1510.76	-1454.46	-7.65	-6.75	-7.35	-1985.85	-1958.34	-1820.90	7
sb10	-16.49	-16.07	-16.13	-33152.80	-31517.80	-29445.10	-8.62	-5.15	-2.40	-620.95	-373.75	-12.50	5
sb16	-4.58	-3.84	-3.35	-776.04	-265.56	-209.59	-10.65	-7.55	-1.85	-113.75	-37.64	-1.85	15
sb18	-4.55	-3.81	-3.80	-1034.80	-775.84	-701.43	-19.01	-1.95	-0.02	-283.00	-6.86	-0.02	20
Avg	-10.99	-10.28	-10.19	-6153.18	-5486.00	-5206.65	-22.87	-12.53	-3.79	-736.32	-417.95	-239.95	10.62
Save	-	8.79%	11.14%	-	25.76%	31.46%	-	22.26%	74.52%	-	56.33%	86.47%	-

2.6 Experimental Results

The proposed flow was implemented in C++ using the open-source RSyn framework [45] as a single-threaded application. RSyn provides all necessary functions for netlist traversal and cell relocation, as well as incremental timing analysis needed by the proposed method. The new method is evaluated using the ICCAD-2015 benchmark set [81]. Table 2.2 shows some of the basic characteristics of each of the benchmarks used as well as the target density and the short and long maximum allowed displacement constraints that all cells should satisfy. All experiments were performed on the same Linux-based workstation using a 3.6 GHz Intel Core i7-4790 with four cores and 32 GB of RAM. The final reported results are validated using the scripts provided by the contest organizers, and OpenTimer [72], which is the reference timer used for evaluation purposes in the above-mentioned contest.

In all cases, our method executes the flow depicted in Figure 2.4, where flip-flop clustering precedes the iterative LR-based timing-driven placement. In each iteration, for

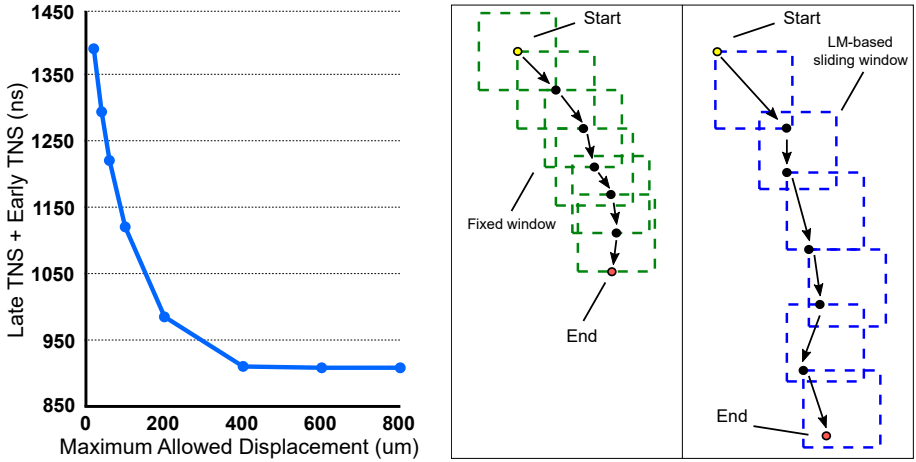
the relocation of each cell, we identify 20 candidate positions uniformly spaced inside a rectangular search window of size W=H=20 rows. For the examined benchmark set, 20 rows correspond roughly to 68μm, which covers almost all of the allowed displacement in the case of the short displacement constraint, and it progressively reaches the maximum displacement in the case of the long displacement constraint.

2.6.1 Comparison with winner of the ICCAD 2015 contest

Table 2.3 summarizes the results achieved by the proposed algorithm, as compared to the initial design characteristics ('Init') and the performance of the first-place winner ('1st') of the ICCAD-2015 contest [53]. As shown in Table 2.3, in all cases, the timing of the designs is either almost the same with that derived by the winner of the contest, or significantly better. Our method smoothly reduces both the late and early timing violations, without creating a tradeoff between the two. For instance, for short displacement, the winner of the contest improves the initial late WNS and TNS by 4.57% and 11.34%, respectively, on average, while the proposed method increases the average savings to 5.35% and 15.66%, respectively. At the same time, WNS and TNS for early timing are significantly more improved, as compared to [53]: early WNS is reduced by a further 40% (69.81% vs. 29.96%), on average, and early TNS by a further 34% (84.29% vs. 50.00%), for the short displacement constraint. The obtained results represent the combined effect of (a) the timing-compatibility clustering that separates incompatible flip-flops, without enabling the dynamic timing update of the flip-flops's profiles, (b) the newly proposed LR-based timing optimization framework, and (c) the LM-driven cell relocation technique that moves all types of cells in a uniform manner. The number of iterations that LR-based cell relocation requires, per design, are depicted in the last column of Table 2.3.

The proposed approach efficiently utilizes the greater placement freedom given by the long displacement limit. In all cases, timing is improved when compared to the results obtained for the short displacement limit, as shown in Table 2.3. To better highlight how the proposed algorithm utilizes the available displacement, we performed timing-driven placement optimization on benchmark sb3 for various displacement constraints. The sum of early and late TNS achieved in each case is depicted in Figure 2.6(a). Timing is improved with increasing displacement constraints, until saturation is reached, which shows that the placement efficiency (in improving timing) has reached a plateau.

This result also stems from the proposed sliding of the search window. Figure 2.6(b) displays the trajectory followed by a cell of the sb3 benchmark, and the progressive placement of the LM-based sliding window and a fixed window of the same size. Using the LM-based window, the cell is allowed to move quickly to its final destination by examining more timing-effective candidate positions, thus helping in converging faster



(a) (b)

Figure 2.6: (a) Final TNS on benchmark *sb3* for various displacement constraints. (b) The evolution of cell relocation using the proposed LM-driven search window and an equally sized search window.

to an overall timing-efficient solution. Additional experimental results reveal that, in all benchmarks, the replacement of the LM-based sliding of the search window by a fixed search window would degrade the overall timing quality by 10% (computed as the average degradation across all WNS/TNS timing metrics.)

The way the LMs are updated enables both fast convergence and better overall timing QoR, since the delay of timing-critical arcs is appropriately emphasized relative to other non-critical timing arcs. Figure 2.7 compares the normalized sum of late and early TNS obtained using the LM update approach of [53] to that obtained when using the proposed LM update technique, for the sb10 benchmark with long displacement limit. For the proposed method, results with different exponents K for the LM updates of (2.18)-(2.19) are also shown in the figure. Obviously, the proposed LM update method yields significantly better TNS results than the LM update approach of [53]. As K is increased, the proposed method exhibits faster convergence and better overall timing QoR. Beyond K=4, further improvement in TNS is marginal, thereby leading us to the selection of K=4 for all our experiments.

The results of the proposed methodology presented thus far were obtained *without dynamic updates* of the timing profiles in the flip-flop clustering step. If dynamic timing update is enabled, the overall timing QoR is improved for the proposed method, as

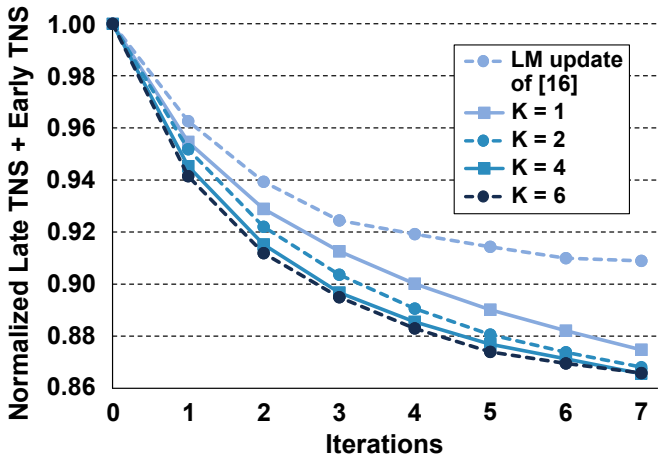


Figure 2.7: TNS comparison between the LM update approach of [53] and the proposed LM update approach for the sb10 benchmark. Higher exponent values of K aim to increase the LMs of timing-critical arcs. As K increases, the TNS decreases more quickly, with diminishing returns beyond K=4.

shown in Table 2.4. As an example, for the short displacement constraint, having the accurate slack values during flip-flop clustering helps improve early WNS by a further 2% (71.84% in Table 2.4 vs. 69.81% in Table 2.3), while early TNS improves by a further 1% (85.44% vs. 84.29%). Correspondingly, late WNS is improved by 0.2% (5.56% vs. 5.35%), and late TNS by 0.5% (16.13% vs. 15.66%). The reason for the minimal improvement is the fact that dynamic timing updates only affect weak flops with small timing slacks. Even with stale timing profiles, those flops are easily corrected by gate/flip-flop relocations in the subsequent optimization process.

Unfortunately, the minimal improvements in timing QoR with dynamic timing updates enabled come at a hefty runtime cost of 6×, on average, relative to disabling the dynamic timing updates. Hence, the decision to enable the dynamic timing update feature is left to the engineer, who may wish to investigate whether a particular design benefits from this additional step, or not.

2.6.2 Comparison with recent state-of-the-art

In the following set of experiments, we compare the proposed algorithm to two recent state-of-the-art timing-driven placement optimization methods; namely, EHC [67] and

Table 2.4: Late/Early WNS (worst negative slack) and late/early TNS (total negative slack) results when dynamic update of timing profiles is enabled in the flip-flop clustering step.

Short Displacement

Circuit	Late		Early	
	WNS (ns)	TNS (ns)	WNS (ps)	TNS (ps)
sb1	-4.55	-363.10	0.00	0.00
sb3	-9.10	-1290.17	-2.12	-7.21
sb4	-5.99	-3083.13	-10.00	-16.67
sb5	-25.11	-6635.68	-14.00	-115.30
sb7	-15.21	-1562.71	-6.80	-1832.40
sb10	-16.27	-31563.40	-1.37	-6.33
sb16	-4.23	-415.31	0.00	0.00
sb18	-4.07	-924.28	0.00	0.00
Avg	-10.57	-5729.72	-4.29	-247.24
Save	5.56%	16.13%	71.84%	85.44%

Long Displacement

Circuit	Late		Early	
	WNS (ns)	TNS (ns)	WNS (ps)	TNS (ps)
sb1	-4.32	-313.12	0.00	0.00
sb3	-8.18	-860.39	-2.88	-7.99
sb4	-5.60	-2580.72	-4.00	-5.95
sb5	-24.29	-6176.79	-15.00	-60.10
sb7	-15.21	-1428.32	-6.87	-1830.00
sb10	-16.12	-28573.2	-1.37	-7.85
sb16	-3.30	-202.68	0.00	0.00
sb18	-3.79	-673.84	0.00	0.00
Avg	-10.10	-5101.13	-3.77	-238.99
Save	11.88%	32.18%	77.24%	86.84%

FPUSM [82]. The obtained results are summarized in Table 2.5. The results of the proposed methodology are obtained without dynamic updates of the timing profiles in the FF clustering step. In most cases, the proposed methodology achieves similar, or better, results, in terms of early timing, and it is always slightly better in late timing. The average savings ('Save') reported in the last rows of Table 2.5 report the average savings achieved by each method relative to the initial designs ('Init') reported in the second column of Table 2.3.

Table 2.5: Timing improvement with short and long displacement limits. Late/early WNS (worst negative slack), late/early TNS (total negative slack) results, as compared to EHC [67] and FPUSM [82].

Short Displacement

Circuit	Late						Early					
	WNS (ns)			TNS (ns)			WNS (ps)			TNS (ps)		
	EHC	FPUSM	Ours	EHC	FPUSM	Ours	EHC	FPUSM	Ours	EHC	FPUSM	Ours
sb1	-4.67	-4.66	-4.60	-373.99	-374.25	-369.19	-0.04	0.00	0.00	-0.04	0.00	0.00
sb3	-9.53	-9.43	-9.12	-1373.51	-1373.13	-1301.26	-59.59	-21.58	-4.74	-393.55	-153.08	-17.44
sb4	-5.94	-5.94	-5.99	-3153.70	-3195.38	-3065.46	-5.50	-2.20	-8.40	-65.32	-3.42	-50.20
sb5	-25.08	-25.07	-25.11	-6775.95	-6779.94	-6639.30	-31.33	-36.77	-12.00	-271.98	-265.37	-111.40
sb7	-15.22	-15.21	-15.21	-1696.02	-1703.81	-1579.86	-6.75	-6.36	-6.80	-1875.86	-1858.34	-1821.19
sb10	-16.19	-16.18	-16.28	-32514.40	-32514.50	-31649.30	-5.87	-2.19	-2.02	-306.75	-9.09	-13.11
sb16	-4.04	-4.36	-4.24	-444.16	-514.25	-418.72	-0.05	0.00	-2.49	-0.06	0.00	-2.49
sb18	-4.08	-4.12	-4.08	-938.77	-943.69	-929.43	-2.56	0.00	-0.03	-22.80	0.00	-0.03
Avg	-10.59	-10.62	-10.58	-5908.81	-5924.87	-5744.06	-13.96	-8.64	-4.56	-367.05	-286.16	-251.98
Save	5.39%	4.57%	5.35%	12.74%	11.34%	15.66%	53.02%	68.29%	69.81%	70.31%	81.12%	84.29%

Long Displacement

Circuit	Late						Early					
	WNS (ns)			TNS (ns)			WNS (ps)			TNS (ps)		
	EHC	FPUSM	Ours	EHC	FPUSM	Ours	EHC	FPUSM	Ours	EHC	FPUSM	Ours
sb1	-4.57	-4.57	-4.42	-351.06	-351.21	-323.94	-0.87	-0.43	0.00	-2.18	-0.43	0.00
sb3	-8.69	-8.70	-8.27	-1159.93	-1160.07	-881.59	-4.46	-5.54	-3.29	-9.10	-29.05	-16.50
sb4	-5.76	-5.76	-5.60	-2437.77	-2462.93	-2309.54	-12.28	0.00	-3.13	-55.62	0.00	-13.80
sb5	-24.29	-24.29	-24.70	-5840.52	-5842.28	-6327.55	-58.34	-36.77	-12.24	-61.10	-268.60	-54.01
sb7	-15.22	-15.21	-15.21	-1510.76	-1510.79	-1454.46	-6.75	-6.38	-7.35	-1958.34	-1858.48	-1820.90
sb10	-16.09	-16.07	-16.13	-31563.90	-31518.00	-29445.10	-2.73	-2.20	-2.40	-40.60	-3.47	-12.50
sb16	-3.69	-3.84	-3.35	-234.07	-265.57	-209.59	-0.20	0.00	-1.85	-0.31	0.00	-1.85
sb18	-3.78	-3.81	-3.80	-771.96	-775.87	-701.43	-0.20	0.00	-0.02	-0.20	0.00	-0.02
Avg	-10.26	-10.28	-10.19	-5483.75	-5485.84	-5206.65	-10.73	-6.42	-3.79	-265.93	-270.00	-239.95
Save	9.27%	8.79%	11.14%	26.40%	25.76%	31.46%	50.70%	72.42%	74.52%	84.02%	82.29%	86.47%

Even though the improvements achieved by the proposed method may initially seem modest, as compared to EHC and FPUSM, the comparison should be viewed in the correct overall context: both EHC and FPUSM are not complete, self-contained optimization methods. Instead, they rely on another preceding technique to first close – as much as possible – the negative slack for late timing. Upon completion of this first step by the other technique, EHC and FPUSM focus on early timing optimization. This is precisely the reason why these two methods were applied to the outcome of the first-place winner of the ICCAD-2015 contest, which significantly pre-optimized (especially for late timing) the designs. On the contrary, the proposed methodology is self-contained and does not rely on other techniques for any pre-optimizations.

It should also be noted that, even though EHC and FPUSM focus on LCB-to-flip-flop re-assignments and LCB/flip-flop movements, the proposed approach *generalizes* this methodology by moving LCBs and flip-flops inside LR-based placement. In general, prior work only moves gates within LR-based placement. The new methodology is holistic and all-encompassing in its optimization approach, surpassing the performance

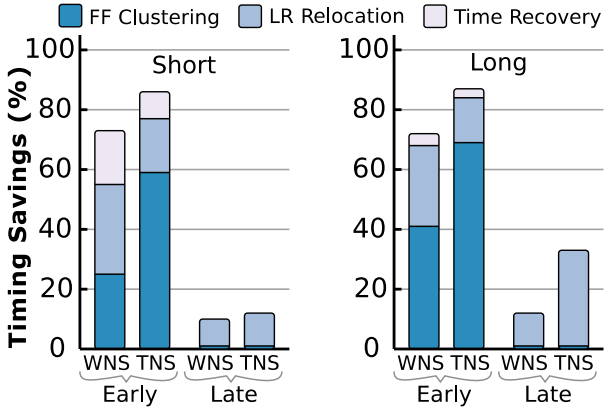


Figure 2.8: The incremental timing savings relative to the initial benchmarks obtained by each step of the proposed method, for early and late timing.

of EHC and FPUSM in most cases for late timing, and closely matching their performance in early timing. More specifically, the new approach always performs better in *sb3* and *sb5* in early timing, and it is always worse in *sb10* in early TNS timing. The latter is attributed to the fact that, in *sb10*, the early critical paths are between flip-flops with very low placement freedom, because they are placed close to macros that block placement.

The obtained timing QoR is the synergistic result of all three algorithms shown in the cell relocation flow of Figure 2.4, i.e., FF clustering, LR-based cell relocation, and early timing recovery. Figure 2.8 depicts the average impact of each step over all ICCAD 2015 benchmarks for short and long displacement limits. Timing-compatibility flip-flop clustering prepares the LCB-to-Flip-Flop connections in such a way that it reduces early timing, but leaves late timing unaffected. In contrast, the LR-based iterative optimization that follows is more complete and provides a combined improvement in both early and late timing. Finally, early timing recovery is a fast step that only contributes in early timing and its effect is more pronounced in short displacement limits.

Finally, the proposed incremental timing-driven placement technique was compared to OWARU [77], which focuses entirely on late timing optimizations. The obtained results are presented in Table 2.6. For a fair comparison with OWARU, we have replaced the performance numbers of our method (when applied as a whole) with the numbers achieved for late timing when performing only LR-based gate relocation, *excluding* FF clustering, FF and LCB movement, and timing recovery. In all cases, the proposed approach achieves better results than OWARU [77].

Table 2.6: Timing improvement of late WNS (worst negative slack) and late TNS (total negative slack) as compared to OWARU [77] using only gate relocation.

Circuit	Late Timing/Short Displacement			
	WNS (ns)		TNS (ns)	
	OWARU	Ours	OWARU	Ours
sb1	-4.8	-4.61	-426	-368.30
sb3	-9.8	-9.12	-1408	-1301.49
sb4	-6.1	-5.99	-3379	-3092.90
sb5	-25.5	-25.13	-6916	-6660.28
sb7	-15.2	-15.21	-1759	-1582.20
sb10	-16.4	-16.31	-32816	-31707.00
sb16	-4.4	-4.25	-605	-424.06
sb18	-4.2	-4.08	-997	-928.58
Avg	-10.8	-10.6	-6038.3	-5758.1
Save	2.75%	5.26%	6.15%	15.44%

2.6.3 Runtime comparisons

In addition to yielding timing improvements, the *runtime* of an optimization methodology is also a critical attribute. The runtime evaluation results in Table 2.7 indicate that the proposed approach is close to the state-of-the-art. Note that the reported runtimes for the competing techniques are taken verbatim from their respective papers. Consequently, those runtimes correspond to other machines with different specifications than the one we used. Therefore, the comparisons can only be broadly and generally indicative. Regardless, we include those runtime numbers here in a big-picture context, to demonstrate that the runtimes of the proposed approach are reasonable, as compared to the others.

The runtimes of the new technique are close – in some cases better, in others worse – to the first-place winner of the ICCAD-2015 contest. However, the new approach utilizes its runtime more effectively, by yielding substantially better QoR, especially for early timing. In the case of long displacement, the new technique's runtimes are longer, since there are more options to search before converging to a solution. Also, the proposed method exhibits more-or-less similar runtimes to EHC and FPUSM. However, for a fair and meaningful runtime comparison to EHC and FPUSM, one would need to also add the runtime of the first-place winner of the ICCAD-2015 contest to those techniques. Recall, that both EHC [67] and FPUSM [82] are applied after the completion of the optimization of the first-place winner.

The runtime of the proposed method is the additive result of the three main steps

Table 2.7: Runtime comparison of state-of-the-art timing-driven placement techniques. Runtimes are reported in minutes.

Circuit	Short Displacement				Long Displacement			
	1st	EHC	FPUSM	Ours	1st	EHC	FPUSM	Ours
sb1	23	84	10	31	32	84	10	43
sb3	23	92	23	21	27	92	25	49
sb4	17	39	11	137	19	39	10	65
sb5	23	88	16	17	25	88	15	20
sb7	43	172	21	38	53	172	20	41
sb10	40	166	18	102	37	166	18	88
sb16	19	73	9	28	22	73	10	41
sb18	15	41	9	10	16	41	10	32

Runtime (in minutes) of each step of the proposed methodology.

Method		sb1	sb3	sb4	sb5	sb7	sb10	sb16	sb18
Short	FF-Clust	3.8	4.8	5.1	2.2	12.4	9.8	3.5	1.9
	LR-Reloc	24.7	13.9	129.2	13.1	22.8	88.3	22.0	6.4
	Tim-Rec	2.5	2.0	2.8	1.7	3.0	4.0	2.1	1.4
Long	FF-Clust	3.8	4.8	5.1	2.2	12.4	9.8	3.5	1.9
	LR-Reloc	36.6	42.1	56.5	15.7	25.8	73.9	35.0	28.2
	Tim-Rec	2.6	2.1	2.9	1.8	3.0	4.1	2.1	1.8

of the overall flow. The contribution of each part is shown at the bottom of Table 2.7 for all benchmarks. As expected, the LR-based cell relocation consumes the majority of the runtime, while early timing recovery has a small marginal contribution. The FF clustering's runtime is always a small, or medium, percentage of the runtime of LR relocation, and its runtime is, by construction, the same for long and short displacement limits.

2.7 Conclusions

Timing-driven placement optimization is an integral cog of the complex process of achieving timing closure, and it is one of the key determinants of the overall QoR. This article presents a novel timing-driven placement optimization methodology based on an extended Lagrange Relaxation formulation. The fundamental contribution of this new approach is the concerted relocation of all types of cells (gates, flip-flops, and LCBs) in a unified manner. The LR-based placement optimization is complemented by a flip-

flop clustering algorithm that ensures the timing compatibility of the flip-flops of each cluster, thus facilitating the timing optimization through LCB movement. Additionally, a simple, yet effective, scaling factor artificially changes the distances of the members of the cluster from the cluster center. This helps create clusters of uneven sizes, thereby appropriately delaying, or speeding up, the clock arrival time. Extensive experimental evaluations using the ICCAD-2015 benchmarks demonstrate the efficacy of the proposed methodology, as compared to four state-of-the-art timing-driven placement-optimization techniques.

3 Incremental Lagrangian-Relaxation based Discrete Gate Sizing and Threshold Voltage Assignment

3.1 Introduction

Timing closure remains one of the most critical challenges of a physical synthesis flow, especially when considering that chip designs usually operate under many different operating conditions (e.g. different temperatures and voltages) with different electrical properties. However, the timing constraints of more than one mode/corner should be satisfied simultaneously at the end [94, 135]. Trying to remove a timing violation from one timing scenario could easily create a new violation in another. This behavior of the multimode multi-corner (MMMC) timing optimization, makes the physical process even more challenging.

Except of the need for timing closure, the design should also be free of any design rule violations such as maximum allowed capacitance and transition time. Large timing and design rule violations are analyzed and removed at the first steps of the design flow using efficient global optimization engines [104]. Still, a small set of remaining violations always exist close to the end of the flow. Repairing such violations requires incremental operations that are non-disruptive and execute as fast as possible. For instance, after routing, we don't want cells' placement to change for improving timing since this would cause re-routing a large part of the design thus possibly introducing new violations.

The problem becomes harder to solve when considering that the introduced timing violations may involve multiple corners that may need significantly different actions to remove them.

The least disruptive operations for improving design's characteristics during physical synthesis involve threshold voltage (V_T) re-assignment and gate sizing [23, 104]. V_T re-assignment tradeoffs smaller delay with increased leakage power and does not perturb routing nor it requires a new parasitics extraction after the change. Gate resizing, even if not as simple as V_T re-assignment, is still considered a fairly non-invasive operation. In the worst case, increasing cell's size (possibly avoiding exceedingly large changes) may

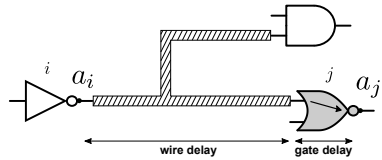
require an additional local legalization step [123, 148] and local re-routing of certain nets [25].

Inserting buffers is still an option at this step [4, 76, 151]. However, buffer insertion may ruin local placement and routing, which may be hard to fix later in highly congested designs. Other highly powerful optimization steps such as useful clock skewing are also considered hard to apply at the end of the flow, unless there is no other practical way to solve the remaining timing violations [42, 82, 151].

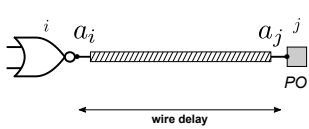
Gate sizing and V_T assignment algorithms have a long history in physical synthesis flows. Initial works assumed continuous sizes for the gates [43] but these approaches had delay inaccuracies compared to the real discrete gate sizes [118]. Coudert *et al.* [29] was from the first ones that proposed a gate sizing method that handles such discrete sizes. Many different methods have been studied to solve the size selection problem effectively. For example, linear programming (LP) has been used widely in the literature [11, 13, 75, 113]. Simulated annealing has been also used to solve the gate sizing problem because it can be applied on circuits containing million gates [129]. Daboul *et al.* [30] used the formulation of resource sharing to select gate sizes. Other approaches have proposed to apply dynamic programming (DP) [64, 92, 117, 124]. Alternative works, use sensitivity functions and from the available sizes, select the size that maximizes the power reduction with the minimal timing degradation [63, 78]. Some of these works have been extended in order to handle multiple timing corners and scenarios for more realistic designs [40, 135]. Even machine learning has been used for gate sizing. The latest work of [101] uses deep reinforcement learning to change the sizes and shows high quality final results.

Among the large set of available solutions, those that rely on Lagrangian Relaxation (LR) achieve significantly better results [47, 97, 118, 140, 142, 146]. However, when applied incrementally they need many iterations to converge even if the number of timing violators is small. Most LR-based sizers assume that they are allowed to initialize every cell of the design to a chosen initial state, e.g., initialize all cells to their minimum size [87], before beginning the optimization. This design disruption may seem reasonable at the early steps of the flow but is not allowed close to the the end.

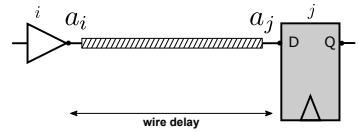
In this work, we propose a novel initialization strategy for multi-corner LR-based timing/power optimizers across multiple operating conditions that combines two useful benefits: On one hand we enjoy the optimization efficiency of an LR-based gate sizer and on the other hand we enjoy fast runtimes and true incremental operation, i.e., the optimized design is only marginally different from the original design but with the timing violations of multiple corners repaired. The method has been evaluated on benchmarks with small timing violations across single and multiple corners and has proven that successfully optimizes the timing with reduced runtime.



(a): Combinational gates



(b): Primary outputs



(c): Flip-flops

Figure 3.1: The definition of the arrival times a_i, a_j and delay $d_{i,j}$ for (a) combinational gates, (b) primary outputs and (c) flip-flops.

3.2 Basics of LR-based gate sizing

A timing-driven optimizer tries to minimize the power (or area) of the design given a set of timing constraints.

$$\begin{aligned} \textbf{minimize:} \quad & \sum_i leakage_i & & (3.1) \\ \textbf{subject to:} \quad & a_i + d_{ij} \leq a_j & \forall \text{ timing arc } i \rightarrow j & \\ & a_k \leq r_k & \forall \text{ endpoint } k & \end{aligned}$$

Variable a_i denotes the arrival time at the output pin of cell i while $d_{i,j}$ is the sum of wire and cell delay of the timing arc $i \rightarrow j$ which is defined from the output pin of the gate i to the output pin of the gate j. Figure 3.1 depicts the delays involved in the computation of d_{ij} for different cases. For combinational gates in the middle of a logic netlist, as shown in Figure 3.1(a), d_{ij} is the summation of the wire delay and the gate delay from output of gate i to the output pin of gate j.

Pin j may represent also a timing endpoint. Timing endpoints can be the primary outputs (POs) of a design or the inputs of flip-flops. When pin j belongs to the set of primary outputs (POs), as highlighted in Figure 3.1(b), delay d_{ij} is equal to the wire delay connecting the output pin of driver i and primary output j. Similarly, when pin j is a flip-flop input, shown in Figure 3.1(c), delay d_{ij} involves only the wire delay from driver i to the input D-pin of the flip-flop j. Parameter r_k is the required arrival time at any timing endpoint k [12].

Associating the constraint for each timing arc with a non-negative Lagrange Multiplier (LM) λ_{ij}, that acts as a penalty factor when the respective constraint gets violated, and computing the KKT optimality conditions [10, 109, 118], allows us to simplify the constrained minimization problem (3.1) to the equivalent unconstrained minimization problem (3.2).

$$\textbf{minimize:} \quad \sum_i leakage_i + \sum_{i \to j} \lambda_{ij} d_{ij} \tag{3.2}$$

The KKT optimality conditions with respect to the values of LMs impose that equation (3.3) should hold during optimization for all pins of the design

$$\sum_{i \to j} \lambda_{ij} = \sum_{j \to k} \lambda_{jk} \tag{3.3}$$

For the example shown in Figure 3.2, equation (3.3) for gate 6 implies that $\lambda_{36} + \lambda_{46} = \lambda_{67} + \lambda_{68}$.

State-of-the-art LR-based optimizers [47, 97, 118, 140, 142] try to minimize the global cost function (3.2) using many iterations of local gate resizing and V_T re-assignment steps. The overall optimization flow is depicted in Figure 3.3.

Initially, all gates are downsized to their least leakage power option (lowest size and highest V_T) [87, 112] that does not violate any design rule constraint. Solving design-rule violations early, simplifies the following local logic tuning steps. In the following,

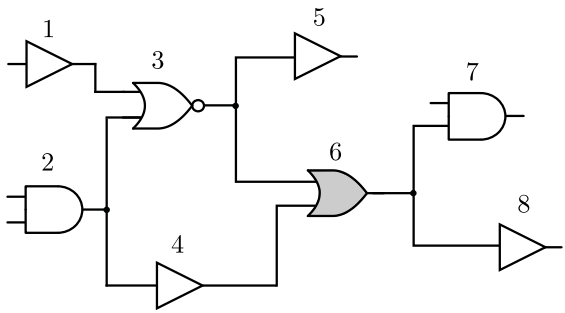


Figure 3.2: Example design to show which LMs are considered in the computation of local cost and to present the LM equalities to preserve optimality KKT conditions which imply that the sum of input LMs must be equal to the sum of output LMs. Thus, for the highlighted gate 6 the LMs $\lambda_{13}, \lambda_{23}, \lambda_{24}, \lambda_{35}, \lambda_{36}, \lambda_{46}, \lambda_{67}$ and λ_{68} multiplied with their corresponding delays will form the local cost. For LM propagation, for gate 6, we should guarantee that $\lambda_{36} + \lambda_{46} = \lambda_{67} + \lambda_{68}$.

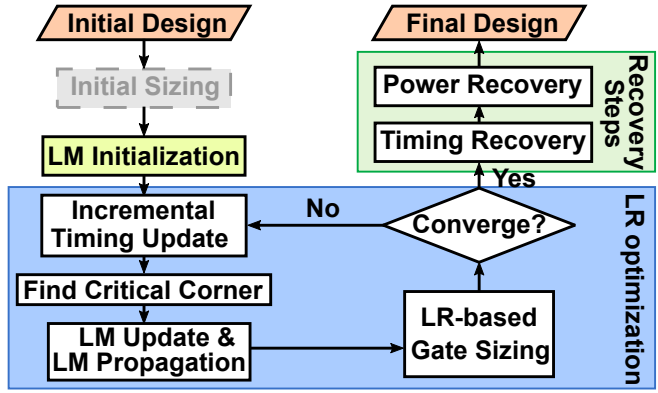


Figure 3.3: The overall LR-based gate sizing optimization flow.

all LMs are set to a starting value, usually to 1, and the main LR optimization loop starts. Each iteration of LR-based gate sizing begins with a full incremental timing update and then evolves in two phases. In the first phase, the LMs are updated and propagated to all gates to reflect the new criticality of the corresponding timing arcs. In the second phase, for each gate, examined in topological order, all possible discrete cell sizes and threshold voltages are tried, assuming constant LMs. The new version selected for the resized gate is the one that minimizes the cost function (3.2) and does not introduce any design-rule violations.

At each iteration, a full incremental timing update on all examined corners is needed to reflect the new timing violations. From all the available corners, the most critical corner is identified [151] for the current iteration. When there aren't any timing violations in any corner, we name critical the timing corner that gives the lowest total slack.

With the new timing information updated, the LMs should be updated too. The update may take different forms and can be either additive ($\lambda_{new} = \gamma + \delta\lambda_{old}$) or multiplicative ($\lambda_{new} = \gamma\lambda_{old}$) [152]. Following the proposal of [47] we use a multiplicative LM update depicted in (3.4):

$$\begin{aligned} \lambda_{ij} &= \lambda_{ij}\left(1+\frac{a_j - r_j}{T}\right)^{1/M} \quad \forall \text{ timing arc } i \to j \text{ with } a_j \geq r_j \\ \lambda_{ij} &= \lambda_{ij}\left(1+\frac{r_j - a_j}{T}\right)^{-M} \quad \forall \text{ timing arc } i \to j \text{ with } a_j < r_j \end{aligned} \tag{3.4}$$

Once the LMs have been updated, LMs must be propagated from output to input following a reverse topological order. In this way, the timing criticality measured at the

Algorithm 5: Find best size for gate g

```
1  min_cost ← inf ;
2  best_size ← size(g) ;
3  foreach equivalent size s of g do
4  |  resize g to s ;
5  |  if violates_Design_Rule_Constraint(g) then
6  |  |  skip s ;
7  |  end
8  |  update_timing_locally(g) ;
9  |  if timing_degradation_around(g) then
10 |  |  skip s ;
11 |  end
   |  // Using Equation (3.2)
12 |  cost ← leakage_g + Σ_{i→j around g} λ_ij d_ij ;
13 |  if (cost < min_cost) then
14 |  |  min_cost ← cost;
15 |  |  best_size ← s;
16 |  end
17 end
18 resize g to best_size ;
19 update_timing_locally(g) ;
```

timing endpoints should be transferred gradually to the internals gates of the design. LM propagation updates the LM values of internal timing arcs while still respecting KKT conditions (3.3)

The value of each LM reflects the timing criticality of each timing arc. LMs increase fast for critical timing arcs and reduce for non-critical timing arcs to favor power reduction. Implicitly, LMs keep also historic information (for the lifetime of an optimization run) with respect to the criticality of each timing arc. If a timing arc remained critical for multiple iterations it is still assumed critical by keeping a high value of LM, even if the slack at its output becomes positive in a certain iteration. In this way, drastic oscillations between critical and non-critical timing arcs are avoided and the optimization evolves smoothly reducing power while satisfying timing constraints.

Later on and assuming constant LMs, all gates are visited in topological order and for each gate the best size is selected using the same procedure described in Algorithm 5. First, the initial size of the gate is stored and then, each equivalent size of the gate

is tried. If the new tried size violates any design rule constraint, this size is rejected. Otherwise, the timing is updated locally, recomputing the new delays and slews of all nets that the examined gate is connected to. To avoid timing degradation, sizes that violate timing constraints are also rejected. If not, the local cost is calculated as the summation of the leakage power of the new size and the neighbor arc delays multiplied by their corresponding LMs.

In the local cost only the arcs whose delay may have changed are included. These are the arcs of the immediate fanin and fanout cells of the examined gate and the arcs of cells driven by the gates fanin cells. Referring to Figure 3.2, changing the size of the highlighted gate 6, the local cost consists of the arcs of its immediate fanin cells ($1 \rightarrow 3$, $2 \rightarrow 3$, $2 \rightarrow 4$), its immediate fanout cells ($6 \rightarrow 7$, $6 \rightarrow 8$) and the arcs of gates driven by the fanin cells of gate 6 ($3 \rightarrow 5$, $3 \rightarrow 6$, $4 \rightarrow 6$). After trying all the equivalent sizes, the size that minimizes the local cost is selected.

The iterative optimization flow stops when the maximum number of iterations is reached or when the Total Negative Slack (TNS) and total leakage power are assumed unchanged. Some timing violations may still remain, if the gate sizing exchanged some marginally positive slack to further reduce the power. The timing recovery step that follows will solve these violations resizing only specific gates that affect many timing endpoints. For these gates, only the next bigger size is tried and full incremental timing update is performed. Once the timing is closed, the final power recovery step will try to save leakage power without creating new timing violations. Again, each gate is resized only to its either next smaller size or exact higher V_T and an incremental timing update is performed after each try, to have the accurate timing information.

3.3 Incremental LR-based gate sizing

The overall effectiveness of an LM-based gate sizer is the combined result of the initialization of gate sizes, the strength of the local optimization and the appropriate update of LMs.

Initializing all cells to their minimum size simplifies the removal of any design rule violations and also may alleviate the design from timing violations because some gates are faster due to the less output load. After initialization, the total leakage power in cost function (3.2) assumes its minimum value. Thus, the sum of $\lambda_{ij} d_{ij}$ products determine which cell should be selected for each gate. This conclusion holds even if leakage and delay participate normalized to the cost function. Increasing fast the LMs of critical timing arcs guides the optimization to reduce their corresponding delay in order to minimize their $\lambda_{ij} d_{ij}$ product. As long as timing constraints are not satisfied, LMs keep increasing thus leading to cells with improved delay.

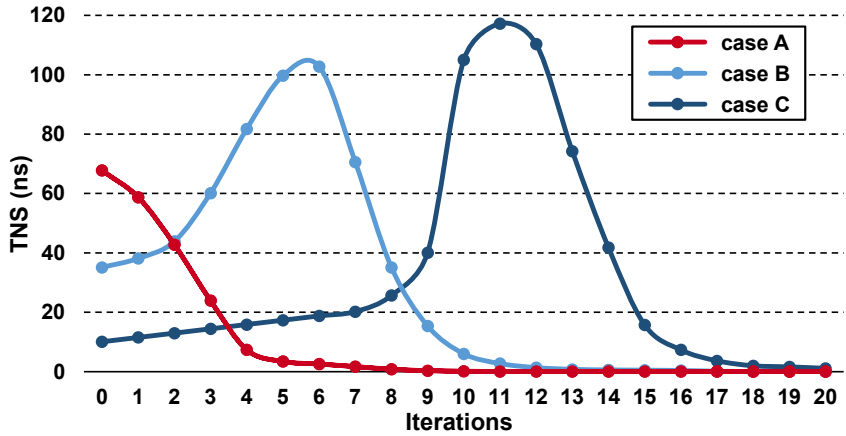


Figure 3.4: The evolution of TNS on each iteration of a LR-based gate sizer for three cases of the same design. In case A the design is not optimized. In cases B and C it is partially optimized thus exhibiting initially less TNS.

3.3.1 What is the problem?

In an incremental optimization scenario, which is the focus of this work, the first step of the state-of-the-art LR-based gate sizing flow as depicted in Figure 3.3 cannot be applied. Since the design is almost finalized, the gate sizer is not allowed to "reset" the state of the design and initialize every gate to its minimum size. Therefore, since all gates keep their already decided size the sum of leakage power in (3.2) may possibly dominate the cost function. The LMs that fit to this occasion are unknown and initializing all of them to an a priori value, e.g. 1, may not be the best choice.

Inevitably, at the first iterations of LR-based gate sizing, lower power cells would be preferred for each gate since they would minimize local cost at the expense of timing. Once timing would starting getting much worse and the corresponding LMs start to take higher values, only then the $\lambda_{ij}d_{ij}$ products would favor the selection of the delay-optimal cells. Due to improper initialization, state-of-the-art LR-based gate sizers exhibit a counterproductive behavior. The less timing critical is the initial state of the design, the more time an LR-based gate sizer would need to optimize it, when resetting the state of the design is not allowed.

To highlight this behavior we performed an experiment using the pci_bridge32_fast design of the ISPD13 benchmark set. Figure 3.4 depicts the evolution of the design's Total Negative Slack (TNS) during LR-based gate sizing for three different cases. When

the design suffers from many timing violations (case A), the LR-based gate sizer is able to find fast the way to improve timing, leading to almost closed timing after the first six iterations. The rest iterations are used to improve leakage power without degrading timing in the meantime.

On the other hand, if the design had initially less TNS (case B), LR-based gate sizer prefers to improve power by degrading timing in the first six iterations before it starts solving timing violations and achieving timing closure in iteration eleven. Similarly, if LR-based gate sizing is applied on an already optimized design with only few timing violations (case C), it will first convert many non timing critical paths to critical before actually reducing TNS to almost zero.

It is clear that regardless of the initial TNS, LR-based gate sizing is powerful enough to solve all timing violations. However, due to improper initialization, it fails to do this fast in cases that it should have. Therefore, for the case of partially optimized designs with a small set of timing violations, like case C of Figure 3.4, we need to derive an incremental version of the LR-based gate sizer that would achieve high quality-of-results and fast convergence.

3.3.2 What can we do about it?

To improve the applicability of the LR-based discrete gate sizer in an incremental optimization context, we propose an efficient method for initializing the values of the LMs so that the value of each LM is adaptive to the initial design state. In this way, the LMs are not set to an a priori chosen value but the values of the LMs would reflect the proper timing criticality of each gate relative to its already selected size, as seen near the end of the physical synthesis flow. The proposed approach is non intrusive, since it deals only with the initialization of the LMs, and can be used with any LR-based gate sizer [47, 118, 140].

Determining the initial values of the LMs should not be based solely on the criticality of the corresponding timing arcs. Assume, for instance, that the design contains a very large gate that contributes a lot to its leakage power and currently has zero timing violations. In fact, we may assume that its output pin has a small positive slack. If we assign to this gate a small initial LM due to its positive slack, we would lead the optimizer to downsize it in the first iterations to save power. This choice may seem reasonable but it fails to answer one critical question: why this gate has not been downsized earlier by the multiple optimization steps that preceded? The most probable answer is that this gate originally belonged to a set of critical timing paths. Optimizing those paths in the first steps of the flow, resulted in selecting for this gate a fast (with small delay) but large cell. Thus, any trial to reduce its size at the end of the flow would directly translate to new timing violations.

Based on this intuition, we choose to initialize the LMs following a balanced approach. We assign increased LMs to timing arcs that are either critical at the moment or belong to high-power cells assuming that those cells may have been timing critical in the past. This approach may lead to a temporary power overhead to cells that are indeed not critical but remained large for the wrong reasons (e.g., a previously applied optimization skipped them to save runtime). However, the first iterations of LR-based gate sizer would identify this by gradually reducing their corresponding LMs thus turning them to good candidates for power reduction.

The initial value for the LM of timing arc $i \rightarrow j$ is set to:

$$\lambda_{ij} = \left(\frac{a_i + d_{ij}}{a_j} \frac{P(g)}{\min P(g)} \right)^K \forall \text{ timing arc } i \rightarrow j \text{ of gate } g \tag{3.5}$$

Gate g refers to the gate where the timing arc $i \rightarrow j$ belongs. The starting value for each LM is the product of two ratios. The first ratio reveals the timing criticality of the arc $i \rightarrow j$. If the corresponding timing arc is responsible for the late arrival time at the output pin of gate j, the sum of a_i and the delay $d_{i,j}$ will be equal to a_j thus setting the ratio to 1. In any other case, a_j will be greater than the numerator and thus the ratio will result to a value less than 1 signifying the non criticality of the arc. The second ratio describes how much more power the current version of the cell spends $P(g)$ relative to the minimum possible leakage power that it can spend using any compatible library cell for gate g. In overall, when timing critical arcs are coupled with high power cells will get much greater LM values. The exponent K helps to increase faster the assigned LMs values and we empirically set it to $K = 2$.

Similarly, for the LMs that correspond to the timing arcs $i \rightarrow k$, where k is a timing endpoint:

$$\lambda_{ik} = \left(\frac{a_k}{r_k} \frac{\sum_{gates} P(g)}{\sum_{gates} \min P(g)} \right)^K \forall \text{ timing endpoint } k \tag{3.6}$$

If the signal arrives at the timing endpoint k earlier than its required time r_k, i.e., $a_k < r_k$, signaling that there is no timing violation, the first ratio will result in a value less than one. On the contrary, in cases that late timing is violated, with $a_k > r_k$, the first ratio will be as big as the actual violation. For the power ratio in the case of timing endpoints, we suggest that it should consider the design as a whole. For this reason, the power ratio that is multiplied to the the timing ratio, divides the current total leakage power of the design relative to the minimum leakage power that the design can achieve after replacing each gate with a minimum leakage power cell. This ratio actually quantifies how far the design is from its virtually minimum leakage power.

Once the LMs have been initialized, they need to be scaled in order to respect the KKT conditions as described in (3.3). Following (3.3) the sum of LMs of the output timing arcs of a gate should be equal to the timing arcs at its input. For instance, for the gate 6 shown in in Figure 2, we should guarantee that $\lambda_{67} + \lambda_{68} = \lambda_{36} + \lambda_{46}$.

To achieve this, each one of the input LMs λ_{36} and λ_{46} receive a percentage of the sum of output LMs $\lambda_{67} + \lambda_{68}$. How much of the sum of output LMs would flow to each input LM is determined by the initial values $\lambda_{36}^{\text{init}}$ and $\lambda_{46}^{\text{init}}$ of timing arcs $3 \rightarrow 6$ and $4 \rightarrow 6$, respectively.

$$\lambda_{36} = \frac{\lambda_{36}^{\text{init}}}{\lambda_{36}^{\text{init}} + \lambda_{46}^{\text{init}}} (\lambda_{67} + \lambda_{68}) \qquad \lambda_{46} = \frac{\lambda_{46}^{\text{init}}}{\lambda_{36}^{\text{init}} + \lambda_{46}^{\text{init}}} (\lambda_{67} + \lambda_{68}) \tag{3.7}$$

The initial values of $\lambda_{36}^{\text{init}}$ and $\lambda_{46}^{\text{init}}$ are derived using equation (3.5). When all gates have been visited in reverse topological order and the LMs of the timing endpoints are propagated internally, the optimization can start.

3.4 Experimental Results

The proposed method was implemented in C++ inside the open-source RSyn physical design framework [45] after extending it for multi-corner timing analysis. The evaluation involves already optimized benchmarks with only few timing violations. For this purpose, we used the fully optimized versions of the benchmarks of the ISPD 2013 gate sizing contest [116]. Those designs exhibit closed timing and minimized leakage power. To introduce additional timing violations, we randomly changed the resistance and capacitance of each net by $\pm 10\%$ thus mimicking local re-routing operation at the end of the physical synthesis flow.

Our approach is experimentally validated using the benchmarks of the ISPD 2013 gate sizing contest considering a single and a multiple-corner scenario. For the case of multiple corners, we created two artificial (but realistic) timing libraries representing the fast (timing derate 1.05) and the slow version (timing derate 0.95) of the main typical library used in the single-corner case. Each cell in timing library has 10 sizes available at 3 Vth, with a total of 30 sizes per cell.

3.4.1 Quality-of-Results and Runtime comparisons

Initially, we report the quality-of-results achieved for the proposed method (New) relative a state-of-the-art LR-gate sizer [47] (called Base) without allowing it to reset the

Table 3.1: The timing and the leakage power of all designs under single corner initially (Init) and at the end of incremental LR-based sizer without (Base) and with (New) the proposed LM initialization.

Single corner

Design	#Cells	Late WNS (ps)			Late TNS (ps)			Leakage (mW)		
		Init	Base	New	Init	Base	New	Init	Base	New
usb_phy_slow	623	-1.53	0.00	0.00	-1.53	0.00	0.00	1	1	1
usb_phy_fast		-0.61	0.00	0.00	-0.61	0.00	0.00	2	2	2
pci_bridge32_slow	30763	-11.21	0.00	0.00	-333.10	0.00	0.00	58	58	58
pci_bridge32_fast		-16.66	-0.44	0.00	-614.66	-0.96	0.00	98	97	100
fft_slow	33792	-16.35	0.00	0.00	-320.92	0.00	0.00	88	88	87
fft_fast		-18.18	-6.58	-1.88	-234.28	-63.37	-4.25	217	228	228
cordic_slow	42937	-13.99	-14.43	-1.24	-801.84	-116.70	-2.11	306	349	309
cordic_fast		-13.26	-4.26	-6.94	-752.72	-30.00	-31.40	1139	1142	933
des_perf_slow	113346	-30.40	-1.88	0.00	-11,920.00	-5.26	0.00	449	410	420
des_perf_fast		-25.80	-3.51	-4.10	-11412.20	-49.94	-8.69	609	522	556
edit_dist_slow	129227	-54.44	0.00	0.00	-21,881.50	0.00	0.00	452	447	445
edit_dist_fast		-63.59	-3.34	0.00	-36,639.50	-15.16	0.00	624	630	610
matrix_mult_slow	159642	-44.00	0.00	0.00	-3292.93	0.00	0.00	481	487	476
matrix_mult_fast		-33.07	0.00	0.00	-2694.75	0.00	0.00	1056	1230	1020
netcard_slow	984094	-30.19	0.00	0.00	-1477.58	0.00	0.00	5160	5101	5102
netcard_fast		-28.97	0.00	0.00	-6394.27	0.00	0.00	5203	5144	5141
Average		**-25.14**	**-2.15**	**-0.89**	**-6173.27**	**-17.59**	**-2.90**	**996**	**996**	**968**

state of the design. In other words, the optimization flow is the same as depicted in Figure 3.3 without performing the initial sizing step. Both cases actually utilize the same LR-based gate sizer. Their only difference is on how they initialize the value of the LMs. The obtained results are shown in Table 3.1 for single corner designs and in Table 3.2 for multi-corner designs. Columns Init correspond to the design produced after randomly perturbing the resistance and capacitance of the wires. In all cases, the optimization stops if the improvement in terms of timing and leakage power across two iterations is less than 1%. Tables 3.1 and 3.2 report the late Worst Negative Slack (WNS), the late TNS and the total leakage power of each design under single and multiple corners, respectively. The final reported timing results are validated by OpenTimer [72]. Please note that ISPD2013 benchmarks do not exhibit early timing violations and thus early timing information is omitted.

The first noticeable result is that "New" offers better timing results than "Base" in the majority of the designs. With the proposed LM initialization, WNS is further decreased by 24% on average, while TNS is improved by more than 36% on average compared to the corresponding results of "Base" with only one corner. In multi-corners designs, "New" helps improve WNS by a further 27% on average, while TNS improves by more 39%. In these cases, when timing slack reported is zero it means that timing constraints

Table 3.2: The timing and the leakage power of all designs under multiple corners initially (Init) and at the end of incremental LR-based sizer without (Base) and with (New) the proposed LM initialization.

Multiple corners

Design	Late WNS (ps)			Late TNS (ps)			Leakage (mW)		
	Init	Base	New	Init	Base	New	Init	Base	New
usb_phy_slow	-0.03	0.00	0.00	-0.03	0.00	0.00	1	1	1
usb_phy_fast	-6.38	-4.99	0.00	-14.39	-8.57	0.00	3	2	3
pci_bridge32_slow	-14.76	0.00	0.00	-485.44	0.00	0.00	60	59	59
pci_bridge32_fast	-21.40	-4.25	0.00	-280.77	-14.78	0.00	194	151	153
fft_slow	-10.74	-0.14	0.00	-194.37	-0.27	0.00	96	97	98
fft_fast	-8.21	0.00	0.00	-449.16	0.00	0.00	356	426	391
cordic_slow	-24.57	-0.68	-2.06	-1000.51	-1.09	-2.06	518	561	527
cordic_fast	-122.26	-92.07	-66.50	-5412.47	-2954.33	-1710.28	2604	3189	3220
des_perf_slow	-34.07	-29.24	-14.08	-11,391.80	-42.13	-26.27	723	704	715
des_perf_fast	-77.49	-46.25	-33.07	-19,884.50	-737.60	-216.15	1272	926	1038
edit_dist_slow	-67.66	0.00	0.00	-36,892.70	0.00	0.00	477	473	471
edit_dist_fast	-68.96	-11.22	0.00	-39,745.10	-77.41	0.00	766	791	754
matrix_mult_slow	-43.79	0.00	0.00	-3254.51	0.00	0.00	576	591	574
matrix_mult_fast	-36.16	-45.23	-33.02	-3243.41	-107.50	-41.07	1876	2357	2302
netcard_slow	-42.25	0.00	0.00	-2251.09	0.00	0.00	5163	5105	5105
netcard_fast	-28.96	-1.23	0.00	-10606.80	-2.34	0.00	5245	5187	5183
Average	**-37.98**	**-14.71**	**-9.3**	**-8444.19**	**-246.63**	**-124.74**	**1246**	**1289**	**1287**

are satisfied in all corners. In all other cases, timing refers to the negative slack of the most critical corner.

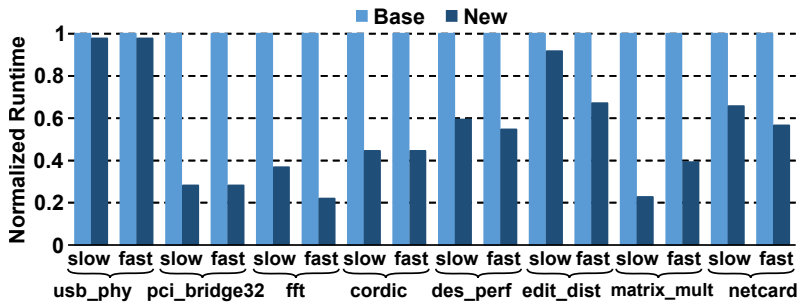


Figure 3.5: The runtime of both methods under comparison for all benchmarks when considering only one corner. The runtime is normalized to the runtime of the "Base" run. In all cases, "New" allows faster convergence saving up to 45% execution time on average in single corner benchmarks.

"New" also achieves slightly better leakage power than "Base". For fair comparison, we take into account only the leakage power savings from designs where both the "Base" and the "New" flow succeeded to resolve all timing violations. In those cases, in single corner designs "New" is 2% better on average, and 1% better on average in multi-corner designs. The reason for choosing only the timing closed designs is that whenever there are timing violations, the design's power is lower than the power of the design with closed timing.

Figure 3.5 compares the two approaches in terms of runtime when the designs have one corner. All experiments were performed on the same Linux-based workstation using a 3.6 GHz Intel Core i7-4790 with four cores and 32 GB of RAM. "New" is able to save up to 45% of runtime on average achieving also better quality-of-results. In terms of absolute runtime, the single corner "Base" finishes optimizing all designs in 9hrs, while the proposed flow needs 5hrs for the same task. The runtime of "Base" and "New" methods for designs usb_phy (slow and fast) is similar due to their small size of the designs.

Similarly, Figure 3.6, reveals the runtime savings of the proposed approach in a multi-corner timing scenario. The runtime of "New" is by 42% on average less than the average runtime of "Base". Multi-corner "Base" finishes optimizing all designs in 12hrs. When the proposed initialization method is used, the total execution time is reduced to 6hrs. The overall increased execution time of multi-corner optimization relative to the single corner scenario is due to the increased runtime of performing timing analysis on all corners.

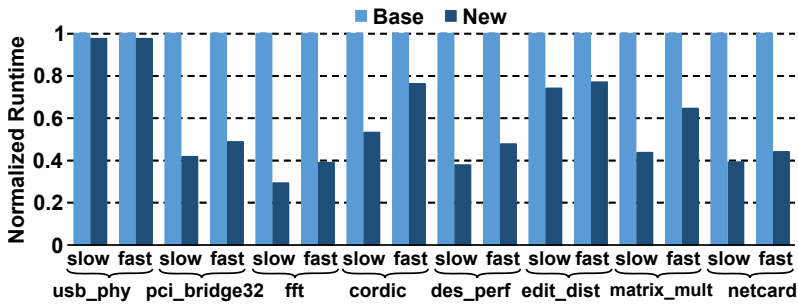


Figure 3.6: The runtime of both methods under comparison for all benchmarks when considering multiple corners. The runtime is normalized to the runtime of the "Base" run. In all cases, "New" allows faster convergence saving up to 42% on average with multiple corners.

3.4.2 Exploring in depth the proposed LM initialization

Additional experimental results reveal that the way the LMs are initialized is crucial for the fast convergence and the overall timing QoR. Figure 3.7 compares the normalized late TNS of fft_fast design with one corner for different exponents K of the proposed Equations (3.5) and (3.6). As the value of exponent K increases, higher LMs are initialized to the timing critical arcs of the design. This means that the timing improves faster with better overall QoR. For all our experiments we have selected $K = 2$ because exponent values above $K = 2$ does not improve any further the QoR.

To observe more clearly how the proposed LM initialization helps the convergence of an LR-based gate sizer, we monitor the evolution of TNS across consecutive iterations initializing the LMs to different values. For the des_perf_fast design, shown in Figure 3.8, "Base" starts degrading the timing until iteration four where the TNS reaches 80ns. From this point, the actual optimization starts and the timing closure is achieved in iteration ten. Applying the proposed Equations (3.5) and (3.6) ("New"), the optimizer starts reducing the timing violations immediately without degrading the initial state of the design and the timing constraints are met in iteration five. To further evaluate our work, we have also tried to initialize the LMs to different values where the starting value of each LM was randomly selected ("Random"). In this case, the peak of the TNS is increased compared to the "Base" run. More specifically, the TNS in

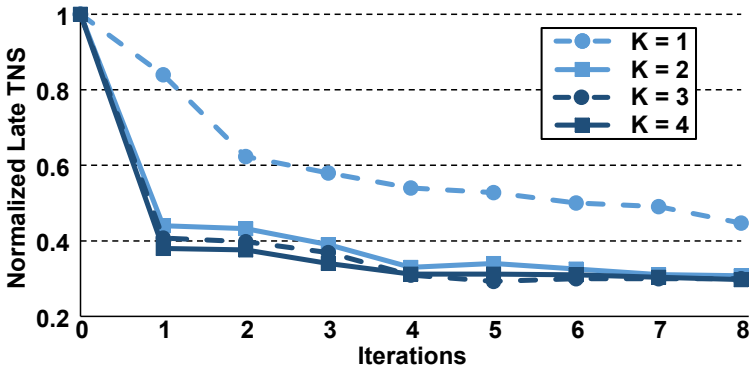


Figure 3.7: TNS comparison for different values of exponent K for LM initialization as proposed in Equations (3.5) and (3.6) for the representative design, fft_fast. Higher values of K increase the LMs of critical arcs leading to faster TNS improvement during optimization iterations. Beyond $K = 2$, there are not sufficient savings.

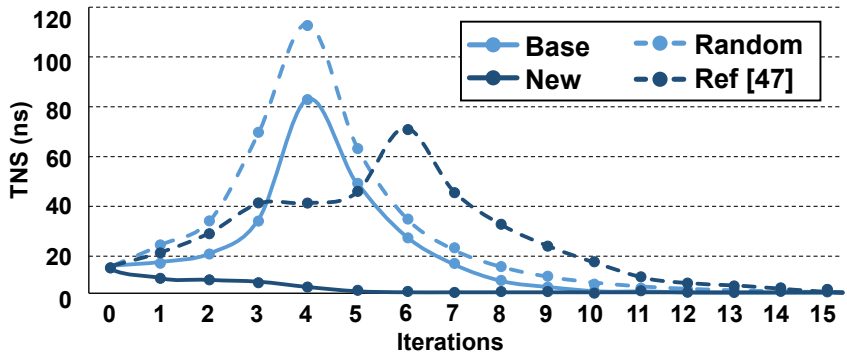


Figure 3.8: The progression of late TNS on des_perf_fast design using different LM initialization methods; initializing the LMs to 1 (Base), using the proposed method (New), initializing each LM to a random value (Random) and using the initialization of [47] (Ref [47]).

iteration four is increased from 80ns to 110ns and thus 3 more iterations were needed, compared to "Base", to close the timing. Finally, we have tested the performance of the LR-based gate sizer adopting the initialization method of work [47], in which the authors start all the LMs from 12. Even though this modification could slightly decrease the highest value of the TNS (compared to "Base"), the optimization showed slower

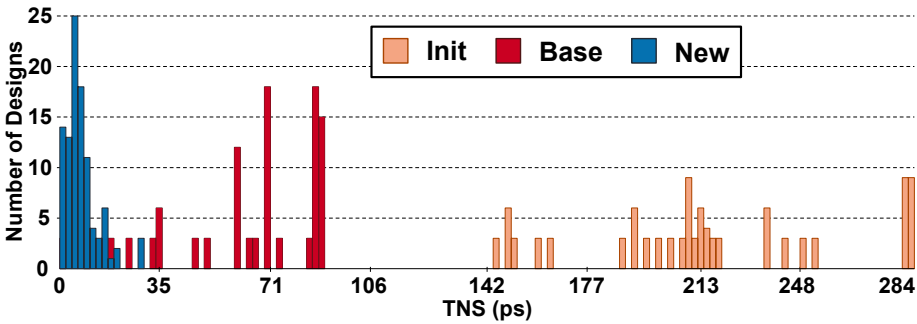


Figure 3.9: The histogram of late TNS initially (Init) and at the end of LR-based gate sizing without (Base) and with (New) the proposed LM initialization. Histograms correspond to 100 versions of fft_fast with randomly perturbed RC characteristics.

Table 3.3: The timing and the leakage power of all designs under single corner with gate size selection restriction without (Base) and with (New) the proposed LM initialization.

Single corner

Design	Late WNS (ps)		Late TNS (ps)		Leakage (mW)	
	Base	**New**	**Base**	**New**	**Base**	**New**
usb_phy_slow	0.00	0.00	0.00	0.00	1	1
usb_phy_fast	0.00	0.00	0.00	0.00	2	2
pci_bridge32_slow	0.00	0.00	0.00	0.00	58	58
pci_bridge32_fast	-1.65	0.00	-6.13	0.00	98	98
fft_slow	0.00	0.00	0.00	0.00	88	87
fft_fast	-6.87	-1.01	-20.18	-2.24	224	221
cordic_slow	-8.79	-2.96	-67.21	-2.96	378	310
cordic_fast	-17.06	-2.73	-133.10	-4.81	1209	942
des_perf_slow	-27.50	-1.40	-67.53	-4.52	480	464
des_perf_fast	-14.41	-7.61	-47.42	-23.30	637	611
edit_dist_slow	0.00	0.00	0.00	0.00	450	449
edit_dist_fast	-20.77	-1.95	-698.80	-2.16	623	619
matrix_mult_slow	0.00	0.00	0.00	0.00	478	479
matrix_mult_fast	0.00	0.00	0.00	0.00	1174	1020
netcard_slow	0.00	0.00	0.00	0.00	5152	5153
netcard_fast	0.00	0.00	0.00	0.00	5197	5194
Average	**-6.07**	**-1.10**	**-65.02**	**-2.50**	**1016**	**982**

convergence. The TNS improvement delayed to start and the timing constraints were finally met after multiple iterations, in iteration 15. From all the LM initialization trials, "New" has shown the fastest convergence of all closing the timing really soon. Similar results are obtained for all other designs. The proposed LM initialization successfully "predicts" the value of the LM that fits better to the status of the design thus avoiding un-necessary power reductions at the first iterations that would hurt timing initially and delay convergence later on.

To be certain for the quality-of-results of the proposed approach, we repeated the same experiment for each benchmark 100 times. Each time, the methods under comparison were applied on designs produced after perturbing randomly the wire parasitics of the already optimized version of each benchmark. The histogram of TNS for the initial design, and the ones produced after applying "Base" and "New" methods are depicted in Figure 3.9 for benchmark fft_fast, while similar results are obtained for all other benchmarks.

Table 3.4: The timing and the leakage power of all designs under multiple corners with gate size selection restriction without (Base) and with (New) the proposed LM initialization.

Design	Multiple corner					
	Late WNS (ps)		**Late TNS (ps)**		**Leakage (mW)**	
	Base	**New**	**Base**	**New**	**Base**	**New**
usb_phy_slow	0.00	0.00	0.00	0.00	1	1
usb_phy_fast	-12.81	0.00	-42.00	0.00	2	2
pci_bridge32_slow	0.00	0.00	0.00	0.00	62	60
pci_bridge32_fast	-22.20	-21.24	-189.58	-154.97	170	170
fft_slow	0.00	0.00	0.00	0.00	100	98
fft_fast	-22.48	0.00	-92.88	0.00	366	365
cordic_slow	-1.48	0.00	-1.86	0.00	705	516
cordic_fast	-113.97	-112.70	-5604.24	-4867.80	3325	3389
des_perf_slow	-30.88	-18.45	-207.30	-125.34	728	713
des_perf_fast	-68.24	-47.37	-1520.11	-386.81	1205	1229
edit_dist_slow	0.00	0.00	0.00	0.00	478	477
edit_dist_fast	-3.11	-0.48	-3.11	-0.85	824	758
matrix_mult_slow	0.00	0.00	0.00	0.00	602	580
matrix_mult_fast	-26.23	-27.31	-42.98	-43.54	2214	2154
netcard_slow	0.00	0.00	0.00	0.00	5172	5158
netcard_fast	-4.70	0.00	-7.60	0.00	5250	5236
Average	**-19.13**	**-14.22**	**-481.98**	**-348.71**	**1325**	**1307**

TNS histograms reveal that both approaches successfully decreased the original TNS. “Base” decreased the mean of initial TNS from 225 ps to 65 ps, while “New” managed to compress the TNS histogram to the left side of the diagram, with the majority of samples gathered close to 5 ps.

3.4.3 Optimization with a restricted number of available gate sizes

For completeness, we evaluated both “Base” and “New” methods under comparison in a more restrictive scenario. In this case, gate sizing is only allowed to resize cells only to their next bigger or smaller size without limiting V_T swapping options, since they do not alter the physical layout. This restriction makes sense at the final steps of physical design flow to preserve as much as possible the already defined detailed wire routes. The obtained results of single corner and multi-corner benchmarks are depicted in Table 3.3 and in Table 3.4, respectively. Besides the restricted availability of gate sizes, “New”

achieves considerable improvements. In single corner designs, late WNS is improved by 36% on average while the savings in TNS reach 39% on average, when compared to the baseline single corner LR-based gate sizer. In terms of leakage power, the restricted "New" method achieves less leakage power by 2% on average, when considering only the designs without negative slack at both methods under comparison. For multiple corners, late WNS is improved by 35% on average, while late TNS improves by 39% on average when compared to the corresponding timing results of "Base". Also "New" achieves slightly less leakage power by 2% on average.

3.5 Conclusions

Efficient incremental and minimally disruptive optimization steps at the end of the design flow are crucial for the overall success of automated physical synthesis. In this work, instead of relying on custom-made timing and power optimization heuristics, we leverage, LR-based optimizers used for the global optimization of the design, as fast incremental optimizers after appropriate initialization. Initialization involves selecting appropriate values for the LMs after taking into account both their timing criticality, in a multi-corner context, as well as the current size of the gates. In this way, we expedite successfully the convergence of the LR-based gate sizer, when applied in an incremental optimization context, without affecting any part of its internal functions and without reducing the achieved quality-of-results.

Experimental results have also shown that relying on constant LM initialization values as done by similar state-of-the-art optimizers or using randomly selected constants do not achieve the smooth convergence needed in the case of last-mile incremental timing optimizations. Initializing the LMs with hand selected constants provides an inaccurate picture of the design to the LR optimizer. This picture translates to un-necessary power reductions and timing degradation at the beginning of the optimization and inevitably leads to many more iterations before re-converging back to an timing optimized solution. This deficit has been corrected by the proposed approach and allows LR-based global optimizers to be successfully used as fast incremental timing optimizers.

4 Task-based Parallel Programming for Gate Sizing

4.1 Introduction

The implementation of physical synthesis algorithms should satisfy multiple contradictory goals [85]. First comes Quality-of-Results (QoR): to place and route a design that satisfies the required timing, area and power constraints [100, 104]. Then comes efficiency that should not compromise QoR: execute physical synthesis algorithms in the least amount of time even for very large designs. Last but not least is performance portability [9]: the implementation should be agnostic to the particularities of the hardware platform thus enabling physical synthesis engines to take advantage of new diverse computing hardware with the least effort for software adaptation.

Currently, the majority of the physical synthesis engines are manually parallelized with custom thread pools and work allocators developed using well-known multi-threaded programming interfaces [31,130,164]. Even if such efforts have shown good scalability, the performance starts to level off after a few active threads. There are multiple reasons for this limitation. In many cases, the algorithms are graph oriented and although they exhibit high degrees of parallelism, the patterns of parallelism involve irregular computations and possibly poor data locality that are harder to exploit than the structured parallelism patterns found in computational science [70, 114]. Also, the combined engineering effort required to describe algorithmic novelty together with multithreaded efficiency *is not trivial* and may lead to sub-optimal results. Additionally, custom thread management and scheduling may be inefficient or hard to adapt to new computing platforms.

To overcome such limitations, our goal is to *separate* algorithm development, that should focus solely at the relevant problem, e.g., placement, routing, or timing optimization, from its parallel execution. To achieve this goal, we argue that *physical synthesis algorithms* should be described as *task-based parallel programs*.

In task-based models, the programmers define elementary blocks of source code as individual tasks and express dependence relationships between those tasks [70]. In this way, the programmers do not manage processes or threads anymore but they focus only

on how to decompose their program into tasks. Contrary to other parallel programming approaches, task-based programming is easier, safer and more efficient to human programmers [9]. In this way, physical synthesis software architecture is stable and can enjoy long-term availability, ease of maintenance and high performance across current and future computing platforms.

In this work, we derive a generic template for timing optimization using task-based parallel programming and apply it to gate sizing, i.e., select for each gate an appropriate size and threshold voltage from a discrete set of library cells [85]. The same template can be used for various forms of timing optimization such as timing-driven placement [109] or logic restructuring [150]. The presented approach can be used both for global timing optimization at the first steps of the physical synthesis flow or close to the end where repairing timing violations requires incremental operations that are nondisruptive and execute as fast as possible [106].

Timing optimization algorithms are often iterative and exhibit irregular computational patterns and complex control flow [109, 118]. For this reason, we selected Taskflow for transforming the multi-step timing optimization to a task-based parallel program. Other candidates, such as Intel oneTBB FlowGraph [130], OpenMP tasks [31] were not chosen. These models require programmers to implement control flow decisions outside the task dependency graph thus creating complicated implementations that compromise parallelism [70]. Taskflow [70] offers a simpler programming interface and allows building hierarchical task graphs. Also, it supports conditional dependencies and cyclic execution patterns that have already been used in accelerating static timing analysis [69] and detailed placement [90].

In overall, this work's contributions are summarized as follows:

1. We introduce a generic task-based parallel programming template for timing optimization and test it on gate sizing algorithms. The presented approach covers all phases of a powerful Langrange-relaxation based gate sizer covering initial sizing, main iterative sizing and final recovery steps. All steps are parallelized for the first time –to the best of our knowledge– without requiring any steps executed serially and without compromising quality of results.

2. For better exploring the runtime vs quality tradeoff, we propose two heuristics that (a) re-evaluate at each iteration the search space of examined sizes per gate and (b) dynamically assess the criticality of local timing arcs. The gates with not-critical timing arcs are pruned from the local timing graph thus speeding up local timing updates.

3. Using the task-based formulation and reducing dynamically the examined sizes per gate gives a speedup of 1.7× to 2.8× when compared to state-of-the-art mul-

tithreaded gate sizers with only a marginal increase in leakage power. When enabling fast local timing updates, runtime is reduced further.

4.2 Related Work

Gate sizing has been traditionally considered as a powerful tool for timing closure and power reduction that could execute in a reasonable runtime even for very large designs [29, 64]. Approaches that used linear programming have been also proposed. In these cases, positive slack was distributed to gates using linear programming with the goal to maximize power savings [23, 113]. Then, gate sizes were selected based on the available slack. Similarly, Held *et al.* [60] assigned slew targets, instead of delay targets and achieved better results.

Langrangian Relaxation (LR) has been widely used for design optimization in recent years. Ozdal *et al.* in [118] proposed a graph model to effectively decide the sizes of the LR-based gate sizing problem. LR-based gate sizing has been refined in [47] and [143]. To resize both data and clock gates Shklover *et al.* [146] extended the traditional LR method with clock-related formulations, while the work of [109] introduced a way to optimize all types of gates (e.g. flip-flops, combinational gates and clock buffers) using the same LR formulation.

Even the most efficient algorithms required parallelism to scale to increasing design sizes. A sensitivity-guided metaheuristic method that optimizes power and timing using parallel execution was proposed in [63] and enhanced in [78]. Sharma *et al.* in [143] implemented a multithreaded gate sizer using OpenMP obtaining good QoR with fast runtimes. This approach has been tested in an industrial setup in [24]. Intel's threading building blocks [130] have been used in the most computational intensive parts of the optimization flow of [87] and [134]. The Galois parallelization framework [114] has been used in [59, 110] to speedup global and maze routing. More closely related to this work, OpenTimer [69] exploited Taskflow to accelerate efficiently static timing analysis with task-based parallelism.

Other approaches have focused on speeding up execution of EDA algorithms in GPUs. Liu *et al.* [93] use a GPU to accelerate a dynamic-programming-based gate-sizer [92]. Also, a GPU has been employed in [52] to accelerate static timing analysis. GPUs were used also for accelerating path-based timing analysis [50, 51] and placement. In the latter case, global [89] and detailed placers [90] executed on GPUs show tremendous speedups in designs with million of gates when compared to multi-threaded implementations for CPUs.

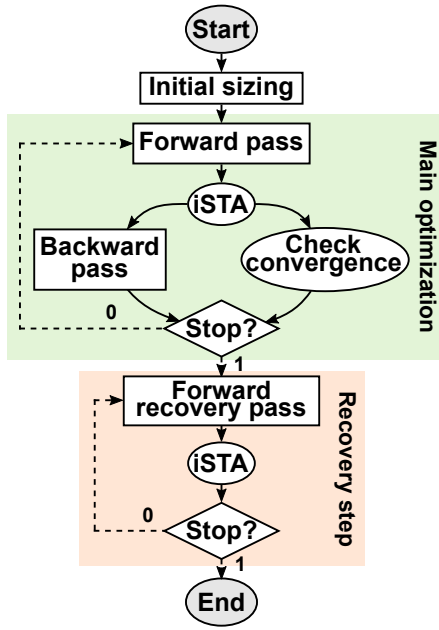


Figure 4.1: The overall task dependency graph that consists of initial sizing, the main optimization loop and the final recovery loop.

4.3 Generic Gate Sizing Template

Timing and power optimization approaches consist mainly of three core steps: initial preparatory optimization, the main iterative optimization, and final timing and power recovery. The organization of these three main steps are depicted in the top-level task graph shown in Figure 4.1. Rectangular tasks represent hierarchical blocks that can be further unwrapped to simpler tasks, round tasks are tasks executed at this level of abstraction, while diamond-shaped tasks represent conditional tasks. Conditional tasks check for a certain condition and determine accordingly the flow of execution. Solid-line edges between tasks represent dependence relations. On the contrary, dashed-line edges are conditional dependencies used in Taskflow [71] for describing cyclic processes.

Initial sizing that is executed first guarantees that gates are properly initialized so that they do not violate maximum load capacitance and maximum input slew design rules [87]. The forward pass (FP) of the main optimization loop selects appropriate sizes for each gate with the goal to minimize the selected cost function that reduces power and

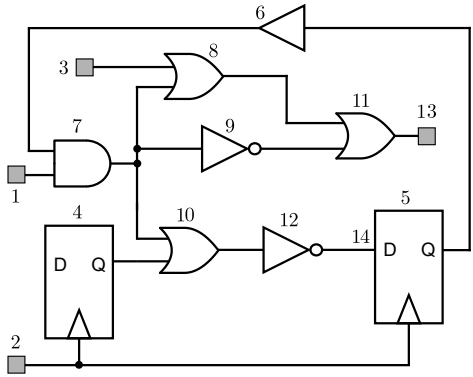


Figure 4.2: The example logic-level netlist used as a running example.

satisfies timing constraints. The result of this pass is quantified by the incremental static timing analysis (iSTA) step that follows. If power has not changed by more than 0.1% for three consecutive iterations the optimization moves to the recovery step. Otherwise, the program flow moves back to FP. This condition is checked by the convergence check task. In parallel, the backward pass (BP) updates the timing criticality of each gate using the updated timing information of iSTA.

The iterative recovery process begins only after the main optimization has converged. The goal of recovery is twofold: to correct the small remaining timing violations and recover power from gates with positive timing slack. To avoid disturbing the already optimized netlist, recovery executes carefully selected resizings with maximum timing accuracy at each step. State-of-the-art optimizers [47, 143] guarantee maximum timing accuracy by touching serially one gate at a time and updating timing globally after every update. In this work, we avoid serial execution and execute recovery in a *conservatively-parallel* way that allows for equivalent power savings but with significantly lower runtime. If timing or power improves, the recovery is repeated until all timing violations are solved and power stops changing. To assess timing improvement at the end of this recovery step a full incremental timing update takes place.

4.4 Initial sizing

The purpose of the initial sizing is to solve any load and slew violation from the beginning. In this way, it is easier for the main optimization loop that follows to preserve this property without introducing new violations.

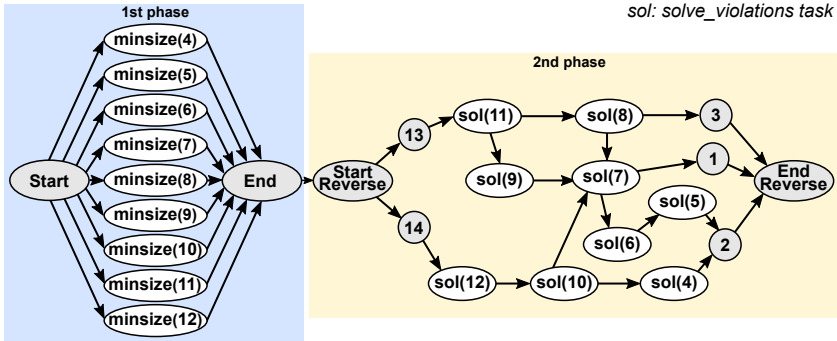


Figure 4.3: The task graph of initial sizing is organized in two phases: In the first phase all cells are downsized in parallel. In the second phase the load and slew violations are removed by visiting gates in reverse topological order.

To implement initial sizing we map its operation to a specific task graph that is organized in two parts. In the first part all gates are downsized in parallel to their lowest size (and lowest leakage). In the second part, the gates are visited in reverse topological order to check and solve if the load and the slew constraints are violated. The first part does not involve any dependencies across tasks, while in the second part the execution of tasks follows the connectivity of the logic-level netlist in reverse order.

The task graph for initial sizing that corresponds to the design of Figure 4.2 is depicted in Figure 4.3 and built using Algorithm 6. The first phase contains one task for each gate or flip-flop of the design without dependencies between them. In the second phase, there are tasks which represent primary inputs (PIs) and outputs (POs), D-pins of the flip-flops, gates as well as flip-flops (FFs). From these tasks, only the tasks that represent gates and flip-flops are assigned with work. Pseudo-tasks are created to help in dependency propagation. Such tasks are represented with grey-colored circles. Dependencies guarantee that each task is executed after the tasks of its fanouts in logic level. Introducing separate tasks for the D and clock/Q pins of the flip-flops breaks any possible sequential loop of the design thus resulting in acyclic task graphs.

The work executed at each task `solve_violations` is described in Algorithm 7. For each gate that is already at its minimum leakage size, we check whether the gate can drive its output load without introducing slew violations. The output slew is computed directly from the output slew lookup tables of the library using the already known output load and assuming the maximum-allowed slew for the inputs. If a load or a slew violation still exists the size of the gate is gradually increased until violations are removed. If none of the available sizes can solve the slew or load violations, only logic restructuring

can fix them (e.g., by adding buffers); but this problem does not occur on the ISPD 2013 benchmarks.

Algorithm 6: Create Task Graph for Initial Sizing

```
1  create_task("Start");create_task("End"); //1st phase
2  foreach g in {Gates ∪ FFs} do
3  |  create_task(g); task(g).assign_work(minsize(g));
4  |  task("Start").precede(task(g));
5  |  task(g).precede(task("End"));
6  end
7  create_task("Start-Reverse"); // 2nd phase
8  foreach po in {POs ∪ D-pins} do
9  |  create_task(po); task("Start-Reverse").precede(task(po));
10 end
11 create_task("End-Reverse");
12 foreach pi in {PIs} do
13 |  create_task(pi); task(pi).precede(task("End-Reverse"));
14 end
15 foreach g in {Gates ∪ FFs} do
16 |  create_task(g); task(g).assign_work(sol(g));
17 end
18 foreach i in {Gates ∪ PIs ∪ FFs} do
19 |  foreach f in { fanouts of i } do
20 |  |  task(f).precede(task(i));
21 |  end
22 end
23 task("End").precede(task("Start-Reverse"));
```

Algorithm 7: solve_violations(gate g)

```
1  sizes ← {equivalent sizes of g from cell library};
2  sizes_sort ← {sort sizes in ascending power order};
3  i ← 0;
4  while violates_load(g) or violates_slew(g) do
5  |  i++; resize g to sizes_sort[i];
6  end
```

4.5 Main gate sizing optimization

Design optimization targets the minimization of the total leakage power without violating any timing constraints:

$$\begin{aligned} \textbf{minimize:} \quad & \sum_{\forall \text{gate } i} leakage_i & & (4.1) \\ \textbf{subject to:} \quad & a_i + d_{ij} \leq a_j, & & \text{for each timing arc } i \rightarrow j \\ & a_k \leq r_k, & & \text{for each endpoint } k \end{aligned}$$

Variable a_i is the arrival time at pin i while r_k is the required arrival time at a primary output or a D-pin of a flip-flop k. d_{ij} is the delay of the timing arc $i \rightarrow j$ that consists of the wire delay from the output pin of gate i to the input pin of gate j plus the cell delay of gate j.

Lagrangian Relaxation associates a non-negative weight λ_{ij}, called Lagrange Multiplier (LM), to each constraint [20]. These weights act as penalty factors whenever the corresponding timing constraints are not met. Incorporating the constraints in the objective function transforms the problem to the following unconstrained one:

$$\textbf{minimize:} \sum_{\forall \text{gate } i} leakage_i + \sum_{\forall \text{arc } i \rightarrow j} (a_i + d_{ij} - a_j)\lambda_{ij} + \sum_{\forall \text{endpoint } k} (a_k - r_k)\lambda_k \qquad (4.2)$$

Differentiating (4.2) with respect to arrival times, according to the Karush-Kuhn-Tucker (KKT) optimality conditions [109, 118], we end up with the following LM conservation rule.

$$\sum_{\forall \text{fanin } i \text{ of } j} \lambda_{ij} = \sum_{\forall \text{fanout } k \text{ of } j} \lambda_{jk} \qquad (4.3)$$

Equation (4.3) implies that the sum of the LMs of the arcs ending to a gate is equal to the sum of the LMs of the arcs starting from this gate. For example, the LM flow for gate 7 of Figure 4.2 implies that $\lambda_{17} + \lambda_{67} = \lambda_{78} + \lambda_{79} + \lambda_{710}$. Replacing the equality condition of (4.3) to (4.2) simplifies the problem to:

$$\textbf{minimize:} \quad \sum_{\forall \text{gate } i} leakage_i + \sum_{\forall \text{arc } i \rightarrow j} \lambda_{ij} d_{ij} \qquad (4.4)$$

4.5.1 Forward Pass

State-of-the-art LR-based optimizers try to minimize cost function (4.4) using many iterations of gate resizing and V_T reassignment steps. At each iteration implemented by FP all gates are visited in topological order and for each gate the best size is selected assuming constant LMs.

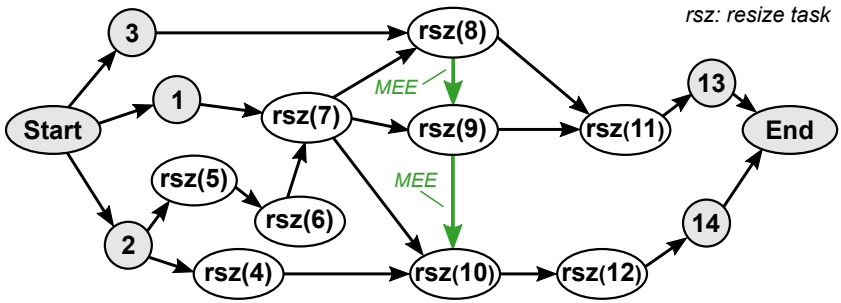


Figure 4.4: In the forward task graph, the gates are visited in topological order to find new size. Apart from the logic-level dependencies (black color), the MEEs (colored green) of [143] are added to prevent the simultaneous sizing of gates driven by the same driver, e.g. gates 8, 9 and 10.

FP Task graph

FP uses a task graph different from the one used for initial sizing. The task graph of FP that corresponds to the example of Figure 4.2 is shown in Figure 4.4 and built using Algorithm 8. Primary inputs, primary outputs, and the D-pins of the flip flops are assigned to pseudo-tasks (grey circles). A task implementing the `resize` function of Algorithm 9 is assigned to each gate and flip-flop of the design (line 9 in Algorithm 8). The dependencies across tasks follow the forward topological order of the logic-level netlist. After "Start" the pseudo-tasks of the primary input pins are visited first, while the rest dependencies follow the netlist connectivity which implies that the task of a gate precedes the tasks of its fanouts (line 14 in Algorithm 8).

According to [143] the tasks that correspond to gates that have a common fanin gate cannot execute in parallel using different threads. If their sizing is done in parallel each gate would estimate the delay change of their common fanin differently thus possibly leading to wrong sizing decisions. Also, when one of them is sized without knowing the size of the other, since it is changing in parallel, it may violate the maximum allowed capacitance of their common fanin gate. To solve this problem, the tasks that correspond to gates with a common driver should be executed serially. To impose this serial execution additional dependencies are added (lines 15–18 in Algorithm 8), called mutual exclusion edges (MEE). For instance in Figure 4.2, gate 7 drives gates 8, 9 and 10. Therefore, besides the normal dependencies $7 \rightarrow 8$, $7 \rightarrow 9$ and $7 \rightarrow 10$ that arise from the forward topological order of the netlist in Figure 4.4, two extra MEE dependencies are added between the tasks 8, 9, 10 to serialize their execution.

Algorithm 8: Create Task Graph for Forward Pass

```
1  create_task("Start"); create_task("End");
2  foreach pi in {PIs} do
3  |  create_task(pi); task("Start").precede(task(pi));
4  end
5  foreach po in {POs ∪ D-pins} do
6  |  create_task(po); task(po).precede(task("End"));
7  end
8  foreach g in {Gates∪FFs} do
9  |  create_task(g); task(g).assign_work(rsz(g));
10 end
11 foreach i of {Gates∪PIs∪FFs} do
12 |  f_i ← {fanouts of i in topological order} ;
13 |  foreach f in f_i do
14 |  |  task(i).precede(task(f));
15 |  |  next_f ← {the fanout next of f in f_i};
16 |  |  if dependency f → next_f doesn't exist then
17 |  |  |  task(f).precede(task(next_f)); // MEE
18 |  |  end
19 |  end
20 end
```

To decide to which pair of tasks we should add an MEE edge we consider the following rule.

An edge $u \rightarrow v$ is added between tasks u and v if:

(a): Tasks u and v correspond to gates that have the same driver in the logic level netlist;

(b): u is visited before v in the forward topological ordering of the netlist.

In this way, the corresponding tasks are executed serially and cyclic dependencies are avoided since an MEE is always a "forward" edge with respect to the topological order.

The task executed per gate

According to Algorithm 9, task `resize` stores first the current size of the gate and computes its local total negative slack (TNS). Local TNS corresponds to the negative slack

Algorithm 9: resize(gate g)

```
1  min_cost ← inf ;
2  best_size ← size(g) ;
3  init_slack ← local_TNS(g) ;
4  trial_sizes ← get_available_sizes(g) ;
5  foreach size s of trial_sizes do
6      resize g to s ;
7      if violates_load(g) or violates_slew(g) then
8          reject s ;
9      end
10     local_timing_update(g);
11     if local_TNS(g) < γ · init_slack then
12         reject s ;
13     end
14     cost ← leakage_g + Σ_{i→j around g} λ_ij d_ij;
15     if (cost < min_cost) then
16         min_cost ← cost;
17         best_size ← s;
18     end
19 end
20 resize g to best_size ;
21 local_timing_update(g);
```

at the output pin of the examined gate and the negative slacks at the output pins of its driving gates. Then, the cell resizing loop examines all available trial sizes and selects the one that minimizes the local cost function (4.4), without introducing load violations and without degrading the local TNS over a threshold γ [47, 151]. To compute the value of γ we use the same approach proposed in [47]: $\gamma = -\min(0, WNS)/T + 1$, where WNS is the worst negative slack of the design and T is the clock period. In this way, γ changes as the optimization evolves. The idea is to allow the local TNS to degrade a little bit for better solution space exploration, but at the same time to keep the local TNS under control. In the first few iterations, γ has a large value to allow the local TNS to have large degradation; and as timing improves, γ allows only fine-grained changes that do not disturb the already optimized design.

The local cost is calculated as the summation of the leakage power of the new size and the neighbor arc delays multiplied by their corresponding LMs. The neighbors of

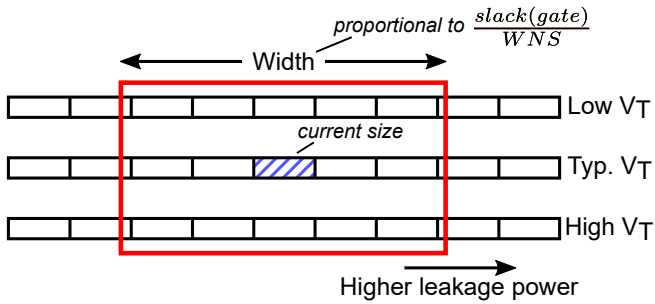


Figure 4.5: The search window is centered around the current gate size and its width changes dynamically in proportion to the gate's slack divided by WNS. The width of the search window cannot reduce below three sizes per threshold.

a gate are the ones connected at its fanin and fanout, including also the side gates, i.e., those that share a driver with the resized gate. For instance, for gate 9 in Figure 4.2 the following arcs are involved in the local cost: its input arc ($7 \rightarrow 9$), the arcs of fanin ($1 \rightarrow 7$, $6 \rightarrow 7$), the arcs of fanout ($9 \rightarrow 11$, $8 \rightarrow 11$) and the input arc of gates 8 and 10 ($3 \rightarrow 8$, $7 \rightarrow 8$, $7 \rightarrow 10$, $4 \rightarrow 10$).

In the baseline case, the available set of trial sizes examined per gate includes all possible alternatives. Also, during local timing update all neighbor arc delays are examined. In an effort to tradeoff runtime with QoR, we present two new approaches that can dynamically narrow down the set of trial sizes and examined timing arcs. Those heuristics are *only selectively enabled* and are not part of the baseline execution flow of gate sizing.

Reduce Trial Sizes (RTS)

Based on the timing slack of each examined gate, we are not obliged to examine all available sizes for each gate. For instance, Sharma *et al.* in [143] observed that by trying all available sizes during the first five iterations of the main optimization loop and using only a subset of them for the rest iterations, is enough to accelerate gate sizing with good final QoR.

In this work, the set of sizes that need to be examined are dynamically decided at each iteration and separately per gate. The smallest set of available sizes corresponds to three sizes for each V_T, i.e., the currently selected size and the immediately smaller and larger gate size, times the number of available thresholds (nine in total for the benchmarks used in the experimental results). On the contrary, the largest set corresponds to all available

sizes per V_T and all available thresholds (thirty sizes for the examined benchmarks). In all cases, the examined set of available sizes lies between those two extremes. How large is the search space depends on the negative slack of the output of the each gate: the more timing critical a gate is, the more options are tried to solve its violation fast.

If a gate has positive slack only the minimum of three sizes per V_T is tried. If the slack s is negative, the width of the search window centered around the currently selected size grows according to the ratio of $\frac{s}{WNS}$, as highlighted in Figure 4.5. Put formally the width W of the window shown in Figure 4.5 equals $W = 1 + \max(2, \#\text{sizes} \times \frac{\min(0,s)}{WNS})$.

Fast Local Timing Update (FLTU)

The local timing update calculates the new arc delays and the slews of the gates which are immediately affected after modifying a gate's size. The computed delays are used for computing the local cost function and the local TNS in Algorithm 9. To speedup this process, we dynamically alter which timing arcs are actually updated. We aim at skipping the update of timing arcs that are associated with relatively small LMs and do not affect the overall local cost. The proposed approach is detailed in Algorithm 10.

Initially, the sum of the delays of the neighboring timing arcs multiplied with their corresponding LMs is computed, which is similar to the local version of Equation (4.4) except of the leakage power term. For each neighbor, the real contribution of each timing arc is computed as the ratio of the arc's delay multiplied with the corresponding LM to the $\sum \lambda_{ij} d_{ij}$ of all neighboring arcs. When at least one of the neighbor's arcs contributes more than the threshold, the corresponding neighbor gate is not skipped. The threshold is set to $\alpha \cdot (1/\#\text{neigh\_arcs})$ where α is a non-negative weight that takes any value in between 0 and 1 and alters the number and which of the neighbors are skipped. For $\alpha = 0$, all the neighboring arcs contribute more than the threshold and thus, all neighbors are updated. On the contrary, for $\alpha = 1$ the threshold allows each arc to contribute equally to the threshold. In our experiments $\alpha = 0.5$ was used, as it minimally impacts the leakage power and provides good runtime savings.

Although a neighbor is ignored, its stale arc delays are still taking part in the local cost. If the local TNS of the examined gate is negative, all neighbors are updated to avoid timing oscillations.

The set of neighboring gates that need local timing update are not statically determined but they are dynamically re-defined for each gate in every iteration. The example in Figure 4.6 illustrates which gates in the neighborhood of gate 9 are skipped during local timing update as the optimization evolves. In every iteration, the criticality of each arc of the neighboring gates (gates 7, 8, 9, 10, 11) is compared to the threshold in which $a = 1$ for simplicity. During the third iteration, the criticality of both arcs of neighbor 10 are less than the threshold and thus gate 10 is skipped. In the next iterations, the same

Algorithm 10: local_timing_update(gate g)

```
1  sum ← 0, neigh_arcs ← 0, upd_gates ← {} ;
2  foreach gate j in get_neighbors(g) do
3  |   foreach input arc i → j do
4  |   |   sum ← sum + λ_ij d_ij; neigh_arcs++ ;
5  |   end
6  end
7  crit_thres ← α · (1/neigh_arcs) ;
8  foreach gate j in get_neighbors(g) do
9  |   foreach input arc i → j do
10 |   |   crit ← λ_ij d_ij / sum ;
11 |   |   if crit > crit_thres then
12 |   |   |   upd_gates ← upd_gates ∪ j ;
13 |   |   end
14 |   end
15 end
16 if fast_local_STA_disabled or local_TNS(g) < 0 then
17 |   upd_gates ← get_neighbors(g);
18 end
19 foreach gate j of upd_gates in topological order do
20 |   update_slews_and_arrival_times(j) ;
21 end
22 foreach gate j of upd_gates in reverse topological order do
23 |   update_required_times_and_slacks(j) ;
24 end
```

checks are performed for each arc thus possibly skipping gates 8 and 10 in the fourth and the fifth iteration, respectively.

4.5.2 Backward pass

Backward pass is responsible for updating the timing criticality of each timing arc of the design by properly updating the values of the LMs. Figure 4.7 depicts the task graph of the backward pass for the same running example. The task graph in this case includes one task for each gate, primary output and D-pin of each flip-flop, connected in reverse topological order. The task graph of the backward pass follows the same structure as the second part of the task graph of initial sizing. Therefore, to build the task graph of the

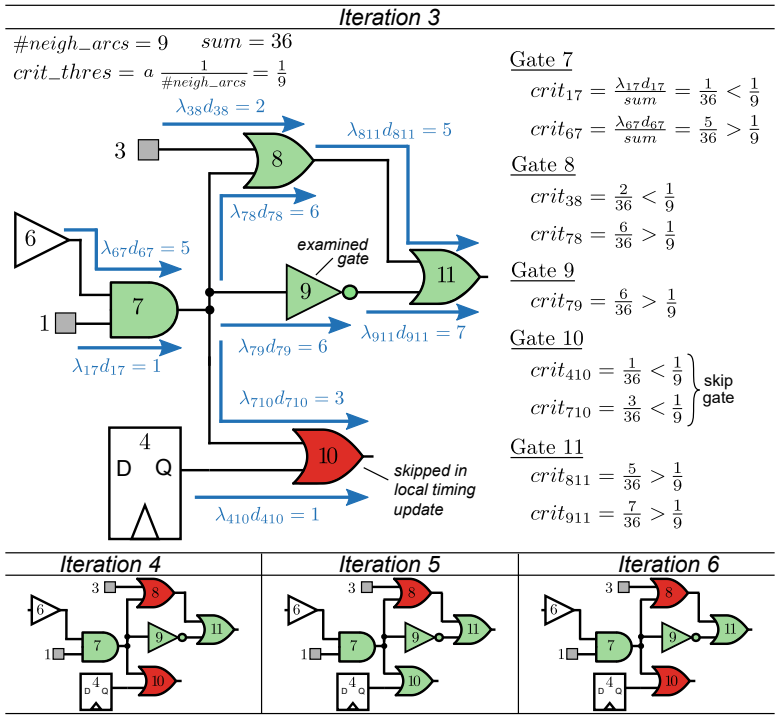


Figure 4.6: Since the criticality of timing arcs 7 → 10 and 4 → 10 of gate 10 are less than the threshold, gate 10 can be removed from the neighbors of gate 9 that participate in local timing update. This decision is dynamic and the neighbors skipped in the next iterations is re-evaluated.

backward pass, we can follow the lines 7–22 of Algorithm 6 and assign the `backward` function to each task that corresponds to a gate, flip-flop, primary output, or D-pin of a flip-flop.

During the backward pass there are no netlist changes. Thus, we can omit the addition of the MEEs which chain the fanout gates. If they were included, they would degrade the parallel performance of the backward step because they would enforce more serialization on the execution of the tasks.

The operation of LM update executed in the backward pass is described in Algorithm 11. The LMs of the input arcs are updated to reflect the timing changes due to the re-sizings of the forward pass. The LMs act as penalty factors and their values should

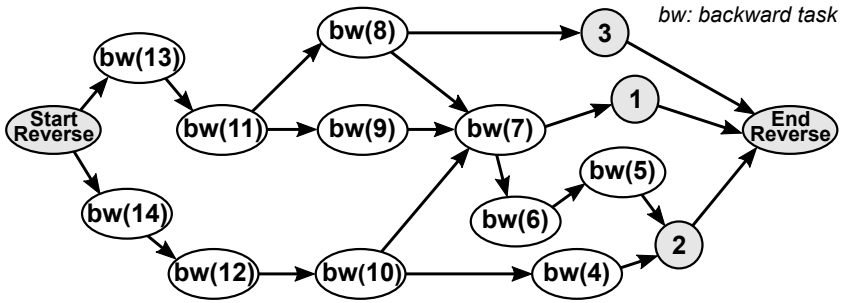


Figure 4.7: In the backward graph the tasks are visited in reverse topological order.

reflect if the timing constraints are met or not. The LMs are updated using the approach proposed in [143]. D_{ij} is the worst path delay that passes through timing arc $i \rightarrow j$ and T indicates the clock period target. Therefore, for a timing arc with negative slack $D_{ij} > T$ and thus the LM increases to reflect the violation of the timing constraint. On the other hand, when $D_{ij} < T$ makes $\frac{D_{ij}}{T} < 1$ and decreases λ_{ij}. The exponent K is used to speedup LM increments and reductions and its value is also adopted from [143]. Initially, the design has multiple timing violations and therefore the exponent for the critical arcs is set to 1 and for the non-critical arcs to 0.25. Once, the TNS becomes less than 20% of the clock target and the majority of the timing violations have been resolved, K=4 is used for the positive arcs because the respective LMs need reduction with higher rate in order to save power.

Algorithm 11: backward(gate j)

1 **foreach** *input arc* $i \rightarrow j$ **do** `// LM update`
2 $\quad \lambda_{ij}^{\text{upd}} = \lambda_{ij} \left(\frac{D_{ij}}{T}\right)^{K}$
3 **end**
4 **foreach** *input arc* $i \rightarrow j$ **do** `// LM scale`
5 $\quad \lambda_{ij} = \frac{\lambda_{ij}^{\text{upd}}}{\sum_{\forall m \rightarrow j} \lambda_{mj}^{\text{upd}}} \left(\sum_{\forall j \rightarrow k} \lambda_{jk}\right)$
6 **end**

Once the LMs of the input arcs are updated, they have to be scaled to respect the KKT condition of (4.3). The sum of the LMs of the output arcs must be equal to the sum of its input arcs LMs. To achieve this, each input LM gets a percentage of the sum of the

output LMs that is proportional to its updated value. For example, for the timing arc 6 $\rightarrow$ 7 of gate 7 (Figure 4.2) it is $\lambda_{67} = \left(\lambda_{67}^{upd} / (\lambda_{17}^{upd} + \lambda_{67}^{upd})\right) \cdot (\lambda_{78} + \lambda_{79} + \lambda_{710})$.

4.6 Timing and Power Recovery

The recovery step aims at identifying and optimizing the gates that were kept in an unoptimized state after main sizing optimization. For instance, it deals with gates that have negative slack or have remained to a high-leakage size but have positive slack to spend. This recovery step is common in many optimization algorithms and especially in those that rely on Langrangian relaxation [47, 143, 151]. In recovery, a small number of new sizes are tried per gate, each one followed by a complete incremental timing propagation to accurately reflect the timing of the affected paths. To keep the timing picture of the design as accurate as possible after each resizing, state-of-the-art sizers execute this step serially using a single thread that operates at one gate at a time.

In this work, to accelerate the execution of the recovery step, we propose its parallel execution *by allowing multiple gates to be sized in parallel but in a more conservative way*. The task graph used for the recovery step is derived from the task graph of FP after adding extra dependency edges called *timing safety edges* (TSEs) and replacing the function for each task to `recover` implemented in Algorithm 13. The addition of TSEs increases timing safety by imposing the recovery task for a gate to execute not only before its fanout but also before the fanout of its side gates.

A TSE is added between tasks u and v on top of the task graph of FP when:

(a): Task u corresponds to a gate that one of its side gates is a fanin of the gate that corresponds to task v and

(b): u is visited before v in the forward topological ordering of the netlist thus avoiding any cyclic dependencies.

The addition of TSEs on top of the task graph of the forward pass is implemented by Algorithm 12.

The forward graph for the recovery step of the example in Figure 4.2 is depicted in Figure 4.8. Gates 8, 9 and 10 share the driver 7 and thus, they are side gates. The task of gate 8 needs to add two TSE dependencies towards the tasks of the fanouts of its side gates 9 and 10, i.e. 11 and 12. But gate 11 is also fanout of gate 8 and therefore a logic-level dependency already exists between them. Therefore, as shown in Figure 4.8, only one TSE dependency is added from the task of gate 8 towards 12. Similarly, the TSE dependencies from task 9 towards 12 and from task 10 towards 11 are added. Even though TSEs enforce more serial execution, they are essential for the timing accuracy

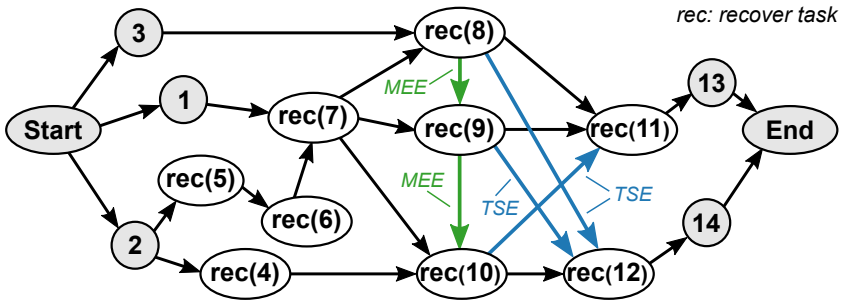


Figure 4.8: The recovery task graph is built on top of the task graph of FP. TSE dependencies are added to eliminate timing inaccuracies. The TSE are added from a gate towards the fanouts of its side gates.

needed by the resizings of this step. For instance, assume the case of tasks 10 and 11 in Figure 4.8. If they were not connected by a TSE it means that they could have executed recovery in parallel. If this was allowed, each task would have affected differently the timing of gate 9 since the latter is a side gate for 10 and a fanin gate for 11. Since the delay of gate 9 affects the local TNS of both gates 10 and 11, resizing them in parallel would have been highly inaccurate for the sensitive recovery step.

Please notice that this inaccuracy in timing does not affect the convergence of the main optimization loop. The main optimization loop is an iterative process that relies on the minimization of the sum of the arc delays multiplied with their corresponding LMs (cost function (4.4)). The LMs are updated gradually during every BP and thus

Algorithm 12: Add Timing Safety Edges

```
1 foreach g in {Gates} do
2     sides ← {side gates of g};
3     sides_f ← {fanouts of sides};
4     foreach f of sides_f do
5         if g seen before f in topological order and dependency g → f doesn't
            exist then
6             task(g).precede(task(f));
7         end
8     end
9 end
```

Algorithm 13: recover(gate g)

```
1  if local_TNS(g) > 0 then // Power recovery
2      resize g to {next increased V_T size of g's initial size};
3      if reject_size(g) then
4          resize g to {next smaller size of g's initial size};
5          if reject_size(g) then
6              resize g to it's initial size;
7          end
8      end
9  else // Timing recovery
10     resize g to {next bigger size of g's initial size};
11     if reject_size(g) then
12         resize g to it's initial size;
13     end
14 end
```

keep historic information with respect to the criticality of the corresponding timing arc. For instance, an arc that was critical for multiple iterations it keeps a high value even if its timing is improved. Therefore, the LMs work as safeguard and any wrong decision due to timing inaccuracy, is fixed in the next iteration.

In contrast, in the recovery step the optimization does not depend on a joined product with LMs, but the local decisions are strictly based on the actual slacks as returned from the timer. Therefore, even the small timing inaccuracies which can harm the convergence of the recovery, must be avoided.

The assigned work to the tasks in the recovery step is described in Algorithm 13. For each visited gate, at first the local TNS is computed and its sign defines whether the gate is sized to save power or to solve timing violations. Therefore, a gate with positive local TNS performs power recovery while a gate with timing violations performs timing recovery.

For power reduction, only two sizes are tried one after the other; the next V_T and the smaller size. After each resize, the timing is updated locally considering all the neighboring gates. The first option that does not violate the design rules and does not degrade the local TNS, is kept. For timing reduction, the gate is sized only to the next bigger size to improve timing. If this size creates any violation of the design rule constraints or after a local timing update it worsens the local TNS, the gate is resized back to its initial size.

4.7 Experimental Results

The proposed approach was implemented in C++ using the RSyn framework [45] that provides the essential functions for netlist traversal and update as well as timing analysis. The creation and the execution of task graphs was performed with Taskflow [71]. All experiments ran on the same CentOS workstation equiped with 128 GB RAM and two Intel Xeon Silver 4214 @ 2.20GHz CPUs with twelve 2-way multithreaded processors each. In all cases, we used the benchmarks of the ISPD 2013 contest [116] and the final timing results are validated with OpenTimer [69]. The final results obtained from the proposed approach meet all the timing, as well as, the load and slew constraints.

4.7.1 The characteristics of the tasks graphs

Before comparing the proposed approach to the state-of-the-art, it would be useful to examine the characteristics of the ISPD13 benchmarks used in the evaluation and how they affect the size and structure of the corresponding task graphs. The characteristics of the task graph depend solely on the structural properties of each design and not on the timing constraints ('slow' or 'fast') associated with each benchmark.

Table 4.1 presents the number of gates of each design together with the properties of each task graph in each case. The number of nodes for all types of tasks graphs are linearly dependent on the number of gates and flip-flops of the design as well as on the number of primary inputs and outputs. The number of simple edges follows the connectivity of the netlist of each design being linearly dependent on the number of design's nets. On the contrary, the MEEs and TSEs added in the fordward pass and in recovery, respectively, depend on the fanout of certain nets of the design. The higher the fanout per net, the more the MEEs and TSEs added.

TSEs are used to maximize timing safety by imposing a more restrictive execution order to certain tasks of the recovery phase. In all cases, the number of TSEs exceed by far the number of simple edges and MEEs. We expect this characteristic to translate to less structural parallelism in the recovery task graph relative to the task graph of the forward pass.

4.7.2 Comparison with state-of-the-art

Initially, we would like to compare the proposed task-based approach with state-of-the-art gate sizers [47] and [143]. The leakage power of state-of-the-art methods are the final results reported in [47, 143] after finishing both main sizing optimization and their corresponding timing/power recovery steps. The runtimes reported [47, 143] are taken verbatim from the respective papers.

Table 4.1: The size of the designs and the properties of the corresponding task graphs (in thousands).

Designs	Gates	Task Graph Properties	Initial Sizing	Forward Pass	Backward Pass	Recovery
usb_phy	0.6	Nodes	1.4	0.7	0.7	0.7
		Edges	2.5	1.3	1.3	1.3
		MEEs	-	0.4	-	0.4
		TSEs	-	-	-	2.7
pci_bridge32	30.8	Nodes	64.9	34.3	34.3	34.3
		Edges	122.0	60.8	60.8	60.8
		MEEs	-	21.8	-	21.8
		TSEs	-	-	-	245.8
fft	33.8	Nodes	70.5	37.8	37.8	37.8
		Edges	142.1	76.6	76.6	76.6
		MEEs	-	27.1	-	27.1
		TSEs	-	-	-	181.6
cordic	42.9	Nodes	87.1	44.2	44.2	44.2
		Edges	167.3	81.5	81.5	81.5
		MEEs	-	28.9	-	28.9
		TSEs	-	-	-	235.9
des_perf	113.3	Nodes	235.4	122.3	122.3	122.3
		Edges	442.1	215.9	215.9	215.9
		MEEs	-	80.2	-	80.2
		TSEs	-	-	-	604.9
edit_dist	129.2	Nodes	261.6	134.9	134.9	134.9
		Edges	506.9	252.6	252.6	252.6
		MEEs	-	95.0	-	95.0
		TSEs	-	-	-	732.3
matrix_mult	159.6	Nodes	320.6	164.1	164.1	164.1
		Edges	620.9	301.2	301.2	301.2
		MEEs	-	110.9	-	110.9
		TSEs	-	-	-	775.6
netcard	984.1	Nodes	2064.2	1081.9	1081.9	1081.9
		Edges	3952.5	1987.9	1987.9	1987.9
		MEEs	-	761.9	-	761.9
		TSEs	-	-	-	10785.5

The runtime results of [47] correspond to single-thread implementations run on an Intel i7-3770 @ 3.40GH, while the runtime results of [143] refer to a mixed multi-threaded and single-thread implementation executed on a system with two quad-core Intel Xeon E3-1240 v5 @ 3.50GHz CPUs and 16GBs of memory. In [143] the main

Table 4.2: The leakage power and runtime of [47] compared to the proposed task-based gate sizing.

Design	Leakage Power (W)		Runtime (min)		Speedup
	[47]	Ours 1 thread	[47]	Ours 1 thread	
usb_phy_slow	0.001	0.001	0.49	0.04	12.25
usb_phy_fast	0.002	0.002	0.42	0.09	4.67
pci_bridge32_slow	0.057	0.058	10.53	3.39	3.11
pci_bridge32_fast	0.085	0.096	22.62	7.06	3.20
fft_slow	0.087	0.088	25.71	6.74	3.81
fft_fast	0.194	0.214	40.43	12.50	3.23
cordic_slow	0.271	0.292	69.04	19.70	3.50
cordic_fast	1.001	1.013	117.08	30.14	3.88
des_perf_slow	0.330	0.342	132.27	23.20	5.70
des_perf_fast	0.649	0.654	347.87	35.94	9.68
edit_dist_slow	0.425	0.451	123.90	21.60	5.74
edit_dist_fast	0.540	0.571	352.96	36.34	9.71
matrix_mult_slow	0.444	0.469	226.13	38.04	5.94
matrix_mult_fast	1.611	1.673	395.96	60.54	6.54
netcard_slow	5.155	5.156	483.55	106.22	4.55
netcard_fast	5.200	5.202	400.89	148.56	2.70
Total	16.050	16.280	2749.85	550.10	-
Geomean	0.220	0.230	57.53	11.57	4.97

optimization step was executed using 8-threads described with OpenMP and the initial sizing as well as the final recovery step ran serially using only one thread. In both [47, 143], the final recovery step is executed on purpose on a single thread to guarantee the maximum timing accuracy.

Table 4.2 highlights the performance of [47] relative to the baseline task-based formulation of the gate sizing problem, without enabling RTS or FLTU that dynamically reduce sizing alternatives and speedup local timing updates. All steps of the proposed gate sizer were also executed on a single thread. The proposed flow achieves similar leakage power results and significant runtime savings when compared to [47]. For instance, the single-threaded execution of the proposed approach achieves 4.97$\times$ speedup at the cost of 5% higher leakage power as reported by the geometric mean average of leakage power and speedup per benchmark, respectively. Geometric mean average is used to facilitate data averaging with a wide range in values. The reduced execution time of the proposed approach is a result of the smaller number of iterations of the

Table 4.3: The leakage power and runtime of [143] compared to the proposed task-based gate sizing.

Designs	Leakage Power (W)			Runtime (min)							
	[143]	Ours		[143]		Ours - Base		Ours with RTS enabled			
		Base	w. RTS	Orig.	Trimmed	8 thr.	24 thr.	8 thr.	Speedup vs Trimmed	24 thr.	Speedup vs Trimmed
usb_phy_slow	0.001	0.001	0.001	0.22	0.05	0.01	0.01	0.01	5.00	0.01	5.00
usb_phy_fast	0.002	0.002	0.002	0.23	0.06	0.02	0.05	0.01	6.00	0.01	6.00
pci_bridge32_slow	0.058	0.058	0.058	0.97	0.40	0.55	0.27	0.37	1.08	0.22	1.82
pci_bridge32_fast	0.090	0.095	0.100	1.54	0.97	1.21	0.53	0.79	1.23	0.40	2.43
fft_slow	0.088	0.087	0.089	1.37	0.73	1.20	0.59	0.73	1.00	0.42	1.74
fft_fast	0.213	0.219	0.230	1.64	1.01	1.51	0.92	1.06	0.95	0.70	1.44
cordic_slow	0.293	0.299	0.338	2.29	1.56	2.62	1.48	1.62	0.96	1.02	1.53
cordic_fast	1.080	1.025	1.100	5.60	4.88	4.77	2.21	2.92	1.67	1.46	3.34
des_perf_slow	0.332	0.339	0.348	7.27	5.82	3.59	1.73	2.56	2.27	1.37	4.25
des_perf_fast	0.639	0.653	0.657	26.16	24.70	5.83	2.33	4.31	5.73	1.71	14.44
edit_dist_slow	0.440	0.451	0.453	4.92	3.15	3.49	1.74	2.24	1.41	1.30	2.42
edit_dist_fast	0.549	0.573	0.587	6.66	4.90	5.99	2.73	4.07	1.20	2.23	2.20
matrix_mult_slow	0.448	0.468	0.475	8.80	6.76	7.76	3.26	4.51	1.50	2.64	2.56
matrix_mult_fast	1.633	1.672	1.680	13.94	11.82	10.97	5.10	6.12	1.93	2.83	4.18
netcard_slow	5.170	5.156	5.155	24.67	12.43	18.91	9.78	11.29	1.10	7.74	1.61
netcard_fast	5.205	5.202	5.200	30.60	18.36	24.81	12.10	13.79	1.33	8.93	2.06
Total	16.241	16.300	16.473	136.88	97.60	93.24	44.83	56.40	-	32.99	-
Geomean	0.230	0.233	0.239	3.70	2.18	1.97	1.10	1.27	1.72	0.76	2.86

main optimization step and the replacement of the full-incremental timing update in the recovery step of [47] with a local timing update.

Similarly, Table 4.3 compares the proposed approach relative to the results reported in [143] for an eight-thread execution. For the proposed sizer we consider the baseline approach that allows each task to examine all possible gate sizes and the one that enables RTS. RTS alters dynamically the number of sizes that are tried for each gate based on the gate's timing slack. Restricting the search space may slow down the convergence of the main optimization since more iterations are needed to find the most suitable size. However, each iteration is faster.

The leakage power achieved in each case is depicted in columns 2–4 of Table 4.3. The two variants of the proposed approach have less than 4% higher leakage power on average than [143]. As expected, the solutions with RTS enabled have higher leakage relative to the baseline approach. This holds for all benchmarks except 'netcard'. In this case, RTS performs marginally better. This result is an artifact caused by multithreading. Every time we run the same experiment with the same input, the final results can be slightly different. Nevertheless, the difference observed in all cases is always at the granularity of the third decimal digit.

The runtime of [143] is shown in columns 5–6 of Table 4.3. Column 'Orig.' refers

to the total runtime reported in [143] for each benchmark. This runtime involves also the time needed for timing-model calibration that is done using an external timer. To have a fair comparison the overhead of communicating with the external timer should be removed from the comparisons. According to [143] the useful runtime for timing calibrations that does not involve TCL and file processing is roughly 20% of the total runtime spent for timing calibrations. To compute the time that should be removed for each benchmark, we used the contribution in runtime of timing calibrations reported in Table V of [143]. The trimmed runtime derived for each benchmark is shown in column 'Trimmed'. In benchmarks that timing calibration was a large part of the overall runtime, the runtime reductions are significant. For instance, for netcard_fast the total runtime reduced from 30.6 to 18.3 minutes.

The runtime of the two variants of the proposed method for 8 and 24 threads and the speedup achieved relative to the trimmed runtime of [143] are depicted in the last six columns of Table 4.3. The baseline version of the proposed approach shows a marginal reduction in the total runtime needed to execute all benchmarks using 8 threads but improves significantly for 24 threads.

The work of [143], after the first five iterations of the LR sizing loop, examines a subset of the available sizes for each gate. This has a significant impact on the overall runtime. Therefore, to have a *fair* comparison with the proposed work we need to compare the trimmed runtimes of [143] with the runtimes achieved by the proposed

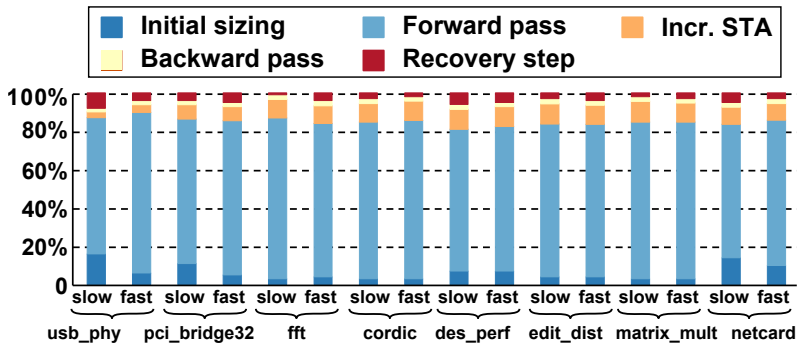


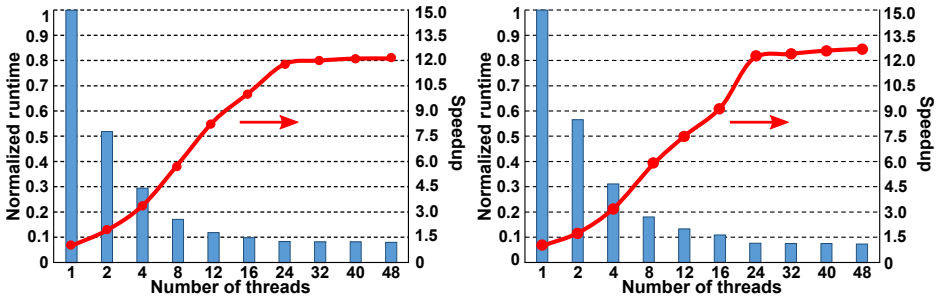
Figure 4.9: The breakdown of the total runtime for all benchmarks. The parallel initial sizing uses the 6% on average of the total runtime. The majority of the time is consumed in the forward pass of the main optimization step and only 9% in the single-threaded incremental timing analysis. The backward pass needs 2% on average while the recovery step utilizes the 4% on average.

method with the RTS heuristic enabled. Under this *apple-to-apple* comparison, i.e., both approaches employ multithreading under an equal number of threads and both use a heuristic that examines only a subset of the available gate sizes, the proposed method achieves a mean speedup improvement of 1.72× for eight threads that improves to 2.86× for 24 threads.

The runtime reduction reported is the combined result of two factors. First, the proposed approach executes in parallel all steps of the optimization including initial sizing, main optimization and final recovery while the method of [143] performs single-threaded initial sizing and recovery. Also, our thread scheduling is not performed manually, as in [143], but done automatically by Taskflow that allows scaling the gate sizer smoothly to a higher number of threads.

Figure 4.9 highlights the contribution in runtime of each step of the proposed task-based parallel gate sizer assuming eight available threads. The FP of the main optimization loop utilizes on overage the 79% of the total runtime while BP takes only 2% of the total runtime. Initial sizing consumes 6% of the total runtime on average, while the parallel implementation of the iterative recovery step reduced its runtime contribution to only 4% on average. Incremental timing analysis accounts for 9% of the total execution time, on average. In our implementation the incremental timing analysis is performed using only one thread. Therefore, a multi-threaded implementation of the timing analysis can further reduce the amount of time consumed by this step.

The scalability of the proposed approach with increasing the number of threads for the two largest designs matrix_mult_fast and netcard_fast is shown in Figure 4.10. The ex-



(a): matrix_mult_fast (b): netcard_fast

Figure 4.10: The normalized runtime and speedup of the proposed approach for increasing number of threads for the two largest benchmarks. The runtime decreases sufficiently until thread count reaches 24 that matches the amount of physical CPUs used in this experiment.

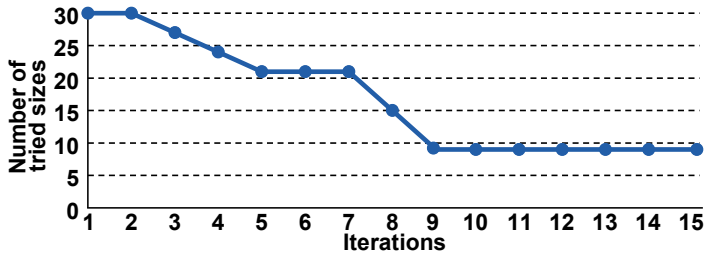


Figure 4.11: The number of sizes that are examined in gate sizing as the optimization evolves for a specific gate of matrix_mult_fast design. At first, the gate has large negative slack and thus all the available sizes are tried. As slack improves the number of examined sizes is reduced to minimum.

ecution time is normalized to the runtime of the single-thread run. Moving from one to two threads speeds the execution of matrix_mult_fast by 1.9× and 1.7× for netcard_fast. Enabling four threads gives an additional speedup of 70% on average for both designs. Beyond 24 threads, there is little reduction in runtime. This is caused by the dependencies of the task graphs that limit the parallelism that can be achieved and, at the same time, with 24 threads we reach the maximum number of physical CPUs.

4.7.3 Highlighting the contribution of RTS and FLTU

The results of Table 4.3 have shown the effectiveness of RTS in reducing the overall runtime. In this section we want to clarify the behavior of RTS and also highlight the contribution of FLTU that reduces dynamically the number of timing arcs included at each local timing update.

To clarify the dynamic behavior of RTS, Figure 4.11 depicts the total number of sizes that are tried in each iteration for a specific gate in matrix_mult_fast benchmark. In the first iteration, the gate is part of the most critical path and therefore all the available sizes are examined. In the next iteration, even though the gates slack and WNS are improved, the gate remains in the critical path and thus again all sizes are examined. As the optimization evolves the number of examined sizes decreases because the ratio of the gates slack to WNS is getting smaller. Once the gate obtains positive slack (iteration 9) the examined sizes are fixed to nine options: three sizes per V_T for three available V_T.

The combined effect of RTS and FLTU relative to gate sizing with only RTS enabled is shown in Table 4.4. In all cases, the task graphs are executed using 24 threads. Enabling FLTU in addition to RTS offers an additional 1.12× speedup on average due to

Table 4.4: The leakage power and the runtime of the proposed flow with only RTS (RTS) and with RTS and FLTU (RTS & FLTU) enabled.

Design	Leakage Power (W)		Total Runtime (min)		Speedup
	RTS	RTS & FLTU	RTS	RTS & FLTU	
usb_phy_slow	0.001	0.001	0.01	0.01	1.00
usb_phy_fast	0.002	0.002	0.01	0.01	1.00
pci_bridge32_slow	0.058	0.058	0.22	0.19	1.16
pci_bridge32_fast	0.100	0.107	0.40	0.36	1.11
fft_slow	0.089	0.089	0.42	0.37	1.14
fft_fast	0.230	0.235	0.70	0.62	1.13
cordic_slow	0.338	0.339	1.02	0.83	1.23
cordic_fast	1.100	1.121	1.46	1.29	1.13
des_perf_slow	0.348	0.349	1.37	1.24	1.10
des_perf_fast	0.657	0.662	1.71	1.39	1.23
edit_dist_slow	0.453	0.455	1.30	1.23	1.06
edit_dist_fast	0.587	0.593	2.23	1.94	1.15
matrix_mult_slow	0.475	0.480	2.64	2.17	1.22
matrix_mult_fast	1.680	1.684	2.83	2.41	1.17
netcard_slow	5.155	5.156	7.74	7.25	1.07
netcard_fast	5.200	5.202	8.93	8.27	1.08
Total	16.473	16.533	32.99	29.58	-
Geomean	0.239	0.241	0.76	0.68	1.12

the simplification of the local timing updates at the cost of less than 1% higher leakage power.

To understand better the application of FLTU, we show in Figure 4.12 the average percentage of the neighbors that are updated during the local timing update at each iteration. The design used in this example is edit_dist_slow. Initially, almost all neighbors are updated because the design starts with many timing violations distributed across all paths. During the next two iterations, the timing improvement is sufficient and thus there are many neighbors for which the timing update locally is not essential. More specifically, in the third iteration only the 72% of the neighbors are considered for local timing update. From this point forward more and more neighbor timing arcs stop contributing to the local cost and therefore can be ignored. Similar results are obtained for all benchmarks.

We should not forget that excluding timing arcs from the local timing update of each gate inevitably keeps their delay unchanged. When those timing arcs participate in the

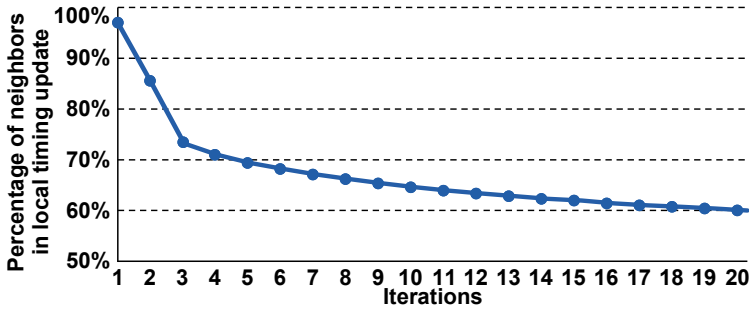


Figure 4.12: The average percentage of the neighbors which are updated when fast local timing update is enabled in edit_dist_slow. Initially, all neighbors are considered for update but during the next two iterations, the number of neighbors considered is significantly reduced.

local cost function with their not-updated (stale) delays may lead to sub-optimal local gate sizing choices. The effect of such choices, translates to designs with increased leakage power (2% more leakage is observed relative to the baseline approach).

The criticality threshold is fundamental in the FLTU because it defines how many and which of the neighbors are skipped from the timing update and therefore can affect the overall QoR. For example, a low value for the threshold implies that the majority of the neighbors are updated, while a high value reduces the neighboring gates to update. As the number of skipped neighbors increases, the runtime reduces but the leakage is degraded. For this reason, to better understand how the criticality threshold affects the overall QoR, we modify its value by increasing the weight α that is part of the threshold. To better evaluate the obtained final results, a loss function is defined which takes into account both the runtime and the power changes. More specifically, the loss for a specific α, $Loss_\alpha$, should be considered minimal if the leakage is not appreciably increased and the runtime is significantly reduced when compared to the corresponding results obtained with $a = 0$ i.e. none of the neighbors is skipped.

$$Loss_\alpha = \frac{leakage_\alpha}{leakage_0} \cdot \frac{runtime_\alpha}{runtime_0}$$

The overall loss for increasing the value of weight α in edit_dist_slow is illustrated in Figure 4.13. As the weight α increases, the threshold also increases and therefore less neighbors are considered for the timing update. Until $\alpha = 0.7$, the overall loss is lower than with $\alpha = 0$, because the runtime is reduced and simultaneously the leakage is marginally increased. For $\alpha \geq 0.8$ more neighbors are skipped from the timing update

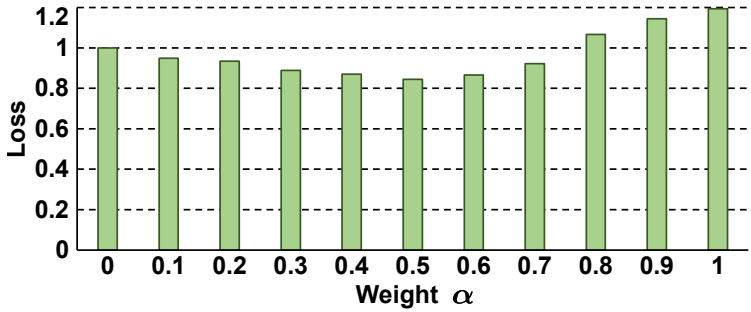


Figure 4.13: The loss for different values of weight α in the criticality threshold in edit_dist_slow. The minimum loss observed in 0.5. Beyond $\alpha = 0.8$ the neighbors which are skipped affect the timing accuracy significantly and therefore more iterations are needed to converge.

and thus even less loss is expected. However, beyond this point, the increased number of skipped neighbors affects negatively the timing accuracy. This leads to sub-optimal solutions with higher leakage power and that causes the gate sizing to run for more iterations. The same behavior is observed in all designs. Therefore, in our experiments $\alpha = 0.5$ was used, as it minimally impacts the leakage power and provides good runtime savings.

4.7.4 The contribution of final timing and power recovery

All three methods under comparison are using a Langrangian Relaxation based formulation for the main optimization step that is accompanied by a timing and power recovery step at the end. This step is crucial in correcting the remaining small timing violations and recovering part of the excessive leakage power. To enable surgical-accuracy resizing decisions, the methods of [47, 143] require examinining serially one gate at a time (from a limited set of gates) and performing a full incremental timing update after each change. This requirement increases inevitably the runtime of the recovery step and limits the applicability of LR-based gate sizers. This limitation is effectively removed by the proposed approach.

Table 4.5 reports in columns 2–4 the leakage power at the end of the main optimization step for all methods under comparison irrespective of the runtime needed to finish the main optimization. The results show that more or less all three methods converge to similar leakage power. If we observe the results more carefully, we see that the method of [47] or the method of [143] achieve the lowest leakage power after main optimiza-

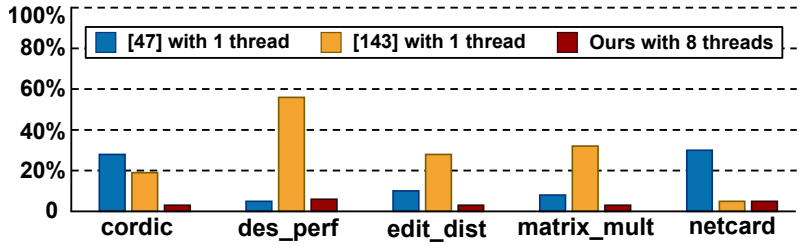
Table 4.5: Leakage power before recovery and the incremental change of power (ΔPower) after recovery.

Design	Leakage Power (W) before recovery			ΔPower (%) after recovery		
	[47]	[143]	Ours	[47]	[143]	Ours
usb_phy_slow	**0.001**	**0.001**	**0.001**	0.0	0.0	0.0
usb_phy_fast	**0.002**	**0.002**	**0.002**	0.0	0.0	0.0
pci_bridge32_slow	**0.057**	**0.057**	0.058	0.0	1.8	0.0
pci_bridge32_fast	**0.088**	**0.088**	0.093	-3.4	2.3	2.2
fft_slow	**0.087**	0.088	0.089	0.0	0.0	-2.2
fft_fast	**0.204**	0.209	0.222	-4.9	1.9	-1.4
cordic_slow	0.309	**0.296**	0.303	-12.3	-1.0	-1.3
cordic_fast	1.665	**1.273**	1.289	-39.9	-15.2	-20.5
des_perf_slow	0.339	**0.328**	0.336	-2.7	1.2	0.9
des_perf_fast	0.750	**0.648**	0.668	-13.5	-1.4	-2.2
edit_dist_slow	**0.429**	0.439	0.454	-0.9	0.2	-0.7
edit_dist_fast	0.573	**0.551**	0.572	-5.8	-0.4	0.2
matrix_mult_slow	0.463	**0.454**	0.485	-4.1	-1.3	-3.5
matrix_mult_fast	2.032	**1.859**	1.894	-20.7	-12.2	-11.7
netcard_slow	**5.117**	5.169	5.156	0.7	0.0	0.0
netcard_fast	**5.148**	5.195	5.205	1.0	0.2	-0.1

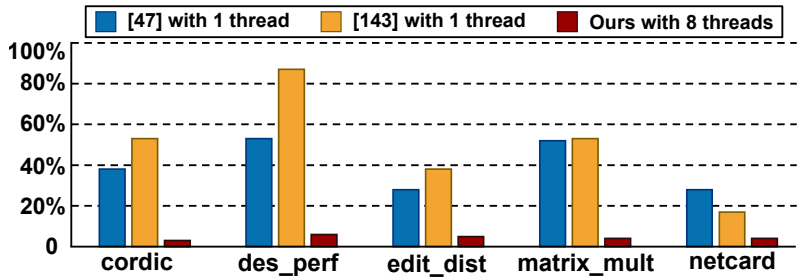
tion. Therefore, the proposed approach has to recover slightly more leakage power than the rest.

Also, columns 5–7 of Table 4.5 depict the reduction in leakage power achieved after recovery (timing is closed in all cases). Even if each method behaves differently during recovery, the overall trend per design remains the same for all methods. For instance, in netcard and edit_dist the recovery step fails to reduce the leakage power. On the contrary, the reductions observed in cordic and matrix_mult are significant.

Even in cases that didn't benefit a lot from timing and power recovery, the runtime spent is not negligible. Figure 4.14 illustrates the percentage of the total runtime consumed by the recovery step in the five largest designs. For each design, we include the percentage of the single-threaded recovery step of [47, 143] and the percentage of the proposed *conservatively-parallel* recovery step executed using 8 threads. The recovery step in [47] and [143] accounts for the 18% and 27% of the total runtime on average, respectively. There are cases, such as des_perf_fast or matrix_mult_fast for [143], where "the last mile" optimization represents more than the 50% of the total runtime. On the contrary, the iterative recovery of the proposed approach performs on average 5 itera-



(a): slow timing constraints

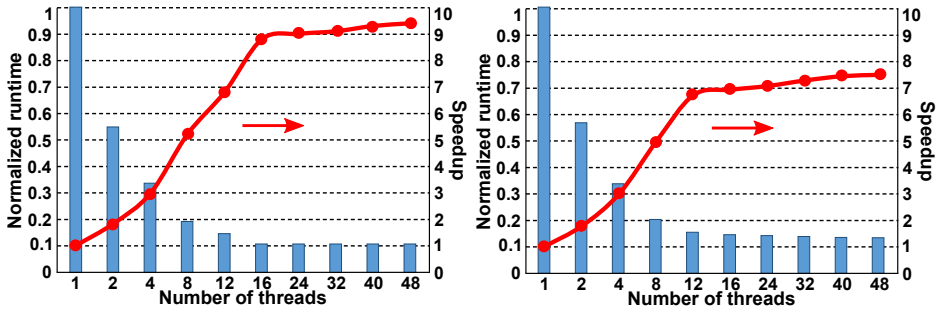


(b): fast timing constraints

Figure 4.14: The percentage of the total runtime utilized by recovery in the five largest designs under (a) slow and (b) fast timing constraints. Both single-threaded recovery steps utilize significant part of the total runtime when compared to the proposed method.

tions and takes 4% of the total runtime. This result stems from the task-based execution of the recovery step and the extra TSE edges added to the task graph that allow for resizing multiple gates in parallel and preserving the timing accuracy needed in this step.

The scalability of the task-based recovery step with respect to number of active threads is depicted in Figure 4.15 for matrix_mult_fast and netcard_fast designs. Initially, the speed up of matrix_mult_fast scales sufficiently until 16 threads. Then, the improvement stops. In the main optimization task graph this behavior is observed at 24 threads. The reason is that the recovery is executed on a more constrained task graph with 3$\times$ more dependencies (due to the TSE edges) that limit inevitably the available parallelism but ensure better timing accuracy after each local timing update. Similarly, at first the runtime of netcard_fast (Figure 4.15(b)) starts improving until 12 threads where speedup levels off for the same reason. The recovery graph of netcard_fast contains 5$\times$ more dependencies and therefore speedup saturates earlier than matrix_mult_fast.



(a): matrix_mult_fast (b): netcard_fast

Figure 4.15: The normalized runtime and speedup of the proposed recovery step for the largest benchmarks using different number of threads. TSEs limit the improvement of speedup at 16 and 12 threads for (a) matrix_mult_fast and (b) netcard_fast, respectively.

4.7.5 Recovery with Composite Tasks

One extra choice that would possibly improve the runtime behavior of the recovery step is to allow the resizing of multiple gates in the same trial. To enable this feature, we should group multiple gates in the same composite task and adjust appropriately the dependency edges of the restructured recovery task graph. Then, inside each composite task all choices are examined allowing two or more gates to change size in the same trial.

To test this feature, we replaced pairs of tasks of the original task graph with composite tasks of two gates. Two tasks are merged when they both represent timing-critical gates (with negative slack at their outputs) or gates with enough positive slack (above 50ps) that can be grouped for power recovery. The pairs of tasks that are grouped should refer to directly connected gates in the netlist and their replacement by a composite task in the recovery task graph should not create any cyclic dependency. This is achieved by performing a depth-first search from the composite task after each merge. If there is a cycle, the merge it is undone, otherwise it is kept.

Inside each composite task, the possible solutions of both gates are enumerated and visited in ascending order of their cumulative power. The first solution that improves local TNS (for timing recovery) or does not degrade local TNS (for power reduction) is selected. The available sizes per gate are the same as the baseline recovery Algorithm 13. However, by allowing the resizing of multiple gates at once, the number of choices increases exponentially.

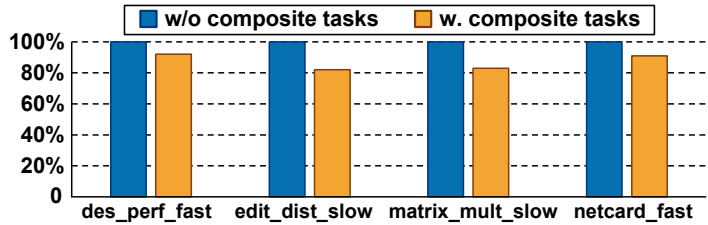


Figure 4.16: The normalized runtime of the proposed recovery without and with composite tasks for the largest benchmarks at 24 threads.

The obtained results using 24 threads are depicted in Figure 4.16. The runtime of the recovery improves in all examined cases by 10% on average without affecting negatively the final leakage power of the design. The number of composite tasks depends on the timing slack of each gate and ranges between 10k and 24k in the examined designs. In other words, composite tasks are between 2.5% and 15% of the total number of tasks.

This feature, even if it seems a promising solution, in its current form offers negligible improvements in the total runtime of gate sizing. Future research work will highlight what is the best approach to group gates in composite tasks, how to examine fast the increased options per composite task, and how to quantify how the formation of composite tasks affects the structural parallelism of the baseline recovery task graph.

4.8 Conclusions

Expressing all parts of timing and power optimization, even those that are traditionally considered as serial operations, as a task-based parallel program allows for scalable runtime improvements and maximum performance portability. The development of the task-parallel gate sizer included the selection of the appropriate optimization kernels for each part of the sizing process and the definition of the dependencies between them. No effort was spent on managing thread execution. Execution order was constrained by the introduced dependencies and handled automatically by Taskflow.

Choosing appropriate dependencies between gate resizing tasks enables their execution with varying levels of timing accuracy. Iterative optimization steps at the beginning of the design flow can operate with relaxed timing accuracy and enjoy many iterations with fast runtime. On the other hand, final recovery steps of gate sizing or when gate sizing is applied at the end of the physical design flow need higher timing accuracy to still enjoy parallel execution but also not compromise the already achieved QoR.

Additional runtime was saved by speeding up the execution of each task. This was

achieved by enabling the two aditional heuristics proposed in this work that dynamically alter the number of examined sizes per gate, or reduce the neighbor gates that participate in local timing update.

5 Flip-flop Placement Targeting Clock-induced OCV

5.1 Introduction

The On-Chip Variation (OCV) effect refers to the intrinsic variability involved in semiconductor manufacturing processes and the fluctuation of operating conditions, such as voltage and temperature, and how they impact a circuit's timing [34].

Due to OCV, some cells may be faster or slower than expected, thus introducing delay uncertainties in data and clock path delays, resulting in more stringent timing constraints. In order to model them, timing derates are introduced that are multiplied with the net and cell delays [12]. For example, in the case of the clock tree, a launch clock path can be slower than expected, while a capture clock path can be faster than expected. In this case, if there is no sufficient positive slack, the increase of clock skew uncertainties may cause a violation of the late (setup) timing constraints. Nevertheless, this opposite derating of the launch and the capture clock paths cannot occur on the part of the clock tree that is common to both paths. The common path should not be derated since the clock can be either slower or faster for both the launch and capture paths. Common path pessimism removal discards this artificial pessimism during timing analysis [12].

Previous research tries to alleviate the impact of OCV either during Clock Tree Synthesis (CTS), or by optimizing already synthesized clock trees. Optimizing the quality of the top-level clock tree by reducing clock divergence and optimizing placement of clock logic and buffers was the goal in [17] and [127]. In [119], a statistical centering based clock routing method is proposed that makes the clock skew more tolerant to interconnect variations. The work of [156] reconstructs the topology of a synthesized clock tree by reconnecting buffers for removing OCV timing violations, while improving the lower bounds on the Worst Negative Slack (WNS) and the Total Negative Slack (TNS).

The work in [136] improves timing in multiple modes and multiple corners by applying useful clock skew on an already constructed clock tree. A formulation based on linear programming similar to [99, 128] computes optimal positive or negative clock skew

offsets that are applied on the clock tree using buffer insertion, removal, or relocation. Similarly, the work of [58] minimizes the sum of skew variations over all adjacent sink pairs using both global and local optimization that includes solving a linear program and utilizing machine learning to predict the impact of local moves on clock latency. Improving the correlation between the predicted clock skew offsets for tackling OCV and the achieved timing quality after CTS was the focus of [36].

Non-tree clock structures have also been tested as a means for reducing clock-induced OCV. Clock meshes [54] and clock trees with cross links [38, 126] represent the most relevant approaches. Non-tree structures reduce clock skew and improve the robustness of clock networks compared to tree-shaped clock networks, but incur an additional non-trivial cost.

In this work, we try to bring appropriately selected flip-flop and clock gaters closer, while respecting both their initial spatial locality and the functional clock-gating hierarchy, in order to create a better seed for CTS to produce clock trees with less path divergence that are inherently less sensitive to clock-induced OCV. Once global placement and in-place datapath optimization have finished, flip-flops and clock gaters are first clustered in a bottom-up fashion, using soft clustering [56], and then moved iteratively closer to the weighted mean location of the center of all neighbor clusters. *Flip-flop-to-cluster membership is not a hard decision and it is not fixed at any stage of the algorithm.* The membership of flip-flops to clusters is quantified by weights that model the physical proximity and timing adjacency of the examined clock cells as well as their timing criticality. Membership weights are not constant but they are dynamically updated as flip-flop/clock-gater relocation evolves. It should be noted that, in this work, clustering is only used for guiding clock cell relocation, and it does not lead to circuit restructuring, as done in clustering-driven CTS engines [32, 144], or in other approaches that use clustering for register clumping [22, 62, 120, 166] or multibit register composition [79, 86, 139].

The proposed method *is applied before CTS* and *is inherently orthogonal* to any previous work that optimizes directly the clock tree topology for reducing clock-induced OCV. Even if clock cell relocation is allowed in modern CTS or post-CTS optimization flows, still the distances that the cells can travel are significantly restricted compared to the pre-CTS stage of the implementation flow that is the focus of this work.

5.2 Motivation–Problem formulation

In this approach, we focus on changing the clock cell placement to allow the CTS engine to produce clock trees with as many common paths as possible thus reducing the effect of clock-induced OCV timing degradation. One generic approach to guide CTS into

placing selected clock cells on the same clock branch is to bring those cells closer in the physical layout. To take advantage of this property, we need to identify (a) which clock cells, when put closer, would alleviate OCV derates, and (b) how to relocate the selected cells. The following example will shed more light on to how we attack the problem of OCV-aware relocation of clock cells.

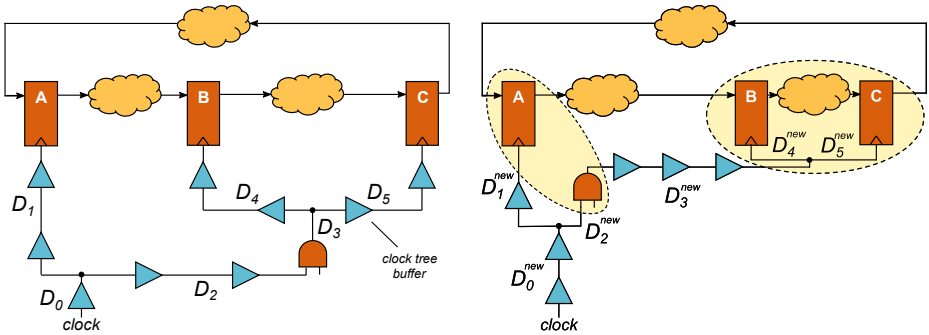


Figure 5.1: Physical proximity of selected clock cells driven by the same clock nets reduces clock divergence and improves common clock path delay. (a) The original clock tree, and its (b) optimized version to tackle clock-induced OCV.

Let us consider the example shown in Figure 5.1(a) that involves three flip-flops A, B, C, and a clock gater that drives flip-flops B and C. The clock skew for each one of the three register-to-register timing paths AB, BC, and CA is the delay difference between the clock launch and the capture path. The delay of any common path between the launch and capture paths is omitted when computing their delay difference. Due to OCV, when considering the worst-case scenario in terms of late constraints, the delay of the launch clock path is increased by a derating factor γ. For the sake of simplicity, we also assume that the delay of the capture clock path is decreased by the same factor. Therefore, the late clock skew of all paths, after the clock divergence point for each path, is the difference between the maximum clock path delay of the launch path and

the minimum clock path delay of the capture path.

$$\begin{aligned}
\text{skew}_{AB}^{\text{late}} &= D_1^{\max} - (D_2^{\min} + D_3^{\min} + D_4^{\min}) \\
&= (\overline{D}_1 - \overline{D}_2 - \overline{D}_3 - \overline{D}_4) + \gamma(\overline{D}_1 + \overline{D}_2 + \overline{D}_3 + \overline{D}_4) \\
\text{skew}_{BC}^{\text{late}} &= D_4^{\max} - D_5^{\min} \\
&= (\overline{D}_4 - \overline{D}_5) + \gamma(\overline{D}_4 + \overline{D}_5) \\
\text{skew}_{CA}^{\text{late}} &= (D_2^{\max} + D_3^{\max} + D_5^{\max}) - D_1^{\min} \\
&= (\overline{D}_2 + \overline{D}_3 + \overline{D}_5 - \overline{D}_1) + \gamma(\overline{D}_2 + \overline{D}_3 + \overline{D}_5 + \overline{D}_1)
\end{aligned}$$

The equations are derived assuming that $D_i^{\max} = \overline{D}_i(1+\gamma)$ and $D_i^{\min} = \overline{D}_i(1-\gamma)$, where $\overline{D}_i$ represents the mean delay (without OCV) and includes both wire and logic delay.

To reduce the impact of OCV, we need to minimize the sum of clock path delays multiplied by γ. To do so for this example, we start from the lower levels of the clock hierarchy and try to minimize delays $\overline{D}_4$ and $\overline{D}_5$. As shown in Figure 5.1(b), this is achieved by moving flip-flops B and C closer relative to Figure 5.1(a). This relocation of flip-flops B and C guides the CTS to place the common branch point of B and C in close proximity, effectively diminishing the derating effect on path BC. This movement is justified, since B and C are the endpoints of the same clock net of the same timing path. Of course, moving B and C closer, for improving OCV, should not degrade data-path timing.

Next, we move up the clock hierarchy and focus on minimizing the OCV derates on paths AB and CA. In this case, our goal is to transfer part of the delays $\overline{D}_1$ and $\overline{D}_2$ to the common path D_0. To achieve this, we need to bring the clock gater and flip-flop A closer, as shown in Figure 5.1(b). Flip-flop A and the clock gater are driven by the same clock net and participate in the same timing paths. Therefore, bringing them closer increases the probability that CTS will drive them with a common clock path.

On the contrary, moving A closer to B, or C, would not improve clock-induced OCV and their relative placement should be decided by other criteria. These flip-flops belong to different clock sub-nets and their last common clock point is above the clock gater. The gater inevitably acts as a divergence point for the clock net, separating the clock branch that drives B and C from the clock branch that drives flip-flop A, irrespective of the physical proximity of A to B and C. The same arguments hold for all flip-flops that belong to different clock nets. Such flip-flops can be considered irrelevant for the interaction of their placement with OCV, since their clock divergence point belongs to an upper level of the clock tree.

In any case, moving clock cells closer should not tradeoff an increase in clock latency. Latency increase is avoided if cells don't move far away from the cells they drive in the

clock tree hierarchy. For instance, in Figure 5.1(b) the clock gater should approach A but at the same time, it should not move too far away from flops B and C. This the reason for not drawing the gater next to A in this example.

This example highlights that, by creating physical clusters of selected clock cells (*timing neighbors*), after examining the cells' launch-capture connectivity and their position on the clock tree hierarchy, we increase the probability that those cells are put on the same clock branch during CTS.

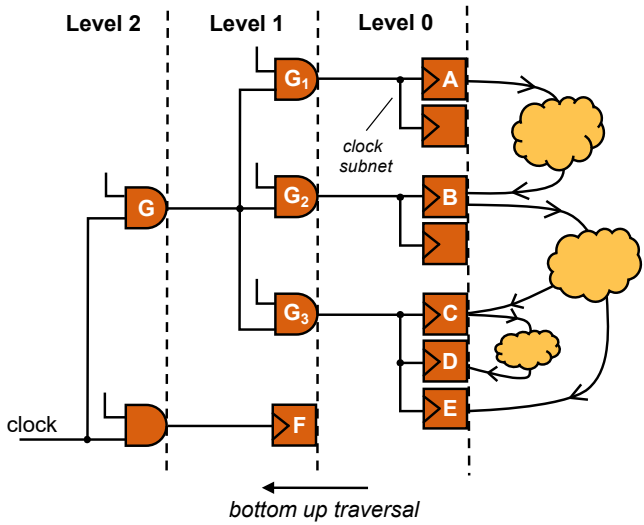


Figure 5.2: An example to explain the definition of timing neighbor clock cells, depending on their connectivity and positions in the functional clock tree hierarchy.

Two clock cells are timing neighbors if they have clock pins on the same net that belong to the launch and capture clock parts of a constrained timing path. For example, in Figure 5.2, flip-flop C is a timing neighbor of flip-flop D and not of flip-flop B. Even if flip-flop C is connected to both flip-flops B and D, it neighbors only with flip-flop D since flip-flop B has its clock pin on different net from C. Also, clock gater G_1 is a timing neighbor cell for gater G_2, since they have their clock pins on the same net (the net driven by gater G) and there is a constrained timing path that connects them through their children ($G_1 \rightarrow A \rightarrow B \rightarrow G_2$). However, the same is not true for G_1 and G_3. Although they are endpoints of the same clock net, there is no timing path that connects their leaves.

Algorithm 14: OCV-aware clock Cell Relocation

```
1  foreach level k of the clock tree - bottom up do
2      foreach clock net n of level k do
3          Cells[n] ← all cells at the endpoints of n;
4          Clusters[n] ← InitClusters(Cells[n]);
5          repeat // Cluster and Relocate
6              repeat // Soft Clustering
7                  Compute m(i, j) ∀i ∈ Cells[n] and j ∈ Clusters[n] using eq. (5.2);
8                  Update the centers of all Clusters[n] using eq. (5.4);
9              until convergence;
               // Cell relocation
10             foreach cell i ∈ Cells[n] do
11                 if i not timing critical && not reached displacement limit then
12                     Move i closer to the weighted mean location of the centers of
                       Clusters[n];
13                 end
14             end
15             UpdateTiming();
16         until no cell moved;
17     end
18 end
```

5.3 Soft Clustering-based Placement

The proposed approach is based on a repetitive process of incremental clustering and clock cell relocation with the goal to bring timing neighbors closer and increase the probability that they are driven by the same clock branch after CTS.

Algorithm 14 depicts the overall structure of the proposed method. The clock cells are examined hierarchically beginning from the leaves of each clock net. The clock cells at the sinks of each clock net are clustered using the k-Harmonic Means (kHM) soft clustering algorithm [169]. kHM begins with an initial guess of the solution (line 4 of Algorithm 14), and then refines the position of the centers until it reaches convergence, i.e., the positions of the cluster centers change by less than 1% per iteration (lines 6–9 of Algorithm 14).

In contrast to hard clustering algorithms [74, 166], kHM is a soft clustering algorithm and allows the cells to belong to more than one cluster [56]. Function $m(s_i, c_j)$, with

$0 \leq m(s_i, c_j) \leq 1$ and $\sum_j^k m(s_i, c_j) = 1$, defines the grade of membership of clock cell s_i to the jth cluster with center c_j. This membership function effectively combines the physical location of the cells with the location of their timing neighbors.

Once the cell-to-cluster memberships have been computed, each examined cell tries to approach the center of the clusters according to the computed membership grades (lines 10–14 of Algorithm 14). The soft membership nature of the proposed clustering algorithm allows *all nearby clusters* to contribute to the movement of each clock cell. This feature would have been impossible with clustering algorithms that employ hard membership functions.

Once all candidate cells have tried one new position, routing and timing are incrementally updated to reflect the available slacks at the inputs and the outputs of the affected cells (line 15 of Algorithm 14).

5.3.1 Initialize cluster centers

The kHM soft clustering algorithm is executed independently on each clock net n, assuming a predetermined number of clusters K. The number of clusters is computed based on a maximum-allowed cluster size that tries to mimic the maximum-fanout constraint imposed on the clock tree.

During initialization, we partition the cells driven by each clock net in equally-sized groups of clock elements (flip-flops and clock gaters), and select randomly the position of one cell from each group as the initial cluster center. As it will be shown in Algorithm 15, the initial cluster centers are derived after taking into account the cells of each net as well as the cluster centers already defined for the hierarchically lower clock nets. This addition is needed to avoid clock cells being placed far away from the cells they drive, which could increase clock wirelength and latency. The upper levels of the clock-tree hierarchy are sparse and involve in most cases a few clock gaters placed far apart from each other. Therefore, bringing those cells closer, as dictated by the proposed method, would risk to separate them from the cells they drive. This risk does not appear on the lower levels of the clock tree where the cells perform only local moves.

The recursive partitioning of the cells driven by each clock net is highlighted in Algorithm 15. In particular, we first define the bounding box that encloses all cells of clock net n, i.e., Cells[n]. Then, we add to Cells[n] all cluster centers of the lower level of the clock tree that are connected to each cell of Cells[n] and placed inside the bounding box of n (line 4 of Algorithm 15). In this way, the cluster initialization of clock net n is done on *AllCells* that includes both the cells connected to n and the pre-defined cluster centers of the hierarchically lower clock sub-nets. Then, the set of points that determine the initialization of the clusters of net n are first sorted geographically, and they are recursively partitioned to equally-sized sets using *RecPartition* function described in Algorithm 15.

Algorithm 15: Initialize Clusters

```
1  function InitClusters (Cells[n])
       // Set #clusters using cells of subnet n
2      K ← Cells[n] / MAX_CLUSTER_SIZE;
3      BB← BoundingBox(Cells[n]);
       // Include cluster centers of next level that are inside BB
4      AllCells← Cells[n] ∪ Valid_Centers;
5      step← AllCells / K;
6      SortedCells ← Sort AllCells according to their (x,y) co-ordinates (x first).;
7      return RecPartition(SortedCells, step);
8  endfunction
   // Recursive partitioning to equal size groups
9  function RecPartition (SortedCells, step)
10     if sizeof(SortedCells) ≤ step then
11         return a random point j ∈ SortedCells;
12     end
       // Split SortedCells in two sets
13     C1 ←SortedCells[1 : step];
14     C2 ←SortedCells[(step+1) : sizeof(SortedCells)];
15     RecPartition(C1, step);
16     RecPartition(C2, step);
17 endfunction
```

5.3.2 Compute membership function

In kHM clustering algorithm [169] the probability of a cell s_i being a member of the jth cluster is determined by the harmonic average of the distances of each cell to the centers of all K clusters and is given by:

$$d(s_i, c_j) = \frac{\left\|s_i - c_j\right\|^{-p-2}}{\sum_{k=1}^{K} \left\|s_i - c_k\right\|^{-p-2}} \tag{5.1}$$

Parameter p is set to 4, to distinguish more clearly the cells that are located far from the center of the cluster relative to those that are placed in a nearby position. For traditional 2D clustering, the scaled physical distance of a cell from the centers of all clusters given by $d(s_i, c_j)$ would have been a sufficient clustering quality metric [166, 169]. However, *for the OCV-aware placement of sequential cells, this is not enough*. A cluster is a good candidate for cell s_i, if the timing neighbors of s_i, denoted as $\mathcal{N}(s_i)$, also belong to the

same cluster, especially the most timing critical ones. In this way, CTS is guided to put them on the same clock branch and effectively reducing clock-induced OCV.

The proposed membership grade

Membership grade $m(s_i, c_j)$ should reflect both the spatial proximity of s_i to the center of the jth cluster c_j, as expressed by $d(s_i, c_j)$, as well as the physical proximity of the neighbors of s_i to the same cluster. Effectively, the closer a timing neighbor of s_i is to cluster j the larger the "pressure" towards cell s_i to group to the same cluster as well.

To express these dependencies, we define the membership grade of cell s_i to the jth cluster as follows:

$$m(s_i, c_j) = a \cdot d(s_i, c_j) + (1-a) \frac{\sum_{k=1}^{|\mathcal{N}(s_i)|} t(s_k, s_i) d(s_k, c_j)}{\sum_{k=1}^{|\mathcal{N}(s_i)|} t(s_k, s_i)} \tag{5.2}$$

The distance $d(s_k, c_j)$ of each timing neighbor s_k of s_i contributes relative to its timing criticality $t(s_k, s_i)$ with respect to s_i. The more critical the timing path that connects s_k and s_i, the stronger the need to bring them closer, expecting that CTS will drive them with a common clock tree path.

For $a = 1$, membership is determined only by the physical distance of each cell to the center of each cluster. This corresponds to traditional flip-flop clumping [166], where flip-flops are clustered together based only on their (x, y) coordinates. On the contrary, $a = 0$ would try to bring closer all timing neighbors ignoring their original locations. Empirically, picking an intermediate value for $a = 0.35$ offers a balanced clustering that could realistically increase the common clock paths and decrease OCV derating.

Timing criticality of timing neighbors

The timing criticality $t(s_k, s_i)$ expresses how critical s_k is, in terms of timing, with respect to the timing paths launching at s_i. It is computed by mapping the effective slack $eslk(s_k, s_i)$ of all neighbors $s_k \in \mathcal{N}(s_i)$ of s_i to a value in the range [0,1], using the function (5.3) suggested in [7]. If s_k is the most critical neighbor of $\mathcal{N}(s_i)$, then $t(s_k, s_i) = 1$, while $t(s_k, s_i) \to 0$ if s_k has much greater slack than the average slack of the rest neighbors.

$$t(s_k, s_i) = e^{b\left(\frac{minES - eslk(s_k, s_i)}{avgES - minES}\right)} \tag{5.3}$$

Effective slack $eslk(s_k, s_i)$, is the total negative slack at s_k due to paths launching at s_i, or the worst positive slack when no negative timing path exists between s_i and s_k. Terms

minES and *avgES* are the minimum and the average effective slack of all neighbors $\mathcal{N}(s_i)$, and b is a tuning parameter; $b = 2$ was used since it consistently gave better results.

To compute the effective slack $eslk(s_k, s_i)$, we need to consider the following cases:

(a): If s_k and s_i are both flip-flops, then we consider the timing paths that connect them directly.

i) If s_k is a launch flip-flop for cell s_i, effective slack corresponds to its output-Q pin slack.

ii) If s_k is a capture flip-flop for s_i, effective slack is the slack at its input-D pin.

(b): If s_k and s_i are clock gaters we consider the D/Q pin slacks of the flip-flops placed at the endpoints of their transitive fanout and not just the slack of their enable pins. This is done, because we want s_i to approach the clock gater (s_k) that drives the more critical flip-flops.

For instance, following the clock tree hierarchy shown in Figure 5.2, $eslk(G_1, G_2)$ involves the timing path $A \rightarrow B$ and is equal to the slack of the Q pin of flip-flop A. Similarly, for $eslk(G_2, G_1)$ we should examine again the path $A \rightarrow B$, but in this case we consider the slack at the input-D pin of flip-flop B. For $eslk(G_3, G_2)$, we examine the paths $B \rightarrow C$ and $B \rightarrow E$ and consider the slack on the input-D pin of flip-flops C and E. Path $C \rightarrow D$ – that is internal to the subnetwork rooted by G_3 – does not contribute to the effective slack of any of its timing neighbors.

5.3.3 Update cluster center

Once the membership grade of each cell s_i placed at (x_{s_i}, y_{s_i}) to all clusters has been computed, the location of the center of each cluster (x_{c_j}, y_{c_j}) needs to be updated using Equation (5.4).

$$x_{c_j} = \frac{\sum_{i=1}^{\#cells} m(s_i, c_j) w(s_i) x_{s_i}}{\sum_{i=1}^{\#cells} m(s_i, c_j) w(s_i)}$$
$$y_{c_j} = \frac{\sum_{i=1}^{\#cells} m(s_i, c_j) w(s_i) y_{s_i}}{\sum_{i=1}^{\#cells} m(s_i, c_j) w(s_i)} \tag{5.4}$$

Computing the position of the cluster center also takes into account the grade of influence w and the grade of membership m of each cell. This is a unique feature of the kHM soft-clustering algorithm, and makes it less sensitive to the initialization of the

cluster centers. The grade of influence $w(s_i)$ of cell s_i to the positions of all clusters is given by [169]:

$$w(s_i) = \frac{\sum_{j=1}^{K} \left\| s_i - c_j \right\|^{-p-2}}{\left(\sum_{j=1}^{K} \left\| s_i - c_j \right\|^{-p} \right)^2} \tag{5.5}$$

By definition, the impact of cells that are not close to any center is increased, while the impact of cells that are close to one or more center is decreased. This principle helps in spreading the centers to cover the positions of all cells.

5.3.4 Relocate Cells

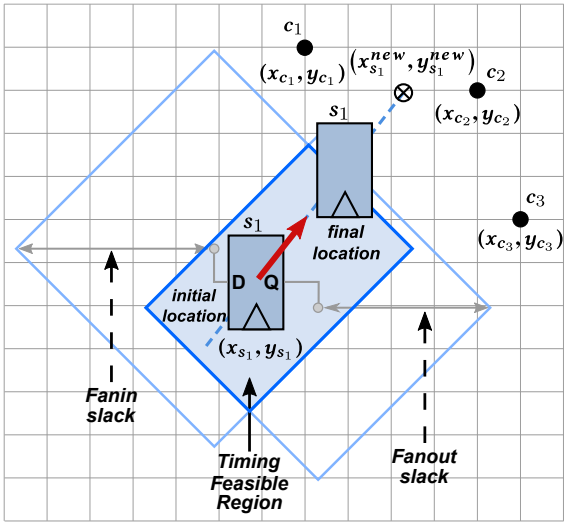
After soft clustering has converged, each cell moves closer to the centers of the more preferable clusters. With this move it attracts its timing neighbors to prefer the same cluster, and increases the probability of sharing a common clock branch with them after CTS.

Clock cell s_i placed at (x_{s_i}, y_{s_i}) in the current iteration is relocated to $(x_{s_i}^{new}, y_{s_i}^{new})$. The new location should be closer to the clusters preferred by s_i, i.e., its membership grade for them is high. For this reason, s_i is relocated to the weighted mean of the location of the centers of all nearby clusters using (5.6). The contribution of each cluster center to the new location is proportional to the membership grade of s_i to each cluster.

$$x_{s_i}^{new} = \frac{\sum_{j=1}^{K} m(s_i, c_j) x_{c_j}}{\sum_{j=1}^{K} m(s_i, c_j)}, \quad y_{s_i}^{new} = \frac{\sum_{j=1}^{K} m(s_i, c_j) y_{c_j}}{\sum_{j=1}^{K} m(s_i, c_j)} \tag{5.6}$$

As long as we completely avoid any hard cell-to-cluster assignment, we allow cell relocation to evolve more smoothly across iterations. If instead we employed hard assignments, the cell would move closer only to the center of its assigned cluster, thus possibly leading to ping-pong movements when cell-to-cluster assignments changed across iterations. Also, since the relocation of each cell is biased by the relocation of its timing neighbors, it means that, after several iterations, a better global solution is reached.

A cell is allowed to move and approach its new location when it has positive slack to spend and has not reached its displacement limit yet. Our goal is to utilize some of the positive slack of certain clock cells to form large common branches in the clock tree in other parts of the design that are more timing critical. In any case, we avoid creating a tradeoff between improving clock-induced OCV and degrading data-path timing. A flip-flop is considered safe to move to improve the common clock branches of its timing



Cluster centers locations

$(x_{c_1}, y_{c_1}) = (1, 7)$
$(x_{c_2}, y_{c_2}) = (5, 6)$
$(x_{c_3}, y_{c_3}) = (6, 3)$

Flop-to-Cluster membership grades

$m(s_1, c_1) = 0.5$
$m(s_1, c_2) = 0.3$
$m(s_1, c_3) = 0.2$

New location co-ordinates

$$x_{s_1}^{new} = \frac{0.5x_{c_1} + 0.3x_{c_2} + 0.2x_{c_3}}{0.5+0.3+0.2} = 3.2$$

$$y_{s_1}^{new} = \frac{0.5y_{c_1} + 0.3y_{c_2} + 0.2y_{c_3}}{0.5+0.3+0.2} = 5.9$$

Figure 5.3: A flip-flop approaching its new location computed as the weighted mean of the three cluster centers. Its relocation is limited by the current timing feasible region.

neighborhood, as long as the input-D and output-Q pin slacks are both positive. Similarly, a clock gater is safe to move when its input-enable pin slack is positive and there is no critical flip-flop at the subtree rooted on this clock gater.

Figure 5.3 illustrates graphically the overall cell relocation process. The new location is the weighted mean of the centers of all nearby clusters and cell movement is bounded by its timing feasible region [21], i.e., the common region formed by the transformation to equivalent distance of the positive slacks of the fanin and fanout nets. Each flip-flop's input and output positive slack define a diamond centered by the fanin and fanout gates of the clock cell, while its half diagonal corresponds to the equivalent distance computed using Elmore delay. In the case of nets with multiple endpoints, each endpoint is considered individually and the intersection of the diamonds of all endpoints is kept.

The cell is legalized instantly to the new suggested position, or to a close-by available position chosen by the legalizer (within 5 rows). Also, when we relocate a cell, the routing congestion and/or row utilization may be degraded due to this movement. To avoid such deteriorations, we do not perform relocations to areas where the routing congestion and/or row utilization is already high, since such movements would affect these metrics even more negatively.

When a cell is relocated it alters its own timing profile as well as the timing profile of all connected cells. Thus, timing and routing needs to be updated to have an accurate view of the timing slacks of each cell. However, performing such incremental updates per cell movement is prohibitive in terms of runtime. As depicted in Algorithm 14, timing and routing are updated once every iteration, after moving many cells. In the meantime, the slacks per cell remain inaccurate. However, respecting the maximum displacement limit and the fact that no cell moves beyond its timing feasible region partially alleviates the problem.

Finally, after each cell relocation two metrics should be updated: its own membership and influence grades relative to every cluster, and the membership grade of every timing neighbor, even if neighbors were not actually moved. After that, the position of the centers of all clusters should be updated, too. Therefore, the re-execution of soft clustering after partial cell relocation needs fewer iterations to converge. At some point, cells and their clusters have been stabilized with no cell being able to move. Please note that once a cell moves closer to the centers of its preferred clusters, it directly impacts the grade of membership of all other connected cells, and all together affect the new position of the centers of all clusters. This orchestrated movement gradually makes the clusters more distinct.

5.3.5 Algorithm complexity

The proposed cell clustering and relocation is executed independently per clock net in a bottom up manner. Let's assume that each clock net consists of N endpoints (clock cells). In the worst case, the N cells can be split to N/K clusters while each cell can have N timing neighbors. Therefore, letting each one of the N cells of the clock net examine all possible cluster centers and all possible timing neighbors leads to a complexity of $O(N^3)$ per clock net. For C clock nets in the functional clock tree hierarchy, the overall complexity is $O(CN^3)$. In practice, the runtime complexity is lower since the designs consist of many small clock nets (i.e, small N per net, large number of nets C) due to the deep functional clock-gating hierarchy seen in modern designs.

In all cases we consider all clock nets. Thus, we cannot reduce the contribution of C in the runtime complexity. To limit the runtime complexity of the proposed method, we restrict the computation involved per clock net. In our experiments, each cell of a clock net cannot examine more than 50 timing neighbors of the same clock net (the ones placed far away are avoided), while it considers at most 20 nearby cluster centers.

5.4 Experimental Results

The proposed flip-flop and clock gater placement methodology has been implemented in C++ and integrated in the Nitro-SoC place-and-route tool. It is executed after global placement and data-path optimization. The former provides valid cell locations, whereas the latter fuels the cells with positive slack allowing them to cover bigger distances. Once the proposed algorithm has concluded, cell group placement constraints are generated for all clustered cells. The cell groups act as fences prohibiting the clustered cells to move away from their initial position. The rest of the implementation flow remains unchanged and runs to completion.

To judge the overall effectiveness of the proposed method, in terms of timing, OCV robustness, and clock tree complexity, we compare it with two versions of the reference flow: The first one "Base" represents the industrial quality flow which was originally used to implement the designs. The second one, denoted as "Cluster", activates physical register clustering at the same point in the flow as the proposed method, and implements the algorithm presented in [166]. In particular, this physical clustering [166] utilizes a modified version of *k*-means algorithm, driven by placement and physical distance criteria, to create physical groups of flip-flops with the goal to simplify the clock tree and reduce clock power. However, we included this technique in our comparisons to highlight that flop grouping techniques that rely on hard flop-to-cluster assignments and use only physical distances for clustering, while ignoring timing criticalities and flip-flop communication, cannot reduce succesfuly the impact of clock-induced OCV. Our implementation of "Cluster" creates groups of tightly placed flip-flops without necessarily forming banks of regularly-placed flip-flops as done in [166]. "Cluster" and the proposed approach cannot move clock cells more than the maximum allowed displacement of 20 rows.

The effectiveness of the proposed method is evaluated on real industrial designs that cover different complexities spanning from 82K up to 1.54M cells and implemented in different technologies between 28 and 14nm. All designs but D2 are constrained with Advanced OCV (AOCV) derates [34]. D2 has simple OCV derates.

5.4.1 Timing comparisons

The results obtained by the three methods under comparison with respect to timing for setup and hold constraints are shown in Table 5.1. The first noticeable result is that in all cases, when applying the proposed flip-flop relocation, the worst-negative slack (WNS) and total negative slack (TNS), for both setup and hold analysis, is reduced; an indication that the criticality of certain paths due to OCV is reduced.

Table 5.1: The timing and row utilization of all designs for the reference implementation flow (Base), the modified flow including the physical register clustering (Cluster) of [166] and the proposed OCV-aware clock cell relocation (New).

Design		Setup		Hold		Util (%)
		WNS (ps)	TNS (ns)	WHS (ps)	THS (ns)	
D1 - 14nm	Base	-337.2	-29.7	0.0	0.0	70.9
82K cells	Cluster	-320.0	-28.8	0.0	0.0	70.5
4.5K regs	New	-297.0	-25.2	0.0	0.0	70.9
D2 - 28nm	Base	-396.0	-885.0	-134.0	-0.6	65.1
199K cells	Cluster	-409.0	-1148.1	-104.0	-7.5	63.6
16K regs	New	-368.0	-768.2	-1.0	-0.1	62.9
D3 - 16nm	Base	-43.0	-0.6	-15.0	-0.6	55.5
542K cells	Cluster	-137.0	-0.9	-17.0	-0.1	55.8
35K regs	New	-24.0	-0.3	-14.0	-0.1	55.7
D4 - 22nm	Base	-232.0	-564.2	0.0	0.0	80.3
557K cells	Cluster	-288.0	-677.0	0.0	0.0	80.8
47K regs	New	-223.0	-392.5	0.0	0.0	80.6
D5 - 16nm	Base	-802.0	-442.9	-35.0	-1.4	56.5
611K cells	Cluster	-668.0	-487.0	-49.0	-0.9	55.6
45K regs	New	-379.0	-100.6	-30.0	-0.6	56.7
D6 - 14nm	Base	-103.0	-41.1	-93.0	-6.0	64.7
1545K cells	Cluster	-68.0	-20.6	-170.0	-20.2	63.8
71K regs	New	-59.0	-16.4	-68.0	-1.9	65.1

Contribution (%) of "New" to the runtime of the full flow

D1	D2	D3	D4	D5	D6
0.83%	0.38%	8.18%	1.23%	6.20%	1.69%

Flip-Flops and clock-gaters have exchanged some of their positive slack to help critical cells reduce the OCV effect on their timing paths. Setup TNS has reduced by 42% on average, while setup WNS is better by 28% on average. Worst Hold Slack (WHS) and Total Hold Slack (THS) have been improved by 45% and 73% respectively. The average savings include the savings of the proposed design relative to both "Base" and "Cluster" method used for comparison. TNS and THS reductions are more distinct since the criticality of the timing neighbors of each cell (see Equation (5.3)) takes into account the sum of the negative slacks of the related timing paths.

At the rightmost column of Table 5.1 we report the utilization for each design. The maximum local utilization does not increase since we employ safeguards to avoid re-

locating cells to areas with already high utilization or routing congestion. The average utilization change is negligible although D2 exhibited a significant utilization improvement with the proposed methodology.

It should be noted that the reported timing violations were collected after the post-CTS optimizations of an industrial quality flow. It is very common in physical design that improvements obtained at specific points in the flow to be partially lost by the optimizations performed later on. However, the proposed cell relocation integrated smoothly with the rest of the flow providing by far the best overall timing Quality-of-Results (QoR).

At the bottom of Table 5.1, the runtime of the proposed work is reported as the contribution (%) of "New" to the runtime of the full flow. The percentages reported correspond to the *single-threaded* execution of the proposed cell relocation algorithm on a machine with four Intel Xeon CPUs at 2.60 GHz and with 250GB memory. The runtime complexity is heavily dependent of the functional clock tree hierarchy, i.e., on the number of clock nets and the endpoints per clock net. Since the proposed method is executed independently on the cells of each clock net, in our future work, we plan to assign the soft-clustering and cell relocation on each clock net to different threads.

5.4.2 Clock-induced OCV redistribution

To observe more clearly how the proposed method guided the CTS engine to produce clock trees with increased common clock tree paths for the communicating sequential elements, we computed the histogram of the impact of clock-induced OCV on late slack on a large set of timing paths.

For the 30K most critical paths of each design, we measured the difference of the path's late slack with and without OCV derates. Then, to produce the required histograms, we split the paths to bins according to the derived slack value. For instance, a bin of 60ps containing a certain number of paths means that those paths are derated in overall by 60ps due to clock-induced OCV relative to the case that OCV is neglected. The histograms of the impact of OCV on late slack for two representative designs and for the three methods under comparison are shown in Figure 5.4. Similar results are obtained for other designs.

Both diagrams of Figure 5.4 reveal that the proposed method ("New") correctly identified the paths most heavily affected by OCV and restructured them to increase their common clock path. For example, for "New" the majority of the paths in the case of D3 in Figure 5.4(a), are affected by 60–110 ps due to OCV. On the contrary, for the baseline design, the impact of OCV is more pronounced since the majority of the paths in this case experience a slack impact of 160ps due to OCV. Similarly, for design D6 in Figure 5.4(b), the proposed method achieved to shift the OCV impact of 210 ps and

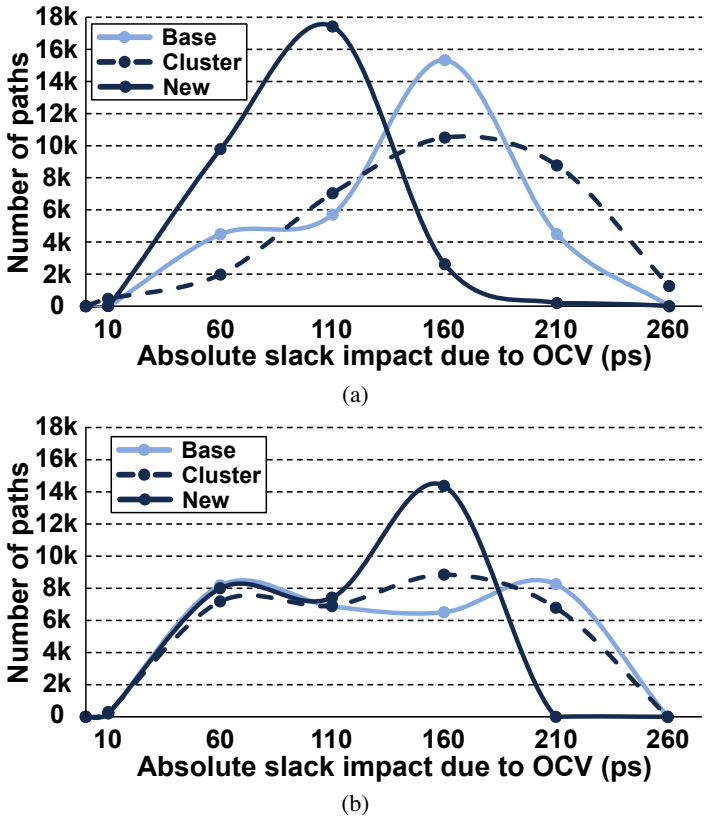


Figure 5.4: The histogram of the impact of clock-induced OCV on late slack on designs (a) D3 and (b) D6.

above to 160 ps. On the other hand, the restructuring of the clock tree triggered by "Cluster" [166] distributes the slack across critical paths in a non-favorable manner. This behavior highlights that simple physical clustering of clock cells is not enough for tackling the effect of clock-induced OCV.

5.4.3 Clock tree complexity

The reduction of slack degradation due to OCV and the overall improvement of timing achieved by the proposed method, as observed at the end of the flow, does not incur any

Table 5.2: Clock tree characteristics for all three methods under comparison.

Design		Clock tree				
		Buffers	WL (mm)	Cap (pF)	Lat (ps)	Skew (ps)
D1	Base	64	18.7	8.4	352	88
	Cluster	65	19.0	8.5	320	88
	New	64	18.1	8.2	341	108
D2	Base	342	103.6	33.6	635	162
	Cluster	300	102.6	33.3	604	134
	New	303	99.5	32.4	535	124
D3	Base	1285	211.9	85.5	690	164
	Cluster	1201	210.8	84.7	740	140
	New	1216	211.2	84.2	677	166
D4	Base	6650	326.1	150.0	661	98
	Cluster	6688	338.2	151.5	599	110
	New	6637	327.2	149.8	679	123
D5	Base	5719	250.4	238.3	1642	143
	Cluster	6009	267.1	250.9	1749	142
	New	5611	253.5	239.9	1646	112
D6	Base	9463	569.6	774.0	1911	197
	Cluster	9540	580.7	778.5	1808	236
	New	9650	571.1	776.0	1540	173

significant overhead to the complexity of the clock tree. This conclusion is supported by the results in Table 5.2 which shows the number of clock buffers, the clock wirelength, and the total clock capacitance including both wire and pin capacitances, as well as the average clock latency and the clock skew. In most of the cases, the clock tree QoR metrics exhibited insignificant differences. However, there were cases where noticeable changes were observed. For instance, the clock latency of "New" for D6 improved compared to the "Base" at the cost of more clock repeaters. Our work did not intend to exercise this trade-off. This was a decision made by the CTS implementation engine.

5.5 Conclusions

Applying iteratively soft clustering and clock-cell relocation improves the physical proximity of timing-neighbor and reduces the clock-induced OCV by increasing the common clock tree paths. To achieve this result at the end of the flow requires a balanced approach that would take into account the spatial proximity and the timing criticality of the cells and their timing neighbors in the functional clock-tree hierarchy. Such metrics

have been gracefully combined in the soft clustering algorithm that drives clock cell relocation. The results across six industrial benchmarks demonstrate the effectiveness of the proposed approach in producing robust clock trees with significantly improved timing.

6 Conclusions

6.1 Summary

As technology scales and the number of cells contained in the design continues to grow, the overall design flow and the timing closure become a major challenge. Even if EDA industry reacted appropriately for several years in developing design methodologies that best address the emerging technology challenges, currently, it faces three intertwined and hard-to-solve challenges: cost, quality and runtime predictability. The focus of this PhD research is next generation physical design automation methods and tools that satisfy the aforementioned design goals. To this end, in this thesis we identified two major aspects that impact physical design scalability: (a) the need to design novel multi-objective algorithms for achieving fast and incremental execution while handling complex multi-mode multi-corner design constraints, and (b) the application of such approaches in a runtime scalable approach supported by efficient parallelism in physical design automation algorithms that enable handling the vast complexity of modern designs with reasonable runtime and without sacrificing quality of results.

The first presented solution optimizes the timing of the design by relocating the cells. The proposed method extends the Lagrangian Relaxation (LR) formulation to relocate all types of cells in a unified manner taking into account both the late and early timing constraints. Before the LR-based placement optimization of the LCBs, a preparatory step is proposed in order to separate flip-flops with incompatible timing profiles. This is achieved using a modified *k*-means flip-flop clustering algorithm in which for the computation of the cost a scaling factor is used that artificially changes the distance of the members of the cluster from the cluster center. In this way, the resulting clusters have uneven sizes delaying or speeding up the clock latency of the appropriate clusters. Using LR, the non-negative weights called Lagrange Multipliers (LMs) act as penalty factors in the global cost function to reflect when the corresponding constraints are violated. However, these LMs are also used as force-like timing vectors to appropriately position the search window. This results to all the tried locations to be beneficial in terms of timing for the examined cell and therefore improves the optimization's convergence. The proposed approach achieves significant improvements in early and late timing results when compared to similar timing-driven techniques.

With the appropriate initialization of the LMs, the second solution presented in this thesis, allows the incremental application of LR-based optimizers at various parts of the design flow. The proposed initialization takes into account both the timing criticality, as well as, the current size of the gates in order to predict the right point to start the optimization. In this way, without affecting any part of the internal functions, we expedite successfully the convergence of the LR-based gate sizer. The presented method has been evaluated on benchmarks considering single and multiple corners. In the latter cases, the optimizer has to meet all the timing constraints across different operating conditions. In all cases, the proposed initialization offers smooth convergence without un-necessary power and timing degradation.

In the third part of this thesis, we expressed all parts of timing and power optimization as a task-based parallel program in order to exploit high CPU parallelism. Even the parts that are traditionally executed serially are converted to parallel processes. To achieve this, we decompose the optimization process into a set of dependent tasks. For each part of the timing optimization algorithm a different task graph is derived. Together with task-based parallel programming, two heuristics are proposed to reduce runtime. In the first one, the number of examined sizes per gate is reduced according to the gates slack, while in the second one, the neighbors that participate in the local timing update are limited accelerating the local decision process sacrificing timing accuracy.

The final contribution of this thesis is an iterative soft clustering and clock-cell relocation method that reduces the clock-induced On-Chip Variations (OCV). This work is applied at the end of the flow and thus is orthogonal to the clock tree synthesis engine. The clock tree is traversed in a bottom up manner starting from the leaves and for each net a modified soft clustering is applied to find the appropriate location for the clock cells. More specifically, the clustering takes into account the spatial proximity and the timing criticality of the clock cells as well as their timing neighbors in the clock-tree hierarchy. Then, following the result of the clustering method, the cells of each cluster are relocated closer to each other so that after the clock tree generation they are driven by the same clock branch reducing the timing impact of the OCV in the clock tree. This novel methodology can effectively produce robust clock trees and gives significant improvements in the late and early timing of the designs with only sight overhead to the final clock tree complexity. The benefits are shown across industrial benchmarks.

6.2 Future Work

The methods proposed in this thesis cover multiple aspects of timing-driven optimizations, offering significant improvements in the timing profile of each design without increasing the final power and area. Although the presented solutions represent a ma-

ture timing optimization portfolio, there are still many opportunities planned for future research.

For instance, in timing compatibility flip-flop clustering we need to derive an efficient approach for deciding beforehand the most appropriate number of clusters. This is highly critical when applying the proposed clustering approach in a pre-CTS phase of the design. When the number of clusters is small, the separation of timing incompatible flip-flops becomes more challenging and there is always the risk to assign to the same cluster slow and fast flip-flops. In addition, it is more difficult to achieve the second target of the clustering that is to create uneven cluster sizes. On the opposite cases that clustering works on too many available cluster centers, each cluster ends up having very few flip-flops. This solution increases the number of clock buffers as well as the clock tree complexity that leads to increased the clock tree power.

The second approach is to generalize the initialization of the Lagrange Multipliers in order to allow other optimization methods that rely on Lagrangian Relaxation to be applied in an incremental application context. In the current version, a new initialization is proposed for the gate sizing methods. The LMs are initialized based not only on the timing criticality of the arcs, but also on the power criticality of their corresponding gates. For similar optimizations such as LR-based timing-driven placement, the LM initialization can tradeoff the timing criticality and the wirelength for better performance. In this way, the LMs of cells with positive slack which are also connected with increased wirelength, will have higher values so that they will be placed closer.

Regarding the LR-based gate sizing with task based parallel programming, we want to apply this approach to other timing optimization methods. For instance, it would be interesting to perform the proposed LR-based timing driven placement using TaskFlow in order to accelerate significantly the execution time. This would allow us also to better tune the formulation of tasks, their size and the dependencies between them. In general, the number of task dependencies affects the final obtained QoR in two ways. First of all, an increased number of dependencies limits the available parallelism. In other words, the number of tasks executed simultaneously is reduced and therefore the runtime is increased. At the same time, as the execution approximates the single-threaded run, the optimization has high timing accuracy. Therefore, it is understood that our future work will focus on identifying how many dependencies can be removed to increase the parallelism without significant degradation of the obtained QoR.

The proposed OCV-aware clock cell relocation method is an effective way to reduce the timing impact of the clock induced OCV. However, there are some improvements planned as future work. First of all, the relocation of the clock cells relies on the result of the modified soft clustering algorithm that takes into account the physical proximity and the timing criticality of the cells and their timing neighbors. But, if the objective of the clustering method, in addition to the current features, takes into account the useful

clock skew of the clock cells, better results are expected. More specifically, the clock tree cells can be assigned with useful clock skew that improves their timing. Including the useful skew in the computation of the membership for each cell results to place closer the elements with similar skew and thus increases the probability of the clock tree engine to drive these cells with the same clock branch.

In parallel, there is the plan to consider the consequences of the clock cells relocation on the clock tree complexity. The current version of the proposed method relocates the clock elements at the end of the pre-CTS phase to invoke the following CTS engine to build clock trees with increased common clock paths. Therefore, the final characteristics of the clock tree such as latency, skew and the number of buffers are decisions made by the CTS implementation engine. However, we want to predict the effects of the clock cell movements happened in the pre-CTS phase on the clock tree. In this way, we will be able to prefer movements that lead to high performance solutions with low power and area.

Finally, we plan to slightly enhance the CTS engine in order to create more robust clock trees for the OCV. Our current solution is orthogonal to the CTS method itself. This means that we change the design's placement pre-CTS in order to produce trees with increased common clock tree paths. It is clear that modifying the CTS methodology to identify the closer placed clock cells, longer common clock paths can be built for them with significant improvements in the final results.

Bibliography

[1] Christoph Albrecht, Shrirang Dhamdhere, Suresh Nair, Krishnan Palaniswami, and Sascha Richter. Sequential Logic Synthesis with Retiming in Encounter RTL Compiler (RC). 2006.

[2] Christoph Albrecht, Bernhard Korte, Jürgen Schietke, and Jens Vygen. Maximum mean weight cycle in a digraph and minimizing cycle time of a logic chip. *Discrete Applied Mathematics*, 123(1):103–127, 2002.

[3] Christoph Albrecht, Pascal Witte, and Andreas Kuehlmann. Performance and Area Optimization using Sequential Flexibility. In *International Workshop on Logic and Synthesis (IWLS)*, 2004.

[4] Charles J. Alpert, Chris Chu, Gopal Gandham, Miloš Hrkić, Jiang Hu, Chandramouli Kashyap, and Stephen Quay. Simultaneous Driver Sizing and Buffer Insertion Using a Delay Penalty Estimation Technique. In *International Symposium on Physical Design (ISPD)*, pages 104 – 109, 2002.

[5] Charles J. Alpert, Gopal Gandham, Miloš Hrkić, Jiang Hu, Stephen T. Quay, and Cliff N. Sze. Porosity-aware buffered Steiner tree construction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):517–526, 2004.

[6] Charles J. Alpert, Milos Hrkic, Jiang Hu, and Stephen T. Quay. Fast and flexible buffer trees that navigate the physical layout environment. In *Design Automation Conference (DAC)*, pages 24–29, 2004.

[7] Charles J. Alpert, Miloš Hrkić, Jhen-Jia Hu, Andrew Kahng, John Lillis, Bao Liu, Stephen T. Quay, Sachin Sapatnekar, AJ Sullivan, and Paul Villarrubia. Buffered Steiner Trees for Difficult Instances. In *International Symposium on Physical Design (ISPD)*, pages 4–9, 2001.

[8] Charles J. Alpert, Zhuo Li, Michael D. Moffitt, Gi-Joon Nam, Jarrod A. Roy, and Gustavo Tellez. What Makes a Design Difficult to Route. In *International Symposium on Physical Design (ISPD)*, page 7–12, 2010.

[9] Olivier Aumage, Paul Carpenter, and Siegfried Benkner. Task-Based Performance Portability in HPC. In *European Technology Platform for High Performance Computing (ETP4HPC)*, 2021.

[10] Michel Berkelaar, Pim Buurman, and Jochen Jess. Computing the entire active area/power consumption versus delay tradeoff curve for gate sizing with a piecewise linear simulator. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(11):1424–1434, 1996.

[11] Michel Berkelaar and Jochen Jess. Gate sizing in MOS digital circuits with linear programming. In *European Design Automation Conference (EDAC)*, pages 217–221, 1990.

[12] Jayaram Bhasker and Rakesh Chadha. *Static Timing Analysis for Nanometer Designs: A Practical Approach*. Springer, 2009.

[13] Koustav Bhattacharya and Nagarajan Ranganathan. A Linear Programming Formulation for Security-Aware Gate Sizing. In *Great Lakes Symposium on VLSI (GLSVLSI)*, page 273–278, 2008.

[14] Kenneth D. Boese, Andrew B. Kahng, and Gabriel Robins. High-Performance Routing Trees with Identified Critical Sinks. In *Design Automation Conference (DAC)*, pages 182–187, 1993.

[15] Stephen Boyd, Seung-Jean Kim, Lieven Vandenberghe, and Arash Hassibi. A Tutorial on Geometric Programming. *Optimization and Engineering*, 8:67–127, 05 2007.

[16] Pak K. Chan. Algorithms for Library-Specific Sizing of Combinational Logic. In *Design Automation Conference (DAC)*, page 353–356, 1991.

[17] Tuck-Boon Chan, Kwangsoo Han, Andrew B. Kahng, Jae-Gon Lee, and Siddhartha Nath. OCV-aware Top-level Clock Tree Optimization. In *Great Lakes Symposium on VLSI (GLSVLSI)*, pages 33–38, 2014.

[18] Tuck-Boon Chan, Andrew B. Kahng, and Jiajia Li. NOLO: A no-loop, predictive useful skew methodology for improved timing in IC implementation. In *International Symposium on Quality Electronic Design (ISQED)*, pages 504–509, 2014.

[19] K. Chaudhary and M. Pedram. Computing the area versus delay trade-off curves in technology mapping. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(12):1480–1489, 1995.

[20] Chung-Ping Chen, Chris Chu, and D. F. Wong. 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, July 1999.

[21] Zhi-Wei Chen and Jin-Tai Yan. Routability-constrained Multi-bit Flip-flop Construction for Clock Power Reduction. *Integration VLSI*, 46(3), June 2013.

[22] Yongseok Cheon, Pei-Hsin Ho, Andrew B. Kahng, Sherief Reda, and Qinke Wang. Power-aware placement. In *Design Automation Conference (DAC)*, pages 795–800, 2005.

[23] David Chinnery and Kurt Keutzer. Linear Programming for Sizing, Vth and Vdd Assignment. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 149–154, 2005.

[24] David Chinnery and Ankur Sharma. Integrating LR Gate Sizing in an Industrial Place-and-Route Flow. In *International Symposium on Physical Design (ISPD)*, page 39–48, 2022.

[25] Amit Chowdhary, Karthik Rajagopal, Satish Venkatesan, Tung Cao, Vladimir Tiourin, Yegna Parasuram, and Bill Halpin. How Accurately Can We Model Timing in a Placement Engine? In *Design Automation Conference (DAC)*, pages 801–806, 2005.

[26] Chris Chu and Yiu-Chung Wong. FLUTE: Fast Lookup Table Based Rectilinear Steiner Minimal Tree Algorithm for VLSI Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(1):70–83, Jan 2008.

[27] Jason Cong, Andrew Kahng, G. Robins, Majid Sarrafzadeh, and C.K. Wong. Provably Good Performance-Driven Global Routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 11(6):739–752, 07 1992.

[28] Jordi Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, 2003.

[29] Olivier Coudert. Gate Sizing for Constrained Delay/Power/Area Optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(4):465–472, 1997.

[30] Siad Daboul, Nicolai Hähnle, Stephan Held, and Ulrike Schorr. Provably Fast and Near-Optimum Gate Sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(12):3163–3176, 2018.

[31] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[32] Chao Deng, Yi-Ci Cai, and Qiang Zhou. Register Clustering Methodology for Low Power Clock Tree Synthesis. *Journal of Computer Science and Technology*, 30(2):391–403, Mar 2015.

[33] Rahul B. Deokar and Sachin S. Sapatnekar. A graph-theoretic approach to clock skew optimization. In *International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 407–410 vol.1, 1994.

[34] Ahran Dunsmoor and Dr. João Geada. Applications and Use of Stage-based OCV. *in EDA Designline*, May 21, 2012.

[35] Hans Eisenmann and Frank M. Johannes. Generic global placement and floorplanning. In *Design and Automation Conference (DAC)*, pages 269–274, 1998.

[36] Rickard Ewetz. A Clock Tree Optimization Framework with Predictable Timing Quality. In *Design Automation Conference (DAC)*, pages 1–6, 2017.

[37] Rickard Ewetz and Cheng-Kok Koh. A Useful Skew Tree Framework for Inserting Large Safety Margins. In *International Symposium on Physical Design (ISPD)*, page 85–92, 2015.

[38] Rickard Ewetz and Cheng-Kok Koh. Cost-Effective Robustness in Clock Networks Using Near-Tree Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(4):515–528, April 2015.

[39] Rickard Ewetz, Chuan Yean Tan, and Cheng-Kok Koh. Construction of Latency-Bounded Clock Trees. In *International Symposium on Physical Design (ISPD)*, page 81–88, 2016.

[40] Hamed Fatemi, Andrew B. Kahng, Hyein Lee, Jiajia Li, and Jose Pineda de Gyvez. Enhancing sensitivity-based power reduction for an industry IC design context. *Integration*, 66:96–111, 2019.

[41] John P. Fishburn. A depth-decreasing heuristic for combinational logic; or how to convert a ripple-carry adder into a carry-lookahead adder or anything in-between. In *Design Automation Conference (DAC)*, pages 361–364, 1990.

[42] John P. Fishburn. Clock Skew Optimization. *IEEE Transactions on Computers*, 39(7):945–951, 1990.

[43] John P. Fishburn and Alfred E. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In *International Conference on Computer-Aided Design (ICCAD)*, 2003.

[44] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. Drive Strength Aware Cell Movement Techniques for Timing Driven Placement. In *International Symposium on Physical Design (ISPD)*, pages 73–80, 2016.

[45] Guilherme Flach, Mateus Fogaça, Jucemar Monteiro, Marcelo Johann, and Ricardo Reis. Rsyn: An Extensible Physical Synthesis Framework. In *International Symposium on Physical Design (ISPD)*, pages 33–40, 2017.

[46] Guilherme Flach, Jucemar Monteiro, Mateus Fogaça, Julia Puget, Paulo Butzen, Marcelo Johann, and Ricardo Reis. An Incremental Timing-Driven Flow Using Quadratic Formulation for Detailed Placement. In *International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 1–6, Oct 2015.

[47] Guilherme Flach, Tiago Reimann, Gracieli Posser, Marcelo Johann, and Ricardo Reis. Effective Method for Simultaneous Gate Sizing and Vth Assignment Using Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(4):546–557, April 2014.

[48] T. Gao, Pravin M. Vaidya, and Chung Laung Liu. A performance driven macrocell placement algorithm. In *Design Automation Conference (DAC)*, pages 147–152, 1992.

[49] Soheil Ghiasi, Eli Bozorgzadeh, Po-Kuan Huang, Roozbeh Jafari, and Majid Sarrafzadeh. A Unified Theory of Timing Budget Management. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25:2364 – 2375, 12 2006.

[50] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Critical Path Generation with Path Constraints. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2021.

[51] Guannan Guo, Tsung-Wei Huang, Yibo Lin, and Martin Wong. GPU-accelerated Path-based Timing Analysis. In *Design Automation Conference (DAC)*, pages 721–726, 2021.

[52] Zizheng Guo, Tsung-Wei Huang, and Yibo Lin. GPU-Accelerated Static Timing Analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2020.

[53] Chrystian Guth, Vinicius Livramento, Renan Netto, Renan Fonseca, José Luís Güntzel, and Luiz Santos. Timing-Driven Placement Based on Dynamic Net-Weighting for Efficient Slack Histogram Compression. In *International Symposium on Physical Design (ISPD)*, pages 141–148, 2015.

[54] Matthew R. Guthaus, Gustavo Wilke, and Ricardo Reis. Revisiting Automated Physical Synthesis of High-performance Clock Networks. *ACM Transactions on Design Automation of Electronic Systems*, 18(2):31:1–31:27, April 2013.

[55] Bill Halpin, C. Y. Roger Chen, and Naresh Sehgal. Timing Driven Placement Using Physical Net Constraints. In *Design Automation Conference (DAC)*, pages 780–783, 2001.

[56] Greg Hamerly and Charles Elkan. Alternatives to the K-means Algorithm That Find Better Clusterings. In *International Conference on Information and Knowledge Management (CIKM)*, pages 600–607, 2002.

[57] Inhak Han, Daijoon Hyun, and Youngsoo Shin. Buffer insertion to remove hold violations at multiple process corners. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 232–237, 2016.

[58] Kwangsoo Han, Jiajia Li, Andrew B. Kahng, Siddhartha Nath, and Jongpil Lee. A Global-Local Optimization Framework for Simultaneous Multi-Mode Multi-Corner Clock Skew Variation Reduction. In *Design Automation Conference (DAC)*, pages 26:1–26:6, 2015.

[59] Jiayuan He, Martin Burtscher, Rajit Manohar, and Keshav Pingali. SPRoute: A Scalable Parallel Negotiation-based Global Router. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[60] Stephan Held. Gate Sizing for Large Cell-based Designs. In *Design, Automation and Test in Europe (DATE)*, pages 827–832, 2009.

[61] Yi-Ju Ho and Wai-Kei Mak. Power and density-aware buffer insertion. In *VLSI Design, Automation and Test (VLSI-DAT)*, pages 287–290, 2008.

[62] Wenting Hou, Dick L. Liu, and Pei-Hsin Ho. Automatic Register Banking for Low-power Clock Trees. In *International Symposium on Quality of Electronic Design (ISQED)*, pages 647–652, 2009.

[63] Jin Hu, Andrew B. Kahng, SeokHyeong Kang, Myung-Chul Kim, and Igor L. Markov. Sensitivity-Guided Metaheuristics for Accurate Discrete Gate Sizing. In *International Conference on Computer-Aided Design (ICCAD)*, page 233–239, 2012.

[64] Shiyan Hu, Mahesh Ketkar, and Jiang Hu. Gate Sizing For Cell Library-Based Designs. In *Design Automation Conference (DAC)*, pages 847–852, June 2007.

[65] Shiyan Hu, Zhuo Li, and Charles J. Alpert. A Faster Approximation Scheme for Timing Driven Minimum Cost Layer Assignment. In *International Symposium on Physical Design (ISPD)*, page 167–174, 2009.

[66] Shiyan Hu, Zhuo Li, and Charles J. Alpert. A fully polynomial time approximation scheme for timing driven minimum cost buffer insertion. In *Design Automation Conference (DAC)*, pages 424–429, 2009.

[67] Chau-Chin Huang, Yen-Chun Liu, Yu-Sheng Lu, Yun-Chih Kuo, Yao-Wen Chang, and Sy-Yen Kuo. Timing-driven Cell Placement Optimization for Early Slack Histogram Compression. In *Design Automation Conference (DAC)*, 2016.

[68] Shih-Hsu Huang, Guan-Yu Jhuo, and Wei-Lun Huang. Minimum buffer insertions for clock period minimization. In *International Symposium on Computer, Communication, Control and Automation (3CA)*, volume 1, pages 426–429, 2010.

[69] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin D. F. Wong. OpenTimer v2: A New Parallel Incremental Timing Analysis Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(4):776–789, 2021.

[70] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems*, 33:1303–1320, 2022.

[71] Tsung-Wei Huang, Yibo Lin, Chun-Xun Lin, Guannan Guo, and Martin D. F. Wong. Cpp-Taskflow: A General-Purpose Parallel Task Programming System at Scale. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(8):1687–1700, 2021.

[72] Tsung-Wei Huang and Martin D. F. Wong. OpenTimer: A high-performance timing analysis tool. In *International Conference on Computer-Aided Design (ICCAD)*, pages 895–902.

[73] Aaron P. Hurst, Philip Chong, and Andreas Kuehlmann. Physical placement driven by sequential timing analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 379–386, 2004.

[74] Anil K. Jain. Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651 – 666, 2010.

[75] Kwangok Jeong, Andrew B. Kahng, and Hailong Yao. Revisiting the Linear Programming Framework for Leakage Power vs. Performance Optimization. In *International Symposium on Quality Electronic Design (ISQED)*, pages 127–134, 2009.

[76] Yanbin Jiang, Sachin S. Sapatnekar, Cyrus S. Bamji, and Juho Kim. Interleaving buffer insertion and transistor sizing into a single optimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(4):625–633, 1998.

[77] Jinwook Jung, Gi-Joon Nam, Lakshmi N. Reddy, Iris Hui-Ru Jiang, and Youngsoo Shin. OWARU: Free Space-Aware Timing-Driven Incremental Placement With Critical Path Smoothing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(9):1825–1838, Sep. 2018.

[78] Andrew B. Kahng, Seokhyeong Kang, Hyein Lee, Igor L. Markov, and Pankit Thapar. High-performance Gate Sizing with a Signoff Timer. In *International Conference on Computer-Aided Design (ICCAD)*, pages 450–457, 2013.

[79] Andrew B. Kahng, Jiajia Li, and Lutong Wang. Improved Flop Tray-based Design Implementation for Power Reduction. In *International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2016.

[80] Andrew B. Kahng and Qinke Wang. An analytic placer for mixed-size placement and timing-driven placement. In *International Conference on Computer-Aided Design (ICCAD)*, pages 565–572, 2004.

[81] Myung-Chul Kim, Jin Hu, Jiajia Li, and Natarajan Viswanathan. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In *International Conference on Computer-Aided Design (ICCAD)*, pages 921–926, 2015.

[82] Seungwon Kim, SangGi Do, and Seokhyeong Kang. Fast Predictive Useful Skew Methodology for Timing-Driven Placement Optimization. In *Design Automation Conference (DAC)*, pages 55:1–55:6, 2017.

[83] Ted Kirkpatrick and Norman Ross Clark. Pert as an Aid to Logic Design. *IBM Journal of Research and Development*, 10(2):135–141, 1966.

[84] Tim Kong. A novel net weighting algorithm for timing-driven placement. In *International Conference on Computer-Aided Design (ICCAD)*, pages 172–176, 2002.

[85] Luciano Lavagno, Igor L. Markov, Grant Martin, and Louis K. Scheffer. *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*. Taylor and Francis group, 2016.

[86] Taehee Lee, David Z. Pan, and Joon-Sung Yang. Clock Network Optimization With Multibit Flip-Flop Generation Considering Multicorner Multimode Timing Constraint. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):245–256, Jan 2018.

[87] Li Li, Peng Kang, Yinghai Lu, and Hai Zhou. An efficient algorithm for library-based cell-type selection in high-performance. In *International Conference on Computer-Aided Design (ICCAD)*, pages 226–232, 2012.

[88] John Lillis, Chung-Kuan Cheng, and Ting-Ting Y. Lin. Optimal wire sizing and buffer insertion for low power and a generalized delay model. *IEEE Journal of Solid-State Circuits*, 31(3):437–447, 1996.

[89] Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. DREAMPIace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement. In *Design Automation Conference (DAC)*, pages 1–6, 2019.

[90] Yibo Lin, Wuxi Li, Jiaqi Gu, Haoxing Ren, Brucek Khailany, and David Z. Pan. ABCDPlace: Accelerated Batch-Based Concurrent Detailed Placement on Multithreaded CPUs and GPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):5083–5096, 2020.

[91] I-Min Liu, A. Aziz, D.F. Wong, and Hai Zhou. An efficient buffer insertion algorithm for large networks based on Lagrangian relaxation. In *International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 210–215, 1999.

[92] Yifang Liu and Jiang Hu. 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, 2010.

[93] Yifang Liu and Jiang Hu. GPU-Based Parallelization for Fast Circuit Optimization. *ACM Transactions on Design Automation of Electronic Systems*, 16(3):24:1–24:14, June 2011.

[94] Yifang Liu, Jiang Hu, and Weiping Shi. Multi-Scenario Buffer Insertion in Multi-Core Processor Designs. In *International Symposium on Physical Design (ISPD)*, page 15–22, 2008.

[95] Vinicius Livramento, Derong Liu, Salim Chowdhury, Bei Yu, Xiaoqing Xu, David Z. Pan, José Luís Güntzel, and Luiz C. V dos Santos. Incremental Layer Assignment Driven by an External Signoff Timing Engine. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(7):1126–1139, 2017.

[96] Vinicius Livramento, Renan Netto, Chrystian Guth, José Luís Güntzel, and Luiz C. V. Dos Santos. Clock-Tree-Aware Incremental Timing-Driven Placement. *ACM Transactions on Design Automation of Electronic Systems*, 21(3):38:1–38:27, April 2016.

[97] Vinicius S. Livramento, Chrystian Guth, José Luís Güntzel, and Marcelo O. Johann. A Hybrid Technique for Discrete Gate Sizing Based on Lagrangian Relaxation. *ACM Transactions on Design Automation of Electronic Systems*, 19(4), August 2014.

[98] Vinicius S. Livramento, Chrystian Guth, José Luís Güntzel, and Marcelo O. Johann. Fast and efficient Lagrangian Relaxation-based Discrete Gate Sizing. In *Design, Automation Test in Europe (DATE)*, pages 1855–1860, 2013.

[99] Jianchao Lu and Baris Taskin. Post-CTS Clock Skew Scheduling with Limited Delay Buffering. In *International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 224 – 227, 09 2009.

[100] Lee-Chung Lu. Physical Design Challenges and Innovations to Meet Power, Speed, and Area Scaling Trend. In *International Symposium on Physical Design (ISPD)*, pages 63–63, 2017.

[101] Yi-Chen Lu, Siddhartha Nath, Vishal Khandelwal, and Sung Kyu Lim. RL-Sizer: VLSI Gate Sizing for Timing Optimization using Deep Reinforcement Learning. In *Design Automation Conference (DAC)*, pages 733–738, 2021.

[102] Chiao-Ling Lung, Hai-Chi Hsiao, Zi-Yi Zeng, and Shih-Chieh Chang. LP-based multi-mode multi-corner clock skew optimization. In *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, pages 335–338, 2010.

[103] Tao Luo, David Newmark, and David Z. Pan. A new LP based incremental timing driven placement for high performance designs. In *Design Automation Conference (DAC)*, pages 1115–1120, July 2006.

[104] Nancy D. MacDonald. Timing Closure in Deep Submicron Designs. In *Design Automation Conference (DAC)*, 2010.

[105] Dimitrios Mangiras, David Chinnery, and Giorgos Dimitrakopoulos. Task-based Parallel Programming for Gate Sizing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2022.

[106] Dimitrios Mangiras and Giorgos Dimitrakopoulos. Incremental Lagrangian Relaxation based Discrete Gate Sizing and Threshold Voltage Assignment. In *International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–5, 2021.

[107] Dimitrios Mangiras and Giorgos Dimitrakopoulos. Incremental Lagrangian Relaxation Based Discrete Gate Sizing and Threshold Voltage Assignment. *Technologies*, 9(4), 2021.

[108] Dimitrios Mangiras, Pavlos Mattheakis, Pierre-Olivier Ribet, and Giorgos Dimitrakopoulos. Soft-Clustering Driven Flip-Flop Placement Targeting Clock-Induced OCV. In *International Symposium on Physical Design (ISPD)*, page 25–32, 2020.

[109] Dimitrios Mangiras, Apostolos Stefanidis, Ioannis Seitanidis, Chrysostomos Nicopoulos, and Giorgos Dimitrakopoulos. Timing-Driven Placement Optimization Facilitated by Timing-Compatibility Flip-Flop Clustering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2835 – 2848, Oct. 2020.

[110] Yehdhih Ould Mohammed Moctar and Philip Brisk. Parallel FPGA routing based on the operator formulation. In *Design Automation Conference (DAC)*, pages 1–6, 2014.

[111] Michael D. Moffitt, David A. Papa, Zhuo Li, and Charles J. Alpert. Path Smoothing via Discrete Optimization. In *Design Automation Conference (DAC)*, page 724–727, 2008.

[112] Juan Antonio Montiel-Nelson, Javier Sosa, Héctor Navarro, Roberto Sarmiento, and Antonio Núñez. Efficient method to obtain the entire active area against circuit delay time trade-off curve in gate sizing. *IEE Proceedings - Circuits, Devices and Systems*, 152:133–145, 2005.

[113] David Nguyen, Abhijit Davare, Michael Orshansky, David Chinnery, Brandon Thompson, and Kurt Keutzer. Minimization of Dynamic and Static Power Through Joint Assignment of Threshold Voltages and Sizing Optimization. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 158–163, 2003.

[114] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *Symposium on Operating Systems Principles (SOSP)*, pages 456–471, 2013.

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

[116] Muhammet Mustafa Ozdal, Chirayu Amin, Andrey Ayupov, Steven M. Burns, Gustavo R. Wilke, and Cheng Zhuo. An Improved Benchmark Suite for the ISPD-2013 Discrete Cell Sizing Contest. In *International Symposium on Physical Design (ISPD)*, page 168–170, 2013.

[117] Muhammet Mustafa Ozdal, Steven Burns, and Jiang Hu. Gate sizing and device technology selection algorithms for high-performance industrial designs. In *International Conference on Computer-Aided Design (ICCAD)*, pages 724–731, 2011.

[118] Muhammet Mustafa Ozdal, Steven Burns, and Jiang Hu. Algorithms for Gate Sizing and Device Parameter Selection for High-Performance Designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(10):1558–1571, October 2012.

[119] Uday Padmanabhan, Janet Meiling Wang, and Jiang Hu. Robust Clock Tree Routing in the Presence of Process Variations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(8):1385–1397, Aug 2008.

[120] David Papa, Charles Alpert, Cliff Sze, Zhuo Li, Natarajan Viswanathan, Gi-Joon Nam, and Igor Markov. Physical Synthesis with Clock-Network Optimization for Large Systems on Chips. *IEEE Micro*, 31(4):51–62, 2011.

[121] David A. Papa, Tao Luo, Michael D. Moffitt, Cliff C. N. Sze, Zhuo Li, Gi-Joon Nam, Charles J. Alpert, and Igor L. Markov. RUMBLE: An Incremental Timing-Driven Physical-Synthesis Optimization Algorithm. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2156–2168, 2008.

[122] Lawrence T. Pillage and Ronald A. Rohrer. Asymptotic waveform evaluation for timing analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(4):352–366, 1990.

[123] Julia Casarin Puget, Guilherme Flach, Marcelo Johann, and Ricardo Reis. Jezz: An Effective Legalization Algorithm for Minimum Displacement. In *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pages 1–5, Aug 2015.

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

[125] Karthik Rajagopal, Tal Shaked, Yegna Parasuram, Tung Cao, Amit Chowdhary, and Bill Halpin. Timing Driven Force Directed Placement with Physical Net Constraints. In *International Symposium on Physical Design (ISPD)*, page 60–66, 2003.

[126] Anand Rajaram and David Z. Pan. Variation Tolerant Buffered Clock Network Synthesis with Cross Links. In *International Symposium on Physical Design (ISPD)*, pages 157–164, 2006.

[127] Anand Rajaram and David Z. Pan. Robust Chip-level Clock Tree Synthesis for SOC Designs. In *Design Automation Conference (DAC)*, pages 720–723, 2008.

[128] Venky Ramachandran. Construction of minimal functional skew clock trees. In *International Symposium on Physical Design (ISPD)*, page 119–120, 2012.

[129] Tiago Reimann, Gracieli Posser, Guilherme Flach, Marcelo Johann, and Ricardo Reis. Simultaneous gate sizing and Vt assignment using Fanin/Fanout ratio and Simulated Annealing. In *International Symposium on Circuits and Systems (ISCAS)*, pages 2549–2552, 2013.

[130] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., USA, 2007.

[131] Haoxing Ren, David Z. Pan, Charles J. Alpert, Gi-Joon Nam, and Paul Villarrubia. Hippocrates: First-Do-No-Harm Detailed Placement. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 141–146, 2007.

[132] Haoxing Ren, David Z. Pan, and David S. Kung. Sensitivity Guided Net Weighting for Placement Driven Synthesis. In *International Symposium on Physical Design (ISPD)*, pages 10–17, 2004.

[133] Bernhard M. Riess and Gisela G. Ettelt. SPEED: fast and efficient timing driven placement. In *International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 377–380, 1995.

[134] Subhendu Roy, Derong Liu, Jagmohan Singh, Junhyung Um, and David Z. Pan. OSFA: A New Paradigm of Aging Aware Gate-Sizing for Power/Performance Optimizations Under Multiple Operating Conditions. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35:1618–1629, Oct 2016.

[135] Subhendu Roy, Derong Liu, Junhyung Um, and David Z. Pan. OSFA: A new paradigm of gate-sizing for power/performance optimizations under multiple operating conditions. In *Design Automation Conference (DAC)*, pages 1–6, 2015.

[136] Subhendu Roy, Pavlos Mattheakis, Laurent Masse-Navette, and David Z. Pan. Clock Tree Resynthesis for Multi-Corner Multi-Mode Timing Closure. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(4):589–602, 2015.

[137] Richard L. Rudell. *Logic Synthesis for VLSI Design*. PhD thesis, EECS Department, University of California, Berkeley, Apr 1989.

[138] Sachin Sapatnekar, Vasant Rao, Pravin Vaidya, and Sung-Mo Kang. An exact solution to the transistor sizing problem for CMOS circuits using convex optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(11):1621–1634, 1993.

[139] Ioannis Seitanidis, Giorgos Dimitrakopoulos, Pavlos Mattheakis, Laurent Masse-Navette, and David Chinnery. Timing-Driven and Placement-Aware Multi-Bit Register Composition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(8):1501–1514, Aug 2019.

[140] Ankur Sharma, David Chinnery, Sarvesh Bhardwaj, and Chris Chu. Fast Lagrangian Relaxation Based Gate Sizing Using Multi-Threading. In *International Conference on Computer-Aided Design (ICCAD)*, pages 426–433, 2015.

[141] Ankur Sharma, David Chinnery, and Chris Chu. Lagrangian Relaxation Based Gate Sizing With Clock Skew Scheduling - A Fast and Effective Approach. In *International Symposium on Physical Design (ISPD)*, pages 129–137, 2019.

[142] Ankur Sharma, David Chinnery, Shrirang Dhamdhere, and Chris Chu. Rapid gate sizing with fewer iterations of Lagrangian Relaxation. In *International Conference on Computer-Aided Design (ICCAD)*, pages 337–343, 2017.

[143] Ankur Sharma, David Chinnery, Tiago Reimann, Sarvesh Bhardwaj, and Chris Chu. Fast Lagrangian Relaxation-Based Multithreaded Gate Sizing Using Simple Timing Calibrations. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(7):1456–1469, 2019.

[144] Rupesh S. Shelar. An Efficent Clustering Algorithm for Low Power Clock Tree Synthesis. In *International Symposium on Physical Design (ISPD)*, pages 181–188, 2007.

[145] Narendra Shenoy. Retiming: Theory and practice. *Integration*, 22(1):1–21, 1997.

[146] Gregory Shklover and Ben Emanuel. Simultaneous Clock and Data Gate Sizing Algorithm with Common Global Objective. In *International Symposium on Physical Design (ISPD)*, pages 145–152, 2012.

[147] Kanwar Jit Singh, Albert R. Wang, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli. Timing optimization of combinational logic. In *International Conference on Computer-Aided Design (ICCAD)*, pages 282–285, 1988.

[148] Peter Spindler, Ulf Schlichtmann, and Frank M. Johannes. Abacus: Fast legalization of standard cell circuits with minimal movement. In *International Symposium on Physical Design (ISPD)*, pages 47–53, 2008.

[149] Arvind Srinivasan, Kamal Chaudhary, and Ernest S. Kuh. RITUAL: a performance driven placement algorithm for small cell ICs. In *International Conference on Computer-Aided Design (ICCAD)*, pages 48–51, 1991.

[150] Apostolos Stefanidis, Dimitrios Mangiras, Chrysostomos Nicopoulos, David Chinnery, and Giorgos Dimitrakopoulos. Design Optimization by Fine-Grained Interleaving of Local Netlist Transformations in Lagrangian Relaxation. In *International Symposium on Physical Design (ISPD)*, pages 87–94, 2020.

[151] Apostolos Stefanidis, Dimitrios Mangiras, Chrysostomos Nicopoulos, David Chinnery, and Giorgos Dimitrakopoulos. Autonomous Application of Netlist

Transformations inside Lagrangian Relaxation-based Optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(8):1672–1686, August 2021.

[152] Hiran Tennakoon and Carl Sechen. Gate sizing using Lagrangian relaxation combined with a fast gradient-based pre-processing step. In *International Conference on Computer-Aided Design (ICCAD)*, pages 395–402, 2002.

[153] Chung-Wen Albert Tsao and Cheng-Kok Koh. UST/DME: A Clock Tree Router for General Skew Constraints. In *International Conference on Computer Aided Design (ICCAD)*, pages 400–405, 2000.

[154] Ren-Song Tsay and Juergen Koehl. An Analytic Net Weighting Approach for Performance Optimization in Circuit Placement. In *Design Automation Conference (DAC)*, page 620–625, 1991.

[155] Wen-Pin Tu, Chung-Han Chou, Shih-Hsu Huang, Shih-Chieh Chang, Yow-Tyng Nieh, and Chien-Yung Chou. Low-power timing closure methodology for ultra-low voltage designs. In *International Conference on Computer-Aided Design (ICCAD)*, pages 697–704, 2013.

[156] Necati Uysal and Rickard Ewetz. OCV Guided Clock Tree Topology Reconstruction. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 494–499, 2018.

[157] Lukas P.P.P. van Ginneken. Buffer placement in distributed RC-tree networks for minimal Elmore delay. In *International Symposium on Circuits and Systems (ISCAS)*, pages 865–868, 1990.

[158] Natarajan Viswanathan, Gi-Joon Nam, Jarrod A. Roy, Zhuo Li, Charles J. Alpert, Shyam Ramji, and Chris Chu. ITOP: Integrating Timing Optimization Within Placement. In *International Symposium on Physical Design (ISPD)*, pages 83–90, 2010.

[159] Kai Wang and Malgorzata Marek-Sadowska. Potential Slack Budgeting with Clock Skew Optimization. In *International Conference on Computer Design (ICCD)*, pages 265– 271, 2004.

[160] Kui Wang, Hao Fang, Hu Xu, and Xu Cheng. A Fast Incremental Clock Skew Scheduling Algorithm for Slack Optimization. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, page 492–497, 2008.

[161] Xinsheng Wang, Wenpan Liu, and Mingyan Yu. A distinctive O(mn) time algorithm for optimal buffer insertions. In *International Symposium on Quality Electronic Design (ISQED)*, pages 293–297, 2015.

[162] Xinjie Wei, Yici Cai, and Xianlong Hong. Effective Acceleration of Iterative Slack Distribution Process. In *International Symposium on Circuits and Systems (ISCAS)*, pages 1077–1080, 2007.

[163] Carl Sechen William Swartz. Timing Driven Placement for Large Standard Cell Circuits. In *Design Automation Conference (DAC)*, pages 211–215, 1995.

[164] Anthony D. Williams. *C++ Concurrency in Action: Practical Multithreading*. Manning Pubs Co Series. Manning, 2012.

[165] Gang Wu and Chris Chu. Two Approaches for Timing-Driven Placement by Lagrangian Relaxation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(12):2093–2105, Dec 2017.

[166] Gang Wu, Yuehuan Xu, Dean Wu, Manoj Ragupathy, Yu-Yen Mo, and Chris Chu. Flip-flop clustering by weighted K-means algorithm. In *Design Automation Conference (DAC)*, pages 1–6, 2016.

[167] Pei-Ci Wu, Martin D. F. Wong, Ivailo Nedelchev, Sarvesh Bhardwaj, and Vidyamani Parkhe. On timing closure: Buffer insertion for hold-violation removal. In *Design Automation Conference (DAC)*, pages 1–6, 2014.

[168] Yue Xu, Yanheng Zhang, and Chris Chu. FastRoute 4.0: Global router with efficient via minimization. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 576–581, 2009.

[169] Bin Zhang. Generalized K-Harmonic Means – Boosting in Unsupervised Learning. Technical report, HPL-2000-137, Hewlett-Packard Labs, 2000.