Algorithms for Circuit Sizing in VLSI Design

Dissertation
zur
Erlangung des Doktorgrades (Dr. rer. nat.)
der
Mathematisch-Naturwissenschaftlichen Fakultät
der
Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von
Ulrike Elisabeth Schorr, geb. Suhl
aus
Grünstadt

Bonn, Dezember 2015

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen
Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter:	Prof. Dr. Jens Vygen
2. Gutachter:	Prof. Dr. Stephan Held
Tag der Promotion:	11. März 2016
Erscheinungsjahr:	2016

Acknowledgments

This work would not have been possible without the support of many people.
First and foremost, I would like to express my gratitude to my supervisors Professor Dr. Jens Vygen and Professor Dr. Stephan Held for their extensive support, and valuable ideas and feedback.
Special thanks go to Professor Dr. Bernhard Korte for providing outstanding working conditions at the Research Institute for Discrete Mathematics at the University of Bonn.
I also wish to thank my past and present colleagues at the institute for the friendly working atmosphere and productive collaboration over the past years. It was a pleasure working in the timing optimization team on various topics.
In particular I would like to thank Dr. Nicolai Hähnle and Daniel Rotter for their support and fruitful discussions.
Further thanks go to Dr. Dirk Müller and Rudi Scheifele for helpful conversations on resource sharing, and to Dr. Ulrich Brenner and Dr. Jan Schneider for their helpful feedback.
I also like to thank past and present students from the timing optimization team for the collaboration, in particular Siad Daboul, Nikolas Kämmerling and Alexander Timmermeister.
I am thankful to all people at IBM who shared their knowledge on VLSI design, especially Karsten Muuss, Lakshmi Reddy and Alexander J. Suess.
I am further grateful to Dr. Ulrich Brenner, Dr. Nicolai Hähnle, Dr. Dirk Müller, Daniel Rotter, Rudi Scheifele and Dr. Jan Schneider for proofreading parts of this thesis. The remarks have been a huge help.
My personal thanks go to my family and friends for their patience and assistance while finishing this thesis, and the reminders that not everything is about chip design.
I wholeheartedly thank my parents and my brother Christian for the best possible support in the past years.
Last but not least I am grateful to my husband Florian for being there, and the never-ending encouragement for half of my life.

To say it with the words of a famous song:

"I've had the timing of my life" [1]

[1] Freely adapted from "(I've Had) The Time of My Life" composed by F. Previte, J. DeNicola, D. Markowitz and recorded by B. Medley and J. Warnes, 1987.

Contents

1 Introduction

The basic building blocks of *computer chips*, also known as *integrated circuits*, are electronic switches called *transistors*. Transistors are connected to realize *circuits* and other features on the chip. Circuits in the combinatorial logic, called *gates*, perform the binary computations, and results are stored in *memory circuits* (*registers*) for a certain amount of time. Since the first integrated circuit was built at Texas Instruments in 1958, the complexity of computer chips has grown exponentially, and today's computer chips consist of millions of circuits and billions of transistors. For example, IBM's Power 8 CPU contains more than 4 billion transistors.
A key problem in the physical design of a computer chip is to choose a physical layout for the circuits. This is a complex task, and has high influence on the power consumption and area of the chip, and also on the speed of electrical signals.
A *library* offers a discrete set of predesigned layouts with different physical properties for each logic function and register type on the chip. The same layout can be used several times on the chip. The most influential characteristics of a circuit defined by the layout are its *size* and its *voltage threshold* (V_t *level*). The tasks to choose a size and V_t level for each circuit are referred to as *circuit sizing* and V_t *optimization*.
Different sizes are realized by varying the transistor areas. The V_t level defines the voltage at which the circuit *switches*, in other words a logical zero becomes a logical one or vice versa. Different V_t levels are realized for example by varying the fabrication material of the transistors, and usually only 3 or 4 levels are available.
While it was relatively easy to meet all constraints imposed on the speed of electrical signals (*timing constraints*) in the early days of chip design, a good choice of size and V_t level for all circuits is nowadays essential. This is illustrated in Figure 1.1, which depicts signal delay through an inverter circuit for different sizes and V_t levels.
Additionally, reducing the power consumption of a chip has become an increasingly important objective in physical design due to the increasing number of transistors on the chip and the continuing technology scaling. Power consumption of a circuit can be divided into *dynamic* and *static* power. Both types scale linearly with the area of a circuit, while static power grows exponentially with a lower V_t level. Figure 1.2 shows the static power consumption of an inverter circuit for different sizes and V_t levels.
Circuit sizing and V_t optimization have been studied extensively, and various heuristic algorithms exist. Both problems were shown to be *NP*-hard for example by Li [Li94]. The continuous relaxation of the circuit sizing problem can be formulated as a convex program and solved in polynomial time. This relaxation poses an in-

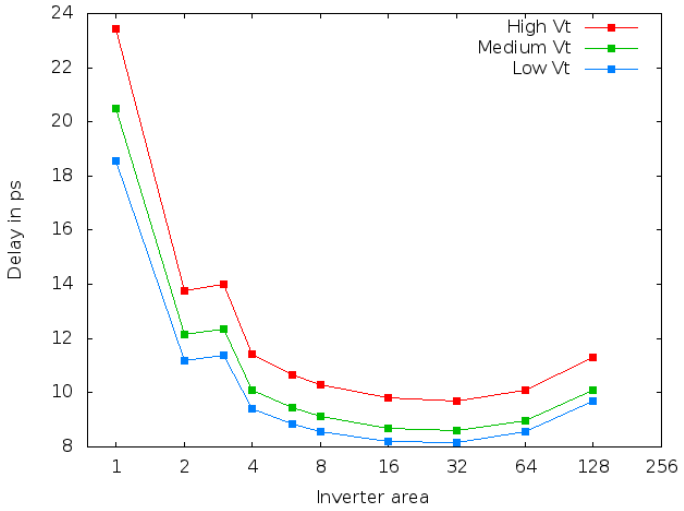


Figure 1.1: Signal delay through an inverter for different sizes and V_t levels, taken from an ISPD 2013 benchmark (Ozdal et al. [Ozd+13]) with a clock cycle time of 300 ps. The delay peak at area 3 is due to the internal structure of the inverter.

teresting challenge to researchers because standard interior point solvers fail for the huge instance sizes occurring in practice.

While both problems have often been treated separately, there is a tendency to optimize them simultaneously. Recently, Intel researchers published realistic benchmark suites for the ISPD 2012 and 2013 Discrete Gate Sizing Contests (Ozdal et al. [Ozd+12; Ozd+13]) that enabled comparison of different approaches for gate sizing and V_t optimization and triggered ongoing research. While none of the contestants dominated on all 2012 benchmarks, the winner team of the 2013 contest achieved the best static power consumption on most benchmarks, and further improved their results in Flach et al. [Fla+14]. The underlying algorithm is a discretized approach based on *Lagrangian relaxation* of the convex program, which has been popular in practice since the groundwork paper of Chen, Chu and Wong [CCW99]. The basic concept consists of using weights that model criticalities of the timing constraints. These weights are updated iteratively, and an oracle algorithm that is guided by these weights computes intermediate solutions until a good solution has been found. Several seemingly heuristic modifications have been proposed to improve the performance of this approach in practice, see for example Tennakoon and Sechen [TS02; TS08], Livramento et al. [Liv+14], Flach et al. [Fla+14].

The main contributions of this thesis are a theoretical analysis of these modifications and the subsequent proposal of a new model for gate sizing as a *min-max resource sharing problem*. With the new model we obtain a fast approximation for the continuous relaxation that improves over the Lagrangian relaxation approach. Under certain assumptions the running time is polynomial. Our experiments illus-

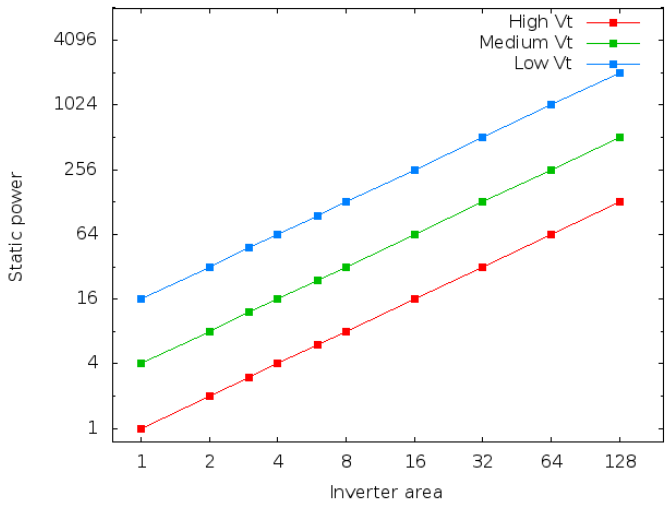


Figure 1.2: Static power consumption of an inverter for different sizes and V_t levels, taken from the ISPD 2013 benchmark library (Ozdal et al. [Ozd+13]).

trate that the new algorithm exhibits the better convergence behavior and results.
This thesis is organized as follows:
Chapter 2 introduces fundamentals and technological aspects of modern computer chip design with a focus on timing optimization in the physical design phase.
Chapter 3 restates known concepts and results from convex optimization.
Gate sizing and V_t optimization are formally introduced in Chapter 4. The chapter further contains an overview of previous work.
Chapter 5 deals with the *power-delay tradeoff problem* that aims to find layouts minimizing a weighted sum of power and signal delays. We will encounter this as a subproblem in the Lagrangian relaxation and resource sharing algorithm. We describe a method that approximates the value of this tradeoff function in pseudopolynomial time for the continuous relaxation. For the discrete problem we provide a fully polynomial approximation scheme under certain assumptions on the topology of the chip.
We give the first comprehensive discussion of the Lagrangian relaxation approach in Chapter 6 and fill gaps in the convergence analysis. Moreover, we show that additional constraints on the local density of circuits on the chip and electrical integrity can also be incorporated into this framework.
In Chapter 7 we analyze heuristic modifications that are usually applied to the Lagrangian relaxation approach in practice. This leads us to the *multiplicative weights method* that implies a certain update rule for the weights, and we use this method to give the first theoretical justification of some of the modifications.
The new model for gate sizing as a min-max resource sharing problem is presented in Chapter 8. This is a well-known problem in mathematical optimization and consists of distributing a limited set of resources among a limited set of customers

who compete for the resources. An optimal solution distributes the resources in such a way that the maximum resource usage is minimized. The model is successfully applied to other problems in chip design, and the fastest algorithm is a variant of the multiplicative weights algorithm (Müller et al. [MRV11]). In our context, the resources are power consumption and signal delays. Although it seems natural to model each gate as a customer, we show that this is not possible, but a single customer representing all gates is sufficient. With this model we obtain a fast approximation for the continuous relaxation.
We further draw comparisons between the performance and running time of the new algorithm and existing ones, and discuss extensions of this model.
Additionally, we implemented a Lagrangian relaxation and resource sharing algorithm for gate sizing and V_t optimization and conducted experiments on the ISPD 2013 benchmarks and state-of-the-art microprocessor designs provided by our industrial partner IBM. Chapter 9 describes our implementations and experiments, which show that the new algorithm improves over our Lagrangian relaxation based implementation. Both algorithms are part of the BonnTools software package, which is developed at the Research Institute for Discrete Mathematics at the University of Bonn in cooperation with IBM.
Finally, we consider an algorithm for timing-driven optimization of memory circuits in Chapter 10. Their sizes and locations on the chip are usually determined during the clock network design phase. As redesigning the clock network is expensive, these remain mostly unchanged afterwards although the timing criticalities on which they were based can change. Our algorithm can be applied after this phase without impairing the clock network, and improves timing of memory circuits on microprocessor designs by up to 7.8% of design cycle time.

2 Timing Optimization in VLSI Design

In this chapter we introduce the fundamentals and technological aspects of modern computer chip design. Today's computer chips consist of millions of tiny modules called *circuits* which implement logic functions or memory elements and are realized by transistors. Our focus is on timing optimization in the physical design phase, and related concepts and notation. Timing optimization algorithms aim to optimize the electrical signals traversing the chip and comprise for example circuit sizing, repeater tree insertion, layer assignment etc.
A comprehensive introduction to modern CMOS VLSI design can be found in Weste and Harris [WH10]. Kahng et al. [Kah+11] and Held [Hel08] give an overview over the VLSI physical design phase.
For graph theory and combinatorial optimization we use the notation from the book *Combinatorial Optimization* by Korte and Vygen [KV12].

2.1 Transistors and Circuits

2.1.1 Transistors

Transistor

Transistors can be seen as electronic switches with three *terminals* called *source*, *drain* and *gate*. A voltage applied to the control terminal (gate) determines if source and drain are connected such that a current can flow between them, or if the transistor is insulating.
The first transistor was build in 1947 by John Bardeen and Walter Brattain at Bell Laboratories. Although there exists a large number of different technical implementations of a transistor, one can distinguish two substantially different types , namely *n-type* and *p-type* transistors. In an n-type transistor, source and drain are connected only if a voltage is applied to the control terminal. A p-type transistor behaves conversely. Modern designs apply *CMOS (Complementary Metal Oxide Semiconductor)* technology to build the circuits on a chip which implement logic functions and memory elements. In this technology, both n-type and p-type transistors are used to realize a circuit. Figure 2.1 shows a sketch of an n-type metal-oxide semiconductor transistor: The n-type source and drain are adjacent to the polysilicon gate (originally made of metal). Additionally, the transistor consists of an insulating oxide layer which is usually made of glass, and the silicon wafer, also called *body*, which is of p-type here. If the voltage applied to the gate is high enough, a thin region below the gate is conducting and a current can flow from source to drain. For a p-type transistor, the situation is reversed.

n-type, p-type

CMOS

The positive voltage applied to the gate is usually called V_{dd} and represents a logic

V_{dd}

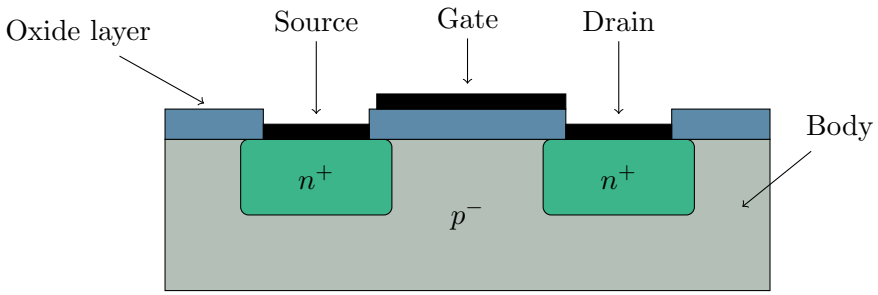


Figure 2.1: n-type metal-oxide semiconductor transistor.

V_0

1 value in digital circuits. The low voltage is called *ground* or V_0 and represents a logic 0 value. The voltage needed at the gate such that current can flow between source and drain is known as *voltage threshold* or V_t level with $V_0 < V_t < V_{dd}$. Different thresholds can be realized by varying the degree to which the body is doped, the thickness of the insulating oxide layer or the fabrication material of the oxide layer. Only a small number of V_t levels is available, as a separate production step is needed for each level. A lower V_t level implies a faster operating transistor but a higher power consumption of the transistor.

Voltage threshold (V_t level)

2.1.2 Circuits

Circuits, Cells

Transistors are connected to realize *circuits*, also called *cells*. We distinguish three main circuit classes according to their function:

Circuit classes

- Combinatorial logic,

Register

- memory circuits (registers) and

Clock driver

- clock drivers.

The *combinatorial logic* performs the binary computations of the chip. Each circuit in that class realizes a logic function like AND, NAND or INVERTER. Memory circuits store the binary information for a certain amount of time, and then feed it back to the combinatorial logic in form of an electrical signal, or the information leaves the chip. The clock drivers control when a memory circuit receives, stores or releases information by sending periodic clock signals. Usually a chip contains many small memory elements that can store one bit at a time (*flip-flops* or *latches*). Often a few large predesigned memory arrays that are able to store many bits simultaneously can be found on the chip.

Flip-flop, latch

The connection points of a circuit with the outer world are called *pins* and consist of a piece of metal (aluminium).

Pin

Figure 2.2 shows the schematic of a CMOS inverter with one n-type transistor connected to ground and one p-type transistor connected to V_{dd}. The gates of both transistors are connected to the input of the circuit, and their drains to the

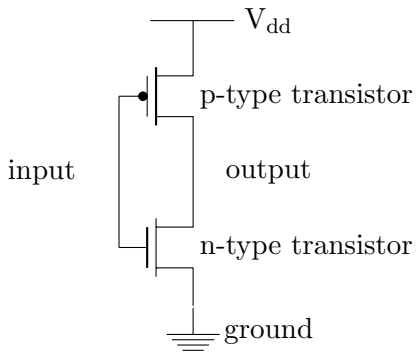


Figure 2.2: CMOS inverter

output. If the input voltage is high, representing a logical 1, the n-type transistor is open, i.e. there is a conducting channel between source and drain. The output of the circuit is then connected to ground, and represents a logical 0. Vice versa, if the input voltage is low, the p-type transistor is open and the output of the circuit is connected to V_{dd}.
Latches have connection points to receive and send binary information, and at least one input for a periodic control signal from a clock driver. The clock signals open and close the latch once per computation cycle. When the latch is open, the information at the data input pin can traverse the latch and is released at the data output pin.
In literature, one often encounters the term *(logic) gate* instead of circuit. Formally, a gate is a logic circuit representing an elementary boolean function with exactly one output signal like AND, NOR etc., and more complex circuits are treated as equivalent to several gates. However, usage of these terms is ambiguous, and often registers are also referred to as gates. In the remainder of this thesis we will refer to gates as circuits that compute a boolean function.

(Logic) gate

Circuit library design

Since the 90's, the focus shifted away from custom circuit design methods, where circuits and transistors were designed individually, to *circuit library design*. The circuit library offers a discrete set of predesigned layouts with different physical properties for each logic function and register type on the chip. This way, the same layout can be used for several circuits that implement the same logic function. Design optimizations are performed on the circuit-level rather than on the transistor-level. A reason for this shift is the increasing number of transistors on a chip. Usually, it is not worth the effort to design each of them individually, because estimating the behaviour of every transistor under realistic assumptions is time-consuming. For example, it requires solving differential equations to determine how fast a transistor can switch. The behaviour of predesigned layouts from the circuit library has usually been tested under realistic circumstances. For each

Circuit library

Timing rules

layout, *timing rules* provide information about the behaviour towards a voltage change. Furthermore, the same library can be used for many different chips.
Schneider [Sch14] gives an historical overview on the design of circuit libraries and presents a tool for fast automatic design of circuit layouts. We only consider digital designs based on circuit libraries in this thesis, as these constitute the majority of today's digital designs.
The most important characteristics of a circuit that influence the speed of electrical signals are its *size* and voltage threshold, also called V_t level. The circuit library provides several layouts for each circuit which implement various sizes and voltage thresholds. Different sizes are realized by modifying the width of the transistors, which also changes the electrical capacitance of the circuit. Thereby the relative sizes between the transistors remain constant. Different V_t levels can be realized by varying the voltage threshold of the transistors in the layout. As only a small number of V_t levels is available for each transistor, there is only a small number of V_t levels available for each circuit.

Circuit size
Circuit V_t level

Power Consumption

Each circuit consumes a certain amount of power which is largely impacted by its size and V_t level. We distinguish between two types of power consumption: The power consumed by a circuit when it is not switching is called *static power* or *leakage power*. The *dynamic power* of a circuit is defined as the power consumed by the circuit due to switching, and charging and discharging capacitive loads. The *total power* consumption, or simply power consumption, of a circuit is the sum of its static and its dynamic power consumption.

Static (leakage), dynamic, and total power

As transistors cannot be fully turned "off", they always leak a small amount of current. Static power grows exponentially with falling threshold voltage and is roughly proportional to

$$\frac{W}{L} \cdot e^{-v_t}, \tag{2.1}$$

where $\mathrm{V}_0 < v_t < \mathrm{V}_{\mathrm{dd}}$ denotes the V_t level, and W, L are the width and length of the circuit, more precisely of the underlying transistors (Sheu et al. [She+87]). This implies that static power consumption of a circuit depends linearly on its width and grows exponentially when lowering the V_t level. We left out some dependencies in (2.1), for example the dependency on thermal voltage which we regard as constant, and the dependency on the voltage at the input pins. Because all possible combinations of voltage states at the input pins (input patterns) cannot be evaluated, and due to the varying process parameters there is always a modeling error when the static power consumption of a circuit is computed.
The dynamic power consumption due to charging and discharging capacitances is roughly proportional to

$$f_{switch} \cdot \frac{1}{2} Cktcap \cdot {\mathrm{V}_{\mathrm{dd}}}^2, \tag{2.2}$$

where f_{switch} is the *switching frequency* of the circuit, i.e. how often the voltage changes at the circuit, and $Cktcap$ is the total capacitance of the circuit (see also Lee and Gupta [LG12]). The relation between the capacitance of a circuit and its area (the area of the underlying transistors) is approximately linear, hence the dynamic power consumption due to charging and discharging scales linearly with the circuit size. The second component of dynamic power is the *short-circuit* power which corresponds to the power that is lost internally when both p-type and n-type transistors are conducting for a short amount of time while the circuit is switching. Based on the α-power law (Sakurai and Newton [SN90]) it is roughly proportional to

Switching frequency

Short-circuit power

$$\tau \cdot \frac{W}{L} \cdot \frac{(\mathrm{V_{dd}} - 2v_t)^{\alpha+1}}{(\mathrm{V_{dd}} - v_t)^{\alpha}}, \tag{2.3}$$

where $\mathrm{V}_0 < v_t < \mathrm{V_{dd}}$ denotes the V_t level, τ is the input transition time, and α is a technology-dependent coefficient (Sakurai and Newton [SN90]). For fast transition times and high V_t levels, it is usually negligible.
In practice, several models are in place to estimate the power consumption of a circuit. We will follow up on these models in Section 2.6.

2.2 Integrated Circuit Design

The basic building blocks of *integrated circuits*, more commonly known as (computer) chips, are transistors. The transistors are fabricated on one piece of semiconductor material, normally silicon, and realize the circuits and other features on the chip. Figure 2.3 shows a computer chip with about 600000 circuits. Electrical wires that connect the transistors are contained on higher *layers* or *planes* of the chip. *Vias* connect these layers. In the manufacturing process, planes are built one by one in a lithographic process.

Integrated circuit

Layer, plane

Via

Electrical signals enter the chip at *primary input pins* (*primary inputs*) and are propagated through the combinatorial logic in each computation cycle until they reach register inputs or leave the chip at the *primary output pins* (*primary outputs*). Registers store the binary information until the next computation cycle begins, and a periodic *clock signal* determines whether a register is open or closed.

Primary input pins

Primary output pins

It is a complex task to design the *clock network* which distributes the clock signals. Often, it is implemented as a *clock tree* or a *clock grid*. A clock tree is a rooted binary tree whose leaves correspond to the registers. In a clock grid, the clock signal is distributed in a grid-like network where clock drivers dispense the clock signal. A chip may have several clock networks with different frequencies. We say that memory elements which are fed by the same clock signals belong to the same *clock domain*.

Clock network

Clock tree

Clock grid

Clock domain

The first integrated circuit was built by Jack Kilby in 1958 and contained two transistors. Since then, the complexity of computer chips has grown enormously: In 1965, Gordon E. Moore [Moo65] predicted that the number of components per

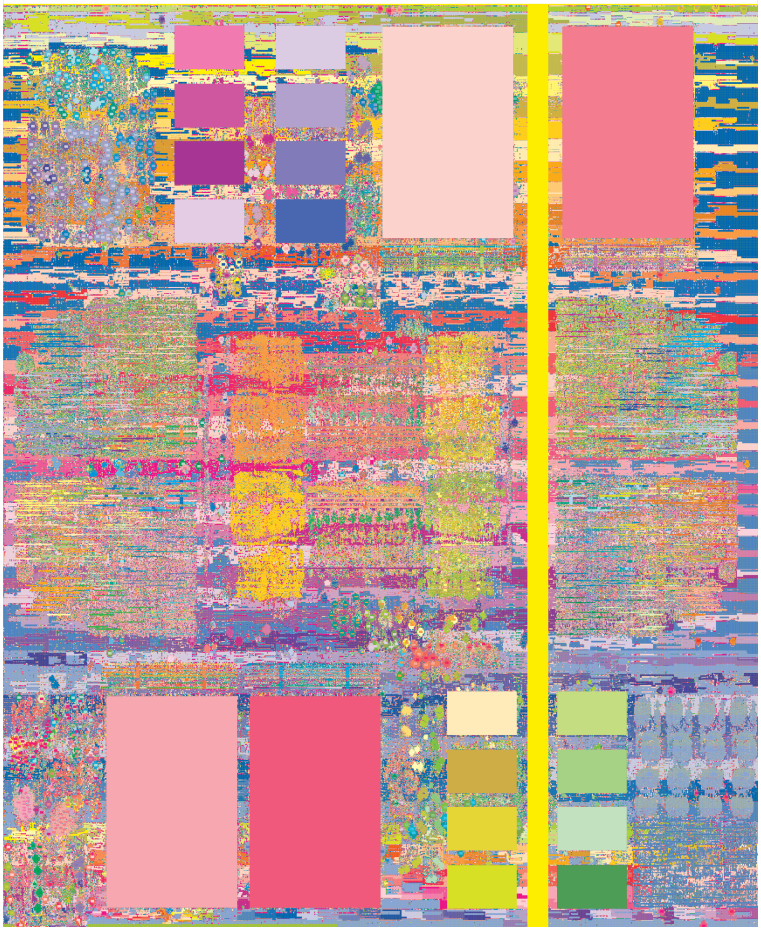


Figure 2.3: The placement of a computer chip with approximately 600000 circuits.

integrated circuit will double every year. This prediction was not completely fulfilled as the number of components doubled every two years, but nonetheless the complexity of computer chips has grown exponentially, and the prediction is today known as Moore's Law.

Moore's Law

Today's computer chips consist of billions of devices, for example IBM's POWER8 CPU contains 4 200 000 000 transistors. The term *VLSI - very large scale integration* - is used to describe this level of integration. Accordingly, today's chips are called *VLSI chips* and the design process is called *VLSI design*.

VLSI design

The continuous growth of complexity was enabled primarily by scaling down transistor sizes. Improvements in manufacturing and the increasing automation of the design process did the rest.

2.3 VLSI Design Flow

The VLSI design process is highly complex and heavily depends on computer software to automate the design steps, so-called *electronic design automation (EDA)* software. EDA tools automate the design process and link the steps into a single flow, which is roughly outlined in Diagram 2.4.
In the first design phase the high-level requirements of the system like functionality, performance and physical dimensions are defined and decisions concerning the design architecture, for example memory management, power requirements etc. are made. Once this is set, a logic description of the design is devised in the functional and logic design phase. Here the functionality and connectivity of each module is specified using a *hardware description language (HDL)*. A compiler translates this description into a *register transfer level (RTL)* description, which maps the desired functionality to a *netlist*: Simply put, a netlist consists of circuits, primary input and output pins of the chip, and information about the connectivity of primary pins and circuits.

EDA

HDL

RTL

Netlist

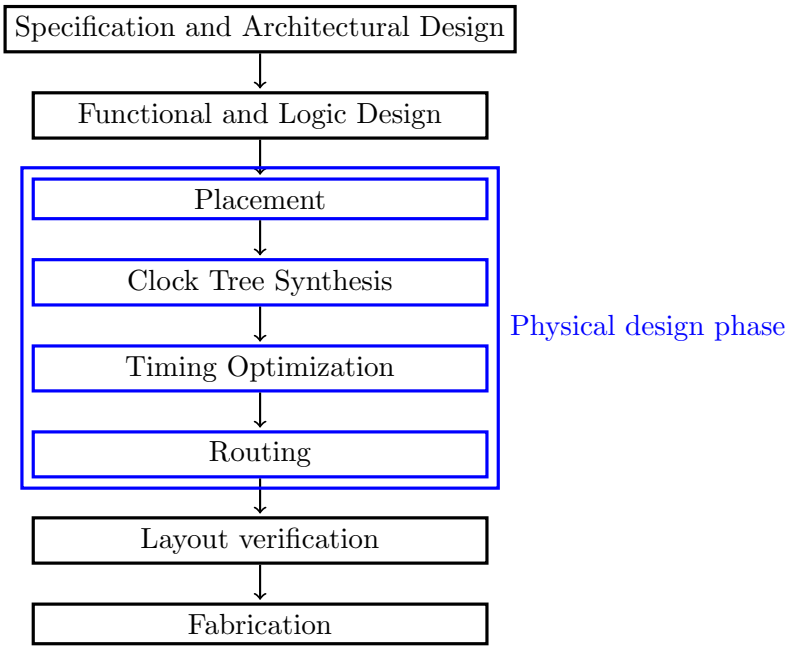


Figure 2.4: VLSI Design Flow

During the physical design phase, the RTL description is transformed into a physical layout. First the circuits are placed on the chip area in the *placement* step. Then the clock network is realized (*clock network design*), and the electrical signals are optimized in the *timing optimization* step. Finally, pins are connected by electrical wires (*routing*). Timing optimization ensures that all signals arrive on time and that all electrical constraints are fulfilled. Among these algorithms are for example

Physical design

Placement

Clock network design

Timing optimization

Routing

circuit sizing, V_t optimization and repeater insertion. In addition, changing the placement often helps to shorten long timing-critical paths on the chip.
Before the chip can be sent to fabrication, the correct functionality of the physical layout has to be verified.
In reality, the diagram is not as straightforward as indicated in Figure 2.4, and some steps are iterated until a certain design goal is achieved. Furthermore, with scaling complexity and decreasing feature sizes the boundaries between the successive (physical) design steps are blurring and are continuing to do so. As a result, the design steps are interleaving, and optimization goals formerly used in later design steps need now be considered in earlier stages. For example, placement must be aware of timing critical paths. Both placement and timing optimization must be aware of routing issues and try to ensure that in each region of the chip there is enough space to route the wires. This is difficult to achieve, as circuit sizing and repeater insertion in turn need information on the placement of the circuits and the rough outline of the wires. Consequently, physical design steps are often iterated or interleave.

2.4 Physical Design Instance

Chip area

The *chip area* is a rectangle $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$ in $z_{max} + 1$ planes with $x_{min}, x_{max}, y_{min}, y_{max}, z_{max} \in \mathbb{N}$. The plane with index 0 is called the *placement plane*, as all circuits are realized on that plane. Planes with higher index are called *routing planes* and are reserved for electrical wires. *Vias* connect two adjacent planes.

Placement plane

Routing planes

Via

Shape

We denote an axis-parallel rectangle on a plane as *shape*. If it is realized on the placement plane, we call it a placement shape. Every object on the chip is given as a set of shapes.

Chip image

I/O-ports

A *chip image* $\mathcal{I}$ consists of the chip area, a set of *blockages* given as a set of shapes, and a set of *I/O-ports* (the primary input and output pins) which connect the chip with the outer world. Blockages are predesigned units with a fixed location on the chip, for example memory arrays or analog circuits, and should not be changed during physical design.

Netlist $(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$

The *netlist* of a chip

$$(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$$

consists of a finite set of circuits $\mathcal{C}$, a finite set of *pins* $\mathcal{P}$, and a finite set of *nets* $\mathcal{N}$. A net is a set of pins, and the nets in $\mathcal{N}$ form a partition of the set of pins, i.e. a family of disjoint subsets that fulfills $\bigcup_{N \in \mathcal{N}} N = \mathcal{P}$. The layout of each pin $p \in \mathcal{P}$ is given as a shape set. A mapping $\gamma : \mathcal{P} \to \mathcal{C} \,\dot\cup\, \mathcal{I}$ assigns each pin either to a circuit, or, if it is an I/O-port, to the chip image itself. We denote with $\mathcal{G} \subset \mathcal{C}$ the set of gates of the chip.

Net

$\mathcal{G} \subset \mathcal{C}$

The *circuit library* $\mathcal{B}$ defines several logically equivalent implementations, so-called *books*, for each logic function and register type on the chip. Each book can be seen as a blueprint or layout of a circuit that can be implemented. For example,

Circuit library $\mathcal{B}$

Book

there are books for different sizes and V_t levels of a circuit. As there can be several circuits on a chip realizing the same function, the chip can contain several instances of the same book. For a circuit $c \in \mathcal{C}$ we denote the set of books that can implement c on the chip by $\mathcal{B}_c \subset \mathcal{B}$ and $\bigcup_{c \in \mathcal{C}} \mathcal{B}_c = \mathcal{B}$. A book $b \in \mathcal{B}$ is described as a set of shapes, and sets of input and output pins of b. When book b is realized on the chip, electrical signals enter at the input pins, and leave at the output pins.

$\mathcal{B}_c$

A *physical design instance* consists of a chip image $\mathcal{I}$, a netlist $(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$ and a circuit library $\mathcal{B}$. Additionally, an initial assignment of circuits to books is given by $\phi : \mathcal{C} \to \mathcal{B}$ with $\phi(c) \in \mathcal{B}_c$ for all $c \in \mathcal{C}$.

Physical design instance

$\phi : \mathcal{C} \to \mathcal{B}$

We assume that physical properties of a book $b \in \mathcal{B}$ transfer to each circuit $c \in \mathcal{C}$ implemented by b, and that the shape set of c equals the shape set of b. The shapes on the placement plane constitute the *placement area* of c. The *placement location* of a circuit on the placement plane is given as a tuple $(x, y) \in [x_{min}, x_{max}] \times [y_{min}, y_{max}]$. This means the circuit is realized on the placement plane such that a predefined anchor point on the placement area of c is located at (x, y).

Circuit area

Placement location

Each net $n \in \mathcal{N}$ has its unique *source pin*, which is either a primary input pin or an output pin of a circuit. The *sink pins* are either primary output pins or input pins of circuits. The source pin is connected to all sinks of the net by electrical wires and distributes electrical signals to all sinks. We also say: the source pin *drives* the sink pins.

2.5 Timing Analysis

In each computation cycle of the chip, electrical signals are propagated through the combinatorial logic. For the chip to operate correctly, signals have to fulfill certain conditions: They need to arrive at the inputs of the memory circuits before these open again and release the data for the next computation cycle. Vice versa, signals should not arrive before the current computation cycle is finished to ensure that the output signal of the memory circuits remains stable. Further, predefined required arrival times for signals exist at primary output pins.

Timing analysis checks if these conditions are fulfilled. If that is the case, we speak of *timing closure* or say the design has *closed timing*. Usually, this is done by means of *static timing analysis* first described by Hitchcock et al. [HSC82]. A detailed introduction to timing analysis in VLSI design can be found in Sapatnekar [Sap04].

Timing closure

Static timing analysis (STA)

2.5.1 Signals and their Shapes

The voltage compared to ground determines the logical state at a given point on the chip: V_{dd} represents a logical 1 or *true*, and V_0 represents a logical 0 or *false*. A *signal* σ is defined as the change of voltage over time. If the potential of the signal changes from V_0 to V_{dd}, we say it is a *rising* signal, otherwise, if it changes from V_{dd} to V_0, we say it is a *falling* signal. We call the direction of σ its *transition*

Signal σ

Signal transition $\tau(\sigma)$

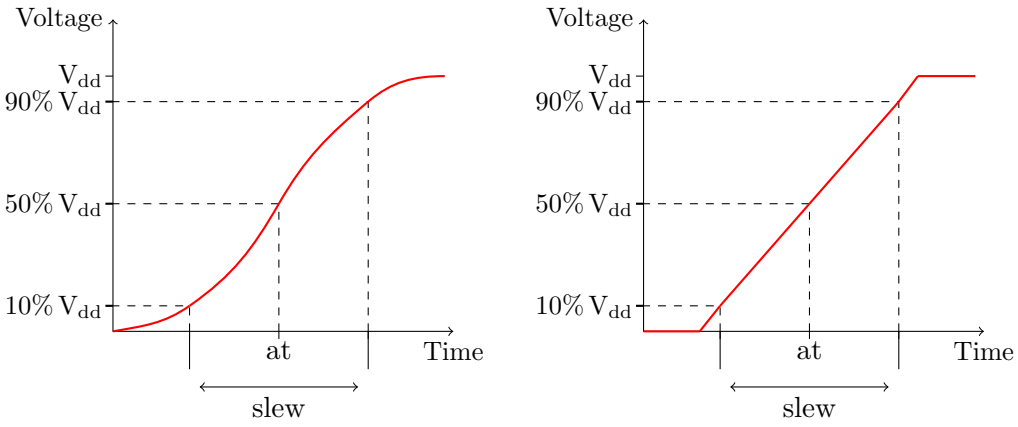


Figure 2.5: A rising signal and its approximation

$\tau(\sigma) \in \{r, f\}$, and denote the possible transitions *rise* and *fall* with r and f, respectively.

Arrival time, slew

A signal is estimated by a piecewise linear function given by its *arrival time* (at) and *slew*, see Figure 2.5. Usually, the arrival time is defined as the time when the voltage change reaches 50%. The slew is usually specified as the time between 10% and 90% of the voltage change, i.e. the range in which the signal is almost linear. Seldomly, other values like the range between 20% and 80% are used in the industry.

Data signals
Clock signals

We distinguish between two types of signals: *Data signals* represent the logical computations of the chip, while periodic *clock signals* control the memory elements.

2.5.2 The Timing Graph and the Gate Graph

Timing points

Static timing analysis measures signals at the *timing points* of the design, which are usually the pins in the netlist. Additionally, some circuits may have internal timing points. Primary input pins and register output pins are called *timing start points*, primary output pins and register input pins are called *timing endpoints*. Together they form the *boundary* of the chip. The timing points form the vertex set of the *timing graph*, which is the basic data structure used in static timing analysis:

Timing start point
Timing endpoint
Boundary

Timing graph

Definition 2.1 (Timing Graph) *The timing graph $G = (V, E)$ of a netlist $(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$ is the directed acyclic graph with one vertex for each timing point, and there is an edge between two vertices p and q if a signal at the pin corresponding to p can immediately cause a signal at the pin corresponding to q, i.e. the pins either belong to the same gate or to the same net. Edges are also called propagation segments. Timing endpoints have no outgoing edges in G. Similarly, timing start points have no entering edges in G. The sets of vertices corresponding to timing start and endpoints are denoted by V_{start} and V_{end}, respectively. The set of vertices $v \in V \setminus \{V_{start} \cup V_{end}\}$ are denoted with V_{inner}.*

Propagation segment

V_{start}, V_{end}, V_{inner}

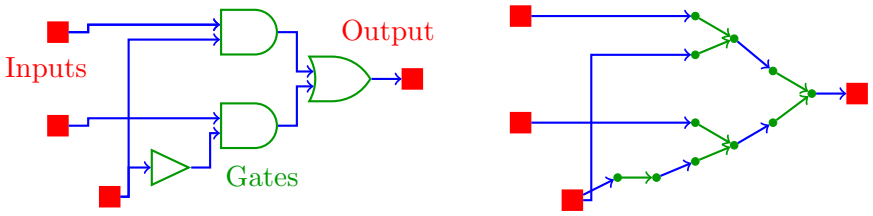


Figure 2.6: A simplified example of a VLSI Chip on the left, and the corresponding timing graph on the right.

Figure 2.6 shows a simplified example of a VLSI chip and the corresponding timing graph. Note that the timing graph does not contain edges traversing memory elements or clock drivers. One reason is that during the timing optimization phase, the locations and sizes of clock driver and registers are usually fixed. Furthermore, it is not uncommon that a signal leaving a register output enters the same register again at a later stage, and including latches with an internal propagation segment (transparent latch) would lead to cycles in the timing graph. Consequently, not every pin $p \in \mathcal{P}$ is represented by a vertex in G.
In practice, cycles can also be introduced by clock gating, which occurs when the combinatorial logic changes the clocking behaviour of memory elements. Such cycles can usually be removed by a two-phase-approach, see for example Szegedy[Sze05], and we consider the timing graph to be acyclic.

Gate graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$

Definition 2.2 (Gate Graph) *The gate graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ of a netlist $(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$ is a directed acyclic graph with one vertex for each gate, each timing start point and each timing endpoint. It can be constructed from the timing graph by contracting vertices corresponding to pins of the same gate to a single vertex. There exists an edge between vertices $v, w \in \mathcal{V}$ if there exists an edge in the timing graph between pins that are assigned to the gates corresponding to v and w, respectively.*

Figure 2.7 shows a simplified example of a VLSI chip and the corresponding gate graph.
In later chapters we will need the concept of a *neighborhood* of pins and a gates: The reason is that analyzing the impact of a local optimization step, for example

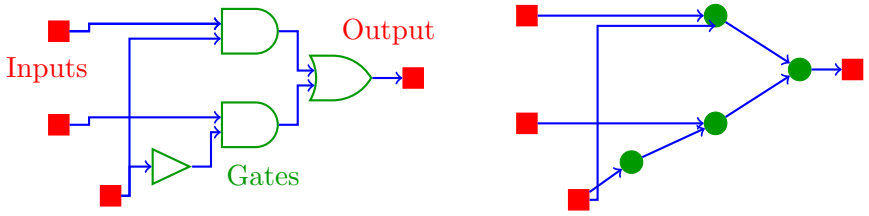


Figure 2.7: A simplified example of a VLSI Chip on the left, and the corresponding gate graph on the right.

changing the size of a gate, on signal delays is time-consuming if signal changes are evaluated in the whole timing graph. Therefore we accept some inaccuracy and evaluate the impact of the optimization step only in a restricted environment of the changed gate that captures most effects.

v_p

For $p \in \mathcal{P}$, let v_p be the corresponding vertex in the timing graph, if existent.

Predecessors
Successors
Siblings
Neighborhood

For an object $o \in \mathcal{P} \cup \mathcal{G}$ we denote with $pred(o)$ its *predecessors*, with $succ(o)$ its *successors* and with $sibl(o)$ its *siblings*. The *neighborhood* of o is the union of its predecessors, successors and siblings, and o itself. For a pin $q \in V$, these sets are defined as follows:

$$\begin{aligned} pred(q) &:= \{p \in V | (p,q) \in E\}, && (2.4)\\ succ(q) &:= \{p \in V | (q,p) \in E\}, && (2.5)\\ sibl(q) &:= \{p \in V | \exists v \in V : (v,p) \in E \text{ and } (v,q) \in E, p \neq q\} && (2.6)\\ &= \left(\bigcup_{p \in pred(q)} succ(p) \right) \setminus \{q\}. && (2.7) \end{aligned}$$

For $g \in \mathcal{G}$, let $P_{in}(g)$ be the set of input pins and $P_{out}(g)$ be the set of output pins of g. Then we have the following definitions:

$$\begin{aligned} pred(g) &:= \{g' \in \mathcal{G} | \exists p \in P_{out}(g'), q \in P_{in}(g) : (v_p, v_q) \in E\}, && (2.8)\\ succ(g) &:= \{g' \in \mathcal{G} | \exists p \in P_{in}(g'), q \in P_{out}(g) : (v_q, v_p) \in E\}, && (2.9)\\ sibl(g) &:= \{g' \in \mathcal{G} \setminus \{g\} | \exists p \in P_{in}(g') \text{ and } q \in P_{in}(g) : && (2.10)\\ &\qquad pred(q) \cap pred(p) \neq \emptyset\}. && (2.11) \end{aligned}$$

Fanout, fanin

The successors of g are also called *fanout* and the predecessors its *fanin*. Figure 2.8(a) shows the neighborhood of a gate.

Neighborhood graph $G_g = (V_g, E_g)$

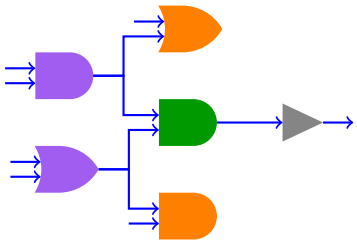
For a gate $g \in \mathcal{G}$ we call the subgraph of the timing graph G that is induced by the neighborhood of g the *neighborhood graph* $G_g = (V_g, E_g)$: The vertex set consists of all vertices that correspond to a pin of a gate in the neighborhood of g, and vertices $v \in V_{start} \cup V_{end}$ that are connected to a pin of g in G. Figure 2.8(b) shows the neighborhood graph of a gate.

2.5.3 Signal Propagation

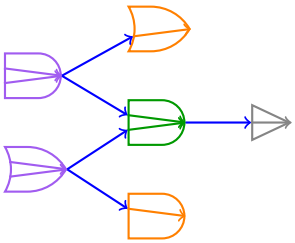
Static timing analysis propagates signals in topological order through the design by means of the timing graph. Each signal σ traversing a timing point p is characterized by its arrival time $at_p(\sigma)$ and slew $slew_p(\sigma)$. If it is clear from the context, we also write at_p and $slew_p$. Signals are characterized by the timing start point at which they are initiated. We say that signals with different origin have a different *phase*. The behaviour of transistors towards different transitions depends on their technology and their size. Similarly, the performance of circuits for rising and falling signals differs, and it is necessary that timing analysis computes arrival times and

$at_p(\sigma)$
$slew_p(\sigma)$

Phase



(a) Predecessor circuits are highlighted in purple, sibling circuits in orange and the successor circuit in gray.



(b) Edges and vertices in the neighborhood graph.

Figure 2.8: Neighborhood (left) and neighborhood graph (right) of a circuit (green).

slews for both transitions separately. Additionally, static timing analysis considers two *timing modes* (early and late) for the earliest and latest signal occurence. The reason is that signals are required not to arrive too early or too late at timing endpoints, see Section 2.5.4 for details. In this thesis we only consider the late timing mode because repeaters can be inserted to slow down signals that are too fast.

Timing mode

Naturally, it takes some time until a signal σ released at pin $p \in V$ arrives at a pin $q \in V$. We call the time it takes a signal to travel over a propagation segment $e = (p, q) \in E$ its *delay*. More formally, the delay is defined as the difference between the arrival times $at_q - at_p$. Also the slew of σ changes during the traversal of segment e. A delay function $delay_e^\tau$ and a slew function $slew_e^\tau$, called *timing functions*, provide the delay and slew of σ for each transition:

Delay

Timing functions $delay_e^\tau, slew_e^\tau$

$$delay_e^\tau : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R} \tag{2.12}$$

$$slew_e^\tau : \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R} \tag{2.13}$$

The first parameter is the total capacitance $cap(N)$ of the net $N \in \mathcal{N}$ containing q, and is defined as the sum of the capacitances of all wires and sink pins of N:

$cap(N)$

$$cap(N) := \sum_{v \in P_{out}(N)} pincap(v) + wirecap(N), \tag{2.14}$$

where $pincap(q)$ is the electrical capacitance of pin q and $wirecap(N)$ is the wire capacitance of N. The second parameter is the slew of σ at p, also called *input slew* of e. If e is a wire propagation segment, the timing functions further depend on the topology of the corresponding net. This will be specified in Section 2.5.6.

Input slew

In reality, delays and slews are influenced by chip operating conditions, for example temperature, and uncertainties during the manufacturing process. These include that actual physical shapes of the objects on the chip cannot be known in advance. Even given this knowledge, complicated non-linear differential equations need to be solved to obtain the exact delays. We will discuss different *delay models* to

Delay model

approximate signal delays and slews for circuit segments in Section 2.5.7 and for wire propagation segments in Section 2.5.6. In VLSI design, a computer program (*timing engine*) is mostly used to compute these values. Usually, several delay models with varying accuracy are implemented, and the designer can choose the suitable model for each application.

Timing engine

Static timing analysis is a variant of the critical path method by Kelley and Walker [JW59]. At each timing start point, a signal is initialized with arrival times and slews for each transition. These signals are propagated simultaneously through the timing graph in topological order. At $q \in V \setminus V_{start}$, the arrival time and slew of signal σ are computed based on the information of all incoming edges that propagate σ, i.e. all edges that lie on a path between q and a timing start point initiating σ. We denote this edge set by $\delta_\sigma^-(q) \subseteq \delta^-(q)$. The arrival time at q is the latest arrival time over all edges $e \in \delta_\sigma^-(q)$ and their edge transitions. The slew propagation on the other hand considers both the slew values and the arrival times associated with each slew. The extent to which the arrival time is considered is controlled by a parameter $\nu \in \mathbb{R}_{\geq 0}$. This model was proposed independently by Vygen [Vyg01] and Lee et al. [Lee+01].

Suppose the arrival times and slews at_p and $slew_p$ of σ for all pins $p \in V$ with $e = (p, q) \in \delta_\sigma^-(q)$ have already been determined. We first define the arrival time of σ propagated over edge $e = (p, q) \in \delta_\sigma^-(q)$ for $\tau \in \{r, f\}$ and $q \in N \in \mathcal{N}$:

$$at_e^\tau(\sigma) \quad := \quad at_p(\sigma) + delay_e^\tau(cap(N), slew_p(\sigma)), \tag{2.15}$$

Then the following holds for the arrival time and slew of σ at pin q:

$$at_q(\sigma) \quad := \quad \max\{at_e^\tau(\sigma) \mid e \in \delta_\sigma^-(q), \tau \in \{r, f\}\}, \tag{2.16}$$

$$slew_q(\sigma) \quad := \quad \max\{slew_e^\tau(cap(N), slew_p(\sigma)) + \nu \cdot (at_e^\tau(\sigma) - at_q(\sigma)) \mid e \in \delta_\sigma^-(q), \tau \in \{r, f\}\}. \tag{2.17}$$

For $\nu = \infty$, the slew of the latest signal is propagated. For $\nu = 0$ the slew at q equals the largest slew. Usually, a timing engine offers a limited set of values for ν. Vygen [Vyg06] described how the parameter can be chosen efficiently.

Remark 2.3 Sometimes an adjust value is added to the arrival time in equality (2.15) that can be user defined or computed. For example, an adjust is needed at latches if the arrival times of data and clock signals do not refer to the same computation cycle. For the simplicity of notation, we ignore this adjust in the remainder of this thesis.

2.5.4 Arrival Time Constraints and Slacks

Static timing analysis checks if all constraints on the arrival times of signals are fulfilled and the design has closed timing. The most typical arrival time constraints are

- the setup test,

- the hold test, and — Setup, hold test
- primary output constraints.

Expressed in simplified terms, the setup test checks whether a signal arrives at a register input before the register closes and releases the data for the next cycle. Similarly, the hold test verifies that a signal does not arrive too early because the voltage at the register output must be stable while it is open. As mentioned before, we are only interested in late mode timing constraints, and do not consider the hold test further.

Primary output constraints require signals to arrive before predefined *required arrival times* $rat_p(\sigma)$ at primary outputs p that indicate the latest feasible arrival time. The setup test can also be transformed into a required arrival time constraint, so for each timing endpoint and each signal σ that reaches this endpoint we have a constraint of the following form: Required arrival time $rat_p(\sigma)$

$$at_p(\sigma) \leq rat_p(\sigma) \ \forall p \in V_{end}, \sigma \in \mathfrak{S}_p. \tag{2.18}$$

For any $p \in V$ we denote with $\mathfrak{S}_p$ the set of signals reaching p, and $\mathfrak{S}$ is the set of signals initialized at any timing start point. Similar to arrival times, required arrival times can be propagated through the timing graph in reverse topological order. We denote the resulting required arrival time at $p \in V$ with $rat_p(\sigma)$ for $\sigma \in \mathfrak{S}_p$. Intuitively, this is the latest arrival time which ensures that for all timing endpoints reachable from p the arrival time constraints are fulfilled. Formally, we have $\mathfrak{S}$

$$\begin{aligned} rat_p(\sigma) \quad := \quad & \min\{rat_q(\sigma) - delay_e^\tau(cap(N), slew_p(\sigma))| \\ & e = (p,q) \in E, \tau \in \{r,f\}, q \in N \in \mathcal{N}\}. \end{aligned} \tag{2.19}$$

The *slack* at p refers to the time a signal σ arrives too late and is defined as $slack_p(\sigma)$

$$slack_p(\sigma) := rat_p(\sigma) - at_p(\sigma). \tag{2.20}$$

Usually, a slack target $slack_{target} \in \mathbb{R}_{\geq 0}$ is defined to take into account uncertainties in the delay models, manufacturing etc. that can cause a signal to arrive later than its estimated arrival time. If the slack is smaller than the target, the signal arrives too late and we say that p is *timing critical*: $slack_{target}$ Timing critical

$$slack_p(\sigma) < slack_{target}. \tag{2.21}$$

The worst slack of a design is defined as WS

$$WS := \min\{slack_p(\sigma) | p \in V_{end}, \ \sigma \in \mathfrak{S}_p\}. \tag{2.22}$$

A design is called timing critical if the worst slack is smaller than the slack target.

SNS
SLS

Apart from the worst design slack, a common measure in timing optimization is the sum of all negative slacks at timing endpoints, in short SNS. Another interesting measure is SLS, which is defined as the sum of negative slacks of all subpaths in the timing graph (see for example Reimann et al. [RSR15] for a definition).

2.5.5 Electrical Constraints

Load capacitance
$loadcap_p$

The *load capacitance* $loadcap_p$, also called *downstream capacitance*, of a primary input pin or circuit output pin $p \in N$ is defined as the capacitance $cap(N)$ of net $N \in \mathcal{N}$. The load capacitance of a primary output pin or circuit input pin is defined as the capacitance of the pin itself. The load capacitance at each primary input pin and circuit output pin should not exceed a certain limit in order to compute valid delays and slews: Each circuit/pin can only drive a certain amount of capacitance. Similarly, the slew at each primary output pin and each circuit input pin needs to obey a certain slew limit.

$\mathcal{P}_{load}$, V_{load}
$\mathcal{P}_{slew}, V_{slew}$

Let $\mathcal{P}_{load} \subset \mathcal{P}$ denote the set of pins with a load capacitance limit, and $\mathcal{P}_{slew} \subset \mathcal{P}$ denote the set of pins with a slew limit. V_{load} and V_{slew} denote the set of timing points in G corresponding to $\mathcal{P}_{load}$ and $\mathcal{P}_{slew}$, respectively.

$loadlim_p$
$slewlim_p$

We denote the load capacitance limit at $p \in \mathcal{P}_{load}$ with $loadlim_p$ and the slew limit at $p \in \mathcal{P}_{slew}$ with $slewlim_p$. We call the following constraints *electrical constraints*:

$$loadcap_p \leq loadlim_p \; \forall p \in \mathcal{P}_{load}, \text{ and} \tag{2.23}$$

$$slew_p(\sigma) \leq slewlim_p \; \forall p \in \mathcal{P}_{slew}, \sigma \in \mathfrak{S}_p. \tag{2.24}$$

Load violation
Slew violation

We call a violation of constraint (2.23) a *load violation*, and a violation of constraint (2.24) a *slew violation*.

In practice, load violations are usually considered to be more severe than slew violations. On the one hand, reasonable slews can only be computed for valid load capacitances. On the other hand, slew limits are often assigned small values by designers, and are considered to be rather a target than a hard limit.

2.5.6 Wire Delay

RC-delay

The delay over a propagation segment e whose endpoints belong to a net $N \in \mathcal{N}$ depends on the *topology* of that net. Usually, a net is modeled as an electrical network which consists of resistance and capacitance elements, and the delay is often called *RC-delay*. The most commonly used model is the *Elmore delay model* [Elm48]. It is a popular delay model because of its simplicity, but it is an upper bound on the actual wire delay and sometimes too pessimistic. More accurate delay models are for example SPICE (Simulation Program with Integrated Circuit Emphasis), which is based on numerical circuit simulation (Nagel and Pederson [NP73]), and RICE (Rapid Interconnect Circuit Evaluation using AWE, Ratzlaff and Pillage [RP94]). In the most simple models signal delay depends linearly on the L_1 distance between two pins. Additionally, the circuit capacitances can be

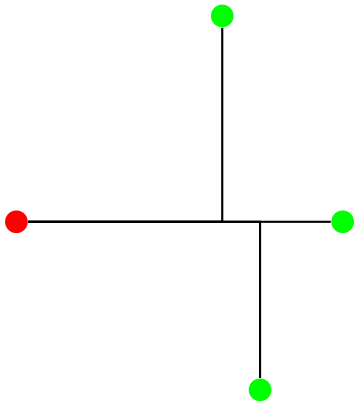


Figure 2.9: A rectilinear Steiner tree connecting the source pin (red) of a net with the sink pins (green).

incorporated. The usage is justified by the assumption that the delay over an optimal buffered wire depends approximately linear on its length, but is usually too inaccurate in the context of circuit sizing.

As the delay along a wire depends roughly quadratically on its length, shorter wires are faster than longer ones. Subdividing a wire by repeaters to refresh the signal decreases its delay. Elmore delay belongs to the class of quadratic delay models.

The accuracy of delay models increases in later steps of the VLSI design flow. The focus of this thesis is circuit sizing and V_t optimization, and due to the complexity of both problems (Section 4) there is little expectation for theoretical guarantees of algorithms if more complex delay models than the Elmore delay model are used to approximate wire delay.

Algorithms described in this thesis (Chapter 9 and Chapter 10) are independent of the delay model in the sense that they use a timing engine as a black box to get signal delay. Industry-standard engines like Synopsis PrimeTime or Cadence Tempus provide several models with different accuracy for signal estimation including the most accurate SPICE simulation.

Elmore Delay Model The Elmore delay model assumes that the physical realization of each net N is modeled by a so-called *RC-tree* consisting of resistance and capacitance elements. On such a tree, Elmore delay can be computed in linear time, see Rubinstein et al. [RPH83]. As the actual net topology and capacitances are not known until after the timing optimization phase, a rectilinear Steiner tree S with default resistances and wire capacitances estimates the actual RC-tree. An example is shown in Figure 2.9. The disjointness of Steiner trees is ignored in this phase of the design flow, and coupling effects of different nets are estimated. The problem to compute a minimum rectilinear Steiner tree is NP-hard (Garey and Johnson [GJ77]), and in practice heuristics are deployed for high fanout nets. For small fanout nets, exact algorithms can still be efficient.

Let $e = (p, q) \in E$ be a propagation segment with $p, q \in N$. Note that we modeled this propagation segment as a single edge in the timing graph, but in the Steiner tree S this propagation segment actually is a (unique) path $S[p, q]$ through the tree.

Wire segment

We call the edges of this path *wire segments* in contrasts to the wire propagation segments.

Each wire segment in the RC-tree is modeled as a resistance element encased by two capacitance elements. We assume that S is oriented from the source to the sinks of N. The Elmore delay on $S[p, q]$ is calculated as

$$rc_{Elmore}(p, q) := \sum_{e'=(v,w)\in S[p,q]} res_{e'} \cdot \left(\frac{cap_{e'}}{2} + load_w \right). \tag{2.25}$$

Here $res_{e'}$ is the (estimated) wire resistance and $cap_{e'}$ the (estimated) capacitance of the wire segment $e' \in S[p, q]$. The total (estimated) capacitance of all wire segments in S and all sink pins of N that are reachable from w is denoted by $load_w$. This implies that the whole Steiner tree needs to be built before the delay of e can be estimated.

It is relatively easy to estimate the resistance of a wire segment as it is approximately proportional to its length and inversely proportional to its width and thickness. Capacitances are harder to estimate as they not only depend on the width, thickness and length of the wire, but also on the capacitances in its environment. As both the resistance and the capacitance of a wire segment contribute to the delay (2.25), it depends quadratically on the length of the wire.

Note that the Elmore delay does not depend on the input slew in contrast to the delay function (2.12) presented earlier. It further approximates the median of an impulse response of an RC-tree, and not the response to a rising or falling signal. Industrial timing engines usually provide delay and slew functions for each $e = (p, q) \in E$, with q in net $N \in \mathcal{N}$, that combine $rc_{Elmore}(p, q)$ with the input slew s and environmental factors:

$$delay_e^\tau(cap(N), s) := rc_{Elmore}(p, q) \cdot delay_{Elmore}(s) \tag{2.26}$$

$$slew_e^\tau(cap(N), s) := s + rc_{Elmore}(p, q) \cdot slew_{Elmore}(s) \tag{2.27}$$

The parameter $cap(N)$ is implicitly used in the calculation of $rc_{Elmore}(p, q)$. We will use the functions $delay_{Elmore}(s)$ and $slew_{Elmore}(s)$ as black box functions. The simplest reasonable estimate sets $delay_{Elmore}(s) = \ln(2)$ and $slew_{Elmore}(s) = \ln(9)$, but more accurate estimates with non-constant slews are common.

2.5.7 Circuit Delay

In this section we are interested in the signal delay over a propagation segment $e = (p, q) \in E$ which traverses a circuit $c \in \mathcal{C}$. For all books from the library, precharacterized delay and slew functions, so-called *timing rules*, are given. They depend on the input slew $slew_p$ and the load capacitance $loadcap_q$, and only return

dependable values if the input slew and the load capacitance are within their limits. Both functions are monotonically increasing and have similar shapes.
With the scaling of CMOS technology and arising design challenges, the focus has been shifted from simple lookup tables to current source based models (CSM) that characterize circuits as non-linear parasitic capacitances connected to a voltage dependent current source, see for example Croix and Wong [CW03]. These models enable more accurate interpolation of electrical effects such as noise.

CSM

2.6 Physical Design Constraints and Objectives

We give a brief overview on physical design constraints and objectives that are considered in this thesis. The focus is on circuit sizing and V_t level optimization in the timing optimization step of the physical design flow, where clock drivers (in particular their sizes) are fixed and we use estimates for the wire topology of nets. For more details and algorithms for physical design we refer to Kahng et al. [Kah+11].

2.6.1 Power Constraints

The power consumption of a chip has become an increasingly important objective in VLSI design. This is due to the increasing number of transistors on a chip (Moore's Law) which dissipate power, but also the breakdown of Dennard's Scaling Law [Den+74]: It roughly states that the power consumption of a transistor scales down in proportion with its area. However, since the 45 nm technology node, leakage current has been growing exponentially, and tradeoffs between power consumption and delay had to be found.

Dennard's Scaling Law

A development of the feature sizes/technology nodes since 1971 is shown in Figure 2.10. Note that the node of a CMOS technology refers to the minimum transistor length that can be built dependably. Recently, a new type of transistor has come into operation (*FinFETs*, or *3d-transistors*) with the prospect of reducing the leakage once again.
Another factor that contributes to the increased power consumption are the higher operating frequencies that are enabled by a higher supply voltage, which contributes quadratically to dynamic power consumption. Recall that dynamic power is proportional to $\frac{1}{2} Cktcap \cdot {\mathrm{V_{dd}}}^2$, but a total of $Cktcap \cdot {\mathrm{V_{dd}}}^2$ is drawn from the power source. The remaining power is dissipated as heat. Chips need to be cooled down as high temperatures cause the chip to fail. Additionally, static power increases rapidly with rising temperature.
In the physical design phase, circuit sizing, V_t optimization and repeater insertion are the most frequently used algorithms to reduce power consumption. Other methods include switching off unused parts of the chip temporarily, or to lower the supply voltage in timing uncritical parts of the chip.
Thereby estimates are used for static and dynamic power consumption. The reason is that power depends on process parameters such as thermal power, switching

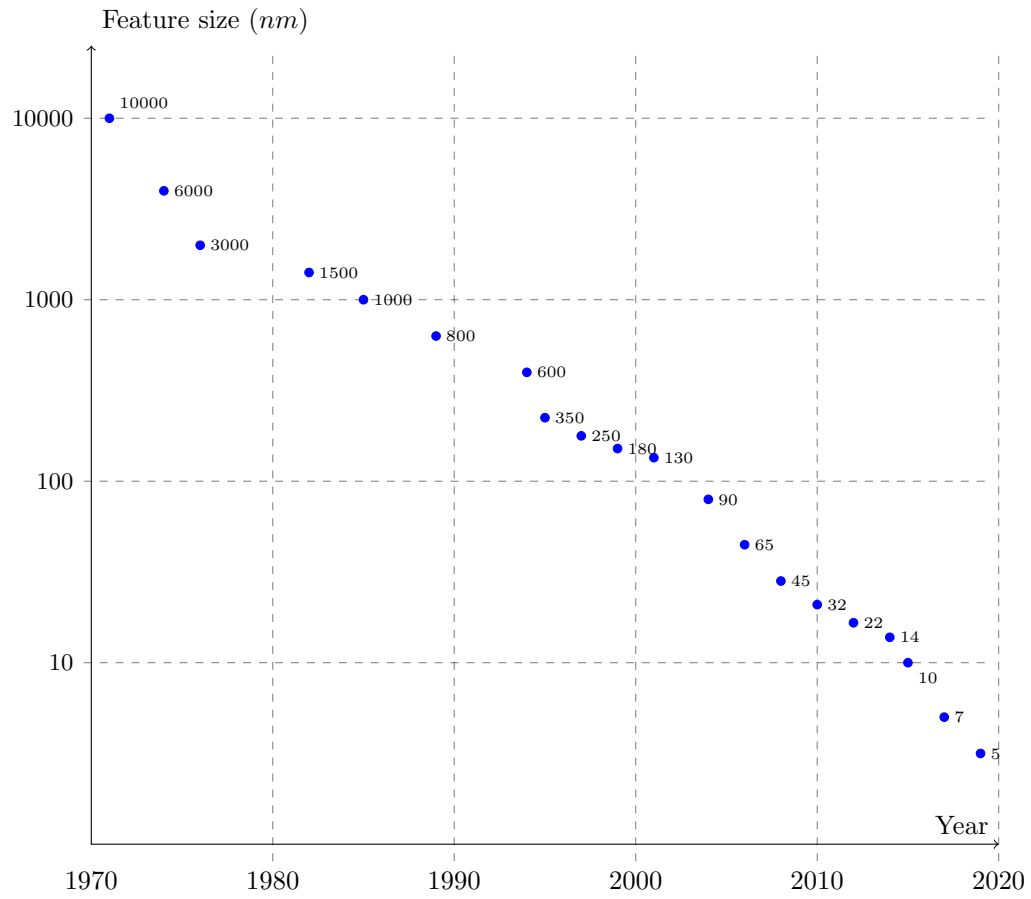


Figure 2.10: Feature size development with future predictions from [SIA13].

factors, input pin patterns etc., which are not always known and cannot be evaluated exactly during physical design due to complexity and runtime reasons. For example, a constant is often used to estimate the switching activity. In the simplest model the same constant is used for all circuits, which implies that dynamic power is proportional to the total capacitance of all input pins on the chip. We assume that we are given independent functions which return static, dynamic and total power consumption of a book $b \in \mathcal{B}$.

We regard process parameters like temperature, supply voltage and operating frequency as constant and consider only the dependency on parameters that are affected by circuit sizing and V_t optimization. Consequently, wire capacitances are neglected as these are affected only marginally when pin locations change due to sizing. Both dynamic and static power functions depend linearly on the size of book b, and lowering the V_t level affects dynamic power linearly. Short-circuit power is considered as constant, which is reasonable when low slew limits are imposed on low V_t circuits. Static power grows exponentially with lower V_t level.

2.6.2 Logical Correctness

During the physical design phase, the netlist is changed by logic restructuring, circuit sizing, repeater insertion etc. In the end, the chip needs to operate logically correct. It is *NP*-hard to decide if two netlists are logically equivalent, and algorithms exploit that in practice usually only local netlist changes are made. During circuit sizing and V_t optimization, we have to ensure that only logically equivalent implementations are chosen for each circuit from the library.

2.6.3 Routing and Placement Constraints

In the routing step, the pins of each net are connected by electrical wires. There are various constraints on the shapes and location of these wires, for example the distance between wires of different nets and the distance to blockages must be sufficiently large. Routing algorithms aim to minimize the total wire length of all nets and the number of vias between different planes, while fulfilling all constraints on the outline and shapes of the wires. In this thesis we will consider routing constraints only indirectly in earlier design steps.
Circuits have to be placed disjointly on the chip area, i.e. their placement shapes should not overlap. Furthermore, their shapes should not overlap with blockage shapes on the chip.
To improve routability of a design, a fast and simple placement approach consists of partitioning the chip area into regions, and to assign each circuit to a region. The *placement density* of a region is defined as the ratio of the placement area covered by the circuits in the region, and the placement area of the region itself. A global target density is prescribed for all regions and should not be violated. The intuition of this concept is that a high placement density often implies in turn a high pin density. This makes it more difficult for routing algorithms to connect the pins without violating any routing constraints on minimum wire distance etc. We denote with $area(c)$ the area of the placement shape of $c \in \mathcal{C}$.

Placement density

$area(b), area(c)$

2.6.4 Timing Constraints

Timing constraints are generally understood as being the constraints on the arrival times of signals, which we discussed in Section 2.5. As the delay and slew functions only return reliable values when load capacitances and slews are below their limit, timing constraints imply that the electrical constraints need to be fulfilled as well. In practice, worst slack (and SNS) improvement and electrical violation removal are usually treated as different objectives and can even be conflicting: Increasing the size of a circuit to remove a load violation at its output can degrade the delay of the entering signal due to the higher input pin capacitances.
Note that arrival time constraints are mostly implied by the clock cycle time of the design, and designers might aim to reduce the clock frequency. This amounts to minimizing the longest path delay in the design.

In the VLSI design flow, a diverse range of timing optimization algorithms is employed to help close timing of the design. Signal delays over wires can be improved by shortening wire lengths on critical paths, as signal delay depends approximately quadratically on wire lengths. This can be achiveved by changing the placement location of circuits on the timing critical paths, or rerouting timing critical nets. Also the logic description of the design can be optimized. Nonetheless, the most important steps to achieve timing closure are circuit sizing, V_t optimization and repeater insertion. Repeaters (inverter or buffer) subdivide long wires into several short ones, thereby reducing capacitances and signal delay. We refer to Bartoschek [Bar14] for an overview of algorithms for the repeater insertion problem.

3 Convex Optimization

Convex optimization is a special class of mathematical optimization methods which studies the problem of minimizing convex functions over convex sets. A fundamental property of convex optimization problems is that each local optimum is always a global optimum, making optimization in a way "easier". Many convex optimization problems can be solved in polynomial time by interior point or ellipsoid methods, which is another advantage.
In this chapter we restate well-known concepts and results from convex optimization needed in later chapters. These include Lagrangian relaxation, descent methods and interior point methods and can be found in most textbooks on nonlinear optimization. We omit proofs of well-known theorems and refer to the textbook of Bazaara, Sherali and Shetty [BSS06]. For a more comprehensive overview see for example [BSS06], Boyd and Vandenberghe [BV04] and Bertsekas [Ber99].

3.1 Basic Concepts

Definition 3.1 (Differentiable, gradient, Hessian matrix) *A function $f : \mathbb{R}^n \to \mathbb{R}^m$ is differentiable at x if the partial derivatives* Differentiable

$$\frac{\partial f_i(x)}{\partial x_j},\ i = 1, \ldots, m,\ j = 1, \ldots, n$$

exist. If f is differentiable at every point in its domain, and its domain is open, we say the function f is differentiable. When f is real-valued, the following vector is called the gradient of the function: Gradient

$$\nabla f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_n} \right)^t.$$

A function is said to be continuously differentiable if its derivative is continuous. A real-valued function f is twice differentiable if its second derivative, or Hessian matrix Continuously differentiable Hessian matrix

$$H(x)_{ij} = \frac{\partial^2 f(x)}{\partial x_i \partial x_j},\ i, j = 1, \ldots, n$$

exists. $H(x) \in \mathbb{R}^{n \times n}$ is symmetric.

Definition 3.2 (Stationary point) *We call $x \in \mathbb{R}^n$ a stationary point of $f : \mathbb{R}^n \to \mathbb{R}$ if $\nabla f(x) = 0$.* Stationary point

Convex

Definition 3.3 (Convex, concave) *A function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}$ is convex if X is convex and if for all $x, y \in X$ and for all $0 \leq \theta \leq 1$ the following holds:*

$$f(\theta x + (1-\theta)y) \leq \theta f(x) + (1-\theta)f(y).$$

f is strictly convex if strict inequality holds for all $0 < \theta < 1$, and strongly convex with parameter $m > 0$ if

$$\langle \nabla f(x) - \nabla f(y), (x-y) \rangle \geq m \left|\left| x - y \right|\right|^2$$

holds for all $x, y \in X$. In other words, f is strongly convex with parameter $m > 0$ if $\nabla^2 f(x) \geq m > 0$ holds for all $x \in X$.

Concave

We say that f is (strictly, strongly) concave if $(-f)$ is (strictly, strongly) convex.

Lipschitz continuous

Definition 3.4 (Lipschitz continuous) *A function $f : X \subseteq \mathbb{R}^n \to \mathbb{R}^m$ is called Lipschitz continuous if there exists a constant $K \in \mathbb{R}_{\geq 0}$ such that for all $x, y \in X$*

$$\left|\left| f(x) - f(y) \right|\right| \leq K \left|\left| x - y \right|\right|.$$

Convex program

Definition 3.5 (Convex program) *A convex program is an optimization problem of the form*

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, l \\ & x \in X \subseteq \mathbb{R}^n \end{aligned} \tag{3.1}$$

where X is a convex set, $f, g_1, \ldots, g_m$ are convex and $h_1, \ldots, h_l$ are affine functions, that is $h := (h_1, \ldots, h_l)$ is of the form $h(x) = Ax - b$ for $A \in \mathbb{R}^{l \times n}$, $b \in \mathbb{R}^l$.

A *concave maximization problem* aims to maximize a concave objective function f' over constraints which have the same form as the constraints in (3.1). We will not distinguish between convex and concave optimization problems, as concave maximization problems can be solved with convex optimization methods by minimizing $(-f')$.

Subgradient

Definition 3.6 (Subgradient) *Given a convex function $f : \mathbb{R}^n \to \mathbb{R}$, a vector $d \in \mathbb{R}^n$ is a subgradient of f at $x \in \mathbb{R}^n$ if*

$$f(z) \geq f(x) + (z-x)^t d \ \forall z \in \mathbb{R}^n.$$

If instead f is a concave function, we say that d is a subgradient of f at x if $(-d)$ is a subgradient of the convex function $(-f)$ at x. The set of all subgradients of a convex (or concave) function f at $x \in \mathbb{R}^n$ is called the subdifferential of f at x.

Each subgradient is a non-descent direction. A convex (concave) function which is differentiable at $x \in \mathbb{R}^n$ has exactly one subgradient at x, namely the gradient.

Definition 3.7 (Posynomial, monomial) *A posynomial is a function* $f : \mathbb{R}^n_{>0} \to \mathbb{R}_{>0}$ *of the form*

$$f(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{K} c_k x_1^{a_{1,k}} \cdots x_n^{a_{n,k}}$$

Posynomial

with $c_k > 0$ *and* $a_{i,k} \in \mathbb{R}$ *for all* $i = 1, \ldots, n$ *and* $1 \leq k \leq K \in \mathbb{N}$. *A monomial is a posynomial with exactly one summand, i.e.* $K = 1$.

Monomial

Definition 3.8 (Geometric program) *A geometric program is an optimization problem of the form*

Geometric program

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 1, \quad i = 1, \ldots, m \\ & h_i(x) = 1, \quad i = 1, \ldots, l \\ & x_j > 0, \quad j = 1, \ldots, n \end{aligned} \tag{3.2}$$

where $f, g_1, \ldots, g_m$ *are posynomials and* $h_1, \ldots, h_l$ *are monomials.*

Geometric programs can be transformed to convex programs by variable transformation $y_i = \log x_i$. The objective function f is transformed to

$$f(x_1, \ldots, x_n) \quad = \quad f(e^{y_1}, \ldots, e^{y_n}) = \sum_{k=1}^{K} c_k \, (e^{y_1})^{a_{1,k}} \cdots (e^{y_n})^{a_{n,k}} \, .$$

The same transformation can be applied to the constraint functions. Note that the equality constraints in (3.1) are affine, but after variable transformation the monomial equality constraints of (3.2) are of the form $e^{a_k^t y + b_k} = 1$. This can be resolved by taking the logarithm of all constraint functions and the objective function. Afterwards, the objective and inequality constraint functions are still convex, and the equality constraint functions are affine.

3.2 Lagrangian Relaxation and Duality

We focus on results for convex optimization problems, and consider a convex problem of the form (3.1). We refer to this problem as *primal problem*. We set $g(x) := (g_1(x), \ldots, g_m(x))$ and $h(x) := (h_1(x), \ldots, h_l(x))$.

Primal problem

Usually, constrained optimization problems are harder to solve than unconstrained problems. A widely used approach is to relax the hard constraints and incorporate them into the objective function with *Lagrange multipliers*:

Lagrange multiplier

Definition 3.9 (Lagrange function) *The Lagrange function associated with problem (3.1) is defined as*

Lagrange function

$$L(x, \lambda, \mu) := f(x) + \lambda^t g(x) + \mu^t h(x),$$

with $\lambda = (\lambda_1, \ldots, \lambda_m)^t \in \mathbb{R}^m$ *and* $\mu = (\mu_1, \ldots, \mu_l)^t \in \mathbb{R}^l$.

We refer to λ_i as the Lagrange multiplier associated with the i-th inequality constraint $g_i(x) \leq 0$, and similarly we refer to μ_j as the Lagrange multiplier associated with the j-th equality constraint $h_j(x) = 0$.

Lagrange primal problem

Definition 3.10 (Lagrange primal problem) *We refer to the problem*

$$\inf_{x \in X} L(x, \lambda, \mu)$$

as Lagrange primal problem.

The Lagrange multipliers can be interpreted as variables of an auxiliary optimization problem:

Dual problem

Definition 3.11 (Lagrange dual problem) *The Lagrange dual problem is the optimization problem*

$$\begin{aligned} \sup \quad & D(\lambda, \mu) := \inf_{x \in X} L(x, \lambda, \mu) \\ \text{subject to} \quad & \lambda \geq 0. \end{aligned} \tag{3.3}$$

The dual function is concave, even if the primal problem (3.1) is not convex (cf. Theorem 6.3.1 of Bazaara et al. [BSS06]). If the infimum of the Lagrange function is unbounded in x, the value of the dual objective function is $-\infty$. If the dual problem is unbounded from above, the primal problem has no feasible solution.

f^*, D^*

We denote with f^* the optimal value of the primal problem (3.1), and with D^* the optimal value of the Lagrange dual problem (3.3). By definition, the value of D^* is the best lower bound on f^* that can be obtained from the Lagrange dual function:

Weak duality

Theorem 3.12 (Weak duality) $f^* \geq D^*$.

Duality gap

Definition 3.13 (Duality gap) *The difference $f^* - D^*$ is referred to as duality gap.*

In general, f^* and D^* are not equal. Under certain conditions the duality gap equals zero and an optimal solution for the primal problem can be found by solving the dual problem, which is often easier to solve. If the duality gap equals zero, we say that *strong duality* holds.

Strong duality

Strongly feasible solution

Definition 3.14 (Strongly feasible solution) *Consider the primal problem (3.1), and let $x \in X$. We say that x is a strongly feasible solution if the following holds:*

$$\begin{aligned} g_i(x) &< 0, \ i = 1, \ldots, m \\ h_i(x) &= 0, \ i = 1, \ldots, l. \end{aligned}$$

The *Slater constraint qualification* guarantees a zero duality gap if a strongly feasible solution exists:

Slater's condition

Theorem 3.15 (Slater's condition) *If there exists a strongly feasible solution $\hat{x} \in X$ for the primal problem (3.1), then strong duality holds. Furthermore, if f^* is finite, D^* is attained at $(\hat{\lambda}, \hat{\mu})$ with $\hat{\lambda} \geq 0$. If f^* is attained at $\hat{x}$, then $\hat{\lambda}^t g(\hat{x}) = 0$.*

A necessary and sufficient condition for strong duality is the existence of a saddle point:

Definition 3.16 (Saddle point) *$(\bar{x}, \bar{\lambda}, \bar{\mu})$ is called saddle point of the Lagrange function $L(x, \lambda, \mu)$ if $\bar{x} \in X, \bar{\lambda} \geq 0$ and*

Saddle point

$$L(\bar{x}, \lambda, \mu) \leq L(\bar{x}, \bar{\lambda}, \bar{\mu}) \leq L(x, \bar{\lambda}, \bar{\mu})$$

holds for all $x \in X$ and all (λ, μ) with $\lambda \geq 0$.

Theorem 3.17 (Saddle point optimality) *A solution $(\bar{x}, \bar{\lambda}, \bar{\mu})$ with $\bar{x} \in X$ and $\bar{\lambda} \geq 0$ is a saddle point for the Lagrange function $L(x, \lambda, \mu)$ if and only if*

Saddle point optimality

- *$L(\bar{x}, \bar{\lambda}, \bar{\mu}) = \min_{x \in X} L(x, \bar{\lambda}, \bar{\mu})$,*
- *$g(\bar{x}) \leq 0,\ h(\bar{x}) = 0$ and*
- *$\bar{\lambda}^t g(\bar{x}) = 0$.*

Furthermore, $(\bar{x}, \bar{\lambda}, \bar{\mu})$ is a saddle point if and only if $\bar{x}$ and $(\bar{\lambda}, \bar{\mu})$ are optimal solutions to the Lagrange primal and dual problems, respectively, and strong duality holds.

Corollary 3.18 *Suppose there exists a strongly feasible solution for problem (3.1). If $\bar{x}$ is an optimal solution to the primal problem, then there exists $(\bar{\lambda}, \bar{\mu})$ with $\bar{\lambda} \geq 0$ such that $(\bar{x}, \bar{\lambda}, \bar{\mu})$ is a saddle point.*

Definition 3.19 (Complementary slackness) *Suppose that f^* and D^* are attained by $\hat{x}$ and $(\hat{\lambda}, \hat{\mu})$, and that strong duality holds. Then*

Complementary slackness

$$\hat{\lambda}_i g_i(\hat{x}) = 0,\ i = 1, \ldots, m,$$

which implies that $g_i(\hat{x}) = 0$ if $\hat{\lambda}_i > 0$, and vice versa that $g_i(\hat{x}) > 0$ implies $\hat{\lambda}_i = 0$.

The Karush-Kuhn-Tucker (KKT) optimality conditions also play an important role in optimization. For convex primal problems, the KKT conditions are sufficient for the optimality of primal and dual solutions:

Definition 3.20 (Karush-Kuhn-Tucker-Point (KKT-point)) *Assume that the functions $f, g_1, \ldots, g_m, h_1, \ldots, h_l$ in problem (3.1) are differentiable, and that $\bar{x}$ is a feasible solution. If there exists a dual feasible solution $(\bar{\lambda}, \bar{\mu})$ such that*

$$\begin{aligned} \nabla f(\bar{x}) + \sum_{i=1}^{m} \bar{\lambda}_i \nabla g_i(\bar{x}) + \sum_{i=1}^{l} \bar{\mu}_i \nabla h_i(\bar{x}) &= 0, \\ \bar{\lambda}_i g_i(\bar{x}) &= 0, \quad i = 1, \ldots, m, \end{aligned}$$

then $\bar{x}$ is called a Karush-Kuhn-Tucker-Point. Furthermore, $\bar{x}$ is primal optimal, $(\bar{\lambda}, \bar{\mu})$ is dual optimal, and strong duality holds.

Karush-Kuhn-Tucker-Point

An interesting relationship exists between the saddle point optimality conditions and the KKT optimality conditions:

KKT optimality conditions

Theorem 3.21 (Karush, Kuhn, Tucker) *If $\bar{x} \in X$ is a KKT-point for dual feasible $(\bar{\lambda}, \bar{\mu})$, then $(\bar{x}, \bar{\lambda}, \bar{\mu})$ is a saddle point. Conversely, if $(\bar{x}, \bar{\lambda}, \bar{\mu})$ is a saddle point, then $\bar{x}$ is a KKT-point.*

The following results characterize subgradients of the dual function and show that under certain conditions, the dual function is differentiable.

Theorem 3.22 *Let X be a compact set, and let $f, g_1, \ldots, g_m, h_1, \ldots, h_l$ be continuous functions. Let*

$$M(\lambda, \mu) := \{x \in X : \ x \text{ minimizes } L(x, \lambda, \mu)\}.$$

If for dual feasible $(\bar{\lambda}, \bar{\mu})$ the set $M(\bar{\lambda}, \bar{\mu})$ contains only one element $\{\bar{x}\}$, then $D(\lambda, \mu)$ is differentiable at $(\bar{\lambda}, \bar{\mu})$ with gradient $\nabla D(\bar{\lambda}, \bar{\mu}) = (g(\bar{x}), h(\bar{x}))^t$.

Note that compactness of the set X is needed to ensure that $M(\lambda, \mu)$ is not empty. If $D(\lambda, \mu)$ is not differentiable, each $\hat{x} \in M(\lambda, \mu)$ yields a subgradient of D at (λ, μ):

Theorem 3.23 *Let X be a compact set and $f, g_1, \ldots, g_m, h_1, \ldots, h_l$ be continuous functions. If $\hat{x} \in M(\lambda, \mu)$ for dual feasible (λ, μ), then $(g(\hat{x}), h(\hat{x}))^t$ is a subgradient of $D(\lambda, \mu)$.*

The subdifferential is characterized by the following theorem:

Theorem 3.24 *Let X be a compact set and $f, g_1, \ldots, g_m, h_1, \ldots, h_l$ be continuous functions. Then ξ is a subgradient of $D(\lambda, \mu)$ at $(\hat{\lambda}, \hat{\mu})$ if and only if ξ is an element of the convex hull of $\{(g(x), h(x))^t \mid x \in M(\hat{\lambda}, \hat{\mu})\}$.*

3.3 Descent Methods for Constrained Optimization

In this section we discuss descent methods to optimize a convex function $f : \mathbb{R}^m \to \mathbb{R}$ over a convex closed set $X \subseteq \mathbb{R}^m$:

$$\begin{aligned} \min \quad & f(x) \\ \text{subject to} \quad & x \in X \end{aligned} \tag{3.4}$$

Descent methods generate a series of feasible solutions (also called iterates), and in each iteration step they advance in a direction of descent at the current iterate with a certain step size. If f is differentiable, the negative gradient is the direction of steepest descent. Otherwise subgradients, which are not necessarily a descent direction, can be used. The literature on descent methods is vast. For constrained optimization, i.e. $X \neq \mathbb{R}^m$, methods can be divided into two classes: Either the descent direction and the step size are chosen such that the next iterate belongs to

X (*methods of feasible directions*), or the iterates are projected to the constraint set X after advancing in a direction of descent (*projected* or *constrained methods*). Variants in both classes differ in their choice of step sizes, the computation of descent directions, problem scaling etc. and give different convergence guarantees. We give a short overview of methods which are suitable for our later application, most notably the *projected gradient method* for f differentiable (in literature also called *constrained gradient method*), and the *conditional gradient method*.

3.3.1 Projection Methods

Orthogonal projection

Definition 3.25 *Let $X \subseteq \mathbb{R}^m$ be convex, and $z \in \mathbb{R}^m$. The orthogonal projection x_0 of z onto X is the "closest" vector to z in X in the sense that*

$$||z - x_0||_2 \leq ||z - x||_2 \ \forall x \in X.$$

We denote the orthogonal projection onto X with $\pi_X : \mathbb{R}^m \to X$.

The Projected Gradient Method

Algorithm 3.1 shows the projected gradient method to solve problem (3.4) for f differentiable. This method was first established by Goldstein [Gol64] and Levitin and Polyak [LP66]. The choice of the step sizes $\rho^{(k)} \in \mathbb{R}_{\geq 0}$ in each iteration influences the convergence of the method.

Projected gradient method

Algorithm 3.1 Projected Gradient Method

Input: $f : X \to \mathbb{R}^n$ convex, $X \subseteq \mathbb{R}^m$ convex
Output: $x \in X$

1: $k \leftarrow 0$
2: Choose starting point $x^{(0)} \in X$
3: **repeat**
4: $\quad g^{(k)} \leftarrow \nabla f(x^{(k)})$
5: $\quad x^{(k+1)} \leftarrow \pi_X(x^{(k)} - \rho^{(k)} g^{(k)})$
6: $\quad k \leftarrow k + 1$
7: **until** stopping criterion is satisfied
8: **return** $x^{(k)}$

Usually, the stopping criterion is of the form $\left|\left|\nabla f(x^{(k)})\right|\right|_2 < \nu$ for some small $\nu > 0$. In most implementations, the stopping criterion is checked immediately after the gradient is computed.

Convergence of the Projected Gradient Method

Generally, convergence rates for Algorithm 3.1 are similar to those of the gradient method. Convergence can be shown for several step size rules $\rho^{(k)}$, more precisely,

for a sequence $\{x^{(k)}\}$ generated by Algorithm 3.1, every limit point is stationary. Even if the step size is a constant, Algorithm 3.1 converges to a stationary point if the gradient of f is Lipschitz continuous. Other rules are for example the exact step size or the inexact Armijo rule, both of which repeatedly evaluate $f(\pi_X(x^{(k)} + \rho^{(k)} g^{(k)}))$ for several choices of $\rho^{(k)}$. The diminishing step size rule fulfills $\sum_{k=0}^{\infty} \rho^{(k)} = \infty$, and $\rho^{(k)}$ approaches 0 for k tending to ∞.
We are also interested in the rate of convergence. Bertsekas [Ber99] (Section 2.3) states that the convergence rates are similar to the convergence rates of the unconstrained steepest descent method ($X = \mathbb{R}^n$). Linear convergence results for the projected gradient method are typically established under assumptions that the objective function is strongly convex, twice differentiable and/or is the combination of a strongly convex function with an affine function, see for example Dunn [Dun87]. For functions with Lipschitz continuous gradient, the convergence rate depends linearly on $\left|\left|x^{(0)} - x^*\right|\right|$, where $x^* \in X$ is an optimal solution, and the Lipschitz constant for certain step sizes. It is known that without further assumptions, the worst-case rate of convergence can be sublinear. For general convex objectives the convergence rate is, to the best of our knowledge, unknown.

The Projected Subgradient Method

Projected subgradient method

The *projected subgradient method* solves problem (3.4) if f is not differentiable. It proceeds in a fashion very similar to that of Algorithm 3.1 except for the computation of a descent direction in line 4 of Algorithm 3.1 and the stopping criterion: In each iteration, it proceeds in the direction of a subgradient, and its convergence depends more heavily on the choice of a step size because a subgradient direction is not always a descent direction. Additionally, the stopping criterion is of the form $\left|\left|g^{(k)}\right|\right|_2 < \eta$ for some small $\eta > 0$, where $g^{(k)}$ is the subgradient in iteration k. The following theorem establishes convergence:

Theorem 3.26 (Polyak [Pol67], Ermoliev [Erm66]) *Let $f : \mathbb{R}^n \to \mathbb{R}^n$ be a convex function, $X \subseteq \mathbb{R}^n$ a closed convex set so that either X is bounded or $f(x) \to \infty$ for $||x|| \to \infty$. If the step sizes $(\rho_k) \in \mathbb{R}^{\mathbb{N}}_{>0}$ are a zero sequence with $\sum_{k=0}^{\infty} \rho_k = \infty$, then the projected subgradient method generates a sequence of feasible solutions that has a subsequence tending to a minimum of f.*

3.3.2 Feasible Directions and the Conditional Gradient Method

We assume that f is continuously differentiable. A feasible direction method starts with a feasible vector $x^{(0)} \in X$ and generates a sequence of feasible iterates $\{x^{(k)}\}$ with $x^{(k+1)} := x^{(k)} + \rho^{(k)} g^{(k)}$. Thereby $g^{(k)}$ is a descent direction that is also a feasible direction, i.e. $x^{(k+1)} \in X$ for all step sizes $\rho^{(k)} > 0$ that are sufficiently small, and $\rho^{(k)}$ is chosen accordingly.
Convergence to a stationary point can be shown for certain step size rules if each $g^{(k)}$ is a descent direction. The *conditional gradient method* or *Frank-Wolfe algorithm*, which was developed by Frank and Wolfe [FW56], belongs to this class of methods.

Conditional gradient method

A new iterate $x^{(k+1)}$ is computed as a convex combination of the previous iterate $x^{(k)}$ and a feasible descent direction. The factor of the descent direction in the convex combination decreases in each iteration. The feasible direction is computed by solving a minimization problem of the form

$$\begin{aligned} \min \quad & \nabla f\left(x^{(k)}\right)^t \left(x - x^{(k)}\right) < 0 \\ \text{subject to} \quad & x \in X. \end{aligned}$$

In order to find a finite solution to the minimization problem, X has to be compact.

3.4 Interior Point Methods

We consider interior point (IP) methods for convex problems of the form (3.1) with inequality constraints only (i.e. $l = 0$) and $X = \mathbb{R}^n$. Interior point methods solve linear and convex optimization problems in polynomial time. The first polynomial time interior point method was developed by Karmarkar [Kar84] for linear programming, and later generalized for convex programming. Many different variants of this method exist, which all have in common that they traverse the interior of the feasible region to find an optimal solution. For nonlinear programming, *primal-dual methods* are increasingly popular (Forsgren et al. [FGW02]), and outperform the *barrier method* on several problem classes including geometric programming (Boyd and Vandenberghe [BV04]). We give a short description of the basic functionality. The interior of the set which is defined by the inequality constraints is incorporated into the objective function by means of a *barrier function*: This continuous function is only defined on the interior of this set, and approaches infinity as any of the inequality constraints approaches 0 from negative values. $b(x) = -\sum_{i=1}^{m} \ln(-g_i(x))$ is an example of such a barrier function.
Starting at a strongly feasible solution x, the barrier method iteratively solves $\min_{x \in X}(f(x) - \sum_{i=1}^{m}(1/t)\ln(-g_i(x)))$, which approximates problem (3.1), with Newton's method for increasing t. It returns the solution from the last iteration.
Starting with a strongly feasible solution, primal-dual interior point methods iteratively step in a so-called primal-dual search direction with a certain step size. The search directions are computed by solving a system of modified KKT-conditions with Newton's method, and are closely related to the search directions used in the barrier method. In contrast to barrier methods, primal and dual variables, which are defined as $\lambda_i := -1/(t \cdot g_i(x))$, are updated in each iteration.
Rather informally, the worst-case complexity is approximately proportional to the number of Newton steps. The idea behind Newton's method is to approximate the objective function by a quadratic function, and then take a *Newton step* towards the minimum or a saddle point of the quadratic function. The Newton step is defined as the negative of the gradient multiplied with the inverse of the Hessian matrix. It is necessary to compute the first and second derivative of the barrier function, and solving the Newton system has complexity $O(n^3)$ in general (Nemirovski [Nem04]).

4 Gate Sizing and V_t Optimization

A key challenge in the physical design of a computer chip is to choose a layout for each gate. Thereby the most common objectives are to minimize the power consumption or area of the chip subject to constraints on the speed of the electrical signals. The most influential characteristics of a gate are its size and V_t level, and we refer to the problem of choosing a layout as gate sizing and V_t optimization problem. With the continuing technology scaling and growing transistor count, gate sizing and V_t optimization have become increasingly important (cf. Section 2.1). Additionally, both characteristics have large impact on signal delays and electrical integrity, and a good choice is essential to achieve timing closure. Often, gate sizing and V_t optimization are treated separately.

Optimization is deployed at several stages during the physical design flow. Besides the versatility of the objective function, both operations are less disruptive than changing the placement of circuits or rerouting nets to fix timing constraint violations, especially in later design stages. The most common application area is in the timing optimization step. At this point, registers, in particular their sizes, have usually been fixed along with the clock net routing, and changing them would require rerouting the clock net. Therefore most algorithms consider only logic gates for optimization. Often gate sizing is performed incrementally and in combination with other algorithms for timing optimization, for example repeater insertion and timing-driven detailed placement.

The application range for sizing and V_t optimization further contains post-routing optimization, where only small changes should be made, and gate sizing that takes place after an initial placement, but before the clocks are fixed. Simplified delay models are used in that early stage.

Having analyzed the power consumption of a gate in Section 2.1, we discuss the dependency of signal delay on gate layouts in Section 4.1. We continue with a formal definition of the gate sizing problem, its continuous relaxation and the geometric program formulation (Section 4.2 - 4.4). In the continuous relaxation, gate sizes are restricted to intervals. The V_t optimization problem is introduced in Section 4.5. Afterwards, we review existing literature on the computational complexity of these problems (Section 4.6) and previous work (Section 4.7). Common approaches can be divided into continuous and discrete approaches. Continuous approaches target the continuous relaxation of the gate sizing problem, which is solvable in polynomial time and both of practical and academic interest. Several algorithms round a continuous solution to a discrete solution. We address the complexity of rounding in Section 4.8, and draw comparisons between discrete and continuous approaches in Section 4.9.

4.1 Delay Characteristics of Gate Sizes and V_t levels

Gate Sizes

The most influential characteristic of a gate is its size. Modern libraries usually contain between ten and twenty different sizes for each elementary logic function which differ in their transistor widths. The number of sizes for registers is usually smaller. Figure 4.1 shows simplified layouts for an inverter gate. Each layout realizes a different size by varying the transistor widths. For larger sizes, the transistors must be *folded* which in turn increases the width of the inverter gate. Gates belong to the class of so-called standard circuits which have in common that they need to fit in the cell rows between the V_{dd} and V_0 rails, and only vary in their widths. For details on transistor layout of circuits we refer to Schneider [Sch14].

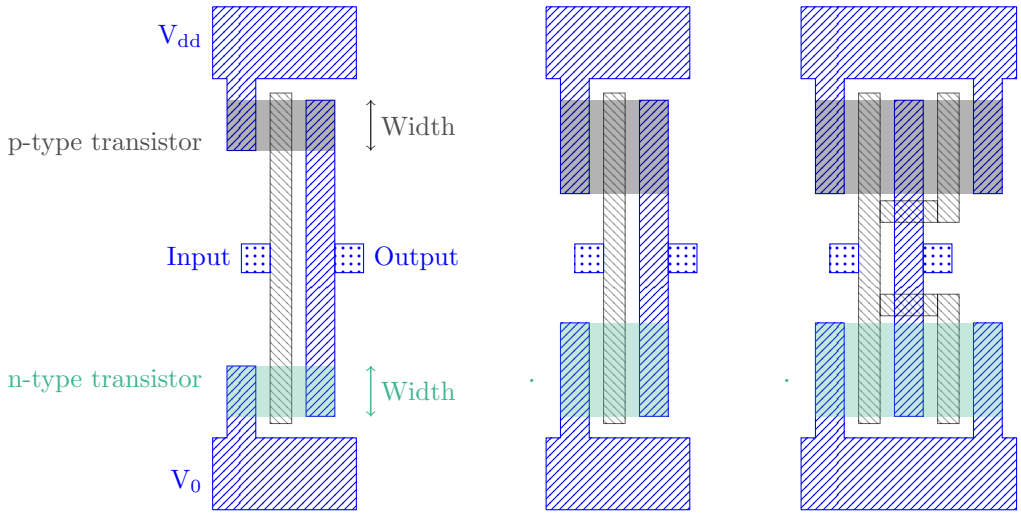


Figure 4.1: Different layouts for an inverter gate realizing different sizes as seen from above. In the layout on the right the transistors have been folded to fit into the gate. Note that the ratio between the sizes of n-type and p-type transistors usually varies.

A larger size has higher drive strength and allows faster charging and discharging of load capacitances. It accelerates a voltage change at the output pin of a circuit, and decreases the outgoing slew. On the downside, a larger size requires more area on the chip and consumes more total power than a smaller size due to larger input pin capacitances. Consequently, delay and slew of signals entering the input pin, which are also propagated to sibling circuits, deteriorate.

$\mathbf{V_t}$ Level

The threshold voltage, or V_t level, defines the voltage at which a gate switches. Usually, only three or four V_t levels are available for each gate (cf. Section 2.1).

The highest V_t level corresponds to the highest threshold. Lowering the V_t level of a gate accelerates signals, as the gate can switch earlier and thereby propagate signals faster. The impact of V_t level changes on input pin capacitances, and hence the impact on delay of input signals, is relatively small and usually neglected.

Beta Ratio and Tapering

Two more important characteristics of a gate that influence signals delays are its *beta ratio* and *tapering*. The beta ratio of a gate is the ratio of the sizes of its n-type and p-type transistors. Changing the ratio either accelerates the rising or the falling signal. Tapering can be applied to gates with more than one input pin that contain serially arranged transistors. Thereby the relative sizes of serially arranged transistors are modified with the aim to improve the delay of certain propagation segments in the gate. In this thesis we focus on optimizing sizes and V_t level.

Beta ratio

Tapering

Common Concepts

Both gate sizing and V_t optimization contain the task to choose layouts that realize a good tradeoff between power consumption or area, and signal speed. V_t optimization is easier than gate sizing in the sense that increasing the power or area consumption more predictably leads to a delay decrease and vice versa, as the impact on input pin capacitances is relatively small. The same is not always true for gate sizing, because larger sizes can slow down predecessor and sibling gates and increase the sum of delays. This can be disadvantageous, as the sum of delays on each path in the timing graph needs to be small enough in order to fulfill the timing constraints.

Gate sizing and V_t optimization are two separate optimization problems in VLSI design, but there is a tendency to handle them simultaneously. In this thesis we focus on theory and algorithms for gate sizing that can easily be extended to incorporate V_t optimization, for example the Lagrange relaxation approach. Heuristic subroutines that evaluate the impact of changing a gate layout on signal delays locally are often used in these algorithms. As changing the V_t level causes less disruption locally than changing the size, it is often easy to extend sizing algorithms to incorporate V_t optimization.

This could also lead to the assumption that changing the V_t level of a gate is preferable to changing its size. On the contrary, the exponential dependency of static power consumption on the V_t level makes changing the size more preferable in many situations. Having said that, lowering the V_t level can be preferable to increasing the gate size, for example if a signal needs to be accelerated, but a larger size would introduce a load or slew violation at its predecessors or would slow down the input signal. Handling gate sizing and V_t optimization simultaneously has the advantage that the better provision can be taken in each situation.

4.2 The Gate Sizing Problem

$size(b)$, $vt(b)$
$size(g)$, $vt(g)$

Let $size(b)$ denote the size of a book $b \in \mathcal{B}$, and $vt(b)$ its V_t level. Similarly, we use $size(g)$ and $vt(g)$ to denote the size and V_t level of gate $g \in \mathcal{G}$, which equals the size and V_t level of the book $\phi(g)$ implementing g.

$\mathcal{B}_g^{\mathcal{V}=vt}$

We assume that the set of books $\mathcal{B}_g$ available for $g \in \mathcal{G}$ contains a book for each combination of size and V_t level, which is reasonable in practice. Let $\mathcal{B}_g^{\mathcal{V}=vt} \subset \mathcal{B}_g$ be the set of available books with V_t level $0 < vt < \mathrm{V_{dd}}$. If it is clear from the context that the V_t level is fixed, as in this section, we use $\mathcal{B}_g$ and $\mathcal{B}_g^{\mathcal{V}=vt}$ synonymously.

Dynamic and static power consumption, and thus total power consumption, of a gate scale approximately linear with its size (cf. Section 2.1). Gate sizes differ in the widths of the transistors, hence the same holds for the area of a gate. We will model the total power consumption of all gates with a linear cost function $cost : \mathcal{B} \to \mathbb{R}_{\geq 0}$

$cost(b)$

of the form

$$cost(b) \quad := \quad \alpha_{f(b)} \cdot size(b), \; b \in \mathcal{B} \tag{4.1}$$

that can also represent other objectives like static or dynamic power consumption or total gate area by varying the factor $\alpha_{f(b)} \in \mathbb{R}_{>0}$. The factor $\alpha_{f(b)}$ further depends on the function $f(b)$ that is implemented by b, and of course on the technology of the underlying design. This mapping can be extended to a mapping $cost : \mathcal{G} \to \mathbb{R}_{\geq 0}$

$cost(g)$

with $cost(g) = cost(\phi(b))$.

Gate Sizing Problem

Instance:

- Physical design instance consisting of
  - a chip image $\mathcal{I}$
  - a netlist $(\mathcal{C}, \mathcal{P}, \gamma, \mathcal{N})$, and
  - a circuit library $\mathcal{B}$.
- Timing constraints and timing rules.
- An objective function $cost : \mathcal{B} \to \mathbb{R}_{\geq 0}$.

Task: Find a size for each gate, i.e. a mapping $\phi : \mathcal{G} \to \mathcal{B}$ with $\phi(g) \in \mathcal{B}_g$ such that

$$\sum_{g \in \mathcal{G}} cost(\phi(g))$$

is minimized and all timing constraints are fulfilled.

The delay-minimizing variant of the gate sizing problem aims to optimize the worst design slack subject to a constraint on total power consumption, and is referred to as the delay-minimizing gate sizing problem.

We fix an ordering of the gates in $\mathcal{G}$ as $g_1, \ldots, g_n$ with $n \in \mathbb{N}$ and introduce a discrete size variable x_i for each gate $g_i \in \mathcal{G}$. Let each book $b \in \mathcal{B}_{g_i}$ be associated with a real number (its "size") which allows us to represent $\mathcal{B}_{g_i}$ by a finite discrete set $X^i_{disc} \subset \mathbb{R}$, i.e we have a bijective mapping

x_i

X^i_{disc}

$$\mathcal{B}_{g_i} \to X^i_{disc}.$$

The discrete set of feasible gate size vectors is defined as

X_{disc}

$$X_{disc} := \{x = (x_1, \ldots, x_n)^t : x_i \in X^i_{disc}\}. \tag{4.2}$$

We get back to the discrete size variables at the end of Section 4.4.

4.3 The Continuous Relaxation of the Gate Sizing Problem

By nature, the gate sizing problem is a discrete problem since the circuit library only offers a discrete set of implementations for each gate. Even so, it can be formulated as a continuous problem by restricting gate sizes to intervals and approximating signal delays and slews. The continuous relaxation of the gate sizing problem is both of academic and practical interest. For example, it can be formulated as a geometric program (see Section 4.4) and solved in polynomial time with interior point methods, but is hard to tackle because of the huge instance sizes occurring in practice. It poses a challenge for researchers because for instance sizes of this magnitude, standard geometric program solvers fail, see Joshi and Boyd [JB08].

The continuous sizing problem appeared in literature first in the context of transistor sizing, and until the early 1990's most research focused on this formulation. Still today some practical approaches are guided by a continuous solution, for example by applying a rounding step in the end to get discrete sizes as proposed by Hu et al. [HKH09] and Xie and Chen [XC15].

We introduce a continuous size variable ξ_g for each gate $g \in \mathcal{G}$, and restrict each ξ_g to an interval $\tilde{I}_g = [\iota_g, \mu_g] \subset \mathbb{R}$. The vector of all gate sizes is denoted by $\xi = (\xi_1, \ldots, \xi_n)$, where we use the previously fixed ordering of the gates. Let further

ξ_g

$\tilde{I}_g = [\iota_g, \mu_g]$

Ξ

$$\Xi := \{\xi \in \mathbb{R}^n : \ \iota_{g_i} \leq \xi_i \leq \mu_{g_i}, \ g_i \in \mathcal{G}\} \tag{4.3}$$

be the set of all feasible continuous gate size vectors.

Power consumption of a gate scales approximately linear with its size, and we can naturally extend the cost function $cost(x)$ of the discrete problem to a function $cost : \Xi \to \mathbb{R}_{\geq 0}$ with

$$cost(\xi) := \sum_{i=1}^{n} cost(\xi_i) := \alpha_i \cdot \xi_i.$$

$cost(\xi)$

with $\alpha_i := \alpha_{f(\phi(g_i))}$ for gate g_i (see also (4.1). The timing constraints on signal arrival times do not change in the continuous formulation. Under the assumption that signal delays can be approximated we obtain:

Continuous Gate Sizing Problem

Instance:

- A physical design instance.
- Timing constraints.
- A linear objective function $cost : \Xi \to \mathbb{R}_{\geq 0}$.

Task: Find a mapping $\phi' : \mathcal{G} \to \bigcup_{g_i \in \mathcal{G}} \tilde{I}_{g_i}$ with $\phi'(g_i) \in \tilde{I}_{g_i}$ such that

$$\sum_{i=1}^{n} cost(\phi'(g_i))$$

is minimized and all timing constraints are fulfilled.

When signal delays and the objective function are approximated by posynomials, the continuous relaxation can be formulated as a geometric program.

4.4 Convex Program for the Continuous Relaxation

The benefit of a geometric program formulation is that it can be solved in polynomial time using for example interior point methods. The first posynomial formulation for transistor/gate sizing was introduced by Fishburn and Dunlop [FD85] and formed the basis for subsequent formulations of the gate sizing problem as geometric program. Recall from Section 3 that in order to get a geometric program, the objective function and the inequality constraints need to be formulated as posynomials. The objective function is of the form $cost(x) = \sum_{i=1}^{n} \alpha_i \xi_i$, which already is a posynomial. Edge delays can be expressed as posynomials by modeling gates and nets as RC-circuits under the Elmore delay model.

4.4.1 Posynomial Delay Models

We model wires and gates likewise as RC-circuits and use the Elmore delay model to compute the actual delays following Shyu et al. [Shy+88] and Chen, Chu and Wong [CCW99]. Figure 4.2 shows a 2-input gate $g_i \in \mathcal{G}$ and its representation by

Figure 4.2: An AND gate and its switch-level RC circuit model: It contains a capacitance element cap_{g_i} for each input pin, and an output resistor res_{g_i}.

two capacitors (one for each input pin) and one resistor. To simplify notation, all input pin capacitances are assumed to be equal. Let ξ_i be the size of g_i. We have

$$\begin{aligned} res_{g_i} &:= \widetilde{res_{g_i}}/\xi_{g_i}, \text{ and} \\ cap_{g_i} &:= \widetilde{cap}_{g_i} \cdot \xi_{g_i} + \tilde{f}_{g_i} \end{aligned}$$

for the resistance res_{g_i} and the capacitance cap_{g_i} of g_i, where $\widetilde{res_{g_i}}, \widetilde{cap}_{g_i}$ and $\tilde{f}_{g_i}$ are gate type specific constants denoting the unit size output resistance, the unit size gate capacitance and the gate perimeter capacitance of g_i, respectively. Wire segments are modeled by a resistor enclosed by two capacitors. We assume wire resistances and capacitances (i.e. wire lengths) to be constant because the impact of gate sizing on wire lengths is marginal. Consequently, load capacitances only depend on the sizes of the gates in the design, and the same holds for the delay and slew functions for all edges in the timing graph if we assume input slews at timing start points to be constant.
Recall from Section 2.5.3 that for each $e = (v, w) \in E$ there is a delay function for each transition and each signal traversing e. For simplicity of notation we assume from now on that the delay for rising and falling transitions do not differ, and that there exists only one phase, i.e. we do not distinguish between signals originating from different timing start points. For $e \in E$ let

$delay'_e(\xi),\ \xi \in \Xi$

$$delay'_e : \Xi \to \mathbb{R}_{\geq 0} \tag{4.4}$$

be the delay function for e that only depends on the gate sizes.
Now let $e = (v, w) \in E$ and let $N \in \mathcal{N}$ be the output net of gate g with wire capacitance $wirecap(N)$. If e is an edge traversing gate g, we have

$$\begin{aligned} delay'_e(\xi) := \frac{\widetilde{res_g}}{\xi_g} \Bigg(& \sum_{g' \in succ(g)} \widetilde{cap}_{g'} \xi_{g'} + \\ & wirecap(N) + \sum_{w' \in V_{end} \cap succ(w)} pincap(w') \Bigg), \end{aligned} \tag{4.5}$$

where ξ is the vector of gate sizes and $pincap(w')$ is constant for $w' \in V_{end}$. Otherwise, if e is a wire edge, its delay is the Elmore delay (2.25) from Section 2.5.6 and can also be expressed as a function depending on the gate sizes:

$$delay'_e(\xi) := \sum_{e'=(p,q)\in S[v,w]} res_{e'} \left(\frac{cap_{e'}}{2} + load_q(\xi) \right), \tag{4.6}$$

where $load_q(\xi)$ is the capacitance of all wire segments in the Steiner tree S realizing N plus the capacitances of all sink pins of N that are reachable from q. The sink pin capacitances depend linearly on the sizes of the sink gate or, if they are timing output pins, are constant.

In reality the delay through a gate edge is a concave function of the load capacitance for constant input slew, and not linear as in (4.5). Furthermore, slews are not considered in this model which tags it as rather inaccurate. Several variants with higher accuracy have been proposed that incorporate for example slews, intrinsic circuit capacitances, rising and falling signal transitions etc. We refer to the tutorial on geometric program-based gate sizing by Boyd et al. [Boy+05] for a more comprehensive overview and references.

4.4.2 Simplifying the Timing Constraints

Recall that the timing constraints require that for each path P in the timing graph, the signal needs to arrive on time at its endpoint p, i.e.

$$at_q + \sum_{e\in P} delay'_e(\xi) \leq rat_p$$

for all paths P in G with required arrival time rat_p at its endpoint, and arrival time at_q at its start point. This formulation is impractical as the number of paths in G depends exponentially on the number of vertices in G. However, the constraints can be partitioned into constraints on delay across the edges by introducing for each $v \in V$ an *arrival time variable* $a_v \in \mathbb{R}$ for the signal arrival time. For $v \in V_{start}$ we fix $a_v := at_v$ and for $v \in V_{end}$ we fix $a_v = rat_v$, where at_v and rat_v are the prescribed signal arrival times and required arrival times, respectively. This leads to the following formulation of the timing constraints:

Arrival time variable a_v

$$a_v + delay'_e(\xi) \leq a_w \quad \forall\ e = (v, w) \in E. \tag{4.7}$$

Remark 4.1 Note that this simplification is independent of the delay model, and also holds for discrete size variables. Transition times and more than one phase can be considered by introducing multiple constraints for each edge.

4.4.3 The Geometric and the Convex Program

With the posynomial delay approximations and the simplified timing constraints we arrive at the geometric program formulation of the continuous relaxation:

GEOMETRIC PROGRAM FOR THE GATE SIZING PROBLEM

$$\begin{aligned} &\min cost(\xi) \\ \text{subject to } &a_v + delay'_e(\xi) \leq a_w \quad \forall\, e = (v,w) \in E \\ &\xi \in \Xi \end{aligned} \tag{4.8}$$

We transform the geometric program into a convex program by variable transformation $x_i = \log(\xi_i)$ for all $i = 1, \ldots, n$ (cf. Section 3.1). Let X_{cont}

$$X_{cont} := \{x \in \mathbb{R}^n :\ l_i := \log(\iota_i) \leq x_i \leq u_i := \log(\mu_i),\ i = 1, \ldots, n\}, \tag{4.9}$$

denote the set of feasible sizes after variable transformation, and $I_c := [l_c, u_c] \subset \mathbb{R}$. $I_c = [l_c, u_c]$
We denote the resulting convex delay functions with $delay_e(x)$ for $x \in X_{cont}$, and $delay_e(x)$
formulate the convex program:

CONVEX PROGRAM FOR THE GATE SIZING PROBLEM

$$\begin{aligned} &\min cost(x) := \sum_{i=1}^{n} \alpha_i e^{x_i} \\ \text{subject to } &a_v + delay_e(x) \leq a_w \qquad \forall\, e = (v,w) \in E \\ &x \in X_{cont} \end{aligned} \tag{4.10}$$

$cost(x)$

In the remainder of this thesis we will assume that the objective function and all delay functions are convex functions of load capacitance and input slew and do not elaborate on the underlying posynomial delay model.
Recall that we have a bijection between $\mathcal{B}_{g_i}$ and the set X^i_{disc} for each $g_i \in \mathcal{G}$. To simplify notation in the following chapters, we assume $X_{disc} \subset X_{cont}$ and $size(g_i) = e^{x_i}$ holds for $x_i \in X^i_{disc}$. We denote the cost induced by size x_i with $cost(x_i) := \alpha_i e^{x_i}$. $cost(x_i)$

Remark 4.2 (Additional constraints) Note that other constraints are also compatible with the geometric program formulation, for example constraints on maximum load capacitances of primary input pins, or slew. We refer to Section 6.6 and the tutorial of Boyd et al. [Boy+05] for an overview.

4.5 The V_t Optimization Problem

We already established that V_t optimization is easier than gate sizing in the sense that the impact of changing the V_t level of a gate on delay of entering signals is smaller: Input pin capacitances only change due to variation in fabrication material or oxide thickness. In practice, algorithms targeting V_t optimization often neglect these changes, which are roughly 10% between two V_t levels (Held [Hel08]). Consequently, signals entering the gate are hardly affected, whereas outgoing signals are accelerated due to a smaller slew. In contrast to gate sizing, a power increase more predictably leads to a delay decrease, and vice versa.
V_t optimization usually targets static power minimization subject to timing constraints. A less common formulation is to minimize the longest path delay (maximize the worst design slack) subject to a constraint on static power consumption. In this thesis we focus on the first variant. An exponential function $cost_{vt} : \mathcal{B} \to \mathbb{R}$ of the form

$cost_{vt}(b)$

$$cost_{vt}(b) \quad := \quad \delta_{f(b)} \cdot e^{-vt(b)},\ b \in \mathcal{B} \tag{4.11}$$

describes the objective. The factor $\delta_{f(b)}$ depends on the logic function $f(b)$ that is implemented by b and of course the technology of the underlying design. This mapping can be extended to a mapping $cost_{vt} : \mathcal{G} \to \mathbb{R}_{\geq 0}$ with $cost_{vt}(g) = cost_{vt}(\phi(g))$.

V_t OPTIMIZATION PROBLEM

Instance

- Physical design instance.
- Timing constraints and timing rules.
- An exponential objective function $cost_{vt} : \mathcal{B} \to \mathbb{R}_{\geq 0}$.

Task Find a V_t level for each gate, i.e. a mapping $\phi : \mathcal{G} \to \mathcal{B}$ with $\phi(g) \in \mathcal{B}_g$ such that

$$\sum_{g \in \mathcal{G}} cost_{vt}(\phi(g))$$

is minimized and all timing constraints are fulfilled.

Connection with the Discrete Time-Cost Tradeoff Problem

V_t optimization is closely connected to the well-known discrete time-cost tradeoff problem, which originally comes from project scheduling:

DISCRETE TIME-COST TRADEOFF PROBLEM

Instance

- Acyclic digraph $H = (V(H), E(H))$.
- Discrete set of edge delays $delay_e \subset \mathbb{R}$ for all $e \in E(H)$.
- A cost function $cost_e : delay_e \to \mathbb{R}$ for all $e \in E(H)$.
- a) Deadline $D \in \mathbb{R}_{\geq 0}$ or b) Budget $B \in \mathbb{R}_{\geq 0}$.

Task Choose an edge delay $y_e \in delay_e$ for each $e \in E(H)$ such that

a) $cost(y) := \sum_{e \in E(H)} cost_e(y_e)$ is minimized subject to

$T(y) := \max_{P \in \mathscr{P}(H)} \sum_{e \in P} y_e \leq D$, or

b) $T(y)$ is minimized subject to $cost(y) \leq B$,

where $\mathscr{P}(H)$ denotes the set of all paths in H.

The discrete time-cost tradeoff problem is well-studied in literature and was shown to be *NP*-hard. Under the simplifying assumption that each gate can be modeled by one edge, and wire delays are considered as constant, V_t optimization can be regarded as discrete time-cost tradeoff problem.
Wittke [Wit13] proposed to model V_t optimization as a variant of the discrete time-cost tradeoff problem that takes several edges within a gate into account. This problem was also shown to be *NP*-hard.
The gate sizing problem can be regarded as a harder variant of the discrete time-cost tradeoff problem, where the sets of available edge delays depend on the current delay assignment of other edges, as the size of a gate also affects the delay of other gates. As a matter of fact, most hardness results for the gate sizing problem arise from modeling it as a discrete time-cost tradeoff problem with simplified delay models that do not take these delay dependencies into account.

$\mathbf{V_t}$ Optimization and the Geometric Program

V_t optimization can be incorporated in the geometric program (4.8), see for example Boyd et al. [Boy+05] and Chou et al. [CWC05]. However, V_t optimization is a highly discrete problem in the sense that only a very small amount of feasible options is available for each variable. Therefore it is even more difficult to force the problem into a continuous framework, and the geometric program approach combined with a rounding step has hardly been used in practice. For this reason, we do not include

V_t optimization in the theoretical discussions on gate sizing algorithms based on the convex program formulation.

4.6 Computational Complexity

Having established that all complexity results for the discrete time-cost tradeoff problem can be transferred to the gate sizing problem and V_t optimization, we also consider results published in the project scheduling context.
The discrete time-cost tradeoff problem was shown to be strongly *NP*-hard and the corresponding decision problem to be strongly *NP*-complete by De et al. [De+97]. Deineko and Woeginger [DW01] prove that the discrete time-cost tradeoff problem is *APX*-hard, and consequently no polynomial time approximation scheme exists unless *P*=*NP*. The problem is at least as hard to approximate as the vertex cover problem and therefore cannot be approximated within a factor of 1.3606 unless *P*=*NP*, see Dinur and Safra [DS05] and Grigoriev and Woeginger [GW04]. Recently, it was even shown to be *NP*-hard to approximate within any constant under the Unique Games Conjecture by Svensson [Sve13].
Independently, the gate sizing problem was shown to be strongly *NP*-hard by Li [Li94], and therefore no pseudo-polynomial algorithm exists unless *P*=*NP*. The author employed a simplified delay model, where the delay of each gate is independent of its load capacitance and wire delays are ignored, which essentially yields the discrete time-cost tradeoff problem. Earlier, Chan [Cha90] established *NP*-hardness under a similar delay model in tree networks, and provided a pseudo-polynomial algorithm for this special case. An even stronger result from Li et al. [Li+93] states that gate sizing is already *NP*-hard for networks consisting of a single chain under this delay model. The authors provided a pseudo-polynomial algorithm for series-parallel graphs.
Under the unrealistic assumption that the size of antichains in the timing graph is bounded by a constant, Liao and Hu [LH11] provide a fully polynomial approximation scheme for the delay-minimizing gate sizing problem under the Elmore delay model. So far, this result has not been extended to general graphs that are prevalent in practice, or to the gate sizing problem.
Approximation algorithms for the gate sizing problem might exist for special cases or simplified delay functions, but for Elmore delay and more complex delay models no approximation algorithms are known.

4.7 Previous Work

Common approaches for the gate sizing problem can be roughly divided into discrete and continuous methods. Continuous methods assume gate sizes to be continuous, and are content with a continuous solution or apply a rounding step in the end. Discrete approaches tackle the discrete problem directly. A comprehensive introduction to gate sizing and V_t optimization and existing work can for example be

found in Lee and Gupta [LG12].
Until recently, it was hard to compare different algorithms because experimental results were often generated on industrial designs that were not publicly available. We start with a brief overview over public benchmarks:

4.7.1 Industrial Benchmarks

The need for publicly available and realistic benchmarks to compare different methods for gate sizing and V_t optimization was addressed by the organizers of the first ISPD 2012 Gate Sizing Contest (Ozdal et al. [Ozd+12]). The benchmark suite contains 14 designs with a realistic standard circuit library and static power information. The benchmarks from the ISPD Gate Sizing Contest 2013 by Ozdal et al. [Ozd+13] constitute a further improvement in this area: Compared to the 2012 contest, these benchmarks now provide a realistic distributed RC model for wires. The largest benchmark chip consists of almost 900000 gates plus latches with fixed size. As in the previous contest benchmarks, the input consists of a realistic standard circuit library, and a netlist with timing constraints. The primary objective is to satisfy all timing constraints, while minimizing the static power consumption. Note that for each gate different V_t levels are available, and also V_t optimization needs to be incorporated in an algorithm to achieve timing closure. The 2013 benchmarks are harder in the sense that more near-critical paths exists even in well-sized situations.
Apart from the ISPD benchmarks there have been other attempts to construct benchmarks that enable a classification of existing algorithms. The ISCAS benchmark suite by Brglez and Fujiwara [BF85] was often used for comparison, but the largest chip in this suite contained less than 4000 gates, whereas state-of-the-art chips contain up to several million gates. Gupta et al. [Gup+10] present the "Eyechart" benchmark suite that can be optimally sized with dynamic programming, but the circuit topologies are not realistic and gate timing is independent of slew. Kahng and Kang [KK12] use a similar strategy to construct benchmarks with more realistic topologies, but with a simplified delay model that is independent of slews. An optimal solution can be found with dynamic programming.

4.7.2 Continuous Approaches

Geometric Program Based

The mathematically best founded approaches rely on the geometric program formulation. A posynomial-based formulation was first presented by Fishburn and Dunlop [FD85] for the transistor sizing problem with the Elmore delay model [Elm48] for transistor delays. Sapatnekar et al. [Sap+93] were the first to compute exact solutions for the transistor sizing problem with a general-purpose solver for convex programs under a variant of the Elmore delay model. Kasamsetty et al. [KKS00] proposed a more accurate posynomial delay model for gate delays than the Elmore delay model, and applied the same optimization approach as Sapatnekar et

al. [Sap+93]. General-purpose solvers for geometric programs usually implement interior point methods with polynomial running time. Unfortunately, interior point methods consume significant memory and running time which makes them unsuitable for large instance sizes as they occur in gate sizing. The largest instance sizes reported to be solved (Joshi and Boyd [JB08]) come from gate sizing and contain up to 100000 gates. The authors reported that for larger instance sizes their customized geometric program solver, which implements a primal-dual interior point method, failed to provide a solution. Additionally, running times were expensive ("tens of hours"). The authors further developed a truncated Pseudo-Newton approach that allowed them to tackle larger instance sizes of up to one million gates. Unfortunately, they could not give any theoretical bounds on the quality of their solution, and they could only establish optimality of their solutions for the smaller instances by comparison with a customized geometric program solver.
Additionally, the geometric program for gate sizing was tackled by several researchers using standard solvers as well as custom methods under different posynomial delay models. For example, Menezes et al. [MBP97] applied sequential quadratic programming and approximated the objective function by a quadratic function, and the constraints by linear functions.
Boyd et al. [Boy+05] provide a tutorial on the geometric program formulation and show how more constraints and objectives such as supply voltage and V_t level optimization can be included in the geometric program.

Lagrangian Relaxation

Marple [Mar89] proposed to relax the timing constraints of the transistor sizing problem with Lagrange multipliers. He applied the Lagrangian augmentation technique, where an extra penalty term is added to the Lagrange function to steer the solution towards the feasible region.
Chen et al. [CCW99] give a thorough introduction into Lagrange relaxation for the gate sizing problem. They show that the Lagrange function can easily be simplified when the Lagrange multipliers are restricted to the non-negative network flow space, in other words they fulfill the flow conservation rule at all vertices $v \in V \setminus \{V_{start} \cup V_{end}\}$. This is not possible with the Lagrangian augmentation technique. The projected subgradient method solves the dual problem, and in each iteration a greedy algorithm for the Lagrange primal problem computes a new subgradient. The Lagrange multiplier projection to the non-negative flow space can be formulated as quadratic cost flow problem and is solvable in polynomial time. Chen et al. [CCW99] observed that exact projection dominated the running time of their algorithm and used a heuristic projection step instead. Further constraints, for example on clock skew, can easily be incorporated into the geometric program and be relaxed by Lagrange multipliers. The approach can be extended to more accurate posynomial delay models, see for example Rahman et al. [RTS11]. The fact that the duality gap is zero if a strongly feasible solution for the geometric program exists further encourages its use. The work of Chen et al. [CCW99] is the

groundwork for Lagrange relaxation based algorithms for both the continuous and the discrete problem. Wang et al. [WDZ07] provide further theoretical analysis on this formulation, and show that the dual objective function is differentiable, allowing the use of the projected gradient method. The Lagrange relaxation approach, related subproblems and extensions will be discussed in detail in Chapter 6 and Chapter 7.

Linear Programming and Network Flows

The simplest model of the continuous relaxation linearizes the delays and the objective function (Berkelaar and Jess [BJ90], Chinnery and Keutzer [CK05]). Nguyen et al. [Ngu+03] use a linear program to distribute slacks to the gates in the design, and realize these targets with gate sizing and V_t assignment. Vygen [Vyg01] and Ren and Dutt [RD08; RD13] model gate sizing as slightly different minimum-cost network flow problems. Available sizes and their costs are encoded in a graph, and the flow indicates which sizes are used in an optimal solution.

Continuous-guided

Continuous-guided approaches compute an optimal continuous solution and use it as a basis for further optimization. However, rounding a continuous solution is not an easy task and the literature on rounding is sparse. It was observed in practice that choosing the "closest" discrete solution can lead to large timing violations, see for example Hu et al. [HKH09] and Rahman et al. [RTS11]. To overcome this problem, Hu et al. [HKH09] propose to evaluate several discrete sizes close to the continuous solution with dynamic programming. The gates are traversed via breadth-first search, and several sizing solutions are propagated for each gate. Solutions are pruned based on path delays and area consumption. In the branch-and-bound approach of Rahman et al. [RTS11], the solution set for each gate consists of a set of sizes close to the continuous solution. Chuang et al. [CSH95] take a linear program solution and allow the next smaller and next larger size as option for each gate. Gates already set to their minimum size are not changed, but for the remaining gates all allowed options are enumerated. This yields near-optimal results, indicating that the sizes in an optimal discrete solution are not necessarily the closest to the continuous solution. Wu and Davoodi [WD08] employ a branch-and-bound algorithm to compute optimal discrete gate sizes, and consider solutions obtained by different rounding strategies for comparison. They conclude that rounding to the closest discrete solution yields large timing violations, which are then fixed by iteratively increasing gate sizes. This comes with an objective function increase by up to 51%. A branch-and-bound technique with a restricted feasible set that consists of the next smaller and the next larger gate size returned solutions with objective function values that are only 0.7% larger than in the optimal solution in average. A recent paper of Xie and Chen [XC15] presents a modified Elmore delay model for the ISPD 2012 benchmarks. A continuous solution is computed with

a Lagrangian relaxation based algorithm, and nearest rounding combined with a postoptimization step returns a discrete solution. Shah et al. [Sha+05] propose a novel formulation for continuous sizing and V_t optimization that models different V_t levels for a gate as a single circuit which is a parallel combination of the high and low V_t gate. Without constraints on the gate sizes, the V_t levels are all set to a feasible discrete value in an optimal solution.
Farshidi et al. [Far+13] regard gate sizing as a multi-objective optimization problem with objectives power, area and delay minimization. The objective of their geometric program formulation is the weighted sum of conflicting objectives for gate sizing, and the weights are also regarded as variables.

4.7.3 Discrete Approaches

Discretized Lagrangian Relaxation

The Lagrange relaxation approach can easily be discretized by solving a discrete primal problem. However, this leads to a gradient with error in each iteration of the projected subgradient method. As there exists no approximation algorithm for minimizing the Lagrange function over a discrete set, the magnitude of this error is unknown and consequently, no convergence guarantees exist for this approach.
Nonetheless, discretized Lagrange relaxation was employed in several gate sizing algorithms, for example by the winning algorithm of the ISPD 2013 Gate Sizing Contest (Ozdal et al. [Ozd+13]), see Flach et al. [Fla+14]. Heuristic variants of the subgradient method to solve the dual problem were proposed by Tennakoon and Sechen [TS08], Huang et al. [HHS11], Ozdal et al. [OBH12] and Livramento et al. [Liv+14]. An advantage of this approach is the easy integration of V_t optimization and other timing constraints. A recent paper by Roy et al. [Roy+15] optimizes power consumption for multiple operating conditions of a chip based on discretized Lagrangian relaxation for gate sizing and V_t optimization. The first multi-threaded version is described in Sharma et al. [Sha+15] and achieves average speedups of 5.23x with 8 threads compared to a sequential implementation.
We refer the reader to Chapter 6 and Chapter 7 for more details on discretized Lagrange relaxation.

Sensitivity-based Heuristics

Lagrange relaxation is often combined with local search heuristics or dynamic programming algorithms for fine-tuning and to further improve path delays and the objective function value, see for example Livramento et al. [Liv+14] for a recent work. Local search heuristics iteratively size gates or change their V_t level, often in the order of their sensitivity to changes. The basic idea behind the sensitivity computation is to first optimize those gates that have a large impact on overall timing, and to achieve a good tradeoff between sometimes conflicting objectives like power consumption and worst slack or SNS optimization. This idea was first

implemented in the TILOS algorithm (Fishburn and Dunlop [FD85]), and since then many variants have been developed.
Heuristic algorithms were also proposed as stand-alone flows. For example, the algorithm of Schietke [Sch99] iteratively chooses an operation that realizes a good tradeoff between objective function and costs. Changing the size of a gate is a possible operation. Different objective functions and costs are considered, for example power consumption and signal delay.
The heuristic gate sizing flow of Hu et al. [Hu+12] consists of two stages: global timing recovery and power reduction with feasible timing. In the first stage, gates are iteratively set to a larger size or lower V_t level in the order of their sensitivity to obtain a timing feasible design. In the second stage, leakage power is reduced by downsizing or increasing V_t levels in order of the gate sensitivity. Thereby a set of sensitivity functions is defined and parameterized, such that it can be traversed in parallel to find the best solution using multistarts. It is remarkable that none of the sensitivity functions dominated the results, in other words the best results on the benchmarks were obtained with different sensitivity functions. [Hu+12] outperformed the best known results on the ISPD 2012 benchmarks, which is surprising given that only local search heuristics are used. The successor paper of Kahng et al. [Kah+13] adapted these algorithms to the ISPD 2013 benchmarks and achieved the second place in the contest.
Li et al. [Li+12a] reduce the power consumption of a Lagrange relaxation solution with an algorithm that is based on a network-flow formulation. Afterwards, gates are iteratively set to books with less power in order of their sensitivity. Here sensitivity is defined as the power change divided by the delay change.

Dynamic Programming

Liu and Hu [LH10] and Ozdal et al. [OBH12] combine Lagrange relaxation with dynamic programming to solve the primal problem. Ozdal et al. [OBH12] further point out the importance of using accurate timing information from an industrial engine to achieve better results on industrial designs.

Delay/Slew Budgeting

Many fast algorithms for gate sizing rely on delay or slew budgeting. In a first step, a delay or slew budget is assigned to each gate. Secondly, a minimum size solution is computed that retains these budgets. Dai and Asada [DA89] proposed the first delay budgeting algorithms for transistor sizing. See for example Nguyen et al. [Ngu+03] and Held [Hel09] for allocation of slack and slew budgets to gates, respectively.

Game theoretic approaches

Game theoretic approaches that model gate sizing as a resource allocation problem and compute a Nash equilibrium can for example be found in Murugavel and

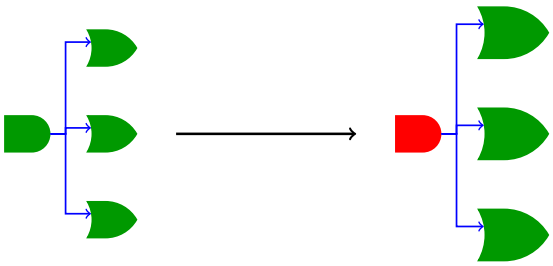


Figure 4.3: Rounding to the nearest discrete solution leads to timing violations.

Ranganathan [MR04] and Hanchate and Ranganathan[HR06]. A recent algorithm of Casagrande and Ranganathan [CR15] is based on fuzzy space games. Thereby each gate is a player, and strategies of players correspond to available sizes. Due to runtime reasons, each game consists of three players only: One gate that consumes a large amount of dynamic power, and its most power-consuming predecessor and successor gate. In each game, the possible strategies of the players are enumerated, and infeasible strategies (for example sizes that cause load violations or timing degradations) are identified.

4.8 Rounding a Continuous Solution

Rather informally, the rounding problem consists of finding a discrete feasible solution that is "close" to the continuous solution in the following sense: The difference between the objective function values is small, and the timing constraints are (still) fulfilled. Ideally, the edge delays in the discrete solution should not be much larger than in the continuous solution. This is not an easy task, and no approximation algorithm is known.
Having discussed practical observations in Section 4.7.2, we now consider specific examples as they might occur in practice that support these observations.

Example 4.3 Consider Figure 4.3: On the left picture you see a continuous solution for gates in a net. The driver gate and three sink gates have approximately the same size. In the discrete solution shown in the picture on the right side the driver has been rounded to a smaller size, while the sink gate sizes have increased. Then the total capacitance of the net cannot be driven by the smaller driver gate, which leads to a load capacitance violation at the driver gate, and possibly slew violations at the sink pins of the net. Even if rounding does not introduce an electrical violation, the net can be substantially slowed down due to the increased load capacitance, which in turn leads to smaller slacks.

In consideration of Example 4.3, it would be straightforward to round down all sizes to a smaller discrete solution. However, power consumption would decrease but timing violations can still be introduced. To see this, consider Example 4.3 and assume that the continuous size of the driver gate is $(s_2 - \epsilon_2)$, with s_2 being the

closest feasible discrete size and $\epsilon_2 > 0$. Similarly, let $(s_1 + \epsilon_1)$ be the continuous sizes of the sink gates with s_1 being the closest feasible discrete size and $\epsilon_1 > 0$. Depending on the ratio of ϵ_1 and ϵ_2, the driver gate might be too small to drive the load capacitance even when all sizes are rounded down.
Alternatively, if optimal continuous arrival times are also known, gate sizes can be rounded such that these arrival times are fulfilled as best as possible. Unfortunately, no performance guarantees exist for this method either.
Now consider the discrete time-cost tradeoff problem as a special case of the gate sizing problem, and suppose we are given a solution of the linear relaxation. It was established by Skutella [Sku97] that no performance guarantee exists if a feasible solution to the discrete problem is compared with the optimal solution of the linear relaxation. We migrated his example to the gate sizing framework and obtain arbitrarily bad approximations in theory:

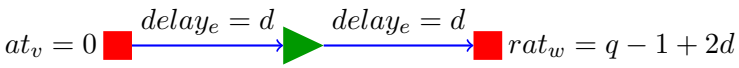


Figure 4.4: No performance guarantee exists if a feasible solution to the discrete problem is compared with the optimal solution of the relaxation.

Example 4.4 ([Sku97]) Figure 4.4 shows a single inverter g connected with a primary input and a primary output pin. We assume that wire delays are constant $d \in \mathbb{R}$, and that two sizes are available for g: Size s_1 has gate delay 0 and a total power consumption of 2. Size s_2 has gate delay $q \in \mathbb{N}$ and consumes 0 total power. The required arrival time at the primary output is $q - 1 + 2d$. Then the optimal solution $\xi \in [s_1, s_2]$ to the continuous relaxation has delay $q - 1$ and power consumption $\frac{2}{q}$. Rounding ξ to s_2 results in a timing violation of 1. Rounding ξ to s_1 leads to a timing feasible solution with power consumption 2, that is larger than the power consumption of ξ by a factor of q. Since q can be chosen arbitrarily large, no approximation guarantee can be given.

This example can be extended to more complex graphs or more complex timing models.

4.9 Comparison of Existing Approaches

It is an interesting question which heuristic performs best in practice, and whether a discrete approach for the gate sizing problem is preferable to a continuous approach. On the one hand, continuous approaches provide a solid mathematical background, except for the rounding step. On the other hand, we deem a discrete approach preferable for several reasons: Foremost is the fact that industrial circuit libraries offer only a discrete set of sizes for each gate. Secondly, rounding a continuous solution destroys all performance guarantees. Additionally, it is hard to find

good posynomials for delay and slew approximation, since real delays are solutions of differential equations, and more complex timing constraints should be taken into account. For example, books realizing the same logic function may scale well with the gate size, but books for different functions have different delay characteristics. This leads to the use of inaccurate delay models in continuous formulations. Discrete algorithms can use more accurate table look-up methods or an industrial timing engine to compute signal delays.
However, there is hope that the error introduced by discretizing a continuous approach is small at least for server designs. This is based on the observation that the discrete problem is at least "close to convexity" in the following sense: Timmermeister [Tim12] reported that a dynamic programming algorithm rarely improved the solution found by a local search algorithm, implying that the local solution is at least close to the global optimum solution.
An interesting research question concerns the integration of algorithms with good performance on the ISPD benchmarks into an industrial design flow. A very recent paper of Reimann et al. [RSR15] deals with the arising challenges. We refer to Chapter 9 for a more detailed discussion and practical observations.

5 Gate Sizing for Power-Delay Tradeoff

This chapter deals with the *power-delay tradeoff problem* which consists of finding gate sizes that minimize a weighted sum of power and signal delays in the timing graph $G = (V, E)$. We will encounter this as a subproblem in later chapters, where the weights are updated iteratively until a good solution has been found.
We are interested in the continuous and discrete version of this problem. More formally, we consider the following problem:

Power-delay tradeoff problem

$tr(x, \omega)$

Power-delay tradeoff problem

Continuous version:

$$\min_{x \in X_{cont}} tr(x, \omega) := \omega_{m+1} cost(x) + \sum_{e \in E} \omega_e delay_e(x), \tag{5.1}$$

Discrete version:

$$\min_{x \in X_{disc}} tr(x, \omega) := \omega_{m+1} cost(x) + \sum_{e \in E} \omega_e delay_e(x). \tag{5.2}$$

Here $\omega \in \mathbb{R}^{m+1}_{\geq 0}$ are the *weights*, with $m = |E|$. We refer to ω_{m+1} as the *power weight* and to the weights ω_e, $e \in E$, as *edge (delay) weights*.
The functions $delay_e(x)$ are the delay functions defined in Section 4.4, and $cost(x)$ is the objective function of the gate sizing problem modeling power consumption. For $x \in X_{cont}$, these functions are continuous and convex.

$\omega \in \mathbb{R}^{m+1}_{\geq 0}$

$\omega_e,\ \omega_{m+1}$

Without loss of generality we can assume that ω_{m+1} equals 1, then $tr(x, \omega)$ is equivalent to the Lagrange primal function for gate sizing (see Chapter 6), and the edge weights correspond to the Lagrange multipliers. The Lagrange primal function needs to be minimized in each iteration of the projected gradient method to solve the Lagrange dual problem. The vector $\big(delay_e(x)\big)_{e \in E}$ will be used as the gradient, and thus we are interested in algorithms with an approximation guarantee for the gate sizes rather than an approximation guarantee for the value of $tr(x, \omega)$.
In Chapter 7 and 8 we are interested in an approximation guarantee for the value of $tr(x, \omega)$. We approximately solve a feasibility problem for gate sizing with the multiplicative weights algorithm in Chapter 7, where the power-delay tradeoff problem occurs as a subproblem. In the resource sharing framework (Chapter 8), we regard

power and edge delays as *resources*. The weighted resource usage of a so-called *gate customer*, which has to be minimized, is of the form $tr(x, \omega)$.
The remainder of this chapter is organized as follows: In Section 5.1 we consider the continuous problem (5.1) which can be solved in polynomial time for example with interior point methods up to any accuracy $\epsilon > 0$. We first investigate properties of $tr(x, \omega)$. Next we consider the so-called *local refinement* algorithm which was originally developed to minimize the Lagrange primal function. Afterwards we show that the conditional gradient method approximates the value of $tr(x, \omega)$ up to any desired accuracy.
The discrete problem (5.2) is the subject of Section 5.2. We show that under simplified delay models, approximation algorithms exist. The local refinement algorithm can be discretized, but no approximation guarantee is known. Section 5.2.3 presents an FPTAS for problem (5.2) under the assumption that the number of gates per level in the gate graph, i.e. the size of antichains, is bounded by a constant.

Extension of the tradeoff function Note that $tr(x, \omega)$ is of the general form

$$tr(x) := \sum_{i=1}^{n} \zeta_i e^{x_i} + \sum_{i=1}^{n} \chi_i e^{-x_i} + \sum_{i,j=1}^{n} \psi_{ij} e^{x_i - x_j}, \tag{5.3}$$

where $\zeta_i, \chi_i, \psi_{ij} \in \mathbb{R}_{\geq 0}$ for all $1 \leq i, j \leq n$ are constants (cf. Section 4.4). Note that no constants occur in the exponents. Here the weights ω are included in the constants, which also incorporate unit capacitance, unit resistance, the objective scaling factor, and wire capacitances and resistances. Algorithms for the continuous relaxation presented in this chapter also minimize functions of the form (5.3). In later chapters we extend the Lagrange function and the resource sharing framework to incorporate additional constraints on placement density, capacitance and slew violations. The resulting subproblems can be brought to the form (5.3).

5.1 The Continuous Power-Delay Tradeoff Problem

Problem (5.1) is solvable in polynomial time for example with interior point methods. The running time complexity of interior point methods is approximately cubic in the number of gates (cf. Section 3.4), and we are interested in faster algorithms. We first examine properties of $tr(x, \omega)$, and consider algorithms afterwards.

5.1.1 Properties of $\mathbf{tr}(\mathbf{x}, \omega)$

Lemma 5.1 *$tr(x, \omega) : X_{cont} \to \mathbb{R}$ is twice differentiable and strictly convex for $x \in X_{cont}$, but in general not strongly convex.*

Proof. As $tr(x, \omega)$ is the sum of exponential functions, it is strictly convex and twice differentiable. It is not necessarily strongly convex because the second deriva-

tive, which also is a sum of exponential functions, can become arbitrarily small, for example when $cost(x)$ and $delay_e(x)$ become arbitrarily small for all $e \in E$. □

For each $g_i \in \mathcal{G}$ we denote with $tr_x(x_i, \omega)$ the terms of $tr(x, \omega)$ which depend on the size variable x_i, and call it the *local refine function of* g_i: Local refine function $tr_x(x_i, \omega)$

$$tr_x(x_i, \omega) := \omega_{m+1} cost(x_i) + \sum_{e \in E_{g_i}} \omega_e delay_e(x), \tag{5.4}$$

where all entries of x are fixed except for the i-th entry. E_{g_i} are the edges in the neighborhood graph G_{g_i} of g_i, i.e. the edges in the timing graph whose delay changes when the size of g_i is altered. This edge set can be extended if more accurate delay models are used.

It is reasonable to assume that the fanout of each gate is bounded by a constant in practice. Similarly, the number of input pins of a gate is bounded by a constant, and thus is the number of edges in the neighborhood graphs. We set $\Lambda := \max_{i=1,\ldots,n} |E_{g_i}|$. $|E_{g_i}| \leq \Lambda$

It was implicitly shown by Chen et al. [CCW99] in the context of gate and wire sizing that the derivatives $\frac{\partial tr_x}{x_i}(x_i, \omega)$ of the local refine functions are the partial derivatives of $tr(x, \omega)$. Langkau [Lan00] explicitly formulated the derivatives for gate sizing. The Lipschitz constant of the gradient depends on the weights and is denoted by $lip(\omega)$: $lip(\omega)$

Lemma 5.2 *The gradient $\nabla tr(x, \omega)$ is Lipschitz continuous on the set X_{cont} with Lipschitz constant $lip(\omega)$ and we have*

$$lip(\omega) \quad \leq \quad \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left(\omega_{m+1} cost(x_i) + \Lambda \cdot \max_{e \in E} \omega_e delay_e(x) \right).$$

Proof. The gradient $\nabla tr(x, \omega)$ is the vector of derivatives of the local refine functions (5.4). $tr(x, \omega)$ is the sum of exponential functions and of the form (5.3), thus each partial derivative $\frac{\partial tr_x}{x_i}(x_i, \omega) : [l_i, u_i] \to \mathbb{R}$ is the sum of (negated) exponential functions of x_i: If g_i is a driver gate, the term $\gamma \cdot e^{-x_i}$ occurs in (5.3), whose derivative is $-\gamma \cdot e^{-x_i}$. The other terms remain. We can therefore bound

$$\frac{\partial tr_x}{x_i}(x_i, \omega) \leq tr_x(x_i, \omega).$$

Now recall the definition of the Lipschitz constant:

$$\begin{aligned} lip(\omega) \quad &= \quad \max_{x,y \in X_{cont}} \frac{||\nabla tr(x, \omega) - \nabla tr(y, \omega)||_\infty}{||x - y||_\infty}, \\ &\leq \quad \max_{x,y \in X_{cont}} \max_{1 \leq i \leq n} \frac{\frac{\partial tr_x}{\partial x_i}(x_i, \omega) - \frac{\partial tr_y}{\partial y_i}(y_i, \omega)}{x_i - y_i} \end{aligned}$$

$$
\begin{aligned}
&\leq \max_{z \in X_{cont}} \max_{1 \leq i \leq n} \frac{\partial tr_z}{\partial z_i^2}(z_i, \omega) \\
&\leq \max_{z \in X_{cont}} \max_{1 \leq i \leq n} tr_z(z_i, \omega) \\
&= \max_{z \in X_{cont}} \max_{1 \leq i \leq n} \left(\omega_{m+1} cost(z_i) + \sum_{e \in E_{g_i}} \omega_e delay_e(z_i) \right)
\end{aligned}
$$

The second inequality follows from the mean value theorem, and for the third inequality we use the same argument as above, namely that $tr_x(x_i, \omega)$ is the sum of exponential functions. Because the number of edges in the neighborhood graphs can be bounded by Λ, we have

$$
\begin{aligned}
lip(\omega) &\leq \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left(\omega_{m+1} cost(x_i) + \sum_{e \in E_{g_i})} \omega_e delay_e(x_i) \right) \\
&\leq \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left(\omega_{m+1} cost(x_i) + \Lambda \cdot \max_{e \in E} \omega_e delay_e(x) \right).
\end{aligned}
$$

□

Note that the gradient $\nabla tr(x, \omega)$ is Lipschitz continuous on X_{cont} because X_{cont} is bounded: On $\mathbb{R}^n$ the exponential function becomes arbitrarily steep, and consequently $tr(x, \omega)$.

5.1.2 Approximating Gate Sizes

We will consider the local refinement algorithm that belongs to the class of coordinate descent algorithms and which minimizes general functions of the form (5.3) (Chu and Wong [CW01]). It consists of a series of local refinement operations:

Definition 5.3 (Local refinement operation (Cong and Hu [CH96])) *Given a function $tr(x)$ of the form (5.3) and a solution $x' \in X_{cont}$, the local refinement operation for any variable x_i, $i = 1, \ldots, n$ is to minimize $tr(x)$ by only varying x_i, with $l_i \leq x_i \leq u_i$, while keeping the value of all other variables x'_j $(j \neq i)$ fixed.*

Local refinement was proposed by several authors to solve the Lagrange primal problem, for example by Cong and Hu [CH96] and Chu and Wong [CW99] in the context of transistor/gate and wire sizing. The linear convergence of this algorithm essentially follows from a general result of Luo and Tseng [LT92], but Chu and Wong [CW99] first give explicit error estimations and a new proof for the special case that the timing graph is a tree. This result has been extended by Langkau [Lan00] and Szegedy [Sze05] in the context of gate sizing for more general topologies and with simpler proofs.

Chen et al. [CCW99] showed that starting with the smallest size for each gate leads to convergence towards the optimal solution of the Lagrange primal problem. Later, Chu and Wong [CW01] extended this result for arbitrary start solutions.

Local refinement algorithm

In the following we formulate the *local refinement algorithm* for minimizing $tr(x, \omega)$ and restate convergence results. The algorithm is summarized in Algorithm 5.1. The local refine function $tr_x(x_i, \omega)$ can be minimized by setting its first derivative with respect to x_i to zero. Let $\tilde{x_i}$ be the solution. Then a vector $\bar{x} \in X_{cont}$ minimizes $tr(x, \omega)$ if for all $\bar{x}_i,\ i = 1, \ldots, n$, the following holds:

$$\bar{x}_i = \min\{u_i, \max\{l_i, \tilde{x_i}\}\}. \tag{5.5}$$

This was proved, for example, by Chen et al. [CCW99] who exploited the convexity of $tr(x, \omega)$. Langkau [Lan00] used Lagrangian relaxation and the KKT-conditions to establish optimality of $\bar{x}$: The conditions on the size boundaries, i.e. $l_i \leq x_i \leq u_i$ for each size variable x_i, were relaxed and incorporated into the objective function. The resulting duality gap equals zero and the KKT-conditions imply (5.5).

Algorithm 5.1 Continuous Local Refinement

```
1:  procedure LocalRefine(x, ω)
2:      k ← 0
3:      x^(0) ← x
4:      while (k = 0 or ∃i : x_i^(k) ≠ x_i^(k−1)) do
5:          x^(k+1) ← x^(k)
6:          for each g_i ∈ G in topologial order do
7:              x̃_i^(k+1) ← arg min tr_{x^(k+1)}(x_i^(k+1), ω) (local refinement operation)
8:              x_i^(k+1) ← min{u_i, max{l_i, x̃_i^(k+1)}}
9:          end for
10:         k ← k + 1
11:     end while
12: return x^(k) = (x_1^(k), ..., x_n^(k))
13: end procedure
```

Chu and Wong [CW01] and Langkau [Lan00] essentially used the same techniques to show convergence of Algorithm 5.1 by generalizing the proofs of Chu and Wong [CW99], but Chu and Wong [CW01] treated a more general class of optimization problems. The following theorem is basically due to these works:

Theorem 5.4 ([CW01], [Lan00]) *Let x^* be the optimal solution of (5.1). If the fanout of each gate is bounded by a constant, Algorithm 5.1 finds a solution $x \in X_{cont}$ with $|(x_i^* - x_i)/x_i^*| \leq \epsilon$ for all $i = 1, \ldots, n$ in $O(n \log(1/\epsilon))$ time for $\epsilon > 0$. Each iteration takes $O(n)$ time.*

Szegedy [Sze05] proposed a slightly different variant of Algorithm 5.1 that sizes all gates simultaneously in each iteration, and obtains slightly different error bounds.

Remark 5.5 The performance of Algorithm 5.1 is unknown if changes are forbidden that increase load or slew violations. If $x_i^{(k)}$ is a forbidden size, the most natural alternative is the closest size that is not forbidden. However, the proof of Theorem 5.4 relies on the fact that the variable changes in each iteration can be bounded from above and below naturally. This is not necessarily possible if some sizes are forbidden.
Similarly, no convergence guarantees exist in general if the discrete size is chosen for which the local refine function is minimal, or if the continuous size is rounded to the closest discrete size in each local refinement step.

Remark 5.6 Chen et al. [CCW99] apply Lagrangian relaxation to the geometric program (4.8) and formulate the local refinement algorithm for a posynomial function. Chu and Wong [CW01] formulate their algorithm for general posynomial functions as well. However, it was pointed out by Wang et al. [WDZ07] that Lagrangian relaxation should rather be applied to the convex program (4.10) obtained by variable transformation, see also Section 6.1.2 for a discussion. As we will work with the convex program in the following chapters, we reformulated the local refinement algorithm and convergence results for the convex problem to avoid confusion, without changing the tenor of the results.

5.1.3 Approximating the Value of $\mathbf{tr}(\mathbf{x}, \omega)$

tr_{opt}

In this section we consider algorithms that provide an approximation guarantee on the optimal value tr_{opt} of (5.1), which is necessary to fit gate sizing into the multiplicative weights and resource sharing framework.
Theorem 5.4 gives an approximation guarantee on the values of the gate sizes returned by Algorithm 5.1, but it is not clear how to translate this guarantee to a bound on the difference $tr(x^{(k)}, \omega) - tr(x^*, \omega)$ for the optimal solution x^*. Such a bound would certainly depend on the value of the weights, which will grow exponentially in our later application. The conditional gradient method (see Section 3.3), yields a better guarantee. It is summarized in Algorithm 5.2. Algorithm 5.3 provides a multiplicative guarantee on tr_{opt}.

An Additive Approximation Guarantee for $\mathbf{tr}(\mathbf{x}, \omega)$

$diam_X$

We denote the diameter of set X_{cont} with

$$diam_X := \max_{1 \le i \le n} (u_i - l_i).$$

Theorem 5.7 *For each $\epsilon > 0$, Algorithm 5.2 finds in $O\left(\frac{n \cdot diam_X^2 \cdot lip(\omega)}{\epsilon}\right)$ time a solution $x \in X_{cont}$ with*

$$tr(x, \omega) \le tr_{opt} + \epsilon.$$

Algorithm 5.2 Conditional Gradient Method

procedure ConditionalGradient(ϵ, ω)
  Initialize $x^{(0)} \in X_{cont}$
  for $k = 0$ to $O\left(\frac{diam_X^2 \cdot lip(\omega)}{\epsilon}\right)$ **do**
    $x^{(k+1)} \leftarrow$ ConditionalGradientIteration($x^{(k)}, \omega$)
    $k \leftarrow k + 1$
  end for
return $x^{(k)} = (x_1^{(k)}, \ldots, x_n^{(k)})$ with $tr(x^{(k)}, \omega) \leq tr_{opt} + \epsilon$
end procedure

procedure ConditionalGradientIteration($x^{(k)}, \omega$)
  Compute gradient $d^{(k)} \leftarrow \nabla tr(x^{(k)}, \omega)$
  Find $s^{(k)} \in \arg\min_{s \in X_{cont}} \langle d^{(k)}, s - x^{(k)} \rangle$
  $\gamma^{(k)} \leftarrow \frac{2}{k+2}$
  $x^{(k+1)} \leftarrow x^{(k)} + \gamma^{(k)}(s^{(k)} - x^{(k)})$
return $x^{(k+1)}$
end procedure

Proof. In iteration k of the conditional gradient method the gradient $d^{(k)} := \nabla tr(x^{(k)}, \omega)$ is computed and a linear subproblem

$$s \in \arg\min_{s \in X_{cont}} \langle d^{(k)}, s - x^{(k)} \rangle \tag{5.6}$$

is solved to obtain a search direction s which guarantees that the next iterate $x^{(k+1)} := x^{(k)} + \gamma^{(k)}(s - x^{(k)})$ belongs to the feasible set X_{cont}. Here $\gamma^{(k)}$ is the step size. In the basic variant, it is set to $\gamma^{(k)} := \frac{2}{k+2}$.

The subproblem (5.6) can be solved in linear time: For each entry s_i we set

$$s_i = \begin{cases} l_i & \text{if } d_i^{(k)} > 0 \\ u_i & \text{if } d_i^{(k)} < 0 \\ x_i^{(k)} & \text{otherwise.} \end{cases} \tag{5.7}$$

Note that $d_i^{(k)} = 0$ implies that s_i can be chosen arbitrarily. However, if we choose s_i as above, the i-th entry of $x^{(k+1)}$ equals the i-th entry of $x^{(k)}$, as it is indicated by the zero gradient. Convergence analysis of this algorithm (see for example Jaggi [Jag13]) yields that after $k \geq 1$ iterations, the following holds for the iterate $x^{(k)}$:

$$tr(x^{(k)}, \omega) - tr_{opt} \leq \frac{2C_L}{k+2}, \tag{5.8}$$

where the constant C_L is defined as

$$C_L := \sup_{x,s \in X_{cont}, \gamma \in [0,1]} \frac{2}{\gamma^2} \Big(tr(x + \gamma(s-x), \omega) - tr(x,\omega) - \gamma \langle s - x, \nabla tr(x,\omega) \rangle \Big).$$

The constant C_L can be seen as capturing the non-linearity of $tr(x,\omega)$, and is zero for linear functions. For linear functions with bounded domain, the optimal solution is hence found in iteration $k = 0$.
The gradient of $tr(x,\omega)$ is Lipschitz continuous on the domain X_{cont} with Lipschitz constant $lip(\omega)$ by Lemma 5.2, in which case C_L can be bounded by

$$C_L \leq \max_{x,y \in X_{cont}} (||x-y||_\infty^2) \cdot lip(\omega) \leq diam_X^2 \cdot lip(\omega).$$

This bound is due to Nesterov [Nes04] (Lemma 1.2.3) and Jaggi [Jag13]. It is notable that the above holds for any norm. Simple transformation of (5.8) yields that a solution with accuracy ϵ can be computed in $\left\lceil \frac{diam_X^2 \cdot lip(\omega)}{\epsilon} - 2 \right\rceil$ iterations.
The total running time of the conditional gradient method is $O\left(\frac{n \cdot diam_X^2 \cdot lip(\omega)}{\epsilon}\right)$, as computing the gradient and solving the linear subproblem takes $O(n)$ time. □

Note that the running time of Algorithm 5.2 depends on the Lipschitz constant $lip(\omega)$, and thus on the weights ω, which can grow exponentially.

Remark 5.8 The gradient is only Lipschitz continuous for $x \in X_{cont}$, but this is sufficient to prove the bound on C_L: Lipschitz continuity is only needed in Lemma 1.2.3 in Nesterov [Nes04] to show that

$$|tr(y,\omega) - tr(x,\omega) - \langle \nabla tr(x,\omega), y - x \rangle| \leq \frac{lip(\omega)}{2} ||y-x||^2$$

holds for $y = x + \gamma(s-x),\ x, s \in X_{cont},\ \gamma \in [0,1]$. Therefore it is sufficient that $\nabla tr(x,\omega)$ is Lipschitz continuous only over the domain X_{cont}.

A Multiplicative Approximation Guarantee for $\mathbf{tr}(\mathbf{x}, \omega)$

In Chapter 7 we are interested in a multiplicative approximation guarantee on the optimal value tr_{opt}. This requires a slightly different precision and analysis, and the algorithm is summarized in Algorithm 5.3. Let

tr_{ratio}

$$tr_{ratio} := \frac{\max_{x \in X_{cont}} \max_{1 \leq i \leq n} \{cost(x_i) + \Lambda \cdot \max_{e \in E} delay_e(x)\}}{\min_{x \in X_{cont}} \{cost(x), \min_{e \in E} delay_e(x)\}}.$$

The running time of Algorithm 5.3 depends on the value of tr_{ratio}.

$\eta > 1$

Theorem 5.9 *Let tr_{opt} be the optimal value of (5.1). For each $\eta > 1$ Algorithm 5.3 finds in $O\left(\frac{n \cdot diam_X^2 \cdot tr_{ratio}}{\eta - 1}\right)$ time a solution $x \in X_{cont}$ with*

$$tr(x,\omega) \leq \eta \cdot tr_{opt}.$$

Proof. Let $\epsilon := (\eta - 1)lb_{opt}$, where lb_{opt} is a lower bound on tr_{opt}. We will run the conditional gradient method up to accuracy $(\eta - 1)lb_{opt}$. By Theorem 5.7, Algorithm 5.2 returns a solution x with

$$tr(x, \omega) \leq tr_{opt} + \epsilon \leq \eta \cdot tr_{opt}$$

in $O(\frac{diam_X^2 \cdot lip(\omega)}{\epsilon})$ iterations. In each iteration, computing the gradient takes $O(n)$ time. It remains to find a good lower bound lb_{opt} to get a better bound on total running time. In particular, we are interested in a running time that is independent of the weights ω, as these can grow exponentially.

tr_{opt} is certainly larger than the largest weight multiplied by the smallest delay or cost, i.e.

$$lb_{opt} \geq \max\left\{\omega_{m+1}, \max_{e \in E} \omega_e\right\} \cdot \min_{x \in X_{cont}} \left\{cost(x), \min_{e \in E} delay_e(x)\right\}.$$

By Lemma 5.2,

$$\begin{aligned} lip(\omega) &\leq \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left\{\omega_{m+1} cost(x_i) + \Lambda \cdot \max_{e \in E} \omega_e delay_e(x)\right\} \\ &\leq \max\{\omega_{m+1}, \max_{e \in E} \omega_e\} \cdot \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left\{cpst(x_i) + \Lambda \cdot \max_{e \in E} delay_e(x)\right\} \end{aligned}$$

Putting together, we obtain

$$\begin{aligned} \frac{lip(\omega)}{lb_{opt}} &\leq \frac{\max_{x \in X_{cont}} \max_{1 \leq i \leq n} \{cost(x_i) + \Lambda \cdot \max_{e \in E} delay_e(x)\}}{\min_{x \in X_{cont}} \{cost(x), \min_{e \in E} delay_e(x)\}} \\ &:= O(tr_{ratio}). \end{aligned}$$

□

Algorithm 5.3 Conditional Gradient Method Multiplicative

1: **procedure** ConditionalGradientMult(η, ω)
2: Initialize $x^{(0)} \in X_{cont}$
3: **for** $k = 0$ to $O\left(\frac{diam_X^2 \cdot tr_{ratio}}{\eta - 1}\right)$ **do**
4: $x^{(k+1)} \leftarrow$ConditionalGradientIteration($x^{(k)}, \omega$)
5: $k \leftarrow k + 1$
6: **end for**
7: Return $x^{(k)} = (x_1^{(k)}, \ldots, x_n^{(k)})$ with $tr(x^{(k)}, \omega) \leq \eta \cdot tr_{opt}$
8: **end procedure**

Other algorithms

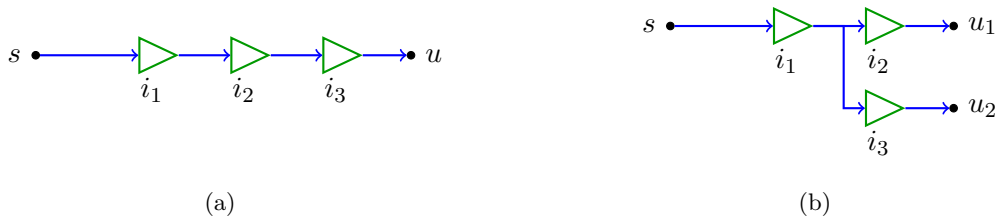
Coordinate descent methods iteratively perform approximate minimization along the coordinate descent directions. In each step, a single coordinate of the current iterate is modified by advancing in the direction of the corresponding coordinate of the gradient. The performance depends on the step size and the choice of the descent directions, and we refer to the recent paper of Wright [Wri15] for an overview. The local refinement algorithm can be regarded as a variant of coordinate descent: The algorithm does not advance in the direction of a gradient component, but changes one coordinate in each step and minimizes the corresponding gradient component. In our application, the conditional gradient method gives a faster approximation than the basic coordinate descent method, where the descent directions are traversed in a cyclic manner, or randomized coordinate descent. For functions with a Lipschitz continuous gradient, the projected gradient method exhibits similar theoretical convergence rates as the conditional gradient method. Coordinate descent methods with descent directions that are more adapted to the problem might perform better for problem (5.1). It is an open question if there exists an algorithm that runs in polynomial time and faster than interior point methods.

5.2 The Discrete Power-Delay Tradeoff Problem

5.2.1 Complexity

Problem (5.2) belongs to the class of convex discrete optimization problems which are *NP*-hard in general. However, no results on the complexity of problem (5.2) exist.
The monograph of Onn [Onn10] gives an overview of recent results in the field of nonlinear discrete optimization. In particular, the author considers convex objectives of the form $\max\{f(Wx) : \ x \in X\}$ with $W \in \mathbb{Z}^{d\times n}$, $X \subseteq \mathbb{Z}^n$ and $f : \mathbb{R}^d \to \mathbb{R}$ convex. If d is a variable part of the input, the maximization problem is *NP*-hard. $tr(x, \omega)$ can be written as $f(Wx)$, but here the dimension d needs to be equal to n. Polynomial time algorithms also exist if $d = n$ and f is separable, which does not hold for $tr(x, \omega)$ because of the third term in (5.3).
Under simplified delay models, problem (5.2) can be solved in polynomial time:

Simplified Delay Model Consider the following delay model: The delay of each edge $e = (v, w)$ is independent of input slew and load capacitance. Wire delays are constant, and several edges within a gate are contracted to a single edge. Let further be $X_{disc} \subset \mathbb{Z}^n$, and gate edge delay be proportional to gate size.
Under this delay model, the gate sizing problem is essentially the discrete time-cost tradeoff problem. Skutella [Sku97] showed that approximation algorithms for instances of the discrete time-cost tradeoff problem with at most two delay alternatives for each edge can be extended to approximation algorithms for arbitrary instances without losing the convergence guarantee.



(a) (b)

Figure 5.1: The discretization error of local refinement depends on the path lengths and the maximum fanout in the design.

We can thus consider the special case of problem (5.2) with at most two alternatives for each edge. Minimizing $tr(x, \omega)$ then amounts to solving a series of independent sizing problems, and can be done in time linear in the number of edges.

5.2.2 Algorithms

Algorithm 5.1 can be discretized by iteratively choosing the feasible discrete size for each gate that minimizes its local refine function, while keeping all other gates fixed. This can lead to a solution that is not "close" to the continuous solution, as the following example depicts:

Example 5.10 Consider the simple inverter chain in Figure 5.1(a). Let $X_{disc} \subset \mathbb{Z}^3$. Suppose Algorithm 5.1 returns continuous sizes $\xi_1 = 1.5$, $\xi_2 = 2.4$ and $\xi_3 = 3.3$. We proceed the gates again in topologial order and for each gate we choose the discrete size minimizing its local refine function. Let the discrete size for i_1 be $x_1 = 2$, i_2 gets assigned size $x_2 = 3$ and i_3 gets assigned size $x_3 = 4$. In the next traversal, x_1 is set to 3 to drive the increased capacitance, and thus the inaccuracies due to discretization are propagated through the inverter chain. The worst case gap between the discrete and continuous solution is in general unknown, but it cannot be arbitrarily large: If the capacitance of i_1 exceeds a certain threshold, the delay benefit of further increasing its size does not compensate the delay decrease of the entering edge, because the input s cannot drive the capacitance anymore. Using the same argument, the sizes of i_2 and i_3 are bounded.
Additionally, the discretization error depends on the maximum fanout in the design. Consider the simplified instance in Figure 5.1(b) and suppose that for all gates, the sizes chosen by the discretized algorithm are larger than the continuous sizes. Then, using the same argument as in the inverter chain example, the size of i_1 increases in the second iteration of the discretized algorithm to drive the increased capacitance. The gap between the continuous and the discrete solution thus also depends on the fanout of i_1.

The discretized version of Algorithm 5.1 and variants are commonly applied to minimize the discrete Lagrange primal function although no convergence guarantees

exist, for example in Huang et al. [HHS11], Flach et al. [Fla+14] and Livramento et al. [Liv+14].
Variants of the algorithm choose the discrete size for a gate for which not only the local refine function is minimized, but which also fulfills additional requirements. Among these are for example that the sum of load and slew violations does not increase, the worst slack seen at the gate or the sum of slacks in a certain neighborhood of the gate does not deteriorate etc. Algorithm 5.4 illustrates this concept.

Algorithm 5.4 Discrete Local Refinement (Variant)

```
1: procedure HeuristicDiscreteLocalRefine(x, ω)
2:     k ← 0
3:     x^(0) ← x
4:     while k = 0 or ∃i : x_i^(k) ≠ x_i^(k−1) do
5:         x^(k+1) ← x^(k)
6:         for each g_i ∈ G (in topologial order) do
7:             best_size ← cur_size(g_i) (current size of g_i)
8:             best_cost ← tr_{x^(k+1)}(best_size, ω)
9:             for each size s ∈ X^i_disc ≠ cur_size(g_i) do
10:                if s increases load or slew violations or degrades the
11:                   worst slack or the sum of slacks in a neighborhood then
12:                    continue
13:                end if
14:                if tr_{x^(k+1)}(s, ω) < best_cost then
15:                    best_cost ← tr_{x^(k+1)}(s, ω)
16:                    best_size ← s
17:                end if
18:            end for
19:            x_i^(k+1) ← best_size
20:        end for
21:        k ← k + 1
22:    end while
23: return x^(k) = (x_1^(k), ..., x_n^(k))
24: end procedure
```

To speed up the discrete local refinement algorithm, Szegedy [Sze05] proposed to only consider gates in iteration k whose neighbors have changed significantly in the previous iteration.
The approach of Ozdal and Burns [OBH12] relies on a graph-theoretic model: The graph contains $|X^i_{disc}|$ vertices for each gate g_i, and vertices corresponding to different gates are connected if their gates are connected in the gate graph. The Lagrange function is encoded in vertex and edge weights. The critical trees in the graph are optimized independently with dynamic programming, where a vertex is chosen for each gate minimizing total weight.

In Algorithm 5.5 the continuous solution serves as guidance. The optimal continuous size is computed iteratively for each gate, and rounded to the next smaller discrete feasible solution. We call such an operation a discrete local refinement operation:

Definition 5.11 (Discrete local refinement operation) *Let $x' \in X_{disc}$ be a solution of problem (5.2). A discrete local refinement operation for any variable x_i minimizes $tr(x, \omega)$ by only varying x_i, while keeping the value of all other variables x'_j $(j \neq i)$ in x' fixed. Let $\tilde{x}_i$ be the continuous solution minimizing $tr(x, \omega)$. Then $\tilde{x}_i$ is rounded down to the next smaller discrete size.*

Algorithm 5.5 DISCRETE LOCAL REFINEMENT ROUNDED

```
1:  procedure DiscreteLocalRefine(ω)
2:      k ← 0
3:      x_i^(k) ← l_i,  i = 1, ..., n
4:      while k = 0 or ∃i : x_i^(k) ≠ x_i^(k−1) do
5:          x^(k+1) ← x^(k)
6:          for each g_i ∈ G (in topological order) do
7:              x̃_i^(k+1) ← arg min tr_{x^(k+1)}(x_i^(k), ω)
8:              x_i^(k+1) ← round x̃_i^(k+1) to the next smaller discrete size
9:              if x_i^(k+1) < x_i^(k) then
10:                 x_i^(k+1) ← x_i^(k)
11:             end if
12:         end for
13:         k ← k + 1
14:     end while
15: return x^(k) = (x_1^(k), ..., x_n^(k))
16: end procedure
```

Theorem 5.12 *Algorithm 5.5 converges to a solution $x \leq x^*$. Each iteration takes $O(n)$ time.*

Proof. The convergence proof for Algorithm 5.1 by Chu and Wong [CW01] can be discretized to show convergence. We omit the rather technical details here because convergence also follows from elementary analysis and the fact that X_{disc} is bounded and sizes are monotonically increasing (line 10). By Theorem 5.4, finding the continuous size for each gate that minimizes the local refine function takes $O(n)$ time. The running time follows because computing the next smaller discrete solution takes constant time for each gate. □

The quality of the solution returned by Algorithm 5.5 is unknown. Consider the more straightforward variant of Algorithm 5.5 where we only change line 8 and round to the closest discrete size. The convergence proof for Algorithm 5.1 by

Chu and Wong [CW01] cannot be extended to show convergence of this variant in general.

5.2.3 FPTAS for Instances with Constant Level Size

In this section we present a fully polynomial time approximation scheme (FPTAS) for instances where the size of the antichains in the gate graph is bounded by a constant l_{max}. However, this assumption is unrealistic as l_{max} also depends on the instance size in practice. It is fulfilled, for example, by timing graphs consisting of a single path only.

The core of the algorithm is a modified binary search technique over the value of $tr(x, \omega)$ due to Ergun et al. [ESZ02], which uses an oracle algorithm to check if the current guess is approximately close to the value of the optimal solution or not. This search technique was already used by Liao and Hu [LH11] to develop an FPTAS for the delay-minimizing gate sizing problem, and our oracle algorithm is a variant of their level based dynamic programming oracle.

To simplify notation we assume that $\omega = \mathbb{1}$ throughout this section, but our results hold for all values of ω. Let $\zeta \in \mathbb{N}$ denote the maximum number of sizes available for a gate $g \in \mathcal{G}$, and let $l_{max} \in \mathbb{N}$ be the maximum number of gates in any antichain. We begin with the oracle algorithm that iteratively enumerates gate sizing solutions. Note that V_t optimization can easily be included: ζ then refers to the maximum number of books (combinations of gate sizes and V_t level) available for a gate.

ζ
l_{max}

Dynamic Programming Oracle Algorithm

The oracle algorithm decides if a guessed value F for the optimal value of $tr(x, \omega)$ can be approximately attained by a solution $x \in X_{disc}$.

Theorem 5.13 *For input (F, R, ϵ) with $F, \epsilon > 0$ and $R \in \{\frac{1}{2}F, F\}$, Algorithm 5.6 either returns a solution x with $tr(x, \omega) \leq (1 + \epsilon)F$, or reports that there exists no solution with $tr(x, \omega) \leq F$ in time $O(n\zeta^{3l_{max}} \cdot (\frac{m}{\epsilon})^{l_{max}})$.*

Proof. We partition the acyclic gate graph into so-called levels, and enumerate sizing solutions iteratively for each level following Liao and Hu [LH11]. The authors polynomially bounded the number of sizing solutions by rounding arrival times. We show that the same holds if we round edge delays appropriately.

The acyclic gate graph $\mathscr{G} = (\mathscr{V}, \mathscr{E})$ is partitioned into $l \in \mathbb{N}$ levels: The level-1 gates are exactly the gates driven by timing start points. For other gates, their level is defined as the number of vertices on a shortest path in the gate graph from any timing start point to the corresponding gate vertex.

Let L_i be the set of gates in level i, $L_{\leq i}$ the set of all gates up to level i, and $E_i := \{e = (v, w) \in E \mid \gamma(w) \in L_i\} \subseteq E$ the set of edges entering or traversing a gate in level L_i. Recall that $\gamma(w) \in \mathcal{G}$ indicates the gate to which pin $w \in V$ is assigned. We have $|E_i| \geq 2$ for all levels i, as each level contains at least one gate.

$L_i, L_{\leq i}$
E_i

The number of edges entering and traversing a gate are obviously bounded by a constant, so we can assume that $|E_i|$ is bounded by $O(l_{max})$.

A *level-i solution* is an assignment of sizes to all gates in $L_{\leq i+1}$. Different level-i solutions will be distinguished by the sizes of gates in levels i and $i+1$, the delays of all edges in E_i and the power of all gates in $L_{\leq i+1}$. Note that a level-i solution also defines the sizes of all gates in level $i+1$, as these influence the delays through gates in level i. The basic idea behind the solution enumeration is to iteratively compute level-i solutions for all $1 \leq i \leq l$.

level-i solution

Level-1 solutions are computed by enumerating all available sizes for the gates in levels 1 and 2. For $i > 1$, the algorithm integrates the size assignments for gates in L_{i+1} into the level-$(i-1)$ solutions by enumerating all possible sizes for gates in level $i+1$. For each combination of level-$(i-1)$ solution and size assignment for L_{i+1}, a new solution is generated.

Thereby solutions are pruned as follows: Given two level-i solutions that only differ in the power consumption of all gates in $L_{\leq i+1}$, we only keep the solution with the smallest power consumption. As both solutions have the same impact on the delays in E_{i+2}, the solution sets generated in the following levels based on these level-i solutions would only differ in their power consumption. As we are interested in minimizing $tr(x, \omega)$, the solutions with larger power consumption are certainly not optimal.

There are at most ζ sizes available for each gate and at most l_{max} gates per level, therefore at most $O\left(\zeta^{2l_{max}}\right)$ level-1 solutions and at most $O\left(\zeta^{3l_{max}}\right)$ level-2 solutions are computed etc. When reaching level l, the number of solutions would depend exponentially on l.

Therefore we bound the number of different edge delays in all E_i, and thus the number of level-i solutions. This can be achieved by rounding edge delays: When enumerating level-i solutions, we round delays in E_i to the nearest integer multiple of $\frac{R\epsilon}{m}$, i.e. $delay_e(x)$ is rounded to $\lfloor delay_e(x) \cdot \frac{m}{R\epsilon} \rfloor \cdot \frac{R\epsilon}{m}$, $x \in X_{disc}$. Then we perform power pruning using the rounded edge delays for comparison, and only keep the solution with the smallest power consumption $cost(x)$.

As we are interested in a solution whose value is approximately bounded by F, we additionally prune solutions with $cost(x) + D_r(x) > F$ in line 13 of Algorithm 5.6, as these cannot occur in any feasible solution. Here $D_r(x) := \sum_{j<i} D_r^j(x) + \sum_{e \in E_i} delay_e(x)$, and $D_r^j(x)$ is the sum of rounded edge delays in E_j. $D(x)$ denotes the sum of unrounded delays for all edges.

$D_r(x)$, $D_r^j(x)$

$D(x)$

It is easy to see that after rounding in line 15 there exist only $O(\frac{m}{\epsilon})$ different delay assignments for each edge: As solutions with $cost(x) + D(x) > F \Leftrightarrow D(x) \leq F - cost(x)$ are pruned, and edge delays are rounded to the nearest integer multiple of $\frac{R\epsilon}{m}$, the largest integer multiple θ that can occur in any solution is bounded:

$$\theta \cdot \frac{R\epsilon}{m} \leq F - cost(x) \leq F \quad \Leftrightarrow \quad \theta \leq \frac{Fm}{R\epsilon} \leq O(\frac{m}{\epsilon}),$$

because $R \in \{\frac{1}{2}F, F\}$. Consequently, there are at most $O(\frac{m}{\epsilon})$ different delay

Algorithm 5.6 Power-delay bounded level based dynamic programming

```
1: procedure FPTASOracle(F, R, ε)
2:     for 1 ≤ i ≤ l do
3:         if i = 1 then
4:             compute all level-1 solutions
5:         else
6:             for each level-(i − 1) solution do
7:                 compute all level-i solutions by enumerating all size
                   assignments for gates in L_i and L_{i+1}
8:             end for
9:         end if
10:        for each level-i solution x do
11:            D_r(x) ← Σ_{j<i} D_r^j(x) + Σ_{e∈E_i} delay_e(x)
12:            if cost(x) + D_r(x) > F then
13:                prune x
14:            else
15:                round edge delays of all e ∈ E_i to the nearest multiple of Rε/m
16:            end if
17:            D_r^i(x) ← sum of rounded delays in E_i
18:            if cost(x) > O(x') and D_r^i(x) = D_r^i(x') for a level-i sol. x' ≠ x then
19:                prune x
20:            else
21:                prune x'
22:            end if
23:        end for
24:        i ← i + 1
25:    end for
26:    return solution x with cost(x)+Σ_{i≤l} D_r^i(x) smallest, or no feasible solution
27: end procedure
```

assignments for each edge in E_i, and $O\left((\frac{m}{\epsilon})^{l_{max}}\right)$ different delay assignments for each combination of gate sizes in L_i and L_{i+1}. Thus, the number of solutions generated at each level is $O\left(\zeta^{2l_{max}}(\frac{m}{\epsilon})^{l_{max}}\zeta^{l_{max}}\right)$, of which $O\left(\zeta^{2l_{max}}(\frac{m}{\epsilon})^{l_{max}}\right)$ are kept.

For all levels we obtain a running time of $O\left(n\zeta^{3l_{max}} \cdot (\frac{m}{\epsilon})^{l_{max}}\right)$: The number of levels is bounded by n, and the running time of the pruning step in each level is $O\left(\zeta^{3l_{max}} \cdot (\frac{m}{\epsilon})^{l_{max}}\right)$ if the solutions are stored in a multidimensional array as in Liao and Hu [LH11]. Computing the cost of a level-i solution and finding the level-i solution with the same size assignment in L_i and L_{i+1} and the same delays in E_i can then be done in $O(l_{max})$ time, which is assumed to be bounded by a constant. The rounding error of each edge is bounded by $\frac{R\epsilon}{m}$. As there are m edges, the rounding error of all edges is bounded by $R\epsilon$. As desired, unscaling the delays of a

solution returned by the algorithm gives

$$cost(x) + D(x) + \epsilon R \leq F + \epsilon R \leq (1+\epsilon)F.$$

□

A Fully Polynomial Time Approximation Scheme

Algorithm 5.7 FPTAS FOR THE DISCRETE POWER-DELAY TRADEOFF PROBLEM

```
1: procedure FPTAS(ε)
2:     Compute lower and upper bounds FL < FU ∈ ℝ on the optimal value F*
3:     k ← 1, FU_1 ← FU, FL_1 ← FL
4:     while FU_k/FL_k > 2 do
5:         ε_k ← sqrt(FU_k/FL_k) − 1, F_k ← sqrt(FU_k FL_k/(1+ε_k))
6:         if FPTASOracle(F_k, F_k, ε_k) returns a feasible solution then
7:             FU_{k+1} ← (1+ε_k)F_k
8:         else
9:             FL_{k+1} ← F_k
10:        end if
11:        k ← k+1
12:    end while
13: return FPTASOracle(FU_k, FL_k, ε)
14: end procedure
```

Algorithm 5.7 presents the FPTAS for problem (5.2). It is based on a binary search technique for the optimal value F^* of (5.2) which was originally developed by Ergun et al. [ESZ02] for the restricted shortest path problem, and adapted by Liao and Hu [LH11] for delay-minimizing gate sizing.
In each step the oracle Algorithm 5.6 checks if the guessed value F is approximately close to F^*. By Theorem 5.13 the oracle either returns a solution $x \in X_{disc}$ with $tr(x,\omega) \leq (1+\epsilon)F$ and we can conclude that $F^* \leq (1+\epsilon)F$, or there exists no solution with $tr(x,\omega) \leq F$. The approximation ratio ϵ decreases in each step until we are sufficiently close to the optimal solution. Hence the last call to Algorithm 5.6 in line 13 with our desired approximation ratio ϵ dominates the running time of the binary search.
Let $FU \in \mathbb{R}_{\geq 0}$ be an upper bound and $FL \in \mathbb{R}_{\geq 0}$ be a lower bound on F^*, respectively. We obtain a lower bound by summing up the power values of the smallest sizes available for all gates, and the minimum delay value for each edge. The upper bound can be computed in a similar fashion.

Theorem 5.14 *Algorithm 5.7 computes a solution $x \in X_{disc}$ with $tr(x,\omega) \leq (1+\epsilon)F^*$ in time*

$$O(n\zeta^{3l_{max}} \cdot (m/\epsilon)^{l_{max}})$$

for $0 < \epsilon < 1$ and constant level size ζ. For $\epsilon \geq 1$ this is $O(n\zeta^{3l_{max}}m)$.

Proof. The proof is basically due to Liao and Hu [LH11] and Ergun et al. [ESZ02].

FU_k, FL_k
ϵ_k, F_k

Let FU_k and FL_k denote the upper and lower bound in iteration k, respectively. We set $\epsilon_k = \sqrt{FU_k/FL_k}-1$ and $F_k = \sqrt{FU_k FL_k/(1+\epsilon_k)}$ and call ORACLE(F_k, F_k, ϵ_k) in iteration k.

If the oracle returns true, FU_{k+1} will be set to $(1+\epsilon)F_k$, and $FU_{k+1}/FL_k = (1+\epsilon)F_k/FL_k$ holds. Otherwise FL_{k+1} will be updated to F_k and we have $FU_{k+1}/FL_k = FU_k/F_k$. The binary search stops as soon as $FU_k/FL_k < 2$ for some $k > 0$. Let $\bar{k}$ be this iteration. In each iteration, the ratio FU_k/FL_k decreases because

$$FU_{k+1}/FL_{k+1} \leq (FU_k/FL_k)^{3/4} \ \forall k, \tag{5.9}$$

which follows from the definitions of F_k and ϵ_k. The running time of the oracle algorithm in iteration k is $O(n\zeta^{3l_{max}} \cdot (\frac{m}{\epsilon_k})^{l_{max}})$. As $\epsilon_k = \sqrt{FU_k/FL_k} - 1$ and $FU_k > 2FL_k$ until the last step of the binary search, we have

$$\sqrt{FL_k/FU_k} \leq 1/\epsilon_k \leq (2+\sqrt{2})\sqrt{FL_k/FU_k}$$

by elementary transformation. The total running time of the binary search then is:

$$\begin{aligned} & O(n\zeta^{3l_{max}} m^{l_{max}} \cdot \textstyle\sum_{k\leq\bar{k}} (1/\epsilon_k)^{l_{max}}) \\ = \ & O(n\zeta^{3l_{max}} m^{l_{max}} \cdot \textstyle\sum_{k\leq\bar{k}} (\sqrt{FL_k/FU_k})^{l_{max}}). \end{aligned}$$

It remains to show that $\sum_k \sqrt{FL_k/FU_k} = O(1)$ to get the desired running time of the binary search:

$$\begin{aligned} \sum_{1\leq k\leq\bar{k}} \sqrt{FL_k/FU_k} &= \sum_{0\leq k<\bar{k}} (FL_k/FU_k)^{\frac{1}{2}\cdot(\frac{4}{3})^k} \\ &\leq \sum_{0\leq k<\bar{k}} 2^{-\frac{1}{2}\cdot(\frac{4}{3})^k} \\ &\leq 2^{-\frac{1}{2}} \sum_{0\leq k<\bar{k}} \delta^k \\ &\leq 2^{-\frac{1}{2}}/(1-\delta) \leq 6.5, \end{aligned}$$

with $\delta = 2^{-\frac{1}{6}} < 1$ and $FL_0 := FL$, $FU_0 := FU$. The first equality follows from equation (5.9). The first and second inequality hold because $FU_k > 2FL_k$ and $2^{-\frac{1}{2}\cdot(\frac{4}{3})^{k+1}} \leq \delta \cdot 2^{-\frac{1}{2}\cdot(\frac{4}{3})^k}$. Putting together, binary search takes $O(n\zeta^{3l_{max}} m^{l_{max}})$.

Now consider the stage when the binary search terminates and $FU_{\bar{k}}/FL_{\bar{k}} < 2$. We call the oracle once more with input $(FU_{\bar{k}}, FL_{\bar{k}}, \epsilon)$, where ϵ is the desired approximation ratio. Solutions x are pruned if $cost(x) + D_r(x) > FU_{\bar{k}}$, which ensures that at least one solution is not pruned and Algorithm 5.6 returns a feasible solution. In the end we choose the solution x with $cost(x) + \sum_{i\leq l} D_r^i(x)$ minimal.

Since we only round down edge delays, the rounded solution fulfills

$$tr(x, \omega) \leq F^* + \epsilon F L_{\bar{k}} \leq (1 + \epsilon) F^*.$$

With the running time from the binary search, the total running time of our algorithm is $O(n\zeta^{3l_{max}} m^{l_{max}}) + O(n\zeta^{3l_{max}} m^{l_{max}} \cdot (\frac{1}{\epsilon})^{l_{max}}) = O(n\zeta^{3l_{max}} m^{l_{max}} \cdot (\frac{1}{\epsilon})^{l_{max}})$. For $\epsilon > 1$ this is $O(n\zeta^{3l_{max}} m^{l_{max}})$. □

Remark 5.15 Depending on the closeness of the upper and lower bound, the delay values might all be rounded to zero. This can be checked easily in advance by rounding the maximum delay value that can occur in any solution. If this value is rounded to zero, it is not necessary to run the oracle algorithm and we can simply check if the power of the minimum size solution is smaller than the guessed value F_k, and use this output to continue with the binary search.

6 Lagrange Relaxation based Gate Sizing

Lagrangian relaxation is one of the mathematically best-founded approaches for gate sizing. A Lagrange multiplier is introduced for each timing constraint in the convex program, and the arrival time variables can be eliminated if the Lagrange multipliers fulfill the flow conservation rule at all vertices $v \in V_{inner}$ in the timing graph. The existence of a strongly feasible solution guarantees a zero duality gap. Despite being based on the convex program formulation of the continuous relaxation, the approach can be discretized by solving a discrete Lagrange primal problem, which is successfully applied in practice.
The projected gradient method solves the dual problem for the continuous relaxation, but the convergence rate is unknown. No convergence guarantee exists for the discretized algorithm.
The literature on the Lagrange relaxation approach is extensive, but theoretical aspects were often not considered. We give the first comprehensive discussion of this approach both from a theoretical and practical perspective.
First we formulate the Lagrange primal and dual problem following the groundwork paper of Chen, Chu and Wong [CCW99]. In practice, the dual problem is usually tackled by variants of the projected subgradient method. Wang et al. [WDZ07] proved that the dual objective function is differentiable, allowing the use of the projected gradient method. In Section 6.2 we analyze properties of the dual function, convergence guarantees of the projected gradient method that have not been considered before, and alternatives to this method.
In each iteration of the projected gradient method, the Lagrange primal problem needs to be solved in order to get a new descent direction (Section 6.3), which is the power-delay tradeoff problem discussed in Chapter 5. Because no approximation algorithms are known for the discrete power-delay tradeoff problem except for special cases, no convergence guarantees for the discretized Lagrangian relaxation approach exist.
The Lagrange multipliers are projected to the space of non-negative network flows in the timing graph in each iteration of the projected gradient method. This can be formulated as a quadratic minimum cost flow problem and be solved in strongly polynomial time. We compare exact, approximate and heuristic projection algorithms in Section 6.4.
The performance of the discretized Lagrangian relaxation approach and the difficulties in obtaining approximations are discussed in Section 6.5.
Finally, we show that convergence of the projected gradient method for the continuous relaxation can still be guaranteed if electrical constraints and constraints on placement density are incorporated into this framework (Section 6.6).

6.1 Lagrangian Relaxation Formulation

Recall the convex program (4.10) for the continuous relaxation:

$$\begin{aligned} &\min cost(x) \\ \text{subject to } &a_v + delay_e(x) \leq a_w \quad \forall\, e = (v,w) \in E. \end{aligned}$$

Recall that the arrival time variables are fixed for all $v \in V_{start} \cup V_{end}$. We relax the timing constraints as in Chen et al. [CCW99] by introducing a non-negative Lagrange multiplier λ_e for each constraint and $e \in E$. Let $\lambda := (\lambda_1, \ldots, \lambda_m)$ be the vector of these multipliers. The upper and lower bounds on the gate sizes (i.e. $x \in X_{cont}$) are kept as these are easier to handle. The Lagrange function augments the objective function with the relaxed constraints:

$\hat{L}(\lambda, a, x)$

$$\hat{L}(\lambda, a, x) \quad := \quad cost(x) + \sum_{e=(v,w)\in E} \lambda_e (a_v + delay_e(x) - a_w), \tag{6.1}$$

and the Lagrange primal problem is to minimize the Lagrange function:

$$\begin{aligned} \inf \quad & \hat{L}(\lambda, a, x) \\ \text{subject to} \quad & x \in X_{cont}. \end{aligned} \tag{6.2}$$

6.1.1 Separation of the Lagrange Function

Chen et al. [CCW99] made the crucial observation that the Lagrange function $\hat{L}(\lambda, a, x)$ can be split into two functions

$$\hat{L}(\lambda, a, x) = L(\lambda, x) + L_{at}(\lambda, a),$$

$L(\lambda, x)$

where

$$L(\lambda, x) := cost(x) + \sum_{e\in E} \lambda_e delay_e(x) \tag{6.3}$$

only depends on the size variables x, and

$$L_{at}(\lambda, a) \quad := \quad \sum_{e=(v,w)\in E} \lambda_e (a_v - a_w) = \sum_{v\in V} a_v \left(\sum_{e\in\delta^+(v)} \lambda_e - \sum_{e\in\delta^-(v)} \lambda_e \right)$$

only depends on the arrival time variables.

Obviously, the infimum of $L_{at}(\lambda, a)$ and thus the corresponding dual function is unbounded if $\lambda \in \mathbb{R}^m_{\geq 0}$ does not satisfy the *flow conservation rule* $\sum_{e\in\delta^+(v)} \lambda_e = \sum_{e\in\delta^-(v)} \lambda_e$ for each $v \in V \setminus \{V_{start} \cup V_{end}\}$:

Flow conservation rule

Suppose the flow conservation rule is violated at v. Then the arrival time a_v can be chosen infinitely large as arrival time variables are unrestricted, and $L_{at}(\lambda, a)$

becomes infinitely small. Otherwise, if λ forms a network flow, the minimum of $D_{at}(\lambda)$ is a finite constant.

We therefore constrain λ to the convex set $\mathcal{F}$ of nonnegative network flows in the timing graph that satisfy the flow conservation rule for all $v \in V \setminus \{V_{start} \cup V_{end}\}$, and consider the dual problem

Flowspace $\mathcal{F}$

Lagrange dual problem

$$\begin{aligned} \sup \quad & D(\lambda) := \inf_{x \in X_{cont}} L(\lambda, x) \\ \text{subject to} \quad & \lambda \in \mathcal{F}, \end{aligned} \tag{6.4}$$

From now on we refer to $L(\lambda, x)$ as Lagrange function and to the problem $\inf_{x \in X_{cont}} L(\lambda, x)$ as Lagrange primal problem. Note that the Lagrange function is of the form $tr(x, \omega)$ defined in Chapter 5 with $w_{m+1} = 1$.

Lagrange primal problem

When solving the dual problem, the Lagrange multipliers can be regarded as a measure for the timing criticality of the edges in the timing graph: A larger multiplier value indicates a higher timing criticality.

Deriving the Flow Constraints from the KKT Conditions

If a strongly feasible solution exists for the primal problem (4.10), i.e. a solution where all constraints are fulfilled by inequality, the duality gap is zero by Slater's condition (Theorem 3.15) and there exists an optimal solution that satisfies the Karush-Kuhn-Tucker-conditions (KKT-conditions, see Theorem 3.21). In that case, the flow conservation rule for λ can also be derived from the KKT-conditions (Chen et al. [CCW99]).

It was pointed out by Wang et al. [WDZ07] that the existence of an optimal solution satisfying the KKT conditions must be guaranteed first before the conditions can be applied, a fact that was neglected by Chen et al. [CCW99]. The reason is that the KKT conditions are a necessary, but not a sufficient condition for the existence of an optimal solution, and it is possible that no feasible solution satisfying the KKT conditions exists. Wang et al. [WDZ07] provide simple examples of such a situation.

Remark 6.1 As described in Langkau [Lan00], it is not necessary to simplify the Lagrange function and restrict the multipliers. She proposed to apply the subgradient method to the dual problem and in each step minimize the Lagrange function independently for gate sizes and arrival times, i.e. $L(\lambda, x)$ and $L_{at}(\lambda, a)$ are minimized separately. We will encounter this problem again in a similar context in Section 7.2.1.

6.1.2 Optimality Conditions

Slater's condition (Theorem 3.15) guarantees a zero duality gap and the existence of a saddle point (x, a, λ) if there exists a strongly feasible solution of the convex program.

Chen et al. [CCW99], whose work laid the foundation of Lagrange relaxation for gate sizing, applied Lagrangian relaxation to their geometric program formulation. They claimed that Slater's condition applies and guarantees a zero duality gap because the geometric program can be transformed into a convex program. It was pointed out by Wang et al. [WDZ07] that this claim does not hold per se. If variable transformations are required to obtain convexity, Lagrangian relaxation should be applied to the transformed problem. Only then Slater's condition can be applied.
Similarly, Chen et al. [CCW99] solve the non-transformed Lagrange primal problem with posynomial delays, although the convergence proof is based on the ability to transform it into a convex problem. Their convergence result was improved by Chu and Wong [CW01] also for the non-transformed problem. In Chapter 5 we adjusted these results to the convex Lagrange primal problem.
Further, Chen et al. [CCW99] did not ensure the existence of a strongly feasible solution, which is necessary for Slater's condition. Wang et al. [WDZ07] guarantee a zero duality gap without the existence of a strongly feasible solution if the objective function and the delay functions are strictly convex. This follows from a result of Rockafellar [Roc71], but does not guarantee a saddle point. Without a saddle point, the dual problem might not have a finite optimal solution, which in turn implies that the Lagrange primal problem is infeasible (cf. Section 3.2).

Infeasible Instances and Strongly Feasible Solutions

Gate sizing is often applied at a stage in the design flow where it is acceptable that not all timing constraints are fulfilled after gate sizing. In other words, the instances are infeasible. This can for example be the case in early stages of the design flow, where the constraints are set according to the designer's experience and small violations are acceptable. In later stages of the design flow, other timing optimization like repeater insertion, logic restructuring etc. are usually necessary to close timing. In those situations, a sizing solution that fulfills all timing constraints does not necessarily exist.
For timing infeasible instances, we aim to maximize the worst slack in practice, which is done implicitly in the Lagrangian relaxation approach because the multiplier values of timing-critical edges increase. But a zero duality gap only exists under the assumption that a strongly feasible solution exists. Alternatively, the existence of a strongly feasible solution can be guaranteed by reasserting required arrival times at timing endpoints:
Given initial sizes $x \in X_{cont}$ for all gates, we compute arrival times a_v at the timing points $v \in V$ with static timing analysis. Let $\theta \in \mathbb{R}$ be the slack target. We reassert required arrival times $\widetilde{rat}_v$ at all timing endpoints $v \in V_{end}$ as follows:

$$\widetilde{rat}_v := rat_v - \theta + \epsilon,\ \epsilon > 0.$$

If the slack target θ equals the worst design slack of the initial solution, the initial solution is already a strongly feasible solution for the modified timing constraints.

The same holds if we assign $\widetilde{rat}_v := a_v + \epsilon,\ \epsilon > 0$. The drawback is that initial timing criticalities are "forgotten", and optimization will focus on optimizing the objective function without degrading worst slack and SNS. Ideally, a slack target should be chosen that is realistic to be achieved by gate sizing.
From now on we assume that a strongly feasible solution exists for the convex program (4.10).

Lagrangian Relaxation for a Modified Geometric Progam

Chou and Chen [CWC05] apply Lagrangian relaxation to a modified version of the geometric program (4.8) which is obtained by dividing each delay and size constraint by the right hand side of the inequality such that the right hand side equals one. Then the variable transformation $x_i = \log(\xi_i)$ is applied and the logarithm of the constraints is taken. This leads to different KKT-conditions, which also allow a restriction of the Lagrange multiplier space. However, this formulation does not allow elimination of the arrival time variables. The authors resort to standard geometric program solvers for the Lagrange primal problem.

Lagrangian Relaxation for Delay-Minimizing Gate Sizing

Chen, Chu and Wong [CCW99] also consider Lagrangian relaxation of the convex program for the delay-minimizing gate sizing problem. It is easy to see that the problem allows a similar simplification of the Lagrange function with a zero duality gap. Results similar to the ones we discuss in the remainder of this chapter can also be established for this problem in a straightforward way.

6.2 The Lagrange Dual Problem

The dual problem can be solved with algorithms for convex optimization. We first study theoretical properties of the dual objective function that allow us to analyze the convergence rate, and consider convex optimization algorithms afterwards.

6.2.1 Properties of the Dual Objective Function

Lemma 6.2 *$D(\lambda)$ is concave and continuous.*

This generally holds for the dual objective function (cf. Section 3.2).
Wang et al. [WDZ07] established that under certain assumptions, the dual objective function is differentiable for a more general class of gate sizing problems where the Hessian matrix of the primal objective is positive definite for any $x \in X_{cont}$. Of course, this can only hold for the simplified dual objective which depends on x and λ. Otherwise, the dual function grows to infinity for all $\lambda \notin \mathcal{F}$, and is therefore neither continuous nor differentiable.
Let $d(x) \in \mathbb{R}^m$ be the vector of delays for $x \in X_{cont}$. We present a simplified proof $d(x) \in \mathbb{R}^m$

of the Theorem of Wang et al. [WDZ07] exploiting the convexity of the Lagrange function:

Theorem 6.3 ([WDZ07]) *The dual function $D(\lambda)$ is differentiable for all $\lambda \geq 0$ with gradient $d(\bar{x})$, where $\bar{x}$ is the unique vector in X_{cont} minimizing $L(\lambda, x)$.*

Proof. The proof is a simplified version of the proof of [WDZ07]. The proof is based on Theorem 3.22 which states that $D(\lambda)$ is differentiable if the set $M(\lambda) := \{x : x \text{ minimizes } L(\lambda, x),\ x \in X_{cont}\}$ is a singleton for all $\lambda \geq 0$, the functions $cost(x)$ and $delay_e(x)$ for $e \in E$ are continuous, and X_{cont} is a convex and compact set. In that case, the gradient is the vector of delays $d(\bar{x})$.
Obviously, the domain X_{cont} of feasible gate sizes is a convex and compact set. The functions $cost(x)$ and $delay_e(x)$ for $e \in E$ are continuous for all $x \in X_{cont}$ because they are the sum of exponential functions, which are continuous on $\mathbb{R}^n$. Hence there exists for all $\lambda \geq 0$ an $x_\lambda \in M(\lambda)$.
It remains to show that $M(\lambda)$ is a singleton for all $\lambda \geq 0$. We already pointed out that $L(\lambda, x)$ is of the form $tr(x, \omega)$ defined in Chapter 5, and therefore is strictly convex (cf. Lemma 5.1). Consequently, $M(\lambda)$ is a singleton.

□

When analyzing a problem, certain properties can be checked with the aim to find the best algorithm with convergence guarantees. The following lemmas analyze if these properties hold for the dual function.

Lemma 6.4 *The dual function is not strictly concave.*

Proof. Consider the following set of inequalities:

$$\begin{aligned}
D((1-t)\lambda + t\mu) &= \inf_{x \in X_{cont}} \left\{ cost(x) + \sum_{e \in E} \Big((1-t)\lambda_e + t\mu_e \Big) delay_e(x) \right\} \\
&= \inf_{x \in X_{cont}} \left\{ \Big(1-t\Big)\Big(cost(x) + \sum_{e \in E} \lambda_e delay_e(x) \Big) + \right. \\
&\qquad \left. t\Big(cost(x) + \sum_{e \in E} \mu_e delay_e(x) \Big) \right\} \\
&\geq (1-t) \cdot \inf_{x \in X_{cont}} \left\{ cost(x) + \sum_{e \in E} \lambda_e delay_e(x) \right\} + \\
&\qquad t \cdot \inf_{x \in X_{cont}} \left\{ cost(x) + \sum_{e \in E} \mu_e delay_e(x) \right\} \qquad (6.5)
\end{aligned}$$

$D(\lambda)$ is strictly concave if strict inequality holds for all $t \in (0,1)$ and for all $\lambda, \mu \in \mathbb{R}^m_{\geq 0}$. Now suppose the timing graph consists of a single path, and that $\lambda_e = \lambda_{e'}$ and $\mu_e = \mu_{e'}$ for all $e, e' \in E$ (i.e. λ, μ fulfill the flow conservation rule). If the entries

of λ, μ are large enough, $D(\lambda) = D(\mu)$ because in that case the unique minimizer x of the Lagrange function is the vector for which the delay is minimized. The same vector minimizes $D((1-t)\lambda + t\mu))$ and equality holds in (6.5). □

Lemma 6.5 *The dual function $D(\lambda)$ is not twice differentiable.*

Proof. The gradient at $\lambda \geq 0$ is the vector of delays, which is independent of λ. The partial derivatives of the delay vector with respect to λ equal zero. □

Lemma 6.6 *The dual function $D(\lambda)$ is not strongly concave.*

Proof. This follows immediately from Definition 3.3 and Lemma 6.5. □

Lemma 6.7 *The dual function $D(\lambda)$ and the gradient $\nabla D(\lambda)$ are Lipschitz continuous.*

Proof. The gradient of the dual function is the vector of delays, and each entry $delay_e(x)$ for $e \in E$ is bounded for $x \in X_{cont}$. Moreover, there exists a real constant $K > 0$ such that

$$||\nabla D(\lambda) - \nabla D(\mu)|| = \left|\left|(delay_e(x^\lambda))_{e\in E} - (delay_e(x^\mu))_{e\in E}\right|\right| \leq K\,||\lambda - \mu||\,,$$

where x^λ, x^μ are the unique minimizers of $D(\lambda, x)$ and $D(\mu, x)$, respectively. □

6.2.2 Solving the Dual Problem

The properties of the dual objective function established in the previous section allow us to analyze the convergence rate of the projected gradient method and alternative algorithms from convex optimization. Thereby the size of the instances occurring in gate sizing certainly restricts the options from a practical point of view. Usually variants of the projected subgradient method are used in practice. Wang et al. [WDZ07] used the conditional gradient method. Szegedy [Sze05] used the projected gradient method, but without knowing that the subgradient he used was in fact the gradient. Nonetheless, the best choice for the multiplier update regarding convergence is not evident, and a seemingly heuristic multiplicative multiplier update is growing in popularity.

Application of the Projected Gradient Method

Let $\bar{x}$ be the unique minimizer of the Lagrange function. By Theorem 6.3, $d(\bar{x})$ is the gradient of the dual objective function. As we aim to maximize the dual function, the projected gradient method (cf. Algorithm 3.1) iteratively advances in the gradient direction, and

$$\lambda^{(k+1)} := \pi_{X_{cont}}(\lambda^{(k)} + \rho^{(k)} d(\bar{x}^k).$$

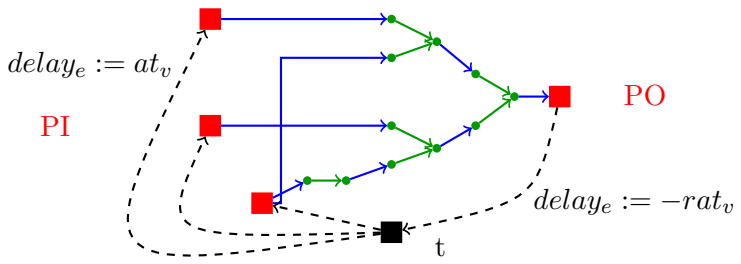


Figure 6.1: Extended timing graph $G' := (V', E')$

Paradoxically, it does not make sense from a practical point of view to update the multipliers with $d(\bar{x})$ because its entries are always positive and multipliers would never decrease. We briefly analyze this paradoxon:

Local edge slack $slack_e(x)$

Chen et al. [CCW99] suggested to propagate the arrival times in each iteration, and to add the negative of the *local edge slacks* $slack_e(x) := a_w - (a_v + delay_e(x))$ to the multipliers. This idea might be based on the fact that arrival times have to be primal feasible in the end, although they are ignored during optimization.

Szegedy [Sze05] observed that arrival time propagation is not necessary because the subsequent multiplier projection cancels the effect of adding the arrival times if the following extension G' of the timing graph G is used:

Extended timing graph $G' := (V', E')$

Add a dummy node t and connect it with all inputs via edges $E'_{in} := \{(t, v) : v \in V_{in}\}$, and with all outputs via edges $E'_{out} := \{(v, t) : v \in V_{out}\}$. We set $E' := E \cup E'_{in} \cup E'_{out}$, $V' := V \cup \{t\}$ and define $G' := (V', E')$.

The signal delay $delay_e$ over an edge $e = (t, v) \in E'_{in}$ is defined as the arrival time at_v at v. The signal delay $delay_e$ over an edge $e = (v, t) \in E'_{out}$ is defined as the negative required arrival time $(-rat_v)$ at v. Figure 6.1 shows the resulting graph. Note that these delays are independent of gate sizes and yield the additional delay constraints

$$\begin{aligned} delay_e &\leq a_v \ \forall e = (t, v) \in E'_{in}, \\ a_v + delay_e &\leq 0 \quad \forall e = (v, t) \in E'_{out}. \end{aligned}$$

Wang et al. [WDZ07], who proved differentiability of the dual objective, used a similar construction with two dummy nodes which models the same effect, but without giving any theoretical justification.

Note that differentiability of the dual function still holds with the additional timing constraints. Furthermore, the problem is more homogeneous as the flow constraints now need to hold for all vertices in the model graph. With the extended timing graph, it makes sense to update the multipliers with the delay vector, as not all edge delays are positive. The arrival time constraints are now encoded in G', and propagation of the timing information in the design is in some sense performed by the multiplier projection.

On the other side, the vector of local edge slacks which was deployed for example

in Chen et al. [CCW99] seemingly worked well in practice. The explanation is the following:

Theorem 6.8 *The vector of local edge slacks in G is a subgradient of the non-separated dual function $\hat{D}(\lambda) := \inf_{x \in X_{cont}, a \in \mathbb{R}^n} \hat{L}(\lambda, a, x)$ defined in equation (6.1).*

Proof. Recall that

$$\hat{L}(\lambda, a, x) \quad = \quad cost(x) + \sum_{e \in E} \lambda_e delay_e(x) + \sum_{v \in V} a_v \left(\sum_{e \in \delta^+(v)} \lambda_e - \sum_{e \in \delta^-(v)} \lambda_e \right)$$

where $const \in \mathbb{R}$ is a constant that accounts for the fixed arrival times and fixed required arrival times at primary input and output pins, respectively.
When λ fulfills the flow constraints, the last non-constant term equals zero and the arrival time variables can be chosen arbitrarily. By Theorem 6.3 there exists a unique $\bar{x}$ that minimizes the first two terms.
Consequently, there exists in general no unique minimizer of $\hat{L}(\lambda, a, x)$, and the Lagrange function has infinitely many subgradients with entries $(a_v + delay_e(\bar{x}) - a_w),\ e \in E$. Theorem 3.24 proves that these are indeed subgradients. □

We conclude that updating the Lagrange multipliers with the delay vector in G' is equivalent to updating the multipliers in G with the negative local edge slacks, where the arrival times are computed by static timing analysis.
The projected gradient method is summarized in Algorithm 6.1. $\pi_{\mathcal{F}}$ denotes the projection to the non-negative network flow space $\mathcal{F}$. Note that we proceed in positive gradient direction, as we aim to maximize the dual objective.

Algorithm 6.1 Projected Gradient Method for Gate Sizing

Input: Dual objective function $D(\lambda)$
Output: $\lambda \in \mathcal{F}, x \in X_{cont}$

1: $k \leftarrow 0$
2: Choose starting point $\lambda^{(0)} \in \mathcal{F}$
3: **repeat**
4: $\quad x^{(k)} \leftarrow \arg\min_{x \in X_{cont}} L(\lambda^{(k)}, x)$
5: $\quad g^{(k)} \leftarrow (delay_e(x^{(k)}))_{e \in E'}$
6: $\quad \lambda^{(k+1)} \leftarrow \pi_{\mathcal{F}} \left(\lambda^{(k)} + \rho^{(k)} \cdot g^{(k)} \right)$
7: $\quad k \leftarrow k + 1$
8: **until** stopping criterion is satisfied
9: Return $\lambda^{(k)}, x^{(k)}$

Convergence and Convergence Rate Given a zero duality gap, the projected gradient method solves the dual problem up to any desired accuracy. To the best of our knowledge, the convergence rate for gate sizing has not been considered before.

Among the drawbacks of the projected gradient method are its sensitivity to the choice of step size and start multipliers, and the slow convergence rate. If the step size degrades too fast, false decisions in early iterations due to inaccurate multipliers cannot be undone in later iterations. Line search repeatedly solves the Lagrange primal problem to determine the step size that maximizes the dual function. However, this is costly and hardly used for gate sizing in practice.
Because the non-negative network flow space $\mathcal{F}$ is convex and closed, Algorithm 6.1 converges to a stationary point for certain step size rules. The fact that the gradient is Lipschitz continuous ensures convergence even if the step size is constant. Additionally, the convergence rate depends linearly on $\left|\left|\lambda^{(0)} - \lambda^*\right|\right|$ and the Lipschitz constant of $\nabla D(\lambda)$ for certain step sizes, where λ^* is an optimal solution to the dual problem. The set $\mathcal{F}$ is unbounded and, to the best of our knowledge, no bounds on the Lagrange multipliers and hence on $\left|\left|\lambda^{(0)} - \lambda^*\right|\right|$ exist.
Linear convergence rates of the projected gradient method have also been established under the assumption that the objective function is strongly convex or twice differentiable. Both assumptions are in general not fulfilled by the dual objective function (Lemma 6.4 and Lemma 6.5). For general convex functions, the convergence rate is unknown (cf. Section 3.3).

Finding a Good Start Solution Tennakoon and Sechen [TS02] propose the following heuristic to find good start multipliers: Firstly, a steepest descent method aims to find gate sizes that satisfy a desired delay target. Secondly, a heuristic aims to find gate sizes inducing the same delays but a better objective function value. Finally, the Lagrange multipliers that imply this sizing solution are estimated and constitute the start multipliers.

Other Methods

Well-known methods like bundle methods or the space dilation method seem to be impractical for large-scale applications because of the additional computational and storage requirements.

Heuristics In recent years, heuristic multiplier updates that are more sensitive to local timing information and independent of a global step size have become more popular. Tennakoon and Sechen [TS02] were the first to use a multiplicative multiplier update of the form

$$\lambda_e^{(k+1)} = \lambda_e^{(k)} \cdot crit_e^{(k)},$$

where $crit_e^{(k)} := \frac{a_v}{a_w - delay_e(x)}$ encodes the violation of the timing constraint corresponding to edge $e = (v, w)$ in iteration k. Since then, several variants have been developed but without any convergence guarantees. We refer to Chapter 7 for a theoretical justification of this and other modifications to the projected subgradient method.

Methods of Feasible Directions Methods of feasible directions generate a sequence of iterates by choosing the descent direction and the step size in such a way that the next iterate is feasible (cf. Section 3.3). The minimization problem that needs to be solved in order to get a feasible descent direction amounts to minimizing the scalar product with the gradient in our application (Wang et al. [WDZ07]):

$$f \in \arg\min_{s \in \mathcal{F}} \langle \nabla D(\lambda), s \rangle. \tag{6.6}$$

This is essentially a minimum cost flow problem in the extended timing graph, where edge costs correspond to the negative delay values. The advantage over projected gradient methods is that no time-consuming projection is necessary, and the feasible search direction can be computed by solving a linear problem. Furthermore, the convergence rate is known.

On the downside, the minimum in (6.6) is generally unbounded because $\mathcal{F}$ is unbounded and the gradient contains negative entries, namely for the artificial edges. Consider for example an instance consisting of a single path only. The extended timing graph is a cycle. If the slack of this path is negative, the minimum of (6.6) is unbounded. Therefore Wang et al. [WDZ07] imposed a non-specified upper bound $u \in \mathbb{R}$ on the Lagrange multipliers, but did not claim convergence.

6.3 The Lagrange Primal Problem

The Lagrange primal problem consists of minimizing the Lagrange function (6.3):

$$L(\lambda, x) := cost(x) + \sum_{e=(u,v) \in E} \lambda_e \cdot delay_e(x).$$

Recall that the Lagrangian relaxation approach can be discretized by solving a discrete Lagrange primal problem. $L(\lambda, x)$ is of the form $tr(x, \omega)$ with $\omega = (\lambda_1, \ldots, \lambda_m, 1)$, and the Lagrange primal problem is an instance of the power-delay tradeoff problem (5.1). Therefore the continuous problem for $x \in X_{cont}$ can be solved up to accuracy ϵ in $O(n \log(1/\epsilon))$ time with the so-called local refinement algorithm (Theorem 5.4): Given an arbitrary starting solution, Algorithm 5.1 iteratively changes the gate sizes until no more improvement can be found. Thus, in the projected gradient method the solution from the previous iteration can be used as a starting solution in the next iteration.

Only heuristics are known for the discrete power-delay tradeoff problem except for special cases (cf. Section 5.2). Algorithm 5.4 is a discrete variant of the local refine algorithm and has been used in practice. In contrast to the continuous relaxation, V_t optimization can easily be incorporated by choosing a size and a V_t level that minimize the Lagrange function locally.

6.4 Multiplier Projection

It is a non-trivial task to compute an exact projection of the Lagrange multipliers to the non-negative flow space. The problem is well-known as the quadratic minimum-cost flow problem:

MULTIPLIER PROJECTION/QUADRATIC MINIMUM-COST FLOW PROBLEM

Instance: Edge weights $\lambda \in \mathbb{R}^m$
Task: Compute edge weights $\lambda' \in \mathbb{R}^m_{\geq 0}$ minimizing $\|\lambda - \lambda'\|_2^2$ subject to $\sum_{e \in \delta^-(v)} \lambda'_e = \sum_{e \in \delta^+(v)} \lambda'_e \quad \forall v \in V_{inner}$.

Although the problem can be solved in strongly polynomial time (Végh [Vég12]), the computation of an exact projection is time-consuming in practice.
In the gate sizing context, heuristics with linear running time have been successfully applied in practice, but theoretical convergence guarantees of Algorithm 6.1 are thereby lost. We discuss exact, approximate and heuristic projection algorithms and the resulting convergence of the projected gradient method.

6.4.1 Exact and Approximate Projections

By (Végh [Vég12], an exact multiplier projection can be computed in $O(m^4 \log(m))$ time. Minoux [Min84] first presented a polynomial extension of the well-known Edmonds-Karp algorithm for linear minimum-cost flows to convex quadratic flows. Ibaraki et al. [IFI91] proposed an algorithm for nonlinear minimum-cost network flow problems, whose application in the gate sizing context is described in detail in Langkau [Lan00].
Exact projections turned out to be computationally quite expensive in practice: Chen et al. [CCW99] reported that the practical running time of their algorithm was about $O(n^{1.7})$ for the whole projected subgradient method, and most of the running time was consumed by the projection step.
Rautenbach and Szegedy [RS04] and Lorenz et al. [LPT14] showed that approximate projections can be used without losing the convergence guarantee in the continuous case. Note that both works consider the projected subgradient method, but their results also hold for the projected gradient method.

6.4.2 Heuristics

The most widely used heuristic with linear running time was presented by Tennakoon and Sechen [TS02]. The timing graph is traversed in reverse topological order. At each vertex $v \in V$ which is not a timing endpoint, the multipliers of incoming and outgoing edges are added: $\mu_{in} := \sum_{e \in \delta^-(v)} \lambda_e$ and $\mu_{out} := \sum_{e \in \delta^+(v)} \lambda_e$.

Note that the multipliers of outgoing edges have already been projected. For each edge $e \in \delta^-(v)$ its contribution λ_e/μ_{in} to μ_{in} is determined, and μ_{out} is distributed to all $e \in \delta^-(v)$ based on this distribution: $\lambda_e := \mu_{out} \cdot \lambda_e/\mu_{in}$.
Szegedy [Sze05] proposed to project the multiplier flow locally optimal at each vertex in the timing graph. A positive outflow is maintained for each timing endpoint in the following sense: The outflow is initialized with a non-negative value depending on its timing criticality. In each iteration of the projected gradient method, the outflow is increased if the endpoint is still timing critical, otherwise the outflow is decreased. The outflow change corresponds to the scaled worst slack at the endpoint. The timing graph is then traversed in reverse topological order, and at each vertex $v \in V$ the multiplier sum of outgoing edges is distributed to the multipliers of entering edges. For timing endpoints, the sum of outgoing multipliers corresponds to the outflow. At $v \in V$ the distribution to the multipliers of incoming edges is chosen to minimize $\sum_{e \in \delta^-(v)} ||\lambda'_e - \lambda_e||^2$. Note that the outflow at the timing endpoints guarantees that the timing criticality of an endpoint is still visible in the next iteration of Algorithm 6.1 even if the endoint is not critical anymore. This prevents the projected gradient method to jump back and forth between solutions.

Comparison with Exact Projection

Heuristic projections are designed to interact well with the often heuristic multiplier update, and it is not easy to predict which combination of multiplier update and projection gives the best results in practice. Example 6.9 points out the differences between an exact projection and the heuristic projections we just described.

Example 6.9 Consider an inverter chain instance $I = (V(I), E(I))$. The extended timing graph for I is shown in Figure 6.2. Let the arrival time at primary input pin s be 0, and assume that $slack_u := rat_u - at_u < 0$ holds for the slack at primary output u. To fulfill the flow constraints, the multiplier values of all edges $e \in E(I)$ need to be equal. Recall from Section 6.2.2 that the gradient is the vector of edge delays, and $d_{(u,t)} = -rat_u$ and $d_{(t,s)} = at_s$ holds. Suppose we are in iteration k of Algorithm 6.1, and update the multipliers with step size 1, i.e. the delay of an edge is added to the corresponding multiplier. Then the sum of all multipliers increases by the slack of I and an exact projection increases the multipliers proportionally to $\frac{-slack_u}{|E(I)|}$. In other words, it calibrates the path length.
The projection from [Sze05] does not consider the extended timing graph during the projection. It increases the outflow at u by $\delta \cdot (-slack_u)$ for some $\delta > 0$, and propagates this increase through the timing graph. Thus it does not calibrate the path length. Depending on the value of δ, this leads to larger multipliers compared to the exact projection, which in turn imply an unnecessary upsizing of all inverters in the next iteration of the projected gradient method.
The projection of [TS02] propagates the multiplier value of the edge entering u backwards. As the multiplier update of each edge is based on the local criticality of the edge itself (a_u/rat_u for the edge entering u), the performance of the projection is hard to compare with the exact projection.

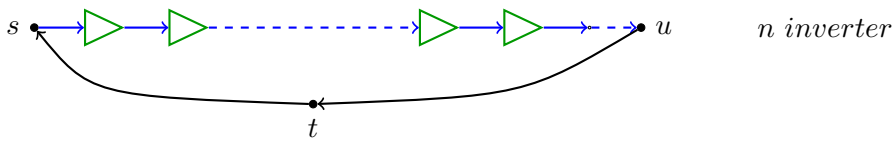


Figure 6.2: Extended timing graph of the inverter chain I.

Considering the Sum of Local Slacks

In timing optimization, usually the sum of negative endpoint slacks SNS and the worst slack of the design are considered to measure the quality of a current realization. However, the sum of local slacks SLS, which is defined as the sum of negative slacks at all gate output pins and timing start points, should not be overlooked. Consider a gate $g \in \mathcal{G}$ with worst negative slack $slack_p$ at its output pin p, and suppose that all paths from g to any timing endpoint merge with more timing critical paths. Then a slack improvement at g does not result in a better SNS. If only SNS and worst design slack are considered, it is not beneficial to improve the slack at g, especially at the cost of higher power and/or area consumption. But still, the slack at g needs to fulfill the slack target at the end of the physical design phase.
Local slack comes into play during the projection step in Lagrangian relaxation based gate sizing: If a path is timing critical, but merges with other paths with smaller slack, the multiplier flow is assigned mostly to the more timing critical paths, and following iterations focus on optimizing these. Once the more critical paths are improved, the multipliers on the formerly less critical path increase.
However, gate sizing is often applied to timing infeasible instances, and thus SLS should be taken into consideration when measuring the quality of a sizing solution, which was also proposed in Reimann et al. [RSR15]. An improvement of worst slack and SNS is not preferable if it comes at the cost of a large SLS degradation that needs to be recovered again.

6.5 Performance Analysis of Discretized Lagrangian Relaxation

In the discretized Lagrange relaxation approach a discrete primal problem needs to be solved (cf. Section 5.2). Compared to the continuous relaxation, signal delays can thereby be computed with accurate delay models or timing engines. Nonetheless, solving a discrete problem results in a gradient with error in each iteration of the projected gradient method: the entries in the error vector $r^{(k)}$ are the difference between the delays induced by the optimal continuous solution and the delays induced by the discrete solution of the Lagrange primal problem.
Approximation algorithms exist only for special cases of the discrete Lagrange primal problem, and consequently no convergence guarantees for the discretized Lagrangian relaxation approach are known.

In theory, the error in the gradient can be bounded because the set of feasible sizes is finite, but it is very large. To see this, suppose the optimal continuous solution realizes the minimum or maximum possible delay for $e \in E$, and the discrete solution realizes the maximum or minimum possible delay over e, respectively.
The literature on projected gradient or subgradient methods with error is sparse. Most works focus on the case where the error decreases in each iteration, which cannot be guaranteed in our application.
Nedic and Bertsekas [NB10] consider the projected subgradient method with a deterministic bounded error $r^{(k)}$: The subgradient directions are $\tilde{g}^{(k)} = g^{(k)} + r(k)$ where $g^{(k)}$ is a subgradient. Their results can be transferred to the projected gradient method. The authors establish convergence for several step size rules, but require a compact constraint set $\mathcal{F}$ or at least a constraint set that fulfills

$$\min_{\lambda^* \in \mathcal{F}^*} \left|\left|\lambda^{(k)} - \lambda^*\right|\right| \leq K \ \forall k \geq 0 \tag{6.7}$$

for some constant $K > 0$, the set of optimal solutions $\mathcal{F}^*$ and the iterates $\lambda^{(k)}$ in the projected gradient method. Because $\mathcal{F}$ is not compact, and no upper bound on the Lagrange multipliers λ exists, condition (6.7) is not fulfilled in our application.
Additionally, Nedic and Bertsekas [NB10] show convergence for the case that $D(\lambda)$ has a so-called *sharp set of minima* over $\mathcal{F}$, in other words

$$D(\lambda) - D^* \geq \mu \cdot \min_{\lambda' \in \mathcal{F}^*} \left|\left|\lambda - \lambda'\right|\right| \tag{6.8}$$

holds for $\lambda \in \mathcal{F}$, where D^* is the optimal value of $D(\lambda)$. $D(\lambda)$ has this property for continuous gate sizes, but not for discrete sizes: If $||\lambda - \lambda'||$ is small, the same discrete sizes are chosen when the Lagrange function is minimized, and $D(\lambda) = D^* = 0 < \mu \cdot \min_{\lambda' \in \mathcal{F}^*} ||\lambda - \lambda'||$.

Performance for the Discrete Time-Cost Tradeoff Problem

It is an interesting question how the discretized Lagrangian relaxation based approach performs for the discrete time-cost tradeoff problem, which can be regarded as an "easier" special case of the gate sizing problem with a simplified delay model. By Skutella [Sku97] we can also assume without loss of generality that only two delays are available for each edge, i.e. $|\tau_e| = 2$ for all $e \in E$.
We demonstrated in Section 4.5 that the discrete primal problem can be solved in polynomial time. Nonetheless, we still have a gradient with error in the projected gradient method: For $\lambda \in \mathcal{F}$, the gradient is the vector of delays induced by the unique minimizer of the Lagrange function, which does not necessarily have to coincide with the optimal discrete solution. But the error can be bounded.
However, conditions (6.7) and (6.8) are not fulfilled because $\mathcal{F}$ is unbounded, and the dual function has no sharp set of minima using the same argumentation as for the gate sizing problem. Hence no convergence guarantees can be deduced.

6.6 Additional Constraints

In this section we incorporate constraints on maximum load capacitance, maximum slew and placement density into the Lagrangian relaxation framework.
Boyd et al. [Boy+05] modeled maximum load capacitance and slew constraints as posynomials and included the constraints in the geometric program. Livramento et al. [Liv+14] incorporated these constraints in their discretized Lagrangian relaxation algorithm with a heuristic multiplier update. Until now, placement density was taken into account during gate sizing only implicitly or for small instances.

6.6.1 Placement Density Constraints

Region, density target

In the VLSI placement step, the chip area is typically divided into so-called *regions*, and a *density target* is dictated for the chip. In each region, the ratio of the placement area of all non-fixed gates and the placement area of the region should not exceed this target. This ratio is called *placement density* (cf. Section 2.6) of the region. In the placement context, non-fixed gates comprise all gates that can change their placement position. In the context of gate sizing, non-fixed gates are those whose size we are allowed to change. A density violation occurs in a region if the current placement density exceeds the target.

Density violation

We discussed in Section 2.6 that high placement density in a region in turn often implies a high pin density, which makes it more difficult for routing algorithms to connect the pins. The density target is an estimate of the density that can be realized in each region without impeding routing steps.
Gate sizing can induce local density violations because larger sizes consume more area. Similarly, gate sizing can create overlaps between gates, which have to be removed in a subsequent legalization step. A high placement density in a region complicates the work of the legalization algorithm. For example, Figure 6.3 shows the movement of gates during legalization after a global gate sizing that neglected density constraints. We conclude that it is beneficial to consider density in a gate sizing algorithm.

Previous works

Previous works on gate sizing usually take placement constraints implicitly into account by minimizing total gate area. Literature on simultaneous placement and gate sizing only considered small instances due to complexity and running time reasons. For example, Liu et al. [LSH08] consider delay-optimal placement and sizing of gate graphs with tree topology. This algorithm is heuristically extended to general acyclic graphs using Lagrange multipliers. Chen and Pedram [CHP00] simultaneously size and place the gates on the k most critical paths in a design based on a geometric program formulation, and round the continuous solutions.
Cong et al. [CLL11] perform sizing and placement with respect to placement density in a Lagrangian relaxation framework. A constraint requires that the total gate area

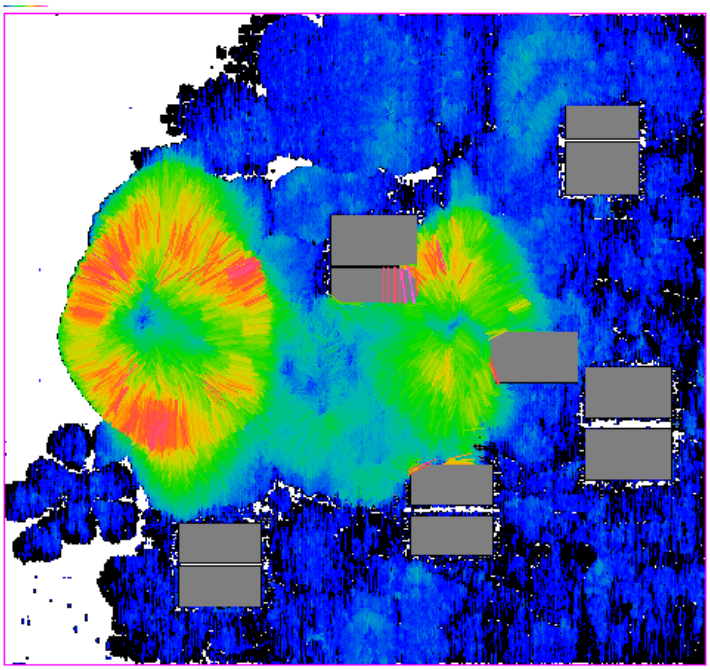


Figure 6.3: The movement of gates during legalization after a global gate sizing. Each colored line connects the placement location of a gate before legalization with the location after legalization. The different colors correspond to the length of the movement from old to new location ranging from blue (shortest) to red (longest).

on the chip does not exceed a prescribed value. A parameterised quadratic term is added to the Lagrange function to punish such a violation. The corresponding parameter μ is refined by iteratively solving the dual problem with the projected subgradient method for different values of μ. In the projected subgradient method, the Lagrange primal problem is solved in each iteration by iteratively minimizing the Lagrange function with respect to the placement variables and the size variables.

We show how placement density can be incorporated more directly:

Incorporating Placement Density Constraints

Suppose the chip area is partitioned into regions $R_1, \ldots, R_q$, $q \in \mathbb{N}$. For each region R_w we are given a placement density target $target_w$ that should be fulfilled. Usually, the density targets are equal for all regions, but may differ if certain regions are known in advance to cause difficulties for the routing tools. We obtain the following

R_w, $target_w$

constraints:

$$\frac{1}{|R_w|} \sum_{g_i \in R_w} area(g_i) \leq target_w, \ w = 1, \ldots, q \tag{6.9}$$

where $|R_w|$ denotes the free area of region R_w and $g_i \in R_w$ means that the area of gate g_i is in region R_w. Note that we assigned each gate to exactly one region. However, there does not necessarily exist a partition of the chip area into regions where the area of each gate lies in exactly one region. It is more likely that the area of some gates contributes to more than one region. In that case we assign the gate to the region that contains its center, or, if the center lies on the border of several regions, we assign the gate arbitrarily to one of these regions. We discuss how this assignment can be improved at the end of this section.
The area usage $area(g_i)$ of a gate g_i depends linearly on its size (cf. Section 4.2) and exponentially on the size variable x_i because of the variable transformation $x_i = \log(\xi_i)$ (cf. Section 4.4.3). We write $area(x_i)$ for the area of gate g_i to clarify the dependency on the size variables. For $x \in X_{cont}$ let $r(x) \in \mathbb{R}^q$ be the vector with entries

$r(x)$

$$r_w(x) := \sum_{g_i \in R_w} area(x_i) - target_w \cdot |R_w|.$$

Obviously, these are strictly convex functions of $x \in X_{cont}$. We introduce a Lagrange multiplier μ_w for each region R_w, $w = 1, \ldots, q$, and relax the constraints (6.9) in the Lagrange function (6.3):

$L(\lambda, \mu, x)$

$$L(\lambda, \mu, x) \ := \ cost(x) + \sum_{e \in E} \lambda_e \cdot delay_e(x) + \sum_{w=1}^{q} \mu_w r_w(x), \tag{6.10}$$

with $\mu = (\mu_1, \ldots, \mu_w)^t$. The Lagrange dual problem is defined as

$D(\lambda, \mu), \ \lambda \in \mathcal{F}, \ \mu \geq 0$

$$\begin{aligned} \sup \quad & D(\lambda, \mu) := \inf_{x \in X_{cont}} L(\lambda, \mu, x) \\ \text{subject to} \quad & \lambda \in \mathcal{F}, \ \mu \geq 0. \end{aligned} \tag{6.11}$$

Solving the Lagrange Primal and Dual Problem

It is easy to see that the Lagrange functions (6.10) and (6.3) are differ by a constant factor for each gate. Therefore Algorithm 5.1 solves the corresponding Lagrange primal problem for the continuous relaxation up to any desired accuracy.

Theorem 6.10 *$D(\lambda, \mu)$ is differentiable for all $\lambda, \mu \geq 0$ with gradient $(d(\bar{x}), r(\bar{x}))^t$, where $\bar{x} \in X_{cont}$ is the unique minimizer of $L(\lambda, \mu, x)$.*

Proof. The proof of Theorem 6.3 can be extended naturally: The newly added constraints $r(x)$ are obviously strictly convex and continuous, therefore there exists a unique minimizer of $L(\lambda, \mu, x)$ for all $\lambda, \mu \geq 0$. The statement follows from Theorem 3.22.

□

Consequently, the projected gradient method finds a solution for the dual problem (6.11) if a strongly feasible solution exists.
Algorithm 5.4 or Algorithm 5.5 can be used to find a solution for the discretized Lagrange primal problem, but without any performance guarantees.

Improving the Assignment of Gates to Regions

In our current assignment each gate belongs to the region that contains its center, and we made the simplifying assumption that this assignment does not change in Algorithm 6.1. However, the placement area of a gate can contribute to more than one region. A partial assignment of gate area to regions would assign several region multipliers to the same gate, each of which relates only to the part of the gate area that lies in the corresponding region. Sizing the gate then requires computing the effects on several regions. We consider this adjustment to reduce the error introduced by allowing assignments to one region only not worth the effort.
Similarly, the location of the center can switch to another region after each sizing step. This is easy to check, and assigning a gate to a different region multiplier does not harm convergence of the projected gradient method for the continuous relaxation. It can be regarded as density decrease in the old region, and as density increase in the new region.

6.6.2 Capacitance and Slew Constraints

Boyd et al. [Boy+05] showed that constraints on the maximum load capacitance of primary input pins and constraints on maximum slews can be incorporated into the geometric program for gate sizing. This was exploited by Livramento et al. [Liv+14] for discretized Lagrangian relaxation, who also incorporate constraints on maximum load capacitances of gates into their framework, but without providing the theoretical background.
Recall that the load capacitance of a pin $p \in V_{load}$ (i.e. a primary input pin or gate output pin) is defined as the capacitance of the net $N \in \mathcal{N}$ driven by p, and consists of the wire capacitance $wirecap(N)$ of N and the capacitances of its sink pins.
In the gate sizing context we assume wire capacitances to be constant. Input pin capacitances scale linearly with the gate size, and we make the realistic assumption that the load limit of a gate output pin scales linearly with the size of the gate. After variable transformation, these values thus scale exponentially with $x \in X_{cont}$ (cf. Section 4.4). Let $loadlim_p(x)$ denote the load capacitance limit associated with pin $p \in V_{load}$ for $x \in X_{cont}$. We assume input pin capacitances to be equal for all pins of a gate, but our model also works for varying capacitances. Let $cap(x_i)$ denote the input pin capacitance of gate $g_i \in \mathcal{G}$ when set to size x_i, and $g^N_{sink_1}, \ldots, g^N_{sink_{l(N)}}$ the sink gates of net N. We set $loadcap_p(x) := \sum_{i=1}^{l(N)} cap\left(x_{g^N_{sink_i}}\right) + wirecap(N)$ for $x \in X_{cont}$ and $p \in V_{load}$ driving net N. Note that $1/loadlim_p(x)$ and $loadcap_p(x)$

$loadlim_p(x_i)$
$cap(x_i)$
$g^N_{sink_i}$
$loadcap_p(x)$

are strictly convex functions in $x \in X_{cont}$.
To impose constraints on the maximum slew, we assume a posynomial delay model that integrates slew, and a posynomial slew function $slew_e$ for each edge $e = (v, p)$ (see Section 4.4) that returns the slew $slew_p$ at p given its load capacitance and the slew entering e at v. After variable transformation $x_i = \log(\xi_i)$, delays and slews are strictly convex in $x \in X_{cont}$. The values $slew_p$ are computed by forward propagation in the timing graph (cf. Section 2.5.3). We impose constraints on the maximum slew over each edge that enters a pin $p \in V_{slew}$ (gate input pins and primary output pins) with a given slew limit, because slew limits are usually independent of gates sizes.
Now we can formulate the $O(n + m)$ electrical constraints (2.23) and (2.24) as strictly convex functions:

$$\begin{aligned} loadcap_p(x)/loadlim_p(x) &\leq 1 \ \forall p \in V_{load} && (6.12)\\ slew_e(x, slew_v) &\leq slewlim_p \ \forall p \in V_{slew},\ e = (v, p) \in E. && (6.13) \end{aligned}$$

Note that the left hand side of (6.12) is a convex constraint because it is the product of convex functions.

$L(\lambda, \mu, \nu, x)$ The constraints (6.12) and (6.13) are relaxed in the Lagrange function:

$$\begin{aligned} L(\lambda, \mu, \nu, x) \ := \ & cost(x) + \sum_{e \in E} \lambda_e \cdot delay_e(x) + \\ & \sum_{p \in V_{load}} \mu_p (loadcap_p(x)/loadlim_p(x) - 1) \\ & + \sum_{p \in V_{slew},\ e=(v,p) \in E} \nu_p (slew_e(x, slew_v) - slewlim_p) \end{aligned} \tag{6.14}$$

with multipliers $\mu = (\mu_1, \ldots, \mu_{|V_{load}|})^t$ and $\nu = (\nu_1, \ldots, \nu_\kappa)^t$, where κ is the number
$D(\lambda, \mu, \nu)$ of slew limit constraints. The Lagrange dual problem is defined as

$$\begin{aligned} \sup \quad & D(\lambda, \mu, \nu) := \inf_{x \in X_{cont}} L(\lambda, \mu, \nu, x), \\ \text{subject to} \quad & \lambda \in \mathcal{F},\ \mu, \nu \geq 0. \end{aligned} \tag{6.15}$$

Theorem 6.11 *Let $\bar{x} \in X_{cont}$ be the unique minimizer of $L(\lambda, \mu, \nu, x)$. The dual objective $D(\lambda, \mu, \nu)$ is differentiable for all $\lambda, \mu, \nu \geq 0$ with gradient*

$$\begin{pmatrix} d(\bar{x}) \\ (loadcap_p(\bar{x})/loadlim_p(x) - 1)_{p \in V_{load}} \\ (slew_e(\bar{x}, slew_v) - slewlim_p)_{p \in V_{slew},\ e=(v,p) \in E} \end{pmatrix}.$$

Proof. The proof of Theorem 6.3 can be extended naturally: The newly added constraints are obviously strictly convex and continuous, therefore there exists a unique minimizer of $L(\lambda, \mu, \nu, x)$ for all $\lambda, \mu, \nu \geq 0$. □

The Lagrange function (6.14) is of the form (5.3), and Algorithm 5.1 solves the corresponding Lagrange primal problem up to any desired accuracy. The algorithm traverses the gates in topological order and can propagate the slews, which ensures that for $e = (v, p) \in E$, the slew at v is available when the gate is optimized to which pin p is assigned.
Similarly, Algorithm 5.4 or Algorithm 5.5 propagate the slew and return a solution to the discretized Lagrange primal problem, but without any performance guarantees.

7 The Multiplicative Weights Method for Gate Sizing

In recent years, the discretized Lagrangian relaxation approach based on the projected gradient method has often been modified to obtain better convergence. Among these seemingly heuristic modifications are a multiplicative Lagrange multiplier update and an additional weight for power consumption, and we will give the first theoretical justification of these ideas based on the multiplicative weights method.

The idea of the multiplicative weights method has been widely used in practice, and lies for example at the core of approximation algorithms for fractional packing and covering LPs (Plotkin et al. [PST95]) and the multicommodity flow problem (Garg and Könemann [GK07]).

Chapter 7 and Chapter 8 are dedicated to new mathematical models and algorithms for the gate sizing problem and its continuous relaxation. Thereby the multiplicative weights method and variants play a central role.

In this chapter we consider the feasibility version of the continuous gate sizing problem which arises from the convex program (4.10) by transforming the objective into a constraint. The multiplicative weights algorithm applied to this problem returns a solution that approximately fulfills all constraints. The discretized algorithm essentially is the modified Lagrangian relaxation approach for the discrete problem.

In the next chapter we go one step further and demonstrate that gate sizing can be modeled as a min-max resource sharing problem. This is a key problem in mathematical optimization and has been successfully applied to (timing-driven) global routing in VLSI design. The fastest algorithm for this problem is a variant of the multiplicative weights algorithm (Müller et al. [MRV11]).

The outline of this chapter is as follows: We introduce the multiplicative weights method in Section 7.1. In Section 7.2, we use the multiplicative weights algorithm to find gate sizes that (approximately) fulfill all constraints of the feasibility version of the convex program. The algorithm can be discretized by solving a discrete feasibility problem, but without any approximation guarantees (cf. Section 7.2.2). Additionally, the final solution needs to be rounded. A bound on the objective function value that is needed to transform the objective into a constraint can be determined with an approximate binary search (Section 7.2.3). In Section 7.2.4 we point out the differences between the continuous multiplicative weights approach and the Lagrangian relaxation approach. Comparison of the discretized algorithms leads to the conclusion that the ideas behind the most prominent heuristic modifications

of the projected gradient method can be justified by the discretized multiplicative weights algorithm.
The algorithms in this chapter can also be applied to the delay-minimizing gate sizing problem in a straightforward way.

7.1 The Multiplicative Weights Method

In this section we review the multiplicative weights algorithm and restate some known results following the survey paper of Arora, Hazan and Kale [AHK12].
The multiplicative weights method was introduced in the approximation algorithms for fractional packing and covering LPs by Plotkin, Shmoys and Tardos [PST95] and has since been used in a wide range of applications, among them the multicommodity flow problem (Garg and Könemann [GK07]) and global routing in VLSI design (Albrecht [Alb01]).
We introduce the multiplicative weights method in the context of solving concave feasibility problems, as we will apply it in the remainder of this chapter. The results are based on the work of Plotkin et al. [PST95] for fractional packing and covering LPs, and have been adapted to concave feasibility problems by Arora et al. [AHK12]. For missing proofs and a more general introduction with a broader overview of applications we refer to this work.

7.1.1 The Multiplicative Weights Algorithm for Feasibility Problems

The multiplicative weights algorithm is often applied to constrained optimization problems. The basic idea is to assign a positive weight to each constraint in the problem and iteratively compute potential solutions to the problem with an oracle algorithm that gets the weights as input. The weights are updated after each oracle call based on how well the corresponding constraints are satisfied in the solution returned by the oracle. This implies that a constraint weight is reduced if the constraint is satisfied, and increased otherwise. In the next iteration, the oracle should implicitly focus on the constraints with high weight that are poorly satisfied.

Feasibility problem

We consider *feasibility problems* of the following form:

$$\exists\ ?\ y \in \mathcal{Y} : f_i(y) \geq 0\ \forall i = 1, \ldots, m, \tag{7.1}$$

where $\mathcal{Y}$ is a convex domain in $\mathbb{R}^n$, and the functions $f_i : \mathcal{Y} \to \mathbb{R},\ i = 1, \ldots, m$, are concave. We use the notation $f := (f_1, \ldots, f_m)^t$.
It is easy to see that optimization problems can be reduced to feasibility problems by transforming the objective function into a constraint. To this end, a bound is imposed on the objective function value, which can be determined by binary search. In each search step, feasibility of the resulting problem is checked. An example is the approximation algorithm of Garg and Könemann [GK07] for the multicommodity flow problem.

We employ the multiplicative weights method to either find a solution $y \in \mathcal{Y}$ that fulfills (7.1) up to a given additive error $\eta > 0$, or to correctly decide that the problem is infeasible.

To this end, we need an oracle algorithm ORACLE that solves the following problem for constraint weights $\omega \in \mathbb{R}^m_{>0}$: ORACLE

$$\exists\ ?\ y \in \mathcal{Y} : \sum_{i=1}^{m} \omega_i f_i(y) \geq 0. \tag{7.2}$$

If there exists a solution $y^* \in \mathcal{Y}$ that fulfills the constraints in (7.1), then y^* is also a feasible solution of problem (7.2) for any $\omega \in \mathbb{R}^m_{>0}$. On the other hand, if there exists $\omega \in \mathbb{R}^m_{>0}$ such that no $y \in \mathcal{Y}$ satisfies (7.2), we can deduce that (7.1) is infeasible.

An oracle algorithm ORACLE for problem (7.2) can be implemented by maximizing $\sum_{i=1}^m \omega_i f_i(y)$ over $y \in \mathcal{Y}$.

Definition 7.1 *An ρ-bounded* ORACLE *for $\rho \geq 0$ is an algorithm that solves the feasibility problem (7.2) for input weights $\omega \in \mathbb{R}^m_{>0}$. Additionally, if* ORACLE *returns $y \in \mathcal{Y}$, we have $f_i(y) \in [-\rho, \rho]$.* ρ-bounded ORACLE

Remark 7.2 We consider ρ-bounded oracles similar to Plotkin et al. [PST95] as this is sufficient for our later application. Arora et al. [AHK12] use a more advanced definition to classify the solutions returned by the oracle in order to derive better running time guarantees.

The ρ-boundedness of ORACLE is needed to obtain convergence of the multiplicative weights algorithm for feasibility problems of the form (7.1). The algorithm is essentially due to Cesa-Bianchi, Mansour and Stoltz [CMS07] and is summarized in Algorithm 7.1. More precisely, $\frac{1}{\rho} f_i(y^{(t)}) \in [-1, 1]$ needs to hold in each iteration t of Algorithm 7.1, where $y^{(t)}$ is the solution returned by ORACLE in that iteration. Algorithm 7.1 assumes the existence of a ρ-bounded ORACLE, and we refer to ρ as the *width* of problem (7.1). Problem width ρ

If $|f_i(y)|$ is bounded for all $y \in \mathcal{Y}$, we can bound $\rho := \max_{y \in \mathcal{Y}} \max_{1 \leq i \leq m} |f_i(y)|$.

Algorithm 7.1 MULTIPLICATIVE WEIGHTS ALGORITHM

Input: Feasibility problem of the form (7.1), ρ-bounded ORACLE, $T \in \mathbb{N}$

1: Fix $0 \leq \nu \leq 0.5$
2: $\omega^{(1)} \leftarrow \mathbb{1}$
3: **for** $t = 1, \ldots, T$ **do**
4: Compute $y^{(t)}$ with ρ-bounded ORACLE for input weights $\omega^{(t)}$
5: **if** ORACLE decides that problem (7.2) is infeasible **then return**
6: **end if**
7: Weight update: $\omega_i^{(t+1)} := \omega_i^{(t)} \left(1 - \nu \cdot \frac{1}{\rho} f_i(y^{(t)})\right)$ for all $i = 1, \ldots, m$
8: **end for**
9: **return** $\bar{y} := \frac{1}{T} \sum_{t=1}^{T} y^{(t)}$

Theorem 7.3 is essentially due to Cesa-Bianchi, Mansour and Stoltz [CMS07]:

Theorem 7.3 ([CMS07]) *After T iterations of Algorithm 7.1 the following holds for any constraint $1 \leq i \leq m$:*

$$\sum_{t=1}^{T} \frac{1}{\rho} f\left(y^{(t)}\right) \cdot \omega^{(t)} \leq \sum_{t=1}^{T} \frac{1}{\rho} f_i\left(y^{(t)}\right) + \nu \sum_{t=1}^{T} \left| \frac{1}{\rho} f_i\left(y^{(t)}\right) \right| + \frac{\log(m)}{\nu}. \tag{7.3}$$

Note that the condition $0 \leq \nu \leq 0.5$ in line 1 of Algorithm 7.1 is necessary for convergence of the algorithm. The following theorem shows that Algorithm 7.1 approximately solves the feasibility problem (7.1).

Theorem 7.4 ([AHK12]) *Let $\eta > 0$ and suppose there exists an ρ-bounded* Oracle *for the feasibility problem (7.2) for $\rho > 0$. Assume that $\rho \geq \frac{\eta}{2}$. Then Algorithm 7.1 either solves problem (7.1) up to an additive error of η, or correctly decides that the problem is infeasible. Thereby $T = O\left(\frac{\rho^2 \log(m)}{\eta^2}\right)$ iterations are needed.*

Proof. The proof is due to Arora et al. [AHK12]. The assumption $\rho \geq \frac{\eta}{2}$ is needed for technical reasons. If it is not fulfilled, we can redefine $\rho = \frac{\eta}{2}$. In each iteration $1 \leq t \leq T$ of Algorithm 7.1 we run Oracle with input weights $\omega^{(t)}$. If Oracle returns that problem (7.2) is infeasible, the algorithm stops because then problem (7.1) is also infeasible.
Now assume that in each iteration t the Oracle returns a solution $y^{(t)} \in \mathbb{R}^m$ such that $\sum_{i=1}^{m} \omega_i^{(t)} f_i\left(y^{(t)}\right) \geq 0$. It is easy to see that

$$\frac{1}{\rho} f\left(y^{(t)}\right) \cdot \omega^{(t)} = \frac{1}{\rho} \sum_{i=1}^{m} \omega_i^{(t)} \cdot f_i\left(y^{(t)}\right) \geq 0.$$

By Theorem 7.3, the following holds after T iterations for all $i = 1, \ldots, m$:

$$\begin{aligned} 0 &\leq \sum_{t=1}^{T} \frac{1}{\rho} f_i\left(y^{(t)}\right) + \nu \sum_{t=1}^{T} \frac{1}{\rho} |f_i\left(y^{(t)}\right)| + \frac{\log(m)}{\nu} \\ &= (1+\nu) \sum_{t=1}^{T} \frac{1}{\rho} f_i\left(y^{(t)}\right) + 2\nu \frac{1}{\rho} \sum_{<0} \left| f_i\left(y^{(t)}\right) \right| + \frac{\log(m)}{\nu} \\ &\leq (1+\nu) \sum_{t=1}^{T} \frac{1}{\rho} f_i\left(y^{(t)}\right) + 2\nu T + \frac{\log(m)}{\nu}. \end{aligned} \tag{7.4}$$

Here the subscript "< 0" beneath the sum $\sum_{<0} |f_i\left(y^{(t)}\right)|$ denotes the subset of iterations t with $f_i\left(y^{(t)}\right) < 0$. The last inequality follows because $\left|f_i\left(y^{(t)}\right)\right| \leq \rho$. We set $\bar{y} := \frac{1}{T} \sum_{t=1}^{T} y^{(t)}$. Because $\mathcal{Y}$ is a convex set, $\bar{y} \in \mathcal{Y}$. By elementary transformations of (7.4) we obtain

$$0 \leq (1+\nu) f_i(\bar{y}) + 2\nu\rho + \frac{\rho \log(m)}{\nu T}$$

for $i = 1, \dots, m$ since all f_i are concave and thus $\frac{1}{T}\sum_{t=1}^{T} f_i\left(y^{(t)}\right) \leq f_i\left(\frac{1}{T}\sum_{t=1}^{T} y^{(t)}\right)$ by Jensen's inequality. We set $\nu = \frac{\eta}{4\rho}$ and $T = \left\lceil \frac{8\rho^2 \log(m)}{\eta^2} \right\rceil$. Then

$$0 \leq (1+\nu) f_i(\bar{y}) + \eta,$$

which implies

$$f_i(\bar{y}) \geq -\eta.$$

Note that $\rho \geq \frac{\eta}{2}$, hence $\nu \leq \frac{1}{2}$. We conclude that $\bar{y}$ returned by Algorithm 7.1 solves problem (7.1) up to an additive error of η. □

Approximate Oracles Algorithm 7.1 approximately solves the feasibility problem (7.1) also if ORACLE solves the weighted feasibility problem (7.2) up to an additive error.

η-approximate ORACLE

Definition 7.5 ([AHK12]) *An η-approximate* ORACLE *for $\eta > 0$ is an algorithm that solves the feasibility problem (7.2) for input weights $\omega \in \mathbb{R}^m_{>0}$ up to an additive error of η: It either returns $y \in \mathcal{Y}$ with $\sum_{i=1}^{m} \omega_i f_i(y) \geq -\eta$, or it decides correctly that (7.2) is infeasible.*

Theorem 7.6 ([AHK12]) *Suppose there exists an ρ-bounded $\frac{\eta}{3}$-approximate* ORACLE *for the feasibility problem (7.2) for $\eta > 0$. Assume that $\rho \geq \frac{\eta}{3}$. Then Algorithm 7.1 either solves problem (7.1) up to an additive error of η, or correctly decides that the problem is infeasible. Thereby $T = O\left(\frac{\rho^2 \log(m)}{\eta^2}\right)$ iterations are needed.*

Proof. The proof is due to Arora et al. [AHK12]. The assumption $\rho \geq \frac{\eta}{3}$ is needed for technical reasons. If is is not fulfilled, we can redefine $\rho = \frac{\eta}{3}$. We set $\nu := \frac{\eta}{6\rho}$. In each iteration $1 \leq t \leq T$ of Algorithm 7.1 we run the approximate ORACLE with input weights $\omega^{(t)}$.
If ORACLE returns in iteration t that there is no $y \in \mathcal{Y}$ with $\sum_{i=1}^{m} \omega_i^{(t)} f_i\left(y^{(t)}\right) \geq -\frac{\eta}{3}$ the algorithm stops because problem (7.1) is infeasible.
Now assume that in each iteration $1 \leq t \leq T$, ORACLE returns a solution $y^{(t)} \in \mathbb{R}^m$ such that $\sum_{i=1}^{m} \omega_i^{(t)} f_i\left(y^{(t)}\right) \geq -\frac{\eta}{3}$. It is easy to see that

$$\frac{1}{\rho} f\left(y^{(t)}\right) \cdot \omega^{(t)} \geq -\frac{\eta}{3\rho}.$$

We simplify as in the proof of Theorem 7.4 and get that after T iterations,

$$-\frac{\eta}{3} \leq (1+\nu) f_i(\bar{y}) + 2\nu\rho + \frac{\rho \log(m)}{\nu T}$$

holds for $i = 1, \dots, m$ and $\bar{y} = \frac{1}{T}\sum_{t=1}^{T} y^{(t)}$. With $T = \left\lceil \frac{18\rho^2 \log(m)}{\eta^2} \right\rceil$ we obtain $f_i(\bar{y}) \geq -\eta$. □

7.2 The Multiplicative Weights Algorithm for Gate Sizing

We consider the multiplicative weights algorithm for gate sizing formulated as feasibility problem based on the convex program (4.10), and assume for now that a sizing solution $x \in X_{cont}$ exists which fulfills the timing constraints. Infeasible timing constraints are treated separately at the end of Section 7.2.1.
In this chapter we use the basic version of the multiplicative weights algorithm as described in Algorithm 7.1 where the problem width contributes to the running time, although more sophisticated variants exist: Firstly to get a better comparison with the modified Lagrangian relaxation algorithm for the discrete problem (cf. Section 7.2.4), and secondly to enable evaluation of the resource sharing model in Section 8.10.

7.2.1 The Continuous Feasibility Problem

Formulation as Feasibility Problem

Recall the convex program (4.10) of the continuous relaxation:

$$\begin{aligned} &\min\ cost(x) \\ &\text{subject to } a_v + delay_e(x) \le a_w \quad \forall\ e = (v,w) \in E \\ &\qquad x \in X_{cont} \end{aligned}$$

The arrival time variables a_v are fixed for all $v \in V_{start} \cup V_{end}$ and unbounded for all $v \in V_{inner}$. Let $\widetilde{\mathcal{A}}$ denote the set of vectors $a \in \mathbb{R}^{|V|}$ with a_v fixed for $v \in V_{start} \cup V_{end}$ to the prescribed arrival time or required arrival time, respectively. We impose an upper bound $budget_{power} \in \mathbb{R}_{\ge 0}$ on the power consumption $cost(x)$, and interpret the gate sizing problem as feasibility problem of the form (7.1):

$\widetilde{\mathcal{A}}$

$budget_{power}$

Feasibility problem for gate sizing

$$\begin{aligned} &\exists?\ x \in X_{cont},\ a \in \widetilde{\mathcal{A}} && && (7.5)\\ \text{subject to}\quad &z_e(a,x) := a_w - (a_v + delay_e(x)) && \ge 0\ \forall e = (v,w) \in E \\ &z_{m+1}(a,x) := budget_{power} - cost(x) && \ge 0 \end{aligned}$$

The functions $z_e(a,x) : \mathbb{R}^{|V|} \times \mathbb{R}^n \to \mathbb{R}$ for all $e \in E$ and $z_{m+1} : \mathbb{R}^{|V|} \times \mathbb{R}^n \to \mathbb{R}$are obviously concave because the delay functions and $cost(x)$ are convex. We assume that the edges $e \in E$ have a fixed ordering $e_1, \ldots, e_m$, and also use the notation $z_i := z_{e_i}$ for all $i = 1, \ldots, m$.

$z_i(a,x)$

We intend to find an approximately feasible solution for problem (7.5) with Algorithm 7.1 and therefore introduce weights $\omega = (\omega_1, \ldots, \omega_{m+1}) \in \mathbb{R}^{m+1}_{>0}$ for the constraints in (7.5). We refer to ω_i for $1 \le i \le m$ as edge weights and to ω_{m+1} as power weight.

Constraint weights $w \in \mathbb{R}^{m+1}_{>0}$

The Oracle Algorithm

Recall that in each iteration of the multiplicative weights algorithm an oracle returns for given constraint weights ω gate sizes and arrival times that fulfill the weighted constraints of the form (7.2) up to a small additive error $\eta > 0$: η

$$\begin{aligned} &\exists?\ x \in X_{cont},\ a \in \widetilde{\mathcal{A}} \\ &\text{subject to } \sum_{i=1}^{m+1} \omega_i \cdot z_i(a,x) \geq -\eta. \end{aligned} \tag{7.6}$$

We follow Arora et al. [AHK12] and maximize

$$\max_{x \in X_{cont},\ a \in \widetilde{\mathcal{A}}} \left(\sum_{i=1}^{m+1} \omega_i \cdot z_i(a,x) \right) \tag{7.7}$$

$$w_e = \max_{x \in X_{cont}} \left(\omega_{m+1} \cdot (budget_{power} - cost(x)) - \sum_{e \in E} \omega_e \cdot delay_e(x) \right) \tag{7.8}$$

$$+ \max_{a \in \widetilde{\mathcal{A}}} \left(\sum_{v \in V} a_v \cdot \left[\sum_{e \in \delta^-(v)} \omega_e - \sum_{e \in \delta^+(v)} \omega_e \right] \right) \tag{7.9}$$

We exploit the fact that the arrival time and size variables are independent and maximize (7.8) and (7.9) separately.

Gate size oracle Obviously, (7.8) is equivalent to

$$\min_{x \in X_{cont}} \left(\omega_{m+1} \cdot cost(x) + \sum_{e \in E} \omega_e \cdot delay_e(x) \right),$$

which is the power-delay tradeoff problem (5.1). By Theorem 5.7, Algorithm 5.2 returns a solution $x \in X_{cont}$ with

$$\omega_{m+1} \cdot cost(x) + \sum_{e \in E} \omega_e \cdot delay_e(x) \geq opt - \eta \tag{7.10}$$

in $O\left(\frac{n \cdot diam_X^2 \cdot lip(\omega)}{\eta}\right)$ time, where $diam_X$ is the diameter of X_{cont}, $lip(\omega)$ is the Lipschitz constant of (7.8) as function of x (cf. Lemma 5.2), and opt the optimal value of (7.8).

Arrival Time Oracle If the edge weights do not form a network flow in the timing graph (i.e. $\sum_{e \in \delta^-(v)} \omega_e = \sum_{e \in \delta^+(v)} \omega_e$ for all $v \in V_{inner}\}$), the maximum of (7.9) is unbounded because the arrival time variables are unbounded.
In the Lagrangian relaxation framework, this obstacle was overcome by restricting the Lagrange multipliers to the non-negative network flow space. However, the

multiplier projection is not in line with the multiplicative weights framework from a theoretical point of view.

We employ the arrival time oracle of Langkau [Lan00] which was developed originally to minimize the terms in the Lagrange function $L(\lambda, a, x)$ depending on the arrival times, which are also of the form (7.9). Held et al. [Hel+15] described essentially the same oracle in a different context.

$l_v,\ u_v$
$\mathcal{A}$

Following Langkau [Lan00], we introduce arrival time bounds $l_v \leq a_v \leq u_v$ for all $v \in V$. For brevity, we denote with $\mathcal{A} := \{a \in \mathbb{R}^{|V|} |\ l_v \leq a_v \leq u_v\} \subset \mathbb{R}^{|V|}$ the set of feasible arrival time assignments, which obviously is a convex set. For timing start and endpoints, both the lower and upper bound are set to the prescribed arrival and required arrival time values, respectively, such that the values of the corresponding arrival times are fixed.

The terms in (7.9) can then be maximized by setting

$$a_v := \begin{cases} l_v & \text{if } \sum_{e \in \delta^-(v)} \omega_e < \sum_{e \in \delta^+(v)} \omega_e \\ u_v & \text{if } \sum_{e \in \delta^-(v)} \omega_e > \sum_{e \in \delta^+(v)} \omega_e. \end{cases} \tag{7.11}$$

Otherwise, for $v \in V_{inner}$ with $\sum_{e \in \delta^-(v)} \omega_e = \sum_{e \in \delta^+(v)} \omega_e$, the value of a_v can be chosen arbitrarily in theory. Again we follow Langkau [Lan00] and set a_v to the minimum of u_v and a'_v, where a'_v is the arrival time computed by static timing analysis with respect to the current gate sizes:

$$a_v := \min\{a'_v, u_v\}.$$

The reason is the following: the constraint weights ω_e reflect the criticality of the timing constraints during the algorithm in the sense that weights increase if the corresponding constraint is violated. (7.11) implies that a_v is set to u_v if the edges entering v are considered to be more critical, and vice versa a_v is set to l_v if the edges leaving v are considered to be more critical. Otherwise, it makes sense to set a_v to the smallest (feasible) value that satisfies all timing constraints of edges that are on a path from a timing start point to v.

Considering this, it is natural to call the gate size oracle before the arrival time oracle.

To determine the lower and upper bounds on the arrival times, we follow Held et al. [Hel+15]. For all $v \in V_{inner}$ we set l_v to the earliest possible signal arrival time at v which can be computed by assuming the minimal possible delay on all edges in the timing graph (see also Lemma 7.7). Similarly, we set u_v to the latest possible arrival time at v which ensures that the timing constraints are fulfilled.

For all $v \in V_{inner}$, the bounds can be computed by propagating the minimum delays through the timing graph in topological and reverse topological order, respectively:

$$\begin{aligned} l_v &:= \max_{e=(u,v) \in \delta^-(v)} l_u + min(delay_e), \text{ and} \\ u_v &:= \min_{e=(v,w) \in \delta^+(v)} u_w - min(delay_e). \end{aligned} \tag{7.12}$$

Here $min(delay_e)$ denotes the minimum delay over edge e that can be computed for each edge independently by sizing all gates with the aim to minimize the delay over e. If $l_v > u_v$ for any $v \in V$, the timing constraints are infeasible and cannot be fulfilled. $min(delay_e)$

Lemma 7.7 *The lower and upper bounds (7.12) on the arrival times can be computed by static timing analysis in $O(m)$ time for all $v \in V_{inner}$.*

Proof. Once the minimum delays $min(delay_e)$ are known for all $e \in E$, the arrival time bounds can be determined in $O(m)$ time by traversing the timing graph once in topological and once in reverse topological order, respectively.
It takes $O(m)$ time to compute the lower delay bounds $min(delay_e)$ for all $e \in E$: The lower delay bounds can be determined independently for each edge by choosing the gate sizes that minimize the load capacitance and resistance, in other words choosing the smallest or the largest available size for all gates whose size has an impact on the delay.

□

Lemma 7.8 *Algorithm 7.2 computes arrival times $a \in \mathcal{A}$ in $O(m)$ time.*

Proof. Static timing analysis traverses each edge once to compute the arrival time at each pin. Similarly, each edge is traversed a constant number of times when the weights of incoming and outgoing edges of each pin are added together. □

Algorithm 7.2 ARRIVAL TIME ORACLE

1: **procedure** ATORACLE(x, ω)
2: Propagate arrival times a'_v through G by STA based on delay for sizes x
3:

$$a_v := \begin{cases} l_v & \text{if } \sum_{e\in\delta^-(v)} \omega_{e_i} < \sum_{e\in\delta^+(v)} \omega_{e_i} \\ u_v & \text{if } \sum_{e\in\delta^-(v)} \omega_{e_i} > \sum_{e\in\delta^+(v)} \omega_{e_i} \\ \min\{a'_v, u_v\} & \text{otherwise} \end{cases} \tag{7.13}$$

return $a \in \mathcal{A}$
4: **end procedure**

Gate Size and Arrival Time Oracle Algorithm 7.3 combines the gate size and the arrival time oracle. Oracle algorithm

Algorithm 7.3 GATE SIZE AND ARRIVAL TIME ORACLE

1: **procedure** ORACLE(η, ω)
2: $x \leftarrow$ CONDITIONALGRADIENT(η, ω) (Algorithm 5.2)
3: $a \leftarrow$ ATORACLE(x, ω)
4: **return** $x \in X_{cont}$ and $a \in \mathcal{A}$
5: **end procedure**

Theorem 7.9 *Algorithm 7.3 is a η-approximate oracle for maximizing (7.7) with running time $O\left(\frac{n \cdot diam_X^2 \cdot lip(\omega)}{\eta} + m\right)$ for $\eta > 0$, i.e. we can deduce from*

$$\omega_{m+1} \cdot \Big(budget_{power} - cost(x)\Big) + \sum_{e \in E} \omega_e \cdot \Big(a_w - (a_v + delay_e(x))\Big) < -\eta$$

that problem (7.5) is infeasible.

Proof. The running time follows from Theorem 5.7 and Lemma 7.8. The arrival time oracle returns the optimum arrival times. The sizing oracle returns a solution with accuracy η, i.e. the sizes fulfill property (7.10). The statement follows from Definition 7.5 because the value *opt* in equation (7.10) is ≥ 0. □

The Problem Width ρ

Width ρ

Algorithm 7.3 is a η-approximate oracle, but it does not give any bounds on the width of problem (7.5) (see Definition 7.1). We bound the width ρ by

$$\begin{aligned} \rho &:= \max_{x \in X_{cont},\ a \in \mathcal{A}} \left\{ \max_{i=1,\dots,m+1} |z_i(a,x)| \right\} \qquad (7.14) \\ &= \max_{x \in X_{cont},\ a \in \mathcal{A}} \left\{ \max_{e=(v,w) \in E} |a_w - (a_v + delay_e(x))|, |budget_{power} - cost(x)| \right\}. \end{aligned}$$

Obviously, ρ is bounded because all variables are bounded. The power constraint can be bounded by the difference $B_u - B_l$ of a lower and upper bound on power consumption (cf. Section 7.2.3). The timing constraints can be bounded as follows:

$$\begin{aligned} & \max_{x \in X_{cont}, a \in \mathcal{A}} \left\{ \max_{e=(v,w) \in E} |a_w - (a_v + delay_e(x))| \right\} \\ = \ & \max \left\{ \max_{e=(v,w)} u_w - (l_v + min(delay_e)), \max_{e=(v,w)} |l_w - (u_v + max(delay_e))| \right\} \\ \leq \ & \max \left\{ \max_{w \in V} u_w, \max_{e=(v,w)} |u_v + max(delay_e)| \right\} \\ = \ & \max_{e=(v,w)} |u_v + max(delay_e)|, \end{aligned}$$

where $max(delay_e)$ denotes an upper bound on the delay over edge e. Multiple cycle paths can exist in the timing graph, therefore the required arrival times at timing endpoints, and consequently the bounds u_v for all $v \in V$, are bounded by $O(D)$, where $D \in \mathbb{R}_{>0}$ is the clock cycle time of the design.

Clock cycle time D

Under the assumption that we are given a fixed gate library and reasonable wire lengths, the maximum delay over an edge can be bounded by a (large) constant. We refer to Section 8.5 for a more detailed discussion.

Constraint Weight Update

In each iteration t of the multiplicative weights algorithm we compute feasible gate sizes $x^{(t)}$ and arrival times $a^{(t)}$, and use the weight update rule

$$\omega_i^{(t+1)} := \omega_i^{(t)} \left(1 - \nu \frac{z_i(a^{(t)}, x^{(t)})}{\rho}\right) i = 1, \ldots, m+1,$$

for $0 \leq \nu \leq 0.5$. The cost of constraint i at point (a, x) is $\frac{z_i(a,x)}{\rho}$. Division by ρ ensures that the costs lie in the range $[-1, 1]$, which is necessary in order to apply Theorem 7.6. Obviously, weights increase if the corresponding constraint is violated, and decrease otherwise.

Algorithm for the Feasibility Problem

Given an instance of the continuous relaxation of the gate sizing problem and an upper bound $budget_{power}$ on power consumption, Algorithm 7.4 returns a convex combination of sizes for each gate $g \in \mathcal{G}$:

Algorithm 7.4 Algorithm for the feasibility problem (7.5)

```
1: procedure FeasibilityProblem(η, budget_power)
2:     Fix 0 ≤ ν ≤ 0.5
3:     ω^(1) ← 𝟙
4:     for t = 1, ..., T do
5:         (a^(t), x^(t)) ← Oracle(η/3, ω^(t))
6:         if Σ_{i=1}^{m+1} ω_i^(t) · z_i(a^(t), x^(t)) < −η/3
7:         then
8:             return 0⃗ //(instance is infeasible)
9:         end if
10:        for i = 1, ..., m + 1 do
11:            ω_i^(t+1) ← ω_i^(t) (1 − ν z_i(a^(t), x^(t))/ρ)
12:        end for
13:    end for
14: return x̄ = (1/T) Σ_{t≤T} x^(t), ā = (1/T) Σ_{t≤T} a^(t)
15: end procedure
```

Theorem 7.10 *Let $budget_{power}$ be an upper bound on the objective function, and $\eta > 0$. Assume that $\rho \geq \frac{\eta}{3}$. Then Algorithm 7.4 returns in $T = O\left(\frac{\log(m)\cdot\rho^2}{\eta^2}\right)$ iterations vectors $\bar{x} \in X_{cont}$, $\bar{a} \in \mathcal{A}$ with $z_i(\bar{a}, \bar{x}) \geq -\eta$ for all $i = 1, \ldots, m+1$, i.e.*

$$\begin{aligned} cost(\bar{x}) &\leq budget_{power} + \eta \quad and \\ \bar{a}_v + delay_e(\bar{x}) &\leq \bar{a}_w + \eta \ \forall e \in E, \end{aligned}$$

or correctly decides that problem (7.5) is infeasible.

Proof. Lines 6-9 correctly ensure that Algorithm 7.4 aborts only if problem (7.5) is infeasible (cf. Theorem 7.9). The condition $\rho \geq \frac{\eta}{3}$ is technical, and if it is not fulfilled we redefine $\rho := \frac{\eta}{3}$.
We apply Theorem 7.6: We call Algorithm 7.3 in line 5 as ρ-bounded $\frac{\eta}{3}$-approximate oracle and set $\nu = \frac{\eta}{\rho \cdot 6} \leq \frac{1}{2}$. We simplify as in the proof of Theorem 7.4 and get that after T iterations, the solutions $\bar{x} = \frac{1}{T}\sum_{t=1}^{T} x^{(t)}$, $\bar{a} = \frac{1}{T}\sum_{t=1}^{T} a^{(t)}$ returned by Algorithm 7.4 satisfy

$$-\frac{\eta}{3} \leq (1+\nu) z_i(\bar{a}, \bar{x}) + 2\nu\rho + \frac{\rho \log(m)}{\nu T}.$$

With $T = \left\lceil \frac{18\rho^2 \log(m)}{\eta^2} \right\rceil$ we obtain

$$\begin{aligned} -\frac{\eta}{3} &\leq (1+\nu) z_i(\bar{a}, \bar{x}) + \frac{\eta}{3} + \frac{\rho \log m}{\frac{\eta}{\rho \cdot 6}\left\lceil \frac{18\rho^2 \log(m)}{\eta^2} \right\rceil} \\ &\leq (1+\nu) z_i(\bar{a}, \bar{x}) + \frac{2}{3}\eta. \end{aligned}$$

It follows that $-\eta \leq z_i(\bar{a}, \bar{x})$ for all $i = 1, \ldots, m+1$, as required. □

It is easy to see that increasing the number of iterations improves the approximation ratio.

Longest path length k

Corollary 7.11 Let k be the length of the longest path in G, and $\bar{x}$, $\bar{a}$ be the solution returned by Algorithm 7.4. Then $\bar{a}_p \leq rat_p + \eta \cdot k$ holds for all $p \in V_{end}$.

Proof. Consider a path in G with endpoint $p \in V_{end}$. For each edge $e = (v, w)$ on this path we have $\bar{a}_v + delay_e(\bar{x}) \leq \bar{a}_w + \eta$. Summing up the timing constraint violations of all edges on this path gives $\bar{a}_p \leq rat_p + \eta \cdot k$. □

Infeasible Timing Constraints

For instances with infeasible timing constraints Algorithm 7.4 never returns a feasible sizing solution. If no timing feasible solution exists, we aim to maximize the worst design slack WS instead. To this end, one can adjust the required arrival times by an estimate of the worst slack s, i.e. we assign required arrival times $rat_w - s$ to all $w \in V_{end}$, and perform binary search over s until a feasible solution has been found. An upper bound for s is 0. A lower bound for s is given by any sizing solution, for example the smallest size solution for all gates.

7.2.2 The Discrete Feasibility Problem

We now consider the discrete feasibility problem:

Discrete feasibility problem for gate sizing

$$\begin{aligned} &\exists?\ x \in X_{disc}, a \in \widetilde{\mathcal{A}} \\ \text{subject to}\ \ z_{m+1}(a,x) &:= budget_{power} - cost(x) \quad \geq 0 \\ z_e(a,x) &:= a_w - (a_v + delay_e(x)) \quad \geq 0 \quad \forall e = (v,w) \in E \end{aligned} \tag{7.15}$$

Algorithm 7.4 can be discretized by calling an oracle for the weighted feasibility problem with $w \in \mathbb{R}^{m+1}_{>0}$:

$$\begin{aligned} &\exists\ ?x \in X_{disc}, a \in \widetilde{\mathcal{A}} \\ \text{subject to}\ &\sum_{i=1}^{m+1} \omega_i \cdot z_i(a,x) \geq 0 \end{aligned} \tag{7.16}$$

in other words find $x \in X_{disc}, a \in \widetilde{\mathcal{A}}$ maximizing $\left(\sum_{i=1}^{m+1} \omega_i \cdot z_i(a,x)\right)$.

Similar to Section 7.2.1, the gate size and arrival time variables can be considered independently, and Algorithm 7.2 returns optimal arrival times $a \in \mathcal{A}$. Algorithm 5.5, for example, returns discrete sizes $x \in X_{disc}$, but with unknown approximation ratio. Putting together, Algorithm 7.5 describes an oracle algorithm that returns arrival times and discrete sizes with unknown oracle error:

Algorithm 7.5 Discrete gate size and arrival time oracle

```
1: procedure DiscreteOracle(ω)
2:     x ← DiscreteLocalRefine(ω) (Algorithm 5.5)
3:     a ← ATOracle(x, ω)
4: return x ∈ X_disc and a ∈ A
5: end procedure
```

Because the oracle error is unknown, Algorithm 7.6 for problem (7.15) cannot decide if an instance is infeasible, and the number of iterations necessary to achieve a certain accuracy cannot be determined. The algorithm is thus a heuristic that stops if no more improvement can be found (line 4), in other words if the weighted sum of cost and delays does not improve further. The convex hull of the vectors in X_{disc} is X_{cont}, and Algorithm 7.6 returns sizes $\bar{x} \in X_{cont}$.

Under the assumption that the error η of the sizing oracle, and thus of Algorithm 7.5, is known, Algorithm 7.6 can decide infeasibility of an instance and determine the required number of iterations to achieve a desired accuracy. In that case we can formulate the analogue to Theorem 7.10, with the same approximation guarantee for the sizes and arrival times.

Algorithm 7.6 Algorithm for the discrete feasibility problem (7.15)

procedure DiscreteFeasibilityProblem($budget_{power}$)
  Fix $0 \leq \nu \leq 0.5$
  $\omega^{(1)} \leftarrow \mathbb{1}$, $t \leftarrow 0$
  while improvement **do**
    $t \leftarrow t + 1$
    $\left(a^{(t)}, x^{(t)}\right) \leftarrow$ DiscreteOracle$\left(\omega^{(t)}\right)$
    for $i = 1, \ldots, m$ **do**
      $\omega_i^{(t+1)} \leftarrow \omega_i^{(t)} \left(1 - \nu \frac{z_i(a^{(t)}, x^{(t)})}{\rho}\right)$
    end for
  end while
return $\bar{x} = \frac{1}{t} \sum_{t' \leq t} x^{(t')}$, $\bar{a} = \frac{1}{t} \sum_{t' \leq t} a^{(t')}$
end procedure

Rounding the Convex Combination

Unfortunately, the vector $\bar{x}$ is the convex combination of discrete sizes, but not necessarily a discrete size itself, and needs to be rounded. Note that the arrival times need not be rounded.

Rounding gate sizes independently can give arbitrarily bad results, see Section 4.8 for a discussion. The sizing oracle determines gate sizes simultaneously based on the current constraint weights, and under the assumption that in later iterations sizes start to converge, the sizes $x^{(t)}$ computed in the last iteration can be a good choice. A possible explanation why this might work well in practice will be given in Section 7.2.4: the heuristic modifications of the Lagrangian relaxation approach in practice essentially lead to Algorithm 7.6 (although variants of the oracle are used), and in this algorithm, the solution from the last iteration yields good results.

Alternatively, we can simply choose the best solution from all iterations.

We will consider rounding a convex combination of discrete sizes and a special case from a more theoretical point of view in Section 8.7.

7.2.3 Binary Search over the Objective Function Value

Algorithm 7.4 approximately solves the feasibility problem 7.5 for a given bound $budget_{power}$ which can be determined with an approximate binary search.

The power consumption induced by the smallest sizes for all gates is a lower bound $B_l \in \mathbb{R}_{\geq 0}$ on power consumption. Similarly, power consumption induced by the largest sizes for all gates is an upper bound $B_u \in \mathbb{R}_{\geq 0}$. It is an interesting problem to find better bounds in polynomial time. For example, the smallest size solution without load capacitance violations can be computed in linear time by traversing the gate graph in reverse topological order.

B_l

B_u

7.2.4 Comparison with Lagrangian Relaxation

Lagrangian Relaxation vs. Multiplicative Weights for the Continuous Relaxation

We highlight the differences between the multiplicative weights algorithm (Algorithm 7.4) and the Lagrangian relaxation approach with the projected gradient method (Algorithm 6.1) for continuous gate sizing from a theoretical point of view.

Objective Function Both approaches are based on the convex program for gate sizing (4.10). While the projected gradient method (Algorithm 6.1) optimizes the objective power consumption directly, Algorithm 7.4 requires an upper bound on the power consumption to transform the objective function into a constraint. The upper bound can be specified by a designer or determined by binary search, which can be time-consuming. On the other hand, a weight is assigned to the new constraint, and updated in Algorithm 7.4 based on its criticality. Intuitively, this makes it easier to find a better tradeoff between power consumption and delays.

Sizing Oracle and Arrival Times Both algorithms call similar oracle algorithms to get feasible gate sizes in each iteration, but with different objectives. In Algorithm 6.1 the gradient is the vector of delays, and it is therefore important to find gate sizes that are close to the optimal sizes. In Algorithm 7.4, one is interested in gate sizes that minimize the power-delay tradeoff function.
Arrival time variables are disregarded in the Lagrangian relaxation approach, instead the Lagrange multipliers are restricted to the non-negative flow space. Algorithm 7.4 iteratively computes arrival times for all $v \in V$ with an arrival time oracle (Algorithm 7.2). The oracle runs in linear time, whereas an exact multiplier projection involves minimizing a quadratic function and is time-consuming in practice. It is easy to see that these are only two equivalent ways of tackling the same problem:
The function that is (approximately) minimized in each iteration of Algorithm 7.4 is equivalent to the Lagrange primal function before eliminating the arrival time variables (cf. Section 7.2.1). We conclude that Algorithm 7.4 approximately minimizes the non-simplified Lagrange function, and Algorithm 6.1 approximately minimizes the simplified Lagrange function in each iteration (cf. Section 6.1).
In the Lagrangian relaxation framework, the non-simplified Lagrange function and an arrival time oracle have been used for example by Langkau [Lan00], but this approach is not common.

Multiplier Update vs. Weight Update The Lagrange multiplier update rule in Algorithm 6.1 is additive: the new multiplier vector is generated by proceeding in gradient direction. The same step size applies to each multiplier, and the convergence of the algorithm is very sensitive to this choice. Algorithm 7.4 uses a

multiplicative update rule, where each weight is updated based on the criticality of the corresponding constraint, which is more sensitive to local information.

Starting Solution In the Lagrangian relaxation framework, the duality gap is zero, and convergence of Algorithm 6.1 can be guaranteed only if a strongly feasible solution exists. In the multiplicative weights framework, a feasible solution is sufficient (cf. Section 7.2.1).
In theory, both algorithms converge independently of the start multipliers/weights, but it is well-known that convergence of the projected gradient method is highly sensitive to the choice of the starting solution. In the gate sizing context, this was for example observed by Tennakoon and Sechen [TS02], who proposed a preprocessing step to find a good starting solution.

Convergence and Running Time An advantage of Algorithm 7.4 is that the number of iterations can be determined depending on the desired accuracy: More iterations yield more accurate solutions. The number of iterations of Algorithm 6.1 that are necessary to achieve a certain accuracy is unknown, and hence it is not clear whether the algorithm even runs in polynomial time (cf. Section 6.2).
Algorithm 7.4 is derived from the basic variant of the multiplicative weights algorithm, where the number of iterations depends on the problem width, and is thus not polynomial. Other variants, for example the scale-free multiplicative weights algorithm (see Hähnle [Häh15]), exhibit better running times. We used the basic variant to enable better comparison between two different models for gate sizing (see Section 8.10). Additionally, we employ the discretized version of Algorithm 7.4 to justify the discretized and heuristically modified version of Algorithm 6.1 in the next section.

Discretized Lagrangian Relaxation in Practice

Since Chen et al. [CCW99] published their groundwork on Lagrangian relaxation for gate sizing, this approach has been widely adopted both for the continuous and the discrete problem. We start with an overview over previous works and the most prominent modifications that are usually applied to the discretized Lagrangian relaxation approach in practice in order to improve convergence of the projected gradient method (Algorithm 6.1). We then observe that the most prominent modifications can be theoretically justified by the discretized multiplicative weights algorithm.

Multiplicative Multiplier Update Tennakoon and Sechen [TS02] were the first to propose a multiplicative Lagrange multiplier update rule of the form:

$$\lambda_e^{(t+1)} := \lambda_e^{(t)} \cdot crit_e^{(t)} \ \forall e \in E,$$

where $crit_e^{(t)}$ encodes the violation of the timing constraint corresponding to edge e in iteration k. Arrival times are computed by static timing analysis. Their motivation was to find a multiplier update that is more sensitive to local information and independent of the global step size.

Although this proposal was made for the continuous relaxation, it was widely adopted by subsequent works on the discretized approach, and variants can for example be found in Ozdal et al. [Ozd+12], Li et al. [Li+12a] and Livramento et al. [Liv+13; Liv+14]. A variant proposed by Flach et al. [Fla+14] incorporates the clock cycle time $D \in \mathbb{R}_{>0}$ in the multiplier update:

$$\lambda_e^{(t+1)} := \begin{cases} \lambda_e^{(t)} \cdot \left(1 + \frac{a_v + delay_e(x) - rat_w}{D}\right)^{1/l}, & \text{if } a_v + delay_e(x) \geq rat_w \\ \lambda_e^{(t)} \cdot \left(1 + \frac{rat_w - a_v - delay_e(x)}{D}\right)^{-l}, & \text{if } a_v + delay_e(x) < rat_w, \end{cases}$$

for $e = (v, w) \in E$, where rat_w is the required arrival time at w and a_v the arrival time at v as computed by static timing analysis based on the sizing solution x in the current iteration. The value of $l \in \mathbb{N}$ changes in the course of the algorithm.

Objective Weight Tennakoon and Sechen [TS08] introduced a weight ω_{power} for the objective function $cost(x)$ to find a better tradeoff between delay optimization and power optimization. The authors optimize a modified Lagrange function of the form $\omega_{power} \cdot cost(x) + \sum_{e \in E} \lambda_e delay_e(x)$. The factor ω_{power} is initialized with a value smaller than 1, and updated in each iteration based on the timing criticality of the design as follows:

$$\omega_{power}^{(t+1)} := \omega_{power}^{(t)} \cdot \min_{v \in V_{end}} \frac{rat_v}{at_v}.$$

Livramento et al. [Liv+13] experimentally observed that adding a power weight not only leads to less power consumption on the ISPD 2012 Gate Sizing Benchmarks (Ozdal et al. [Ozd+12]), but also to less timing constraint violations compared to running their algorithm without the power weight.

Multiplier Projection In practice, heuristics with linear running time but without any approximation guarantees estimate the multiplier projection, as computing an exact projection is time-consuming in practice, see for example Tennakoon and Sechen [TS02] and Szegedy [Sze05].

Comparison with Algorithm 7.6 We now compare the discretized projected gradient method including the modifications presented above with the discretized multiplicative weights algorithm (Algorithm 7.6) for the feasibility problem. Both algorithms are heuristics and do not necessarily terminate.

FeasibilityProblem($budget_{power}$)

Set $\omega^{(1)} := \mathbb{1} \in \mathbb{R}^{m+1}$, $0 \leq \nu \leq 0.5$, $t \leftarrow 0$
while improvement **do**
$t \leftarrow t + 1$
$x^{(t)} \leftarrow \text{DiscreteLocalRefine}\left(\omega^{(t)}\right)$
$a^{(t)} \leftarrow \text{ATOracle}\left(x^{(t)}, \omega^{(t)}\right)$
for $e \in E$ **do**
$\omega_e^{(t+1)} \leftarrow \omega_e^{(t)}\left(1 - \nu \frac{a_w^{(t)} - a_v^{(t)} - delay_e\left(x^{(t)}\right)}{\rho}\right)$
end for
$\omega_{m+1}^{(t+1)} \leftarrow \omega_{m+1}^{(t)}\left(1 - \nu \frac{budget_{power} - cost\left(x^{(t)}\right)}{\rho}\right)$
end while
Return $\bar{x} = \frac{1}{t}\sum_{t' \leq t} x^{(t')}$

Modified projected gradient

Initialize $\lambda^{(1)} \in \mathbb{R}^m$, $\omega_{power}^{(1)} \in \mathbb{R}$, $t \leftarrow 0$
while improvement **do**
$t \leftarrow t + 1$
$x^{(t)} \leftarrow \text{DiscreteLocalRefine}\left(\lambda^{(t)}, \omega_{power}^{(t)}\right)$
Compute $a^{(t)}, rat^{(t)} \in \mathbb{R}^{|V|}$ by STA
for $e \in E$ **do**
$\lambda_e^{(t+1)} \leftarrow \lambda_e^{(t)}\left(1 + \frac{\left|a_v^{(t)} + delay_e\left(x^{(t)}\right) - rat_w^{(t)}\right|}{D}\right)^{\theta_e^{(t)}}$
end for
$\omega_{power}^{(t+1)} := \omega_{power}^{(t)} \cdot \max_{v \in V_{end}} \frac{rat_v^{(t)}}{a_v^{(t)}}$.
Project $\lambda^{(t+1)}$ to flow space $\mathcal{F}$
end while
return $x^{(t)}$

Here $\theta_e^{(t)} = 1/l$ if $a_v^{(t)} + delay_e(x^{(t)}) \geq rat_w^{(t)}$ and $-l$ otherwise. STA is the abbreviation of static timing analysis

The ideas behind the modifications of the discretized projected gradient algorithm essentially yield the discretized multiplicative weights algorithm.
We already established in this section that the multiplicative weights algorithm (Algorithm 7.6) tackles the non-simplified Lagrange function, which contains the arrival time variables, in each iteration. Together with the arrival time oracle, this is equivalent to optimizing the simplified Lagrange function combined with the multiplier projection.
Although the multiplier update in the modified projected gradient algorithm is heuristic and based on arrival times and required arrival times computed by static timing analysis, the ideas behind the multiplicative update rule and the power weight ω_{power} for the objective function can now be theoretically justified by Algorithm 7.6.
Recall that even under the assumption that an approximation algorithm is known for the discrete power-delay tradeoff problem, it is unknown if the projected gradient method converges because no bound on the Lagrange multipliers exist (cf. Section 6.5). In the multiplicative weights algorithm, the oracle error would directly translate into the final approximation. The number of iterations required to get a desired accuracy can still be determined and depends on the oracle error (cf. Section 7.2.2).
A drawback of Algorithm 7.6 is that the vector $\bar{x}$ returned is not necessarily feasible discrete. However, the modified projected gradient has been successfully used in practice, which indicates that the solution vector from the last iteration can be a good choice. Additionally, a binary search for $budget_{power}$ is necessary.

8 The Resource Sharing Framework for Gate Sizing

The min-max resource sharing problem is a fundamental problem in mathematical optimization. It consists of distributing a limited set of resources among a limited set of customers who compete for the resources. An optimal solution distributes the resources in such a way that the maximum resource usage is minimized. This model has been successfully applied to (timing-driven) global routing in VLSI design, and the fastest approximation algorithm for this problem is a variant of the multiplicative weights algorithm. Having established the effectiveness of the multiplicative weights method for gate sizing, this chapter is dedicated to gate sizing as a min-max resource sharing problem.

We begin with a formal problem definition in Section 8.1 and demonstrate in Sections 8.2 and 8.3 how the continuous relaxation of the gate sizing problem fits into this framework using a single gate customer. With the algorithm of Müller, Radke and Vygen [MRV11] for the min-max resource sharing problem we get a fast approximation for the continuous relaxation of the gate sizing problem in Section 8.4. Under certain assumptions the running time is polynomial.

Subsequently, we compare its running time with existing algorithms for the continuous relaxation in Section 8.5. Section 8.6 models gate sizing as min-max resource sharing problem with path delay resources. This model was proposed by Hähnle [Häh15]. The approximation can easily be discretized, although with unknown performance guarantee because the discrete power-delay tradeoff problem occurs as a subproblem (cf. Section 5.2). Additionally, the convex combination returned by the resource sharing algorithm needs to be rounded. We consider rounding in Section 8.7 and give a bound on the approximation guarantee of randomized rounding for a special case. Section 8.8 describes how constraints on load capacitance, slew and placement density, which need to be taken into account in practice, fit into the resource sharing framework. Integration with timing-driven global routing and repeater insertion is the subject of Section 8.9. The chapter concludes with an evaluation of the resource sharing model in Section 8.10.

The interpretation of gate sizing as a resource sharing problem has been used before in game-theoretic approaches for the discrete problem, but without any performance guarantee (cf. Section 4.7). However, our interpretation as min-max resource sharing problem with one gate customer is novel.

The results in this chapter are joint work with Nicolai Hähnle and Stephan Held.

8.1 The Min-Max Resource Sharing Problem

Resources $\mathscr{R}$
Customers $\mathscr{C}$
Feas. solution $\mathscr{B}_c$

In the min-max resource sharing problem, we are given a finite set $\mathscr{R}$ of *resources*, a finite set $\mathscr{C}$ of *customers*, and for each customer $c \in \mathscr{C}$ an implicitly given convex set $\mathscr{B}_c$ of feasible solutions and a convex resource usage function $g_c : \mathscr{B}_c \to \mathbb{R}_{\geq 0}^{|\mathscr{R}|}$. We assume that these functions can be computed efficiently.

Reource weights $\omega \in \mathbb{R}^{|\mathscr{R}|}$
$opt_c(\omega)$

We are further given an oracle function $f_c : \mathbb{R}_{\geq 0}^{|\mathscr{R}|} \to \mathscr{B}_c$ for each customer, also called *block solver*, which computes for given *resource weights* $\omega \in \mathbb{R}_{>0}^{|\mathscr{R}|}$ a feasible solution $b_c \in \mathscr{B}_c$ with $\omega^t g_c(b_c) \leq \eta \cdot opt_c(\omega)$, where $opt_c(\omega) := \inf_{b \in \mathscr{B}_c} \omega^t g_c(b)$, and $\eta \geq 1$ is the approximation factor of the oracle.

The task is to find a feasible solution $b_c \in \mathscr{B}_c$ for each customer $c \in \mathscr{C}$ such that the maximum resource usage $\max_{r \in \mathscr{R}} \sum_{c \in \mathscr{C}} (g_c(b_c))_r$ is approximately minimized.

Note that the block solver for customer c requires input resource weights $\omega \in \mathbb{R}^{|\mathscr{R}|}$. The algorithms presented in this chapter are based on the multiplicative weights algorithm, and the resource weights grow proportionally to their usages.

Grigoriadis and Khachiyan [GK94] presented the first combinatorial fully polynomial approximation scheme for the general problem. The problem was also studied by Khandekar [Kha04], Jansen and Zhang [JZ08], and the fastest algorithm for the general problem was developed by Müller et al. [MRV11] for application in global routing in chip design. We refer to this paper for a broader overview of previous work.

κ^*

MIN-MAX RESOURCE SHARING PROBLEM

Instance:

- A finite set $\mathscr{R}$ of *resources*
- A finite set $\mathscr{C}$ of *customers*
- For each customer $c \in \mathscr{C}$
  - a convex set $\mathscr{B}_c$ of feasible solutions
  - a convex function $g_c : \mathscr{B}_c \to \mathbb{R}_{\geq 0}^{\mathscr{R}}$ that describes its resource usages

Task: Find feasible solutions $b_c \in \mathscr{B}_c$ for each $c \in \mathscr{C}$ minimizing the largest resource usage, i.e.

$$\kappa^* := \inf\{\max_{r \in \mathscr{R}} \sum_{c \in \mathscr{C}} (g_c(b_c))_r \mid b_c \in \mathscr{B}_c\}$$

is approximately attained.

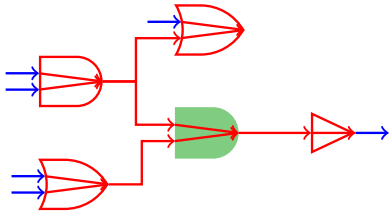


Figure 8.1: The delay of the red edges is affected when the green gate is sized.

8.2 Customers and Resources

We first introduce the resources and customers that are needed to model gate sizing as a resource sharing problem.

8.2.1 Resources

Edge delay resources Recall that in the convex program (4.10) we have a constraint for each edge in the timing graph. Similarly, we introduce a delay resource for each edge.

Power resource Additionally, we add a power resource to represent the objective function. Note that the objective power consumption of the convex program (4.10) can be transformed into a constraint by imposing an upper bound on the power consumption of all gates, which can be determined by binary search.

8.2.2 Customers

Gate customer The most natural choice is to introduce one customer for each gate, and to define that each customer consumes delay from the edge resources in its neighborhood graph. Figure 8.1 shows the neighborhood graph of the green gate in the center.
However, most edge delays depend on several gates in a non-separable way, because the load capacitance and resistance usually depend on the size of more than one gate. The same holds for the input slew of an edge in a delay model that incorporates slew effects. Thus changing the size of one gate, while keeping all other gates fixed, also impairs the delay usage of other gates. This is illustrated in Figure 8.1: Changing the size of the green gate in the center impacts the delays of all edges indicated in red. Thus the delay usage of the customers corresponding to the red gates is affected, even though these customers are not changed. This implies that we cannot specify convex resource usage functions for the gate customers.
We overcome this difficulty with one gate customer that represents all gates. The convex set of feasible solutions for this customer is the set X_{cont}, and the customer uses power from the power resource and delay from all edge resources.

Arrival time customer The task of gate sizing is to find a sizing solution such that the path delays in the timing graph do not exceed the delay limits imposed by (required) arrival times at timing start and endpoints, respectively. In other words, we aim to find a good distribution of the available delay to the edges, and thus to the gate customer, such that these constraints are fulfilled.

Arrival time customer $atcus_v$

To this end, an *arrival time customer* $atcus_v$ is introduced for each vertex $v \in V$ in the timing graph, which consumes delay of all edges entering and leaving v in the timing graph. Feasible solutions for arrival time customers are arrival times a_v with $atmin_v \leq a_v \leq atmax_v$ for all $v \in V$ similar to Section 7.2.1. For $v \in V_{start} \cup V_{end}$, arrival times are fixed and we set both $atmin_v$ and $atmax_v$ to the fixed value. For the other vertices, we will specify the values of the bounds in Section 8.3.3. We denote with $\mathcal{A} = \{a \in \mathbb{R}^{|V|} | atmin_v \leq a_v \leq atmax_v\}$ the set of feasible solutions for all arrival time customers to simplify notation.

$atmax_w$, $atmin_v$

$a \in \mathcal{A}$

The idea behind the arrival time customers originates from Held et al. [Hel+15] in the context of timing-driven global routing. Their purpose is to implicitly perform a delay budgeting of the available path delays, and to introduce a global view to the delay usage. Whereas Held et al. [Hel+15] work with a reduced graph, the set of our arrival time customers comprises all vertices in the timing graph.

Note that equivalently one arrival time customer modeling all vertices can be used. This increases the problem width, but is not apparent in the running time of the resource sharing algorithm.

Remark 8.1 (Path resources) With the above considerations, it seems more natural to use paths as resources rather than edges, and to omit the arrival time customers. However, the number of paths in the timing graph depends exponentially on its size. It was shown in Hähnle [Häh15] how timing-driven global routing can be modeled with path resources without the exponential dependency, and we adapt this model to gate sizing in Section 8.6.

8.3 Resource Usages and Oracle Functions

In this section we specify the resource usages and oracle functions of the gate and arrival time customers, and discuss how minimizing the maximum resource usage corresponds to the objectives in timing optimization.

$budget_{power}$

We denote with $budget_{power} \in \mathbb{R}_{\geq 0}$ the upper bound imposed on the power consumption of all gates (cf. Section 8.2.1). Let further $budget_e := atmax_w - atmin_v + h_e \in \mathbb{R}_{\geq 0}$ be a *delay budget* for each edge $e = (v, w) \in E$ in the timing graph, where $h_e > 0$ is a constant for all $e \in E$. The budgets will be chosen as in Held et al. [Hel+15] and Traub [Tra15] in such a way that minimizing the maximum resource usage is equivalent to maximizing the worst design slack. Before we consider the choice of the delay budgets in Section 8.3.3, we describe the resource usages of the customers.

$budget_e$, h_e

8.3.1 Gate Customer

The resource usage of the gate customer *gatecus* is *gatecus*

$$\frac{delay_e(x) + h_e}{budget_e} \quad \text{for each } e = (v, w) \in E, \text{ and} \tag{8.1}$$

$$\frac{cost(x)}{budget_{power}} \quad \text{for power,} \tag{8.2}$$

where $delay_e(x)$ and $cost(x)$ are the delay and cost functions as in the convex program (4.10). All resource usages are convex as required in the problem definition. The oracle function for the gate customer minimizes its weighted resource usage

$$\omega_{m+1}\frac{cost(x)}{budget_{power}} + \sum_{e\in E}\omega_e\frac{delay_e(x)}{budget_e} + constant, \tag{8.3}$$

which is the power-delay tradeoff problem (5.1). The weight ω_{m+1} is referred to as power weight. The constant comprises the terms h_e for all $e \in E$ and can be disregarded in the oracle algorithm. We set $\widetilde{\omega} := \left(\frac{\omega_{e_1}}{budget_{e_1}}, \ldots, \frac{\omega_{e_m}}{budget_{e_m}}, \frac{\omega_{m+1}}{budget_{power}}\right)$ and choose $\eta > 1$. Algorithm 5.3 returns for input $(\eta, \widetilde{\omega})$ a solution $x \in X_{cont}$ with $\eta > 1$

$$\omega_{m+1}\frac{cost(x)}{budget_{power}} + \sum_{e\in E}\omega_e\frac{delay_e(x)}{budget_e} \leq \eta \cdot opt_{gatecus}(\omega)$$

in $O\left(\frac{|\mathcal{G}|\cdot diam_X^2 \cdot tr_{ratio}}{\eta - 1}\right)$ time, with $tr_{ratio} = \frac{\max_{x\in X_{cont}} \max_i\{cost(x_i) + \Lambda \cdot \max_{e\in E} delay_e(x)\}}{\min_{x\in X_{cont}}\{cost(x),\ \min_{e\in E} delay_e(x)\}}$ and $diam_X = \max_{g\in\mathcal{G}} ||u_g - l_g||$ (Theorem 5.9).

8.3.2 Arrival Time Customers

The resource usage of customer $atcus_v$ is

$$\frac{a_v - atmin_v}{budget_e} \quad \text{for } e = (v, w) \in E, \text{ and} \tag{8.4}$$

$$\frac{atmax_v - a_v}{budget_e} \quad \text{for } e = (u, v) \in E, \tag{8.5}$$

with $budget_e = atmax_v - atmin_v + h_e$.
Putting this together with the gate customer, we get the delay resource usages

$$\frac{atmax_{w_i} - a_{w_i} + a_{v_i} - atmin_{v_i} + delay_{e_i}(x) + h_{e_i}}{budget_{e_i}} \tag{8.6}$$

for $i = 1, \ldots, m$, and $e_i = (v_i, w_i)$. As the arrival time customers are needed to implicitly perform a delay budgeting, the choice of the budgets naturally plays an important role:

8.3.3 Modeling Timing Objectives

Recall that the Lagrangian relaxation approach (Algorithm 6.1) and the multiplicative weights algorithm (Algorithm 7.4) aim to solve the convex program (4.10) where the constraints on path delays are split into constraints on edges. They require the existence of a (strongly) timing feasible solution. Algorithm 7.4 then returns a solution that approximately fulfills the timing constraints on the edges. By Corollary 7.11, this implies $a_p \leq rat_p + \eta \cdot k$ for all $p \in V_{end}$, such that a lower bound on the worst slack of the solution depends on the length k of the longest path in the timing graph G.

Additionally, we established in Section 6.1.2 that gate sizing is often applied at a stage in the design flow where a sizing solution that fulfills all timing constraints does not necessarily exist, and we aim to maximize the worst slack instead. This can be achieved by relaxing the required arrival times rat_w at timing endpoints $w \in V_{end}$ by a lower bound $s \in \mathbb{R}$ on worst slack and a binary search over s: In each step we relax the required arrival times to $rat_w - s$ for all $w \in V_{end}$, and test if a feasible solution for the gate sizing problem exists. However, initial timing criticalities can thereby be "forgotten" in the sense that timing critical paths can appear to be uncritical, and are therefore not optimized.

We conclude that Algorithm 7.4 is disadvantageous in the sense that a binary search is necessary to optimize the worst slack, and that the approximation guarantee on the edge constraints introduces a dependency on k in the worst slack bound.

The situation is different in the resource sharing framework. It turns out that for an appropriate choice of delay budgets, minimizing the maximum resource usage is equivalent to maximizing the worst slack:

Theorem 8.2 ([Hel+15; Tra15]) *Given a lower bound $slackmin \in \mathbb{R}$ on the worst design slack, then intervals $[atmin_v, atmax_v]$, constants h_e and arrival times $a \in \mathcal{A}$ can be computed in linear time such that for each sizing solution $x \in X_{cont}$, where the slack at each vertex v is larger than $\min\{u_v - l_v, slackmin\}$, the following holds:*

$$\min\left\{\max_{e=(v,w)\in E} \frac{atmax_w - a_w + a_v - atmin_v + delay_e(x) + h_e}{budget_e}\right\} = 1 - \frac{WS_x}{h_{out}}.$$

Here WS_x is the worst slack of solution x, and

$$h_{out} = \max_{P \text{ path in } G} \sum_{e=(v,w)\in P} atmax_w - atmin_v.$$

The idea is to choose the delay budgets $budget_e$ in such a way that each inclusion-wise maximal path in the timing graph has the same delay budget.

Originally, Theorem 8.2 was formulated in the context of timing-driven global routing, but can be transferred to gate sizing in a straightforward way.

8.4 Minimizing the Maximum Resource Usage

In the previous sections we have outlined how gate sizing can be formulated as a min-max resource sharing problem. The algorithm of Müller et al. [MRV11] returns an $\eta(1+\epsilon)$ approximation of the optimal solution for any given $\epsilon > 0$ and oracle error $\eta > 1$. Combined with a binary search over the power budget this is a fast approximation for the continuous relaxation of the gate sizing problem (cf. Section 8.5). In particular, we apply the following theorem:

Theorem 8.3 ([MRV11]) *The min-max resource sharing problem can be solved with approximation ratio $\eta(1+\epsilon)$ in $O(\theta(|\mathscr{C}|+|\mathscr{R}|)\log|\mathscr{R}|(\log\log|\mathscr{R}|+\epsilon^{-2}))$ time for any $\epsilon > 0$. Here $\eta \geq 1$ is a constant bounding the approximation ratio of the oracle functions, and θ bounds the running time of an oracle call. The running time reduces to $O(\theta(|\mathscr{C}|+|\mathscr{R}|)\epsilon^{-2}\log|\mathscr{R}|)$ if $\frac{1}{2} \leq \kappa^* \leq 2$ holds for the optimum κ^*.*

The underlying algorithm is a variant of the multiplicative weights algorithm with approximation guarantee $\eta(1+\epsilon)$ if the optimum κ^* lies within the interval $[\frac{1}{2}, 2]$. Otherwise a preceding binary search finds an appropriate scaling factor for the resource usages. During the binary seach, the algorithm is called $\log\log|\mathscr{R}|$ times with an early stopping criterion, which explains this factor in the running time depicted in Theorem 8.3. We give a short description of the underlying algorithm, and refer to it as the resource sharing algorithm:

Resource sharing algorithm

Resource weights are initialized with 1. In each iteration t, the customers are processed in arbitrary order, and each customer at least once per iteration. For each customer, its oracle function is called with the current resource weights, and subsequently the resource weights ω_i are updated based on their resource usages as follows:

$$\omega_i^{(t+1)} := \omega_i^{(t)} e^{\delta\alpha_r},$$

where α_r is the scaled usage of resource i by the customer ($\alpha_r \leq 1$ holds), and the parameter δ is part of the input. In the end the algorithm returns the scaled arithmetic mean of the solutions computed in the course of the algorithm.
Note that resource weights grow exponentially in the course of the algorithm.
In Theorem 8.3, the running time of an oracle call is bounded by θ. In other words, θ is bounded by the highest running time of all oracles, which is the gate customer oracle in our application. Together with the results from the previous sections, we get the following approximation ratio for gate sizing as min-max resource sharing problem:

Theorem 8.4 *For $\eta > 1$ fixed, the continuous relaxation of the gate sizing problem modeled as min-max resource sharing problem can be solved with approximation ratio $\eta(1+\epsilon)$ in time*

$$O(\theta(|V|+|E|)\log|E|(\log\log|E|+\epsilon^{-2}))$$

for any $\epsilon > 0$, where θ is bounded by the running time of the sizing oracle

$O\left(\frac{|\mathcal{G}|\cdot diam_X^2 \cdot tr_{ratio}}{\eta-1}\right)$. *The running time reduces to* $O(\theta(|V|+|E|)\log|E|\epsilon^{-2})$ *if* $\frac{1}{2} \leq \kappa^* \leq 2$.

η is the approximation ratio of the sizing oracle and can be chosen arbitrarily close to 1. If η cannot be regarded as bounded by a constant, the running time depends quadratically on η and is $O(\theta \log|E| \left[(|V|+|E|)\log\log|E| + (|V|+|E|\eta)\eta\epsilon^{-2}\right])$ (Müller et al. [MRV11]).

Remark 8.5 In the context of global routing, the resource sharing algorithm converges faster if the arrival time oracles are run multiple times in the algorithm before continuing with the next customer type. Details can be found in Held et al. [Hel+15].

8.5 Fast Approximation of the Continuous Relaxation

The continuous relaxation of the gate sizing problem can be solved up to a desired accuracy $\epsilon > 0$ in polynomial time for example with interior point methods (cf. Section 4.7). However, the complexity of interior point methods is approximately cubic in the number of variables (cf. Section 3.4). The instance sizes reported to be solved with interior point methods are usually small. The largest instance contained 100000 gates, and it took "tens of hours" to solve it, see Joshi and Boyd [JB08].
The projected gradient method for the Lagrange dual problem converges to an optimal solution if a strongly feasible solution exists for the convex program. However, the convergence rate is unknown, and it is not clear if the running time is polynomial (cf. Section 6.2).
In Section 8.4 we established that for a given power budget the resource sharing algorithm approximates the continuous relaxation of the gate sizing problem up to a factor of $\eta(1+\epsilon)$ in time $O(\theta(|V|+|E|)\log|E|(\log\log|E|+\epsilon^{-2}))$. Combined with a binary search, this gives an approximation of the continuous relaxation:

Theorem 8.6 *For* $\eta > 1$ *fixed, the continuous relaxation of the gate sizing problem can be approximated up to accuracy* $\eta(1+\epsilon)$ *in time*

$$O(\theta(|V|+|E|)\log|E|(\log\log|E|\log\log(K)+\epsilon^{-2})),$$

min_{power}, max_{power}

where $K := \frac{max_{power}}{min_{power}}$, *and* $min_{power} > 0$ *and* $max_{power} > 0$ *denote the minimum and maximum power consumption of any book in the gate library, respectively.*

Proof. We start with $B_u := |\mathcal{G}| \cdot max_{power}$ and $B_l := |\mathcal{G}| \cdot min_{power}$ as an upper and lower bound on the power budget, respectively. Note that initially, $B_u = K \cdot B_l$. With a binary search technique described in Young [You01] and Müller et al. [MRV11] we reduce the ratio between B_u and B_l such that they differ by only a factor of 2 after $O(\log\log(K))$ search steps. In each step, the resource sharing algorithm is called with $\epsilon = 1$, and feasibility of the current power budget is verified if the power resource usage is less or equal than $\eta(1+\epsilon) = 2\eta$. In a second phase,

the precision ϵ of the resource sharing algorithm is iteratively decreased such that the running time is dominated by the last call to the algorithm. It was shown that for $\zeta > 0$ the ratio B_u/B_l can be reduced to $(1+\zeta)$ in $O(\log(1/\zeta))$ binary search steps. We run the second phase until $B_u/B_l = (1+\epsilon)$. □

The running time is pseudopolynomial because the running time of the sizing oracle $\theta = O\left(\frac{|\mathcal{G}| \cdot diam_X^2 \cdot tr_{ratio}}{\eta - 1}\right)$ is pseudopolynomial (Algorithm 5.3). Of course, interior point methods can also be used as oracle algorithms, but this introduces a cubic dependency on $|\mathcal{G}|$ (cf. Section 5.1).
We consider the running time of the resource sharing algorithm more closely:
The term $\log\log|E|$ is introduced by a binary search to find an appropriate scaling factor for the resource usages (cf. Section 8.4). The term $(|V|+|E|)\log|E|$ bounds the number of oracle calls of all customers during the algorithm. In each iteration of the algorithm, a customer oracle is called more than once only if the corresponding customer usage of any resource is larger than 1. This introduces the term $|E|\log|E|$ (see Müller et al. [MRV11]). It is easy to see that this situation can occur only for the gate customer (consider (8.2) and (8.6)).

Fixed Gate Library and Reasonable Wire Lengths We consider the pseudopolynomial terms $diam_X$ and $tr_{ratio} = \frac{\max_{x \in X_{cont}} \max_i \{cost(x_i) + \Lambda \cdot \max_{e \in E} delay_e(x)\}}{\min_{x \in X_{cont}} \{cost(x),\ \min_{e \in E} delay_e(x)\}}$ in the running time of the sizing oracle under the assumption that we are given a fixed gate library:
Under this assumption, $diam_X = \max_{g \in \mathcal{G}} ||u_g - l_g||$ can be regarded as bounded by a constant. Similarly, min_{power} and max_{power} can be regarded as bounded by a constant from below and above, respectively, and hence K.
Then $\frac{1}{\min_{x \in X_{cont}} cost(x)} \leq \frac{1}{|\mathcal{G}| \cdot min_{power}} \leq \frac{1}{min_{power}}$. Additionally, it is realistic to assume that the bound Λ on the maximum fanout of a gate is constant (cf. Section 5.1).
The situation is more involved for the delay of an edge $e \in E$: It is reasonable to assume that the minimum delay of any edge is bounded by a constant from below. However, the maximum delay can be very large due to large input slews or load capacitances. If e is a wire edge, the underlying wires can be very long such that the speed of the signals degrades.
Therefore we make the additional assumption that our algorithm is called at a state of the design flow where wires are reasonably long due to repeater insertion, and thus their length does not induce huge load capacitances or degrade slews significantly. Large load capacitances and slews are then induced by inadequate gate sizes, but our gate library is fixed and thus, in combination with a bounded fanout, load capacitances and slews are bounded by a constant. Then the maximum delay over an edge is bounded by a constant, but can become very large for inadequate gate sizes.
Putting all preceding considerations together, the terms in tr_{ratio} can be regarded

as bounded by a constant, and consequently tr_{ratio}. We regard the oracle approximation ratio η as fixed. This yields a running time of

$$O\left(|\mathcal{G}| \cdot (|V| + |E|)) \log |E| (\log \log |E| + \epsilon^{-2})\right)$$

for the approximation of the continuous relaxation of the gate sizing problem.

8.6 Path Resources instead of Edge Resources

Hähnle [Häh15] proposed to treat paths as delay resources in the context of timing-driven global routing. More precisely, the timing graph G is extended by a unique source s and a unique sink t, and the set of resources corresponds to the set $\mathscr{P}$ of directed s-t-paths. This reduces the number of customers significantly, as the arrival time customers are no longer necessary to perform delay budgeting. But most importantly, the approximation guarantee for κ^* now directly translates to the path delays without the construction of Theorem 8.2.

Hähnle [Häh15] uses a scale-free variant of the multiplicative weights algorithm, where resource usages $y \in \mathbb{R}^{|\mathscr{R}|}$, with $y_r := \sum_{c \in \mathscr{C}} (g_c(b_c))_r$ for resource r, are computed by the customer oracles in each iteration and then scaled with $1/\left|\left|y\right|\right|_\infty$ in the weight update. Formally, the algorithm optimizes over a set of resource usages and only implicitly over the set of gate sizes, and a convex combination of resource usages is returned in the end. As resource usages are convex functions, the approximation guarantee of the convex combination of resource usages transfers to the convex combination of the sizes computed in each iteration with the same factors.

In the context of gate sizing, the resources are now the set of s-t paths and the power resource. The gate customer is the only customer, and Algorithm 5.3 serves as sizing oracle because the path resource weights and budgets can be decomposed into weights and budgets on the edges in the timing graph. For more details we refer to Hähnle [Häh15] and Section 9.3, where we describe an implementation of this algorithm. The following theorem is implied by Hähnle [Häh15]:

Theorem 8.7 *Given $\eta > 1$ fixed, the continuous relaxation of the gate sizing problem modeled as min-max resource sharing problem can be solved with approximation ratio $\eta(1+\epsilon)$ in*

$$O\left((\theta + |E|)\epsilon^{-2}|E| \log |\mathscr{P}|\right)$$

time for any $\epsilon > 0$, where $\theta = O\left(\frac{|\mathcal{G}| \cdot diam_X^2 \cdot tr_{ratio}}{\eta - 1}\right)$ is the running time bound for the sizing oracle.

Note that the number of $s-t$ paths depends exponentially on the instance size, but the running time of the algorithm depends on $\log |\mathscr{P}|$ only.

With a binary search as described in the proof of Theorem 8.6, we obtain a fast approximation of the continuous relaxation of the gate sizing problem.

Under the assumption that we are given a fixed gate library and reasonable wire lengths as discussed in Section 8.5, the running time is polynomial.
Note that the oracle approximation ratio η does not contribute to the number of oracle calls even if it is variable, in contrast to Theorem 8.4.
We discuss in Section 8.7 that this model is useful when it comes to randomized rounding of special cases of the discrete problem that allow more than one gate customer. In Chapter 9 we compare an implementation of this algorithm for gate sizing with an implementation of the discretized Lagrangian relaxation approach, and refer to is as the path resource sharing algorithm.

8.7 Resource Sharing for the Discrete Problem and Special Cases

Given the good approximation for the continuous relaxation of the gate sizing problem, we now turn towards the discrete problem. The obstacles that lie in the way of an approximation algorithm are a discrete sizing oracle, which is equivalent to the discrete power-delay tradeoff problem (5.2) treated in Section 5.2, and rounding.
Under the assumption that an approximation algorithm for the discrete sizing oracle exists, the oracle error would directly translate into the final approximation ratio and running time, and pseudopolynomial running time could be guaranteed.
We now analyze rounding from a theoretical point of view and provide a bound on the rounding error for a special case, the discrete time-cost tradeoff problem.

Discrete Gate Sizing as Resource Sharing Problem

The resource sharing algorithms from Section 8.4 and Section 8.6 can be discretized by calling a discrete sizing oracle. In both cases, the resource sharing algorithm returns a convex combination of gate size vectors, which is not necessarily a discrete solution and needs to be rounded. The arrival time variables, if existent, need not be rounded.
We discussed rounding of a convex combination of discrete sizes already in Section 7.2.2, and similar considerations apply here. It seems natural to take the best solution over all iterations, i.e. the sizes that minimize the maximum resource usage, but no approximation guarantees can be given. Alternatively, the solution from the last iteration can be a good choice.
We are also interested in theoretical guarantees that can be obtained by randomized rounding, which is used in Müller et al. [MRV11] in the context of global routing. Here random variables are scaled resource usages of the customers, in other words one random variable is introduced for each combination of customer and resource.
In our application we only have one customer whose solution needs to be rounded, and we shortly point out why randomized rounding is not reasonable here. At the end of this section we transfer these considerations to a special case of gate sizing.

By a lemma from Raghavan [Rag88], the sum of usages of resource i for a rounded solution is larger than a certain threshold with a probability that depends on

$$\rho_i := \max\{g_c(b)/\kappa \mid b \in \mathscr{B}_c, c \in \mathscr{C}, b \text{ occurs in convex combination}\}. \tag{8.7}$$

Here we used the general notation from Section 8.1, and κ is the solution returned by the resource sharing algorithm. Let $\hat{\kappa}$ be the maximum resource usage after rounding. Then the probability that $\hat{\kappa} \leq \kappa(1+\delta)$ is at least $1-\sum_{resources\ i} e^{-h(\delta)/\rho_i}$, with $h(\delta) := (1+\delta)\ln(1+\delta) - \delta$ for $\delta > 0$ (see Müller et al. [MRV11]). Obviously, δ cannot be chosen arbitrarily. In the global routing context, the values of ρ_i are small, and thus δ can be chosen quite small in order to have a positive probability. In the context of gate sizing, we have only one customer and thus one random variable for each resource. Consequently, the value of each ρ_i will be relatively large, irrespective of whether we model timing constraints by edge or by path resources: For the model with path resources, ρ_i will be larger than one for most resources. For edge resources, the arrival time customers additionally consume from the delay resources. Here the magnitude of each ρ_i depends on the ratio of the delay budget to the maximum edge delay.

Discrete Time-Cost Tradeoff as Resource Sharing Problem

We proceed with the analysis of randomized rounding for a special case of the gate sizing problem. The discrete time-cost tradeoff problem can be regarded as gate sizing with a very simplified, and unrealistic, delay model, see also Section 5.2: The delay of each edge is independent of input slew and load capacitance. Wire edge delays are constant, and several edges within a gate are contracted to a single edge. Let further be $X_{disc} \subset \mathbb{Z}^n$, and assume that each gate edge delay is proportional to the gate size. In particular, edge delays only depend on the size of one gate, and we can model each gate as a separate customer. We already established in Section 5.2.1 that the corresponding discrete sizing oracle can be implemented in polynomial time.

Edge Resources Consider the resource sharing model with edge delay resources. We have one random variable for each resource, and randomized rounding introduces an additional factor of $(1+\delta)$ into the approximation of each edge resource usage. It is easy to see that the timing constraint violation at any timing endpoint $w \in V$ is bounded by the sum of timing constraint violations on a path to w, and that this bound depends on the length k of a longest path in the timing graph. Due to randomized rounding, this bound increases by a factor of $(1+\delta)$.

Path Resources In view of that, we model paths as resources (cf. Section 8.6) such that the additional factor $(1+\delta)$ in the approximation guarantee due to rounding directly translates to the path delays, and the bound on timing constraint violations at timing endpoints does not depend on k.

As before, the random variables are the scaled resource usages of the customers, and again we consider the probability that $\hat{\kappa} \leq \kappa(1+\delta)$, where $\hat{\kappa}$ is the maximum resource usage after rounding the gate sizes. Obviously, δ cannot be chosen arbitrarily small, because the probability $1 - \sum_{resources\ i} e^{-h(\delta)/\rho_i}$ needs to be larger than zero, and preferably be larger than $\frac{1}{2}$. This condition is certainly fulfilled if for all resources i

$$\begin{aligned} e^{-h(\delta)/\rho_i} &< 1/(2|\mathscr{P}|+1) \Leftrightarrow \\ h(\delta) &> \ln(2|\mathscr{P}|+1)\rho_i. \end{aligned}$$

holds. In contrast to gate sizing with one gate customer, ρ_i as defined in (8.7) is usually quite small for path resources and the power resource, because one single gate customer does not contribute much to the delay of a path in G or total power consumption. Nonetheless, in the worst case ρ_i can still be larger than one.

Lemma 8.8 *The discrete time-cost tradeoff problem modeled as a min-max resource sharing problem can be solved with approximation ratio $\eta(1+\epsilon)(1+\delta)$ in*

$$O(\theta\epsilon^{-2}|E|\log|\mathscr{P}|)$$

time for any $\epsilon > 0$, where θ is the running time bound for the discrete sizing oracles, $\eta > 1$ is the oracle error, and δ is chosen such that $h(\delta) > \max_i \ln(2|\mathscr{P}|+1)\rho_i$.

The discrete sizing oracle can be implemented to run in polynomial time, but $\eta > 1$ holds for the oracle error: Similar to Section 6.5, the min-max resource sharing problem is a convex problem in the sense that resource usages and feasible solutions are convex. For each customer, the minimum weighted resource usage is not necessarily attained by the discrete oracle solution. This implies a sizing oracle error $\eta > 1$. We shortly discuss a bound on η. Assume that the optimal continuous oracle solution $\tilde{x_i} \in X_{cont}$ of gate g_i lies between two discrete solutions $\underline{x_i} \leq \tilde{x_i} \leq \bar{x_i}$. One of these is the optimal discrete solution. Thus η can be bounded by $\max_{1\leq i\leq n} \frac{2tr(\bar{x_i},\omega)}{tr(\bar{x_i},\omega)-tr(\underline{x_i},\omega)}$ for $\omega \in \mathbb{R}^{|\mathscr{R}|}$, where e_i is the single edge traversing gate g_i whose delay only depends on the size of g_i by construction.
It is an interesting problem to decide if the dependency of δ on the number of paths can be transferred to a dependency on the number of edges in the timing graph, which would lead to a better approximation ratio.

8.8 Capacitance, Slew and Placement Density Resources

Similar to the Lagrangian relaxation approach (cf. Section 6.6), we can incorporate constraints on load capacitance, slew and placement density into the resource sharing framework under certain assumptions. We use the notation from Section 6.6.

Capacitance Resources We introduce a resource for each net $N \in \mathcal{N}$ to model the convex constraints on load capacitances (6.12):

$$loadcap_p(x)/loadlim_p(x) \leq 1 \ \forall p \in V_{load}, x \in X_{cont}.$$

The gate customer uses from all capacitance resources. Intuitively, the resource usage of a net N increases if the size of the source gate decreases and the size of a sink gate increases. It is specified as:

$$\frac{loadcap_{p(N)}(x)}{loadlim(x)}, \tag{8.8}$$

where $p(N) \in V_{load}$ is the driver pin of net N. In other words, the budget of the net resource equals 1. With the considerations from Section 6.6.2, this is a convex function under the assumption that the load limit of a gate scales linearly with its size.

Slew Resources Recall the convex slew constraints (6.13)

$$slew_e(x, slew_v) \leq slewlim_p \ \forall p \in V_{slew}, \ e = (v, p) \in E, \ x \in X_{cont}.$$

Slew effects can be incorporated into our convex delay model, and we assume a convex slew function $slew_e(x, slew_v), \ x \in X_{cont}$ for each edge $e = (v, p) \in E$ (cf. Section 4.4.1 and Section 6.6). Similar to delay functions, the slew over an edge depends on the sizes of several gates. We introduce a slew resource for each $e = (v, p) \in E$ with $p \in V_{slew}$ and budget $slewlim_p$, which is usually independent of the size of gate $\gamma(p)$. The slew resource usage by the gate customer is

$$\frac{slew_e(x, slew_v)}{slewlim_p}, \ x \in X_{cont}, \tag{8.9}$$

and it is easy to see that this is convex. Thereby we can assume the input slew to be a default slew, or we can use input slews computed by the oracle algorithm, as will be described at the end of this section.

Alternatively, slew constraints can be modeled by requesting that edge delays should not exceed a certain budget, because slew functions have shapes similar to delay functions. However, the model we propose is more accurate.

Remark 8.9 (V_t optimization) The slew limit of a gate input pin $p \in V_{slew}$ is usually independent of the size, but not the V_t level of the gate. We can model a changing slew limit $slewlim_p$ similar to the way we modeled a changing load capacitance limit in (8.8), however, the resulting resource usage is in general not convex.

Placement Density Resources Recall the placement density constraints (6.9):

$$\frac{1}{|R_w|}\sum_{g_i\in R_w} area(x_i) \le target_w,\ w = 1,\dots,q. \tag{8.10}$$

The chip area is partitioned into regions $R_1,\dots,R_q$, and for each region R_w we are given a placement density target $target_w$. Gates are assigned to regions as described in Section 6.6.
We introduce a resource for each region, and each gate uses area from the region it is assigned to. The area budget of region R_w is $|R_w|\cdot target_w$, and the resource usage of a single gate g_i with size x_i of region R_w is $\frac{area(x_i)}{|R_w|\cdot target_w}$. Consequently, the usage of a region resource by the gate customer is

$$\sum_{g_i\in R_w} \frac{area(x_i)}{|R_w|\cdot target_w}, \tag{8.11}$$

where $g_i \in R_w$ means that the area of gate g_i is in region R_w. As required, this is a convex function in $x \in X_{cont}$, see Section 6.6. Similar considerations as in Section 6.6 to improve the assignment of gates to regions apply here.

Oracle Function Putting together, the weighted resource usage of the gate customer, from all resources we have introduced until now, is

$$\begin{aligned}&\omega_{m+1}\frac{cost(x)}{budget_{power}} + \sum_{e\in E}\omega_e\frac{delay_e(x)}{budget_e} + \omega_{R_w}\sum_{g_i\in R_w}\frac{area(x_i)}{|R_w|\cdot target_w} \\ &+ \sum_{N\in\mathcal{N}}\omega_N\frac{loadcap_{p(N)}(x)}{loadlim(x)} + \sum_{e=(v,p)\in E}\omega_p\frac{slew_e(x,slew_v)}{slewlim_p}\ .\end{aligned} \tag{8.12}$$

The resource usages defined in this section are convex functions of $x \in X_{cont}$, and it is easy to see that (8.12) is of the form (5.3). As before, the conditional gradient method can be used as oracle (cf. Section 5.1.3). In each iteration, the input slews induced by the previous iterate can serve to compute the slew resource usages.
For discrete gate sizes, we can for example deploy Algorithm 5.5, but without any performance guarantee. Here the gate graph is traversed in topological order, which enables us to propagate the correct input slew values.

8.9 Integration with Global Routing and Repeater Insertion

Modeling gate sizing as a resource sharing problem allows a better integration with other optimization in VLSI design steps like timing-driven global routing and repeater insertion, which can also be interpreted as a min-max resource sharing problem (Müller et al. [MRV11], Held et al. [Hel+15]).

Timing-Driven Global Routing

Global routing algorithms generate an approximate wiring of each net in a coarse global routing graph, which restricts the search space for the actual connections of a net in the subsequent detailed routing step. Global routing is also used in earlier design steps for congestion estimation, for example in the placement stage by Brenner et al. [Bre+15]. Müller et al. [MRV11] model global routing as a min-max resource sharing problem and show that it is easy to include several objectives into this framework, among them power usage, wiring length, manufacturing yield etc. In this context, customers are nets and each edge in the global routing graph corresponds to a resource. The usage of an edge resource is the usage of the Steiner tree realizing this net. The algorithm proposed by Müller et al. [MRV11] deploys oracles that return convex combinations of Steiner trees. In the end, a Steiner tree for each net is computed by randomized rounding of the fractional solution.
As mentioned before, Held et al. [Hel+15] integrated timing constraints into this framework in the form of arrival time customers and a timing- and congestion-aware computation of Steiner trees. They introduced an arrival time customer for each pin in the timing graph except for gate output pins, and a resource for each net edge in the timing graph. Gate delays are integrated in the delay over nets. A notable difference to the algorithm described in Müller et al. [MRV11] is that first the net customers are processed, and then an internal loop processes the arrival time customers for a fixed number of iterations to obtain a better fractional solution.

Integrating Gate Sizing

In order to incorporate gate sizing into this framework, we extend the set of customers such that each pin in the timing graph is represented by an arrival time customer, and add the gate customer. Additionally, each edge in the timing graph is now a resource and not only the net edges.
However, the delay usage of a net customer cannot be specified independent of the gate customer and vice versa, as both customer types have an impact on the delay usage of the other one. This is illustrated in Figure 8.2:
For net customers, the delay usage of wire edges is determined by the Steiner tree realizing the net. For a gate edge, the load capacitance of the Steiner tree contributes to the load capacitance at the gate output pin. Changing the sizes of the green gates in the left picture affects the delay usages of the net customers corresponding to the red nets, even though these customers are not changed. Similarly, rerouting the purple net in the right picture affects the delays of all gate edges indicated in red. Thus we cannot specify convex resource usage functions for the gate and net customers.
For the gate sizing problem we resolved this dependency with a single gate customer. For combined sizing and routing with a single customer for all gates and nets it is an interesting question whether an oracle algorithm for simultaneous gate sizing and routing exists.

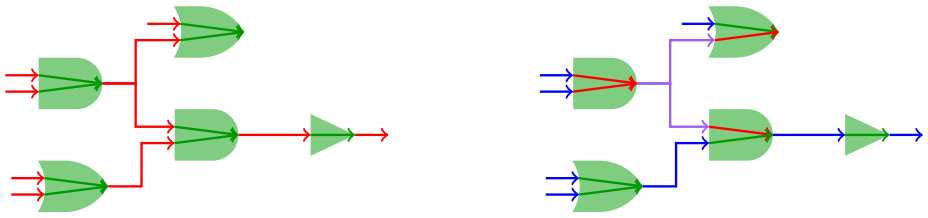


Figure 8.2: Changing the gate customer (left side) or the purple net customer (right side) impairs the delay usage of other customer types.

For practical application we propose a heuristic algorithm that alternately proceeds the net customers, the gate customer and the arrival time customers in each iteration. For each customer type, several iterations are performed before moving on with the next customer type. This algorithm has no theoretical performance guarantee, even for the continuous relaxation.
We further need lower and upper bounds on earliest and latest possible arrival times for each pin to compute the delay budgets. Non-trivial bounds can be computed in polynomial time for all $v \in V_{inner}$ for example as follows: Given a lower bound on all edge delays, earliest and latest arrival times can be computed by forward and backward propagation through the timing graph as in (7.12). We assign the smallest possible load capacitance at the output pin of each gate, and choose the largest size for the gate. The resulting delay is a lower bound on gate delays. The load capacitance is the sum of wire capacitances of its net and the input pin capacitances of the sinks. We get a lower bound on load capacitance when the sink pin capacitances are smallest possible (for gate input pins we assign the smallest size to the gate), and the capacitance of the Steiner tree is smallest possible, i.e. it is a shortest Steiner tree. The shortest Steiner tree problem is *NP*-hard, but for example the minimum maximum capacitance of a shortest source-sink connection in the net is a lower bound.
For net edges, we assign the largest size to the driver gate (if the driver pin is a gate pin) and the smallest size to the sink gates. As in Held et al. [Hel+15] we choose as lower delay bound the delay which results from connecting source and sink individually with a shortest connection.

8.10 Evaluation of the Resource Sharing Model

This section is dedicated to an evaluation of the resource sharing model for gate sizing by comparison with the approaches already discussed in this thesis. To this end, we formulate the continuous relaxation of gate sizing modeled as a min-max resource sharing problem as feasibility problem of the form (7.1) and apply the multiplicative weights algorithm. We compare its running time and performance with that of Algorithm 7.4 for problem (7.5). We will again use the basic variant of the multiplicative weights algorithm that divides the costs by the problem width

instead of the more sophisticated scaled variant to point out the differences in the models. Furthermore, we use the resource sharing model with edge delay resources, as this is compliant with the convex program.

8.10.1 Comparison with Lagrangian Relaxation

The resource sharing algorithm is a variant of the multiplicative weights algorithm, and a detailed comparison between the Lagrangian relaxation approach and the multiplicative weights approach for gate sizing can be found in Section 7.2.4. The main insights concerning the objective function handling, the oracle functions, the weight update and starting solution are also valid in the resource sharing framework.

Additionally, we now optimize over the set of resource usage vectors and only implicitly over the set of gate sizes. This impacts the approximation of the timing constraints and will become clear in Section 8.10.3 when we compare the multiplicative weights algorithm applied to the different problem formulations.

The running times of the resource sharing algorithm and the projected gradient method for the continuous relaxation have already been discussed in Section 8.5.

8.10.2 Formulation as Feasibility Problem

We assume we are given a power budget $budget_{power} \in \mathbb{R}_{\geq 0}$ that can be determined by binary search (cf. Section 7.2.3). Further assume that $h_e = 0$ for all $e \in E$ and we set $atmin_v := l_v, atmax_v := u_v$ for earliest and latest signal arrival times l_v and u_v at $v \in V$, respectively, such that we have the same constraints on arrival times as in Algorithm 7.4 (cf. Section 7.2.1). This implies $budget_e = u_v - l_v$. For feasible arrival times and sizes $(a, x) \in \mathcal{A} \times X_{cont}$ we denote the power and delay resource usages by $y_i(a, x)$ for $i = 1, \ldots, m+1$. Let $\mathcal{Y} \subset \mathbb{R}^{m+1}_{\geq 0}$ denote the set of feasible resource usage vectors induced by $(a, x) \in \mathcal{A} \times X_{cont}$, and let further $u := \min\{budget_{power}, \min_{e \in E} budget_e\}$, $U := \max\{budget_{power}, \max_{e \in E} budget_e\}$.

$y_i(a,x)$, $\mathcal{Y}$

u, U

Lemma 8.10 ([Hel+15]) *The power budget $budget_{power}$ is met and the timing constraints of the convex program (4.10) are fulfilled if $y_i \leq 1$ for all $i = 1, \ldots, m+1$.*

The feasibility problem of the form (7.1) for gate sizing modeled as min-max resource sharing problem differs from the feasibility version of the convex program (7.5) insofar as we are now optimizing over the set of resource usage vectors and only implicitly over the gate sizes:

FEASIBILITY PROBLEM FOR RESOURCE SHARING

$$\begin{aligned} & \exists?\ y \in \mathcal{Y} \\ \text{subject to } & 1 - y_{m+1}(a,x) = 1 - \frac{cost(x)}{budget_{power}} && \geq 0 \\ & 1 - y_e(a,x) = \frac{a_w - (a_v + delay_e(x))}{budget_e} && \geq 0 \ \forall\ e \in E \end{aligned} \tag{8.13}$$

We introduce a weight ω_i for each constraint/resource, and the multiplicative weights algorithm increases a weight if the corresponding constraint is violated.

The width $\rho_{\mathbf{r}}$ According to Definition 7.1, the width of problem (8.13) can be bounded by the smallest number $\rho_r \geq 0$ such that $\frac{1-y_i(a,x)}{\rho_r} \leq 1$ for $i = 1, \ldots, m+1$: Width ρ_r

$$\rho_r \ := \max_{x \in X_{cont}, a \in \mathcal{A}} \left\{ \max_{e \in E} \left| \frac{a_w - (a_v + delay_e(x))}{budget_e} \right|, \left| 1 - \frac{cost(x)}{budget_{power}} \right| \right\}. \tag{8.14}$$

This can be bounded by $O(\frac{\rho}{u})$, where ρ is the width of problem (7.5) defined in equation (7.14).

Oracle Similar to Section 7.2.1, the multiplicative weights algorithm requires an oracle that solves a weighted feasibility problem of the form (7.2) up to accuracy $\eta > 0$ for resource weights $\omega \in \mathbb{R}^{m+1}_{>0}$:

$$\begin{aligned} & \max_{x \in X_{cont}, a \in \mathcal{A}} \left(\sum_{i=1}^{m+1} \omega_i \cdot (1 - y_i(a,x)) \right) \\ = \quad & \max_{x \in X_{cont}} \left(\omega_{m+1} \cdot \left(1 - \frac{cost(x)}{budget_{power}} \right) - \sum_{e \in E} \omega_e \frac{delay_e(x)}{budget_e} \right) && (8.15) \\ & + \max_{a \in \mathcal{A}} \left(\sum_{v \in V} a_v \left[\sum_{e \in \delta^-(v)} \frac{\omega_e}{budget_e} - \sum_{e \in \delta^+(v)} \frac{\omega_e}{budget_e} \right] \right). && (8.16) \end{aligned}$$

Finding the maximum in (8.15) is equivalent to problem (7.8), whereas (8.16) can be solved with Algorithm 7.2 as in Section 7.2.1. Hence we can use Algorithm 7.3 as η-approximate oracle (Theorem 7.9) with running time $O(\frac{n \cdot diam_X^2 \cdot lip(\widetilde{\omega})}{\eta} + m)$, where $\widetilde{\omega} := (\frac{\omega_{e_1}}{budget_{e_1}}, \ldots, \frac{\omega_{e_m}}{budget_{e_m}}, \frac{\omega_{m+1}}{budget_{power}})$.

Algorithm Algorithm 8.1 is essentially the same as Algorithm 7.4, but the algorithms differ in the weight update. Moreover, Algorithm 7.4 calls Algorithm 7.3

(for the weighted feasibility problem) in line 5 with weights $\omega \in \mathbb{R}^{m+1}_{>0}$ not scaled by budgets.

Algorithm 8.1 Solving the feasibility problem for resource sharing

1: **procedure** FeasibilityProblemRS($\eta, budget_{e_1}, \ldots, budget_{e_m}, budget_{power}$)
2: Fix $\nu \leq 0.5$
3: $\omega^{(1)} \leftarrow \mathbb{1}$
4: **for** $t = 1, \ldots, T$ **do**
5: $(a^{(t)}, x^{(t)}) \leftarrow \text{ORACLE}\left(\frac{\eta}{3}, (\frac{\omega^{(t)}_{e_1}}{budget_{e_1}}, \ldots, \frac{\omega^{(t)}_{e_m}}{budget_{e_m}}, \frac{\omega^{(t)}_{m+1}}{budget_{power}})\right)$
6: $y^{(t)}_e \leftarrow \frac{atmax_w - a^{(t)}_w + a^{(t)}_v - atmin_v + delay_e(x^{(t)})}{budget_e}$ $\forall e = (v, w) \in E$
7: $y^{(t)}_{m+1} \leftarrow \frac{cost(x^{(t)})}{budget_{power}}$
8: **if** $\left(\sum_{i=1}^{m+1} \omega^{(t)}_i \cdot (1 - y_i(a^{(t)}, x^{(t)}))\right) < -\frac{\eta}{3}$ **then**
9: **return** $\vec{0}$
10: **end if**
11: $\omega^{(t+1)}_i \leftarrow \omega^{(t)}_i (1 - \frac{\nu}{\rho_r} \cdot (1 - y^{(t)}_i))$ for all $i = 1, \ldots, m$
12: **end for**
13: Return $\bar{y} := \frac{1}{T} \sum_{t \leq T} y^{(t)}$, $\bar{x} := \frac{1}{T} \sum_{t \leq T} x^{(t)}$, $\bar{a} := \frac{1}{T} \sum_{t \leq T} a^{(t)}$
14: **end procedure**

Theorem 8.11 *Let $budget_{power}$ be an upper bound on the objective function, and $\eta > 0$. Assume that $\rho_r \geq \frac{\eta}{3}$. Then Algorithm 8.1 returns in $T = O(\frac{\log(m) \cdot \rho_r^2}{\eta^2})$ iterations vectors $\bar{x} \in X_{cont}$, $\bar{a} \in \mathcal{A}$ and $\bar{y} \in \mathcal{Y}$ with $\bar{y} \leq \mathbb{1} + \eta$, in other words*

$$\begin{aligned} cost(\bar{x}) &\leq budget_{power}(1 + \eta), \text{ and} \\ \bar{a}_v + delay_e(\bar{x}) &\leq \bar{a}_w + \eta \cdot budget_e \ \forall e \in E, \end{aligned}$$

or correctly decides that problem (8.13) is infeasible.

Proof. The proof is essentially the same as for Theorem 7.10, which is based on Theorem 7.6. We set $\nu = \frac{\eta}{\rho_r \cdot 6} \leq \frac{1}{2}$ and $T = \lceil \frac{18 \rho_r^2 \log(m)}{\eta^2} \rceil$.

□

Corollary 8.12 follows immediately from Corollary 7.11:

Corollary 8.12 Let k be the length of the longest path in G, and $\bar{x}$, $\bar{a}$ be the solution returned by Algorithm 8.1. Then $\bar{a}_p \leq rat_p + U \cdot \eta \cdot k$ holds for all $p \in V_{end}$.

8.10.3 Comparison with Algorithm 7.4

Running Times By Theorem 8.11 and Theorem 7.10, the running times of Algorithm 8.1 and Algorithm 7.4 depend quadratically on the problem widths ρ_r and ρ, respectively. Thus the number of iterations in Algorithm 8.1 is by factor u^2 smaller

than in Algorithm 7.4. The same holds for the upper bounds on the constraint weights. Furthermore, for $\omega \in \mathbb{R}^{m+1}_{>0}$ and $\widetilde{\omega} = (\frac{\omega_{e_1}}{budget_{e_1}}, \ldots, \frac{\omega_{e_m}}{budget_{e_m}}, \frac{\omega_{m+1}}{budget_{m+1}})$ we have that

$$\begin{aligned} lip(\widetilde{\omega}) &\leq \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left\{ \frac{\omega_{m+1}}{budget_{power}} cost(x_i) + \Lambda \cdot \max_{e \in E} \frac{\omega_e}{budget_e} delay_e(x) \right\} \\ &\leq \frac{1}{u} \cdot \max_{x \in X_{cont}} \max_{1 \leq i \leq n} \left(\omega_{m+1} cost(x_i) + \Lambda \cdot \max_{e \in E} \omega_e delay_e(x) \right) = \frac{1}{u} \cdot lip(\omega) \end{aligned}$$

holds for the Lipschitz constants $lip(\widetilde{\omega})$ and $lip(\omega)$ of the sizing subproblems (8.15) and (7.8) that arise in each iteration of Algorithm 8.1 and Algorithm 7.4, respectively. Algorithm 7.3 approximately solves the sizing subproblems, and its running time depends linearly on the Lipschitz constants, which in turn depend on the constraint weights. As the weights computed in the course of Algorithm 8.1 and Algorithm 7.4 differ, we cannot directly compare running times of these algorithms. Nonetheless, the worst case running time of an iteration is larger in Algorithm 7.4 due to the larger Lipschitz constant of the sizing subproblem, and the larger upper bound on the weights that is implied by the larger number of iterations.

Worst Slack Maximization By Corollary 7.11, $\bar{a}_w \leq rat_w + \eta \cdot k$ holds for all $w \in V_{end}$ and the solution returned by Algorithm 7.4. Similarly, we have by Corollary 8.12 that $\bar{a}_w \leq rat_w + \eta \cdot k \cdot U$ holds for all $w \in V_{end}$ and for the solution returned by Algorithm 8.1. This implies an inferior guarantee on the timing constraints for the solution returned by Algorithm 8.1 if $U > 1$, which is a reasonable assumption. Additionally, more accurate (smaller) budgets imply a better approximation ratio. On the other hand, maximizing the weighted feasibility problem (8.15) can be done independently for the size variables and arrival time variables. Therefore we can also use, without a running time degradation, the delay budgets of Held et al. [Hel+15] with $h_e \neq 0$ (cf. Theorem 8.2). It is easy to see that this does not increase the width ρ_r. Then minimizing the maximum resource usage, which is equivalent to maximizing $1 - y_i(a, x)$ for all $i = 1, \ldots, m + 1$, is equivalent to maximizing the worst slack.

8.10.4 Conclusion

The resource sharing algorithms described in Section 8.5 and Section 8.6 improve over Algorithm 8.1 and Algorithm 7.4 because their running times are independent of the resource weights and problem widths. We conclude that the resource sharing model for the gate sizing problem leads to a fast approximation of the continuous relaxation, and allows to model timing objectives like worst slack maximization more directly (see also Section 8.3.3).

9 Experimental Results

Having compared the resource sharing with the Lagrangian relaxation approach for gate sizing from a theoretical point of view, we now turn towards a comparison in practice.
We implemented a Lagrangian relaxation approach and a new resource sharing algorithm with path resources for discrete sizing and V_t optimization. Both algorithms are built around a common oracle algorithm for sizing and V_t optimization that can be run in parallel. Resource weights and Lagrange multipliers are updated sequentially in each iteration and fed into the oracle algorithm, which queries the resource weights and Lagrange multipliers as needed.
Using the same oracle algorithm enables us to directly compare the different weight update schemes. The purpose of this chapter is to get a direct comparison between both algorithms, and a first evaluation of the resource sharing model for gate sizing and V_t optimization in practice.
As a sizing oracle we extend the local search based sizing tool BonnRefine (Held [Hel09]) that is part of the BonnTools optimization suite for VLSI physical design.
We start with a description of the sizing oracle in Section 9.1 followed by implementation details of our Lagrangian relaxation (LR) algorithm in Section 9.2 and path resource sharing (RS) algorithm in Section 9.3. The framework for the path resource weights was provided by S. Daboul. We describe our testbed and setup in Section 9.4, including our choice of starting solutions, evaluation metrics and different optimization modes. Finally, both algorithms are compared in Section 9.5 on a testbed consisting of 8 microprocessor units provided by our industrial partner IBM with 22 nm and 14 nm technology, and the ISPD 2013 benchmarks (Ozdal et al. [Ozd+13]) in Section 9.6. We conclude the chapter with a short summary and an outlook on future research.

9.1 BonnRefine as Oracle Algorithm

Our oracle algorithm returns a solution to the discrete power-delay tradeoff problem (5.2), but in general not an optimal solution. As discussed in Section 5.2.1, no approximation algorithms are known for this problem.
We employ a discretized version of Algorithm 5.1 that is widely used in practice. This algorithm optimizes gates iteratively in reverse topological order, and for each gate the discrete solution which minimizes its local refine function as defined in (5.4) is chosen. Recall that the local refine function for a gate $g_i \in \mathcal{G}$ is the weighted sum of its power consumption and the edge delays in its neighborhood graph E_{g_i}

(cf. page 24). More formally,

$$tr_x(x_i, \omega) := \omega_{m+1} cost(x_i) + \sum_{e \in E_{g_i}} \omega_e delay_e(x)$$

for $x \in X_{disc}$ and weights $\omega_{m+1} \in \mathbb{R}_{\geq 0}$, $\omega_e \in \mathbb{R}_{\geq 0}$ for $e \in E$. Here all entries of x are fixed except for the i-th entry.

In the LR algorithm, ω_{m+1} equals 1 and the weights ω_e for $e \in E$ correspond to the Lagrange multipliers. In the RS algorithm, the weights correspond to the edge weights that are derived from the path resource weights, divided by the resource budgets. This will become clear in Section 9.3. We integrated the budgets into the weights to simplify notation.

Recall that in the LR algorithm we aim to find sizes and V_t levels that are close to the optimal solution, while in the RS algorithm we are interested in sizes and V_t levels such that the value of the power-delay tradeoff function is close to the optimum. In both cases, no approximation algorithms are known, and we do not distinguish between the two objectives in the following.

Furthermore, recall that the local refine function of a gate g_i depends on the sizes of other gates. The oracle algorithm aims to minimize the power-delay tradeoff function for given weights, but in practice it is not clear with which sizes and V_t levels to start when optimizing the gates iteratively. It is reasonable in practice to start with the solution computed in the last iteration of the LR and RS algorithm, which is what we did in our implementations. In the RS algorithm, a convex combination of the solutions computed in the previous iterations can also be used by assigning capacitances and slews appropriately, because existing convergence guarantees refer to convex combinations of solutions.

Our oracle uses the infrastructure of the sizing tool BonnRefine, which, in its general setting, computes local slack optima under arbitrary delay models based on local search. We added as new solution evaluation metric the value of the local refine function, to which we refer from now on as *refine cost*. Furthermore, we integrated V_t optimization into BonnRefine and refer to a new size or V_t level for a gate, or a combination of both, as *solution candidate*.

Refine cost

Solution candidate

The industrial timing engine IBM EinsTimer is used for all delay, slew and slack calculations under the Elmore delay model [Elm48]. Wires are estimated as approximately shortest Steiner trees. To bound the running time of the algorithm, delay recalculations are restricted to a bounded number of logic levels around each gate by the timing engine. This prevents propagation of delay and slew changes through the whole timing graph in each sizing and V_t optimization step, but introduces small inaccuracies. In our setting, we restrict delay recalculations to the neighborhood graph of the gate.

Nonetheless, it is time-consuming to evaluate the local refine function for each solution candidate of a gate. Therefore we skip some candidates as follows: Let s be the size of the current solution of a gate $g \in \mathcal{G}$. Starting with the next smaller size than s, all smaller sizes are tested in order of decreasing size. For each size,

all V_t levels are evaluated. Now let $\tilde{s}$ be a size that is smaller than s. If for all V_t levels of $\tilde{s}$, the refine costs are larger than the refine cost of the current solution, no size smaller than $\tilde{s}$ is evaluated. Afterwards, larger sizes are evaluated and pruned similarly once no V_t levels of a size could improve upon the initial solution.
We largely neglected electrical constraints in our theoretical discussions, which is not reasonable in practice. Solution candidates that increase the sum of load or slew violations of the vertices in the neighborhood graph of the currently optimized gate are rejected. A solution that improves either the sum of load or slew violations, but does not degrade the other, is accepted if the worst slack in the neighborhood graph is not degraded. The reason is that a large increase of violations can be very hard to fix afterwards, and designs have to be free of violations in the end.
For gates with positive slack we added a heuristic check not to choose a larger size or lower V_t level even if this improves the refine cost, as our optimization goals are timing closure and power minimization.

Translating Power Consumption into Delay

Local refine functions are weighted sums of power and signal delays, both measured in different units. In the RS algorithm, power and signal delays are divided by their budgets and we optimize weighted resource usages, therefore it is not necessary to translate between different units.
To allow a meaningful evaluation of the local refine functions in the LR algorithm, we translate power to delay by means of a technology-dependent translation factor. This factor is computed by simulating infinite and optimally buffered repeater chains as in Bartoschek [Bar14]. This gives a notion of the power consumption per unit distance induced by the repeaters in the chain, and the delay per unit distance. We perform this simulation for different repeater sizes and V_t level, and choose the average ratio of power per unit distance and delay per unit distance for the different repeater types as our translation factor.

9.2 Implementation of a Discrete Lagrangian Relaxation Algorithm

For the continuous relaxation of the gate sizing problem, the dual objective function is differentiable, and the gradient is the vector of delays. We conducted experiments in which we employed the extended timing graph, and updated the Lagrange multipliers with the vector of delays as described in Section 6.2. The multiplier projection was computed with the IBM ILOG CPLEX Optimizer. For each edge in the extended timing graph, the delay which induced the worst slack over this edge was used in the multiplier update.
For comparison, we implemented an alternative multiplier update and a heuristic projection step: The Lagrange multipliers are updated with the local edge slacks of the corresponding edges in the timing graph, which are defined as the worst slack

induced by the edge over all phases and transitions (cf. Section 2.5.1 and 2.5.3). We established in Section 6.2 that this is equivalent to updating the multipliers in the extended timing graph with the delay vector. As a heuristic multiplier projection we implemented the algorithm proposed by Tennakoon and Sechen [TS02], which is also described in Section 6.4.2. The running time is linear in the number of edges in the timing graph.
We observed in practice that the results obtained with the exact projection and delay multiplier update were not superior to the results obtained with the local edge slacks in the multiplier update and the heuristic projection, both with respect to timing constraints and power consumption. We thus use the latter variant in our implementation, as it is faster than an exact projection. For example, exact projection of an instance with approximately 2.3 million edges and 1.86 million vertices took 206 seconds on an Intel Xeon CPU with a clock frequency of 3.47 GHz, whereas the heuristic projection took only a few seconds.
Lagrange multipliers are initialized with a small percentage of the local edge slacks instead of a fixed value for each multiplier, such that the current timing criticalities are reflected from the start. As step size during the multiplier update we deploy the ratio 1/(current iteration).
We did not implement a stopping criterion for the algorithm but rather performed a predefined number of iterations as in the RS algorithm to enable a comparison of both algorithms over a fixed number of iterations.

9.3 Implementation of a Discrete Resource Sharing Algorithm

We implemented a resource sharing algorithm with path resources and a power resource as described in Section 8.6. The timing graph G is extended by a unique source s and a unique sink t, and the set of resources $\mathscr{R}$ corresponds to the set of s-t-paths plus a power resource.
Let $\mathscr{P}_{vw}$ denote the edge sets of all paths from v to w for $v, w \in V \cup \{s\} \cup \{t\}$, and we set $\mathscr{P} := \mathscr{P}_{st}$. The path resource budgets correspond to the clock cycle time D of the design, and the resource usage of a path $P \in \mathscr{P}$ is given as $\frac{delay_P(x)}{D}$. Here $delay_P(x)$ is the signal delay of P as incurred by sizes $x \in X_{disc}$, more formally $delay_P(x) = \sum_{e \in P} delay_e(x)$.
The only customer is the gate customer. Resource weights are updated exponentially based on their scaled resource consumption. More precisely, let $y^{(t)} \in \mathbb{R}^{|\mathscr{R}|}$ be the vector of resource usages computed in iteration t of the path resource sharing algorithm. We set

$$\omega_r^{(t+1)} := \omega_r^{(t)} \cdot \exp\left(\delta \cdot \frac{y_r^{(t)}}{\left|\left|y^{(t)}\right|\right|_\infty}\right), \tag{9.1}$$

for the weight of resource r in iteration $t+1$ for $\delta > 0$.

Path resources were proposed in Hähnle [Häh15] in the context of timing-driven global routing, and it was also shown in this work that the path resource weights can be decomposed into weights on the edges in the timing graph in linear time.
To see this, suppose we are in iteration t of the RS algorithm, and let ω_P be the current weight of path $P \in \mathscr{P}$. Let further $\widetilde{delay_P}$ and $\widetilde{delay_e}$ be the convex combinations of delays of path P and edge $e \in E$ computed in the previous iterations, respectively.
To simplify notation, we leave out the division by $\left|\left|y^{(t)}\right|\right|_\infty$ and D in the following formula. The weight ω_e of $e = (v, w) \in E$ can then be computed as follows:

$$\begin{aligned}
\omega_e &:= \sum_{P \in \mathscr{P}, P \ni e} \omega_P = \sum_{P \in \mathscr{P}, P \ni e} \exp\left(\delta \cdot \widetilde{delay_P}\right) \\
&= \sum_{P \in \mathscr{P}_{sv}} \sum_{Q \in \mathscr{P}_{wt}} \exp\left(\delta \cdot \sum_{f \in P \cup Q \cup \{e\}} \widetilde{delay_f}\right) \\
&= \exp(\delta \cdot \widetilde{delay_e}) \cdot \underbrace{\left(\sum_{P \in \mathscr{P}_{sv}} \exp\left(\delta \cdot \sum_{f \in P} \widetilde{delay_f}\right)\right)}_{\omega_{sv}} \\
&\quad \cdot \underbrace{\left(\sum_{Q \in \mathscr{P}_{wt}} \exp\left(\delta \cdot \sum_{f \in Q} \widetilde{delay_f}\right)\right)}_{\omega_{wt}}
\end{aligned} \tag{9.2}$$

All edge weights can be computed by traversing the timing graph once in topological and once in reverse topological order.
This implies that minimizing the weighted path resource usages of the gate customer is equivalent to minimizing the weighted sum of edge delays in the timing graph.
Hence minimizing the weighted resource usage of the gate customer is the power-delay tradeoff problem, and we can deploy BonnRefine as gate customer oracle that gets as input the edge weights and the power weight.
Path weights are only computed implicitly when computing the edge weights.
The edge delays used in the weight computation are the delays that correspond to the phase and transition attaining the worst slack over this edge. These are divided by the cycle time of the phase inducing the slack. We come back to these inaccuracies in Section 9.7. When analyzing our experimental results we make the simplifying assumption that we have only one clock phase and clock cycle time and that the required arrival times at timing endpoints are induced by this clock cycle time. Under this assumption the worst path resource usage is induced by a path attaining the worst slack.
Edge weights are initialized based on the resource usages of the starting solution. Note that starting with a value of 1 for each weight, as it is done in theory, would lead to large timing and power degradations until the weights are more balanced and

reflect the status of the design. Furthermore, more iterations of the algorithm would be needed. In an industrial design flow, instances have already been optimized by other algorithms before gate sizing, and it is not reasonable to "forget" about this information by starting with uniform weights.
Recall that we need to specify a power budget in order to determine the power resource usage of a solution. For experimental purposes, we specified a budget for each instance individually. This will be described in Section 9.5 and Section 9.6, respectively.

9.4 Testbed and Setup

All algorithms are implemented in C++ and the industrial timing engine IBM EinsTimer is used for all delay computations. Tests on microprocessor instances and the ISPD 2013 benchmarks were performed on Intel XEON machines with clock frequencies of 3.1 GHz and 2.9 GHz, respectively.
Table 9.1 shows our industrial testbed consisting of 8 microprocessor units.

Design	Technology	# Circuits	Cycle time (ps)	Edges	Pins
Unit1	22nm	50520	208	218507	170630
Unit2	22nm	54709	174	277141	259207
Unit3	14nm	66843	240	168093	148929
Unit4	14nm	74136	174	263197	228221
Unit5	14nm	169327	174	637107	501126
Unit6	14nm	219749	264	917523	726164
Unit7	22nm	474312	208	2745919	2166512
Unit8	22nm	542544	208	2222887	1790123

Table 9.1: Microprocessor units used in our experiments. Column two and three indicate the technology of the units and the number of circuits. The last three columns show the cycle time, and the number of edges and pins in the timing graph, respectively.

In the previous chapters we considered algorithms for gate sizing only, as latches (registers) are usually fixed along with the clock net routing in earlier design stages. On the ISPD 2013 benchmarks, latch sizes are fixed, but on the microprocessor designs we include non-fixed latches in our optimization. We will treat latch sizing again in Chapter 10.
The oracle algorithm is called once in each iteration of the LR and RS algorithms in parallel with 8 threads. Thereby the netlist is partitioned into logical regions with an algorithm provided by IBM, and threads iteratively process the regions. Thereby circuits that lie in the boundary of two regions, which means that they are connected with circuits in another region by an edge in the timing graph, are not optimized. For this reason the netlist is partitioned a second time after the first

processing of all regions. This is done in such a way that each former boundary circuit now lies within a single region. Afterwards, each circuit has been processed at least once. Within the regions, circuits are traversed in reverse topological order. We performed 25 iterations of each algorithm as the RS algorithm runs a specified number of iterations in theory. Placement locations of the circuits on the ISPD 2013 benchmarks benchmarks are unknown, and designs cannot be legalized. We also did not run a legalization algorithm on the microprocessor designs in order to evaluate the raw behavior of the LR and RS algorithm without considering effects that are caused by legalization.

9.4.1 Starting Solutions

Algorithms were run incrementally on optimized instances instead of starting with the smallest size and highest V_t level solution as it is usually done on the ISPD 2013 benchmarks. The microprocessor designs are placed and already timing optimized with tools for repeater insertion, layer assignment, sizing and V_t optimization etc. For the ISPD 2013 benchmarks, we generated a starting solution similar to Li et al. [Li+12a] and Flach et al. [Fla+14]: Starting with the smallest available sizes and highest V_t levels, all gates were traversed in topological order and the smallest solution that incurred no load violations was chosen for each gate.

9.4.2 Evaluation Metrics

Existing convergence guarantees of the LR algorithm refer to the solution computed in the last iteration of the algorithm, while for the RS algorithm estimations refer to the weighted average of the solutions computed in the course of the algorithm. This is not necessarily a feasible discrete solution. Given that, it is not clear which solution to analyze from our experiments because rounding can be arbitrarily bad (cf. Section 8.7). We can choose for example the solution that minimizes the maximum resource usage. Note that under our simplifying assumptions (cf. Section 9.3) the worst path resource usage is induced by a path that attains the worst slack WS of the design. However, the sum of negative endpoint slacks SNS and the sum of negative slacks of all subpaths SLS in the timing graph can be poor in this solution (see page 27 and 28 for a definition of these metrics). In our evaluation we consider the solutions returned after the last iteration of the algorithms. We will see that this usually is a good choice.

We shortly describe the metrics we employ to evaluate the solutions returned by our algorithms. These metrics are also depicted in our result tables (Table 9.3 - 9.6).

We measured the worst slack of the design WS, the sum of negative endpoint slacks SNS, and the sum of negative slacks of all subpaths SLS , which was also proposed as evaluation metric in Reimann et al. [RSR15]. We refer to the these metrics as *timing metrics*.

Timing metrics

In literature, SLS is often not considered. We use it as an additional measure for the following reason: In an industrial design flow, gate sizing is often applied at a stage where the design cannot become timing clean by sizing and V_t optimization only, and further optimizations like repeater insertion are necessary. Using edge weights in the oracle algorithm, regardless if they are Lagrange multipliers or resource weights, sometimes leads to delay improvements on timing critical paths, which do not appear in the SNS that is most commonly used as a measure for the timing quality of a design. Consider for example a gate with 2 inputs a and b, and suppose the edge from input a to the gate output pin c lies on a very critical path that cannot be significantly improved by gate sizing. If the edge from input b to c is less timing critical than the edge from a to c, a delay improvement of this edge will not change the worst slack at c. Thus improving the delay of the edge from b to c will only be visible in the SLS.
Additionally, a WS and SNS improvement is not preferable if it comes at the cost of a large SLS degradation that needs to be recovered again by later algorithms to achieve timing closure.
"P_{static}" denotes the static power consumption of all gates. Minimizing the static power consumption is the objective for the ISPD 2013 benchmarks. On the microprocessor instances, the relation between the total power consumption of the solutions returned by our algorithms was usually of the same magnitude as the relation between the static power consumptions except for one design. Therefore we omit the total power consumption in our result tables and point out the discrepancy in Section 9.5.
The column headed by "P_{usage}" shows the usage of the static power resource, i.e. the ratio of static power consumption and its budget. Due to the exponential number of paths in the timing graph, we do not evaluate the usage of all path resources. Instead we depict the worst usage, which is induced by a path that attains the worst slack under our simplifying assumptions, in column "WS_{usage}". Both metrics are also evaluated for the LR algorithm to provide a better comparison.
As performance guarantees for the RS algorithm refer to the convex combination of resource usages (and the convex combination of the solutions) computed in the course of the algorithm, the two columns headed with "$\varnothing$ Usages" show the convex combination of the power and worst path resource usage over all iterations of this algorithm, respectively.
Column "RT" denotes the running times of the algorithms in minutes. Most of it is spent for delay computations in the solution evaluation of the oracle, and only to a minor extent for updating the resource weights. It serves as an indicator for the number of solutions that have been evaluated in the course of the algorithm, and usually the number of gates that have been changed correlates with the running time.
However, updating the resource weights takes approximately three times as long as updating the Lagrange multipliers. This can be contributed to the fact that here the timing graph needs to be traversed twice in order to compute the new edge weights. Additionally, the worst path resource usage needs to be determined first

to compute $\left|\left|y^{(t)}\right|\right|_\infty$ for $y^{(t)} \in \mathbb{R}^{|\mathscr{R}|}$ in each iteration t of the RS algorithm, which is also needed in the weight computation (cf. equation (9.1)).
As the sum of load and slew violations were of the same magnitude for the LR and RS algorithm, and never degraded significantly except for two microprocessor designs, we omit these numbers in our tables and indicate the exceptions in Section 9.5.

9.4.3 Optimization Modes

Mode	Weight update rule	Local oracle objective	Power weight
Init	-	-	-
LR	Lagrangian relaxation	Sum of weighted delays	No
RS	Resource sharing	Sum of weighted delays	Yes
LRM	Multiplicative	Sum of weighted delays	No
LRH	Lagrangian relaxation	Sum of weighted delays and local SNS	No
RSH	Resource sharing	Sum of weighted delays and local SNS	Yes
APPR	Resource sharing	Weighted local SNS	Yes

Table 9.2: Optimization modes.

Table 9.2 shows our optimization modes that will be explained in the following. The modes are classified by the update rule for the edge weights and multipliers, the objective that is optimized locally in the oracle algorithm (the refine costs), and whether a power weight is considered.
Mode “Init” refers to the initial values before optimization. Modes ”LR“ and ”RS“ indicate the base LR and RS algorithm whose implementations we have described in Section 9.2 and Section 9.3, respectively.
Recall that our analysis of the heuristic modifications of the LR algorithm in practice led us to the multiplicative weights algorithm (cf. Section 7.2.4) and further to the new model as a min-max resource sharing problem. Naturally, the question arises how the LR algorithm with heuristic modifications performs compared to the base LR algorithm, and in particular to the RS algorithm. We divide these modifications into two classes: Firstly, changes proposed for the projected gradient method, and secondly changes to the oracle algorithm described in Section 9.1. As various heuristics exist and it is impossible to implement and test all the combinations, we implemented only a selected choice that we deem reasonable.
Among these are a multiplicative multiplier update proposed in Flach et al. [Fla+14] that was employed in the winning algorithm of the ISPD 2013 Discrete Gate Sizing Contest. This mode is indicated by ”LRM“ in our tables. Here the Lagrange multipliers are updated multiplicatively based on their local timing criticality (cf. Section

7.2.4). Although the weight update rule in the RS algorithm is also multiplicative, it is based on the resource usages and differs significantly from this heuristic rule. Similar update rules were proposed for example by Tennakoon and Sechen [TS02] and Livramento et al. [Liv+14].
It also is an interesting question how other heuristic oracles influence the performance of the algorithms. As mentioned before, our oracle algorithm is a heuristic and no approximation guarantees are known. Based on our observations in practice, we believe that better results can be obtained when the relevant timing metrics like WS, SNS and SLS are considered more directly in the oracle algorithm instead of letting the weights guide the optimization.
Modes "LRH" and "RSH" refer to the LR and RS algorithm with such a heuristic oracle algorithm, which we will shortly describe. Let $g \in \mathcal{G}$. We refer to the *local SNS* of g as the sum of negative slacks at the sibling and predecessor pins of the input pins of g and the successor pins of the data output pin of g (cf. Section 2.5.2). Based on our practical observation that the LR and RS algorithm tend to use larger sizes and lower V_t levels unnecessarily, we included both the negative slacks at predecessor and sibling pins in this metric to put more emphasis on signals entering g. We deploy as refine costs the sum of weighted delays in the neighborhood graph as before, but reject solutions that degrade the local SNS of the initial solution by more than a factor of $\Phi > 0$ similar to Flach et al. [Fla+14]. For the ISPD 2013 benchmarks, the value of Φ is updated after each iteration based on the criticality of the design: If the worst slack improves, Φ decreases. For the microprocessor designs the worst slack does not necessarily improve by sizing and V_t optimization. On designs with a worst slack that is strongly negative, Φ would then remain large and hardly any solution would be rejected due to a local SNS degradation. For this reason we set Φ to a fixed value of 1.05. This proved to be efficient in practice, but can possibly be improved by fine-tuning. We also prohibit degradations of the 1% most critical slacks of the design. This oracle algorithm is a variant of Algorithm 5.4.

Local SNS

Additionally, we evaluate an optimization mode called "APPR" with another heuristic oracle algorithm that optimizes the weighted local SNS for each gate in the oracle algorithm similar to Daboul [Dab15]. Here the slack at $v \in V$ is weighted with $\omega_{sv} \cdot \omega_{vt}$, and these pin weights can be computed simultaneously with the edge weights by equation (9.2). In each iteration, the refine costs of $g \in \mathcal{G}$ are the sum of the weighted negative slacks at the pins that are considered in the local SNS of g. The differences to minimizing the weighted sum of delays are as follows: Firstly, the delays of a few edges are not "captured", which is illustrated in Figure 9.1. The picture shows the neighborhood graph of gate g in the center. Edges whose delay contributes to the local SNS are indicated in green and blue, otherwise edges are colored red. The reason why the edge delays of some edges are not considered is that at predecessor pins and output pins of g only the criticalities and thus the delays of the most timing critical edges are propagated further. We conclude that the weighted sum of negative slacks at sibling and successor pins equals the sum of weighted delays when the currently sized gate and the predecessor gates are invert-

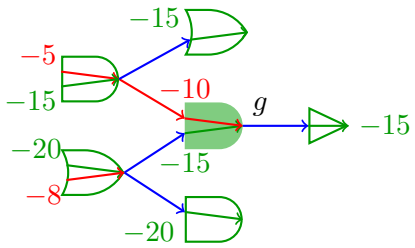


Figure 9.1: The weighted sum of negative slacks in the neighborhood of gate g approximates the weighted sum of delays.

ers, as these have only one traversing edge. Otherwise, it is a lower bound on the sum of weighted delays.

Secondly, we include the weighted negative slacks at the predecessor pins in the refine cost function to put more emphasis on signals entering g.

9.5 Results on Microprocessor Instances

For our microprocessor instances, we choose a power budget individually for each design. It was obtained by heuristic sizing and V_t optimization algorithms on our instances, which also improved the timing metrics of the initial solution (Init). Hence, these budgets are realistic targets for our RS algorithm.

It requires some tuning to find a good number of iterations and a corresponding value of δ in the update of the resource weights. We ran 25 iterations of the RS algorithm and compared different values for δ. It turned out that $\delta = 1$ provided the best results: For a larger value of 5, power consumption improved significantly at the cost of large timing degradations. This is probably caused by numerical issues due to very large values of the resource weights in the sense that all resource weights are very large and timing criticalities are overlooked. An implementation that is more aware of numerical issues might solve this problem.

Note that although the sum of weights in the last iterations can be bounded (see Hähnle [Häh15]), the weights grow exponentially with the resource usages, and can become rather large.

A smaller δ value of 0.2 improved SNS on some designs at the cost of higher power consumption, and led to inferior timing and power consumption on others. Running 100 iterations with $\delta = 0.25$ yielded almost the same results as 25 iterations with $\delta = 1$ both with respect to timing metrics and power consumption. This shows that the behavior of the RS algorithm is stable, because in theory the number of iterations depends linearly on $\frac{1}{\delta}$.

9.5.1 Without V_t Optimization

We first conducted experiments with fixed V_t levels, as including V_t optimization makes the problem in a way harder. This will become clear in Section 9.5.2.
Table 9.3 shows the results after 25 iterations of the LR and RS algorithm with fixed V_t levels.

		Timing metrics			**Power**	**Usages**		**∅ Usages**		
Chip	Mode	WS $[ps]$	SNS $[ns]$	SLS $[ns]$	P_{static} $[\mu W]$	P_{static}	WS	P_{static}	WS	RT [min]
Unit1	Init	-47.1	-30.7	-189.5	42.57	1.01	1.23			
	LR	-77.2	-52.3	-319.6	45.91	1.09	1.37			13.4
	RS	-54.1	-33.4	-193.6	44.74	1.06	1.26	1.10	1.29	14.7
Unit2	Init	-545.1	-879.3	-1490.4	437.06	1.04	1.78			
	LR	-545.1	-917.3	-1552.0	409.12	0.97	1.78			9.0
	RS	-545.1	-895.2	-1500.6	446.47	1.06	1.78	1.03	1.78	14.3
Unit3	Init	-310.5	-88.3	-297.5	19.00	1.14	1.49			
	LR	-335.1	-102.9	-339.0	20.79	1.25	1.52			20.4
	RS	-307.2	-92.4	-298.7	16.98	1.02	1.48	0.94	1.48	23.3
Unit4	Init	-292.7	-300.0	-1153.0	92.07	1.27	2.68			
	LR	-292.5	-298.3	-1160.2	89.82	1.24	2.68			22.0
	RS	-292.3	-291.8	-1107.1	85.03	1.17	2.68	1.18	2.69	25.4
Unit5	Init	-153.8	-216.4	-2750.9	125.63	1.02	1.88			
	LR	-151.2	-244.3	-3126.8	119.28	0.97	1.87			46.4
	RS	-151.2	-210.6	-2640.7	122.06	0.99	1.87	1.00	1.87	55.8
Unit6	Init	-196.3	-846.8	-6198.2	130.09	1.17	1.74			
	LR	-205.3	-864.8	-6717.8	125.79	1.13	1.78			75.0
	RS	-194.9	-749.7	-5634.9	128.00	1.15	1.74	1.18	1.75	89.9
Unit7	Init	-338.2	-1443.2	-10460.9	339.82	0.92	2.63			
	LR	-372.3	-1502.5	-10206.9	369.67	1.00	2.79			320.6
	RS	-353.3	-1225.8	-8267.6	403.19	1.09	2.70	1.07	2.69	348.8
Unit8	Init	-182.8	-445.3	-4341.3	416.37	0.93	1.88			
	LR	-185.8	-482.2	-4381.8	476.49	1.07	1.89			135.8
	RS	-186.0	-419.6	-3923.5	496.47	1.11	1.89	1.10	1.88	171.9

Table 9.3: 25 iterations of the LR and RS algorithms with fixed V_t levels on microprocessor designs.

Compared with the LR algorithm, the RS algorithm exhibits better SNS and SLS on all designs. On most of the designs, this comes with a higher power consumption, for example Unit7.
Compared to the starting solution, the LR algorithm degrades either SNS, SLS or both metrics on all designs, and WS degrades on all units except Unit2, Unit4 and Unit5.
The RS algorithm degrades WS on Unit7 and Unit8, but at the same time improves SNS and SLS. On the three smallest designs, it degrades either SNS or SLS, or both metrics.
On the smallest design Unit1, both algorithms degraded not only the timing metrics, but also power consumption. However, degradations are more significant for the LR algorithm.
An exception where the LR algorithm performs better than the RS algorithm is Unit2, where both algorithms degrade timing, but the LR algorithm causes a significantly smaller power consumption and even improves over the initial value.

The running time differences between the LR and RS algorithm are mostly due to the slower weight computation for the RS algorithm.
We believe that the timing and power degradations and the improvements of small magnitude can be contributed to the fact that increasing the size of a gate often improves the sum of weighted delays, but actually degrades the slacks in the neighborhood graph of the gate.
Consider for example a gate g with more than one edge traversing the gate. Increasing the size of g has the largest effect on the delay of the edges traversing g. In comparison, delay changes of the other edges in the neighborhood graph are relatively small. If the edges traversing g are timing critical and g only has a few timing critical sibling gates, it takes several iterations until the weights of the edges in the neighborhood have increased sufficiently such that a larger size does not improve the sum of weighted delays in the neighborhood graph. This effect is enhanced if additionally g has a large fanout with critical gates.
It can be reduced by choosing a larger value for δ, such that differences in timing criticality are more pronounced from the beginning, and running more iterations.
To verify our hypothesis, we ran 100 iterations of the RS algorithm on two smaller designs. On Unit3 and Unit1, this further improved the worst slack by a few pico seconds, and static power consumption by approximately 10%.
SNS and SLS further degraded on both designs. This is in accordance with the objective of the RS algorithm to minimize the maximum resource usage, and shows that the weights of the most critical path resources increased sufficiently. The usage of less critical path resources thereby degraded.
Running 100 iterations of the LR algorithm did not improve the results significantly.

9.5.2 Including V_t Optimization

We also ran the LR and RS algorithm with an oracle that optimizes both sizes and V_t level. The results are presented in row 2 and 3 of each design in Table 9.4.
Including V_t optimization makes the problem in a way harder because a lower V_t level does not have such a large impact on entering signals, and thus will improve the weighted sum of delays in most cases. A power weight is necessary to prevent setting too many gates to the lowest V_t level, which is also illustrated by our results.
While the power consumption of the LR algorithm without V_t optimization was on some designs smaller than after the RS algorithm, the situation is reverted when V_t optimization is included, for example on Unit5, Unit6 and Unit8.
For the LR algorithm, power is larger on all designs except Unit2 compared to optimization with fixed V_t levels. This does not come with a timing improvement, in fact timing degraded significantly on several designs, for example on Unit6 and Unit7. This can be contributed to the fact that there is no power weight in the LR algorithm, and that minimizing the sum of weighted delays locally degraded the timing metrics.
Similar considerations as in the previous section apply when comparing the LR and RS algorithm with respect to the timing metrics. The RS algorithm exhibits better

SNS and SLS than the LR algorithm on all designs except Unit3, and improves over the initial solution on all designs except the smallest ones. The better timing of the RS algorithm comes with a higher power consumption on Unit2 and Unit7. Power consumption is also larger than for the RS algorithm with fixed V_t levels on Unit3, Unit5, Unit7 and Unit8. Compared to this algorithm, timing metrics have slightly degraded on most designs. This shows that often a lower V_t level or larger size was chosen, although the timing metrics degraded. While the average worst slack usage is almost the same with and without V_t optimization, the average power usage is significantly larger with V_t optimization on the larger designs, for example by 9% on Unit5 and 8% on Unit8, respectively. More iterations are necessary to reduce power again. On Unit2, Unit3 and Unit4, the average power usage improved.
Another explanation for the SLS and SNS degradation caused by the LR algorithm and the RS algorithm on the smaller designs is that on real-life designs, the worst slacks cannot always be improved by gate sizing and V_t optimization. In that case the weights of edges on these paths will continue to grow and can dominate the weights of other, less critical edges. If only the sum of weighted delays is considered in the oracle algorithm, slacks on less critical paths can decrease.
We mentioned in Section 9.4.2 that the relation between the total power consumption and the static power consumption of the solutions were of the same magnitude except for one design. On Unit3, static power consumption is larger for mode LR, but total power consumption is the same as for mode RS, which indicates that more gates were set to a lower V_t level, but to smaller sizes than compared to the RS algorithm. The same holds for mode LRM, whose results we will now describe.

9.5.3 Multiplicative Multiplier Update

Row 4 in Table 9.4 depicts the mode LRM with a multiplicative multiplier update in the Lagrangian relaxation algorithm. Interestingly, this update rule degraded timing metrics compared to the base LR approach, but improved static power on all designs. For example on Unit6, static power was improved by almost 10%. This leads to the conclusion that the Lagrange multipliers are smaller and weighted delays are not as dominating in the local refine functions as before. Furthermore, the Lagrange multipliers reflect the timing criticality of the design not as good as in the base LR algorithm.
Multiplicative update rules were often used in combination with a heuristic oracle. Therefore we also conducted experiments of this update rule in combination with the heuristic oracle that considers the local SNS of the gates and which is also used in mode LRH and RSH. Using this oracle algorithm improved both power and timing metrics significantly. In comparison with mode LRH, which is the base LR algorithm combined with this oracle, LRM revealed less power on all units except Unit3, and worse SNS and SLS. Hence we omitted these results in Table 9.4.
Given that this multiplicative update rule was successfully applied to the ISPD 2013 benchmark suite by Flach et al. [Fla+14], our results indicate that more careful tuning of the algorithmic components is necessary to obtain similar power im-

provements and dispose of the timing degradations. This assumption is supported by the fact that on the ISPD 2013 benchmarks, this mode performs better than on the microprocessor designs compared to mode LR. A variant of this update rule was also successfully applied to industrial microprocessor designs by Reimann et al. [RSR15]. Here the timing criticalities were measured in relation to the criticality of the starting solution instead of the slack target of the design. This is more adapted to the fact that real-life instances are usually not timing clean after sizing and V_t optimization.

9.5.4 Heuristic Oracles

Finally, we consider the optimization modes LRH, RSH and APPR with the heuristic oracle algorithms that consider the local SNS or weighted local SNS.
Avoiding large degradations of the local SNS in the oracle algorithms as in mode LRH and RSH introduced more stability to the LR and RS algorithms. The results are shown in line 5 and 6 of Table 9.4, respectively.
We observe that modes LRH and RSH improve over modes LR and RS, respectively, regarding timing metrics and power consumption. Timing metrics are better for mode RSH compared to LRH on all designs except Unit3. Better timing metrics come at the cost of a higher power consumption on Unit2 and Unit7.
Line 7 shows the results for the optimization mode APPR, which finds the best tradeoff between power consumption and timing metrics on these designs. It improves over mode RSH with respect to power consumption except for Unit3, Unit4 and Unit6.
We believe this is due to the fact that the objective function in the oracle approximates the weighted sum of delays as in the RS algorithm, but is more aware of the actual timing criticalities and considers the timing metrics more directly.

9.5.5 Running Times

Running times are of approximately the same magnitude for all modes except APPR, which is significantly faster especially on the larger designs. This indicates that a smaller number of solutions has been evaluated in the course of the algorithm. Modes LRH and RSH are only slightly faster than modes LR and RS. The contribution of computing the resource weights to the running time of mode RS is approximately 10%, and thus larger for modes RSH and APPR.

9.5.6 Electrical Violations

We already mentioned that the sum of load capacitance and slew violations were of approximately the same magnitude for all designs, and improved the values of the initial solution except for the sum of slew violations on Unit1 and Unit8. We observed in practice that restricting delay and slew calculations to a bounded number of logic levels when optimizing a gate in the oracle algorithm (cf. Section

		Timing metrics			Power	Usages		∅ Usages		
Chip	Mode	WS [ps]	SNS [ns]	SLS [ns]	P_{static} [μW]	P_{static}	WS	P_{static}	WS	RT [min]
Unit1	Init	-47.1	-30.7	-189.5	42.57	1.01	1.23			
	LR	-81.7	-52.0	-298.1	53.71	1.28	1.39			31.8
	RS	-51.2	-39.5	-231.4	43.92	1.04	1.25	1.14	1.30	36.2
	LRM	-79.1	-67.3	-490.0	52.25	1.24	1.38			31.5
	LRH	-47.1	-29.8	-170.4	45.35	1.08	1.23			28.2
	RSH	-47.1	-26.6	-150.5	44.28	1.05	1.23	1.06	1.23	33.4
	APPR	-47.1	-23.7	-134.5	42.93	1.02	1.23	1.01	1.23	19.8
Unit2	Init	-545.1	-879.3	-1490.4	437.06	1.04	1.78			
	LR	-545.1	-916.0	-1570.8	384.58	0.91	1.78			27.7
	RS	-545.1	-895.1	-1506.3	442.45	1.05	1.78	1.00	1.78	35.1
	LRM	-545.1	-921.9	-1595.9	371.24	0.88	1.78			28.3
	LRH	-545.1	-912.7	-1561.7	381.45	0.90	1.78			27.2
	RSH	-545.1	-892.6	-1502.9	444.89	1.05	1.78	1.00	1.78	34.5
	APPR	-545.1	-880.2	-1490.0	431.11	1.02	1.78	1.02	1.78	27.1
Unit3	Init	-310.5	-88.3	-297.5	19.00	1.14	1.49			
	LR	-332.7	-94.7	-312.6	23.54	1.41	1.52			69.4
	RS	-312.1	-96.5	-308.3	17.43	1.05	1.49	0.91	1.49	72.8
	LRM	-323.8	-102.7	-339.3	22.55	1.35	1.51			71.0
	LRH	-295.1	-87.4	-282.7	18.98	1.14	1.46			66.0
	RSH	-295.2	-91.6	-293.0	16.82	1.01	1.46	0.86	1.46	73.5
	APPR	-295.1	-90.6	-297.9	17.84	1.07	1.46	1.07	1.46	53.0
Unit4	Init	-292.7	-300.0	-1153.0	92.07	1.27	2.68			
	LR	-302.1	-354.5	-1532.1	91.70	1.26	2.74			75.2
	RS	-292.2	-291.0	-1098.9	80.16	1.10	2.68	1.11	2.69	78.1
	LRM	-301.7	-446.3	-2011.9	83.16	1.14	2.74			73.2
	LRH	-292.8	-322.6	-1327.3	86.62	1.19	2.68			72.0
	RSH	-292.8	-287.6	-1086.2	81.03	1.11	2.68	1.09	2.68	77.6
	APPR	-292.8	-294.8	-1115.0	84.78	1.17	2.68	1.19	2.68	58.1
Unit5	Init	-153.8	-216.4	-2750.9	125.63	1.02	1.88			
	LR	-150.4	-233.7	-3135.4	136.72	1.11	1.86			149.9
	RS	-150.4	-215.2	-2670.2	132.28	1.08	1.86	1.09	1.86	158.6
	LRM	-150.4	-268.6	-3900.6	129.47	1.05	1.86			151.2
	LRH	-150.4	-223.4	-2966.0	127.51	1.04	1.86			148.3
	RSH	-150.4	-211.2	-2624.7	124.22	1.01	1.86	1.02	1.86	154.1
	APPR	-150.4	-209.9	-2612.7	118.69	0.97	1.86	0.97	1.86	120.8
Unit6	Init	-196.3	-846.8	-6198.2	130.09	1.17	1.74			
	LR	-222.3	-1082.2	-7576.7	136.94	1.23	1.84			184.4
	RS	-194.3	-737.6	-5711.2	124.61	1.12	1.74	1.19	1.74	193.7
	LRM	-246.4	-1287.5	-10315.1	125.33	1.13	1.93			180.9
	LRH	-194.8	-1005.5	-6951.6	132.95	1.20	1.74			182.3
	RSH	-194.7	-719.3	-5682.7	126.66	1.14	1.74	1.18	1.74	193.1
	APPR	-194.7	-742.4	-5987.8	126.65	1.14	1.74	1.15	1.74	141.1
Unit7	Init	-338.2	-1443.2	-10460.9	339.82	0.92	2.63			
	LR	-392.8	-1594.9	-10512.9	397.97	1.07	2.89			879.6
	RS	-353.3	-1233.9	-8342.9	412.45	1.11	2.70	1.10	2.69	820.3
	LRM	-392.8	-1786.6	-15040.4	383.16	1.03	2.89			805.7
	LRH	-325.0	-1509.2	-9939.4	379.82	1.02	2.56			810.0
	RSH	-325.0	-1214.9	-8319.0	392.65	1.06	2.56	1.05	2.57	791.7
	APPR	-325.0	-1198.6	-8275.5	355.23	0.96	2.56	0.95	2.56	531.0
Unit8	Init	-182.8	-445.3	-4341.3	416.37	0.93	1.88			
	LR	-186.1	-483.7	-4515.1	536.30	1.20	1.89			334.3
	RS	-187.6	-426.3	-4010.5	516.96	1.16	1.90	1.18	1.89	354.7
	LRM	-186.1	-573.1	-6269.2	519.55	1.16	1.89			338.5
	LRH	-181.2	-426.7	-3987.7	486.96	1.09	1.87			329.4
	RSH	-181.2	-385.8	-3682.4	479.06	1.07	1.87	1.05	1.87	333.0
	APPR	-181.2	-390.7	-3580.8	427.12	0.96	1.87	0.94	1.87	204.8

Table 9.4: 25 iterations of all optimization modes with sizing and V_t optimization on microprocessor designs.

9.1) sometimes leads to slew violations in the logic levels that were not included. This can happen for example when the slew is just slightly below the limit at a pin within a considered logic level, and above the limit in the next level that is not considered.
Recall that the oracle algorithms reject solution candidates that increase the sum of load or slew violations in all optimization modes. Hence small differences in the sum of violations can rather be contributed to random effects and the limited delay and slew propagation when sizing a gate, and not to differences between the optimization modes.

9.5.7 Convergence Plots

Figure 9.2, Figure 9.3 and Figure 9.4 show the WS, static power consumption and SNS, respectively, in each iteration of our optimization modes for Unit6. The number of iterations is plotted on the x-axis. Depicted on the y-axis are the WS, static power consumption and SNS, respectively. For the resource sharing based optimization modes we also plotted the bare values to get a notion on the convergence behavior. The resource sharing based algorithms (mode RS, RSH and APPR) exhibit better and more predictable convergence behavior in all three metrics. In mode LR and LRM the changes between WS and SNS are more significant between iterations, and both metrics jump back and forth. The worst slack is relatively stable in mode LRH, as we prohibit degradations of the most critical slacks locally.

9.6 Results on the ISPD 2013 Benchmarks

We ran the optimization modes as depicted in Table 9.2 also on the ISPD 2013 benchmarks. We chose as static power budget for each benchmark 105% of the best static power consumption reported to be achieved without timing or electrical violations by Flach et al. [Fla+14].
However, in our first experiments the RS algorithm increased power consumption dramatically in the first iterations, and power consumption not nearly recovered after 25 iterations. The reason was that the power consumption of our starting solution with mostly small sizes and high V_t levels (cf. Section 9.4.1) was significantly smaller than the power budget, which led to a very small power weight that was hardly considered in the first iterations.
Therefore we generated new starting solutions by running a global gate sizing algorithm (Held [Hel08]) and a global V_t optimization algorithm (Wittke [Wit13] and Daboul [Dab15]) to further optimize the design. Afterwards, the power consumption was larger than our budget.
Table 9.5 and Table 9.6 show our results after 25 iterations of all our optimization modes on the ISPD 2013 benchmarks for the fast designs and the slow designs, respectively. The fast designs and the corresponding slow designs have the same netlist, but the clock cycle time is smaller for the fast designs. Note that also the cycle times of the netlists differ.

For δ we chose a value of 0.1 instead of 1 as paths in the timing graph tend to be longer than on the microprocessor instances. A larger value of δ led to numerical issues similar to the ones described in the previous section.

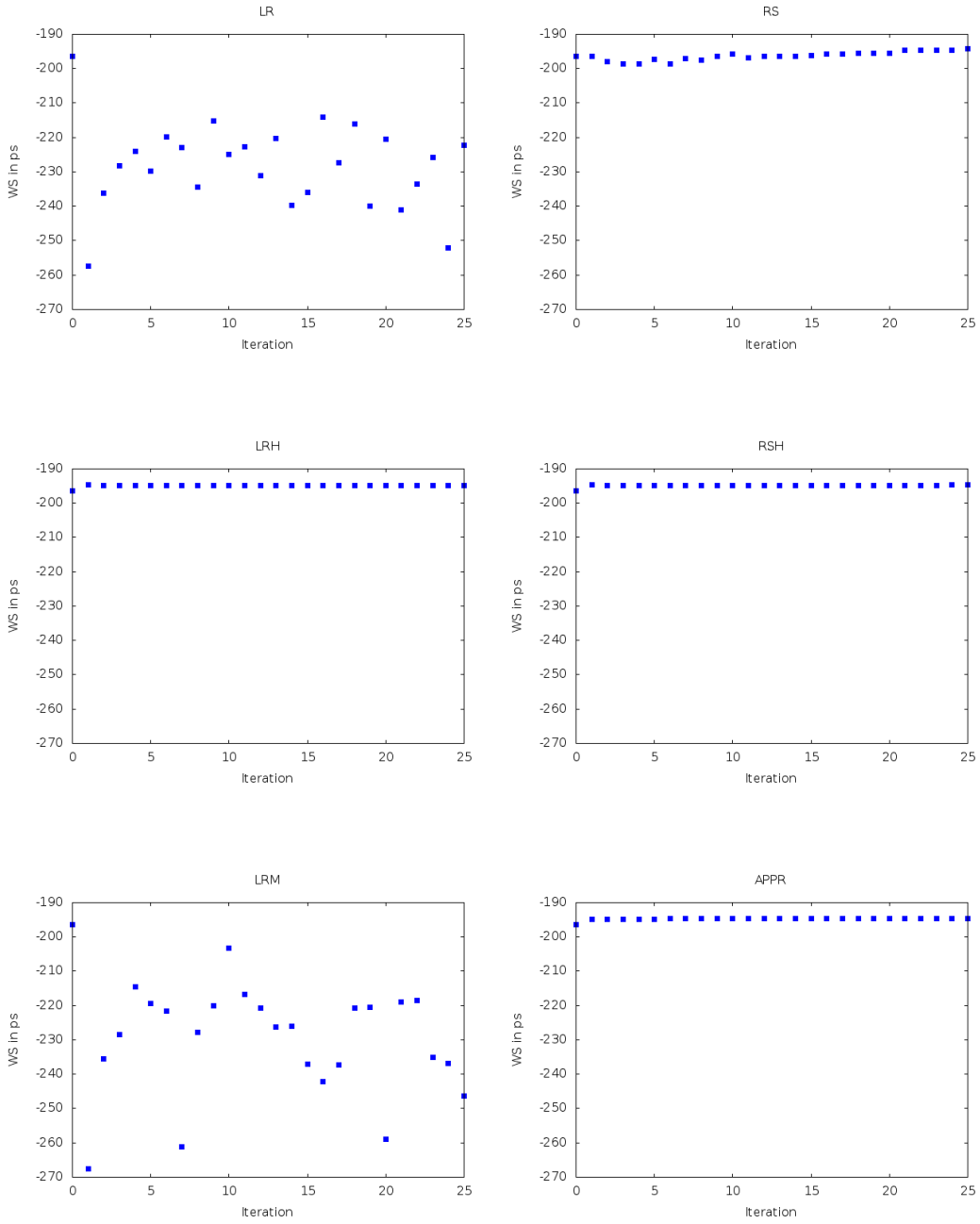


Figure 9.2: Convergence of WS for Unit6 and all optimization modes.

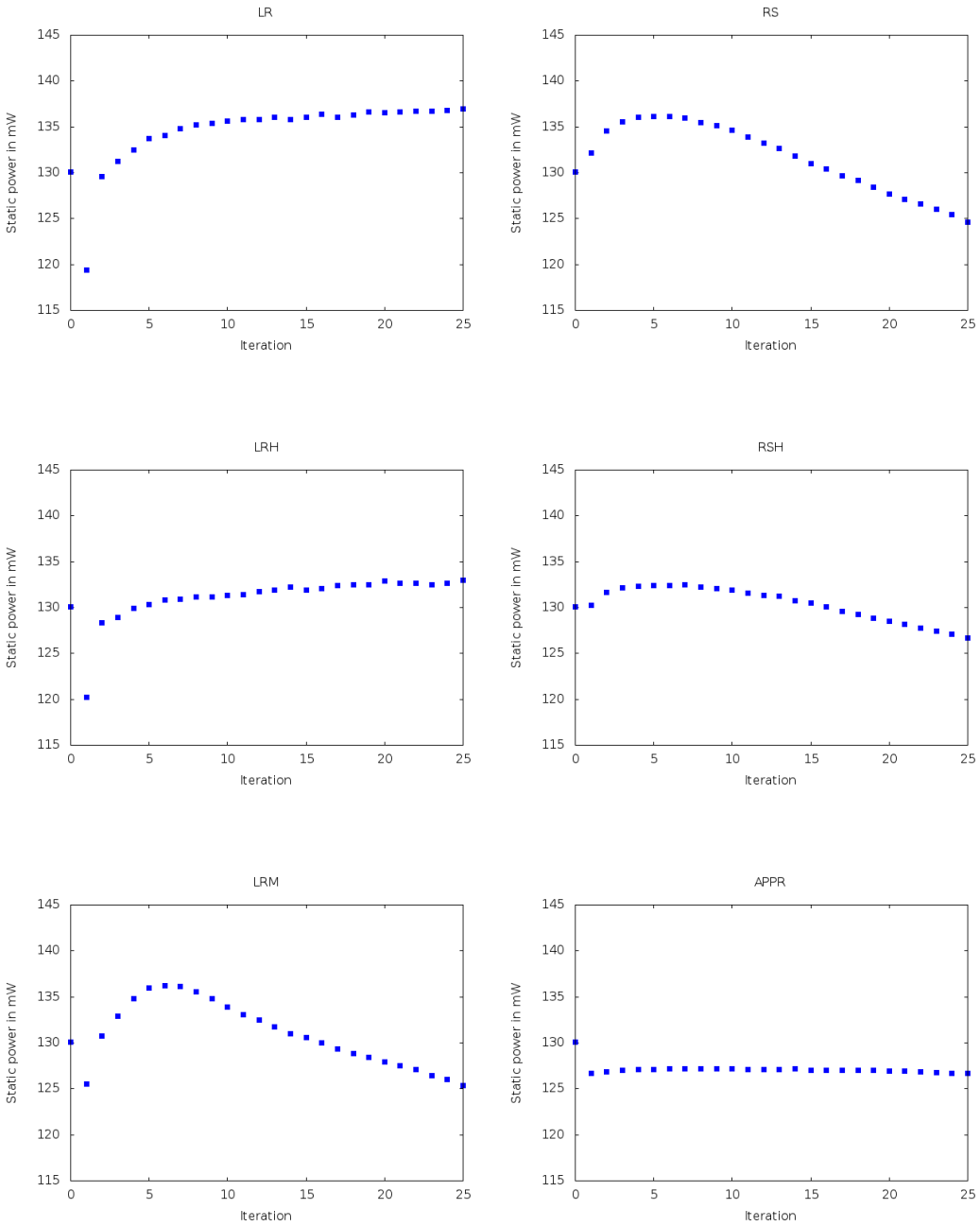


Figure 9.3: Convergence of static power consumption for Unit6 and all optimization modes.

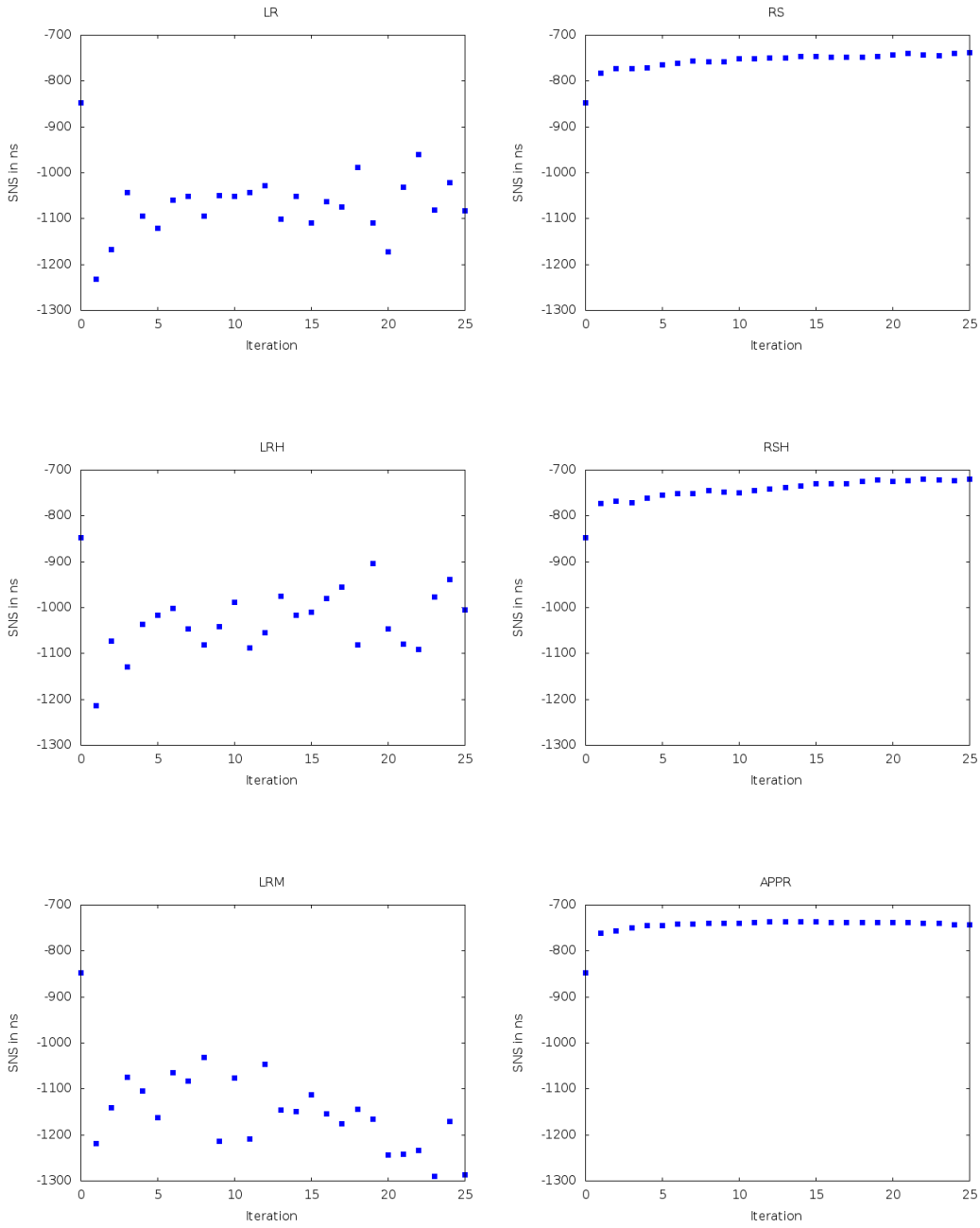


Figure 9.4: Convergence of SNS for Unit6 and all optimization modes.

On these benchmarks, the differences we observed between mode LR and mode RS for the microprocessor designs are more pronounced: Not only does the RS algorithm exhibit the better timing metrics, also power consumption is smaller on all designs except for the smallest netlist usb_phy in fast and slow mode. Designs are also close to timing closure. In both modes, designs were free of load capacitance and slew violations

The multiplicative multiplier update in mode LRM improved power significantly compared to mode LR, in some cases at the cost of timing degradations. With the oracle algorithm used in mode LRH and RSH, results were further enhanced. Nonetheless, the RS algorithm improved over mode LRM on almost all designs in all metrics.

On these benchmarks, the improvements achieved by using heuristic sizing oracles that consider the local SNS (mode LRH and RSH) are enormous: Mode LRH improved all metrics on all designs, and power consumption was sometimes decreased by more than 50% for example on cordic and matrix_mult with fast clock period compared to mode LR.

Mode RSH improved power consumption on all designs, and timing metrics on 10 out of 16 designs compared to mode RS. On the remaining 6 designs, the power improvement caused a timing degradation, which is most pronounced on the fast and slow versions of netlist des_perf and edit_dist.

On these two netlists, the timing degradations are caused by numerical issues similar to the ones we have observed on the microprocessor designs for a larger value of δ.

It is interesting that mode APPR does not improve over the RS based modes as significantly as for the microprocessor designs. This can be contributed to the fact that the ISPD 2013 benchmarks can become timing clean by sizing and V_t optimization, such that it is not as important to optimize the timing metrics as directly as for designs for which no timing feasible solution for sizing and V_t optimization exists.

Similar to the microprocessor designs, running times are of approximately the same magnitude for all modes except APPR, which is significantly faster. Modes LRH and RSH are slightly faster than mode LR and RSH, and on some larger designs LRH is faster than RS, for example on netcard with slow clock period.

The contribution of computing the resource weights to the running time of mode RS is approximately 10%, and thus larger for modes RSH and APPR. The contribution is slightly larger than for the microprocessor designs.

The high power consumption of the solutions returned at the end of the algorithms certainly is an issue. We chose a rather small and ambitious power budget for each design, hence this cannot be attributed to a small power weight. The problem is that minimizing the sum of delays locally in the sizing oracle often leads to unnecessarily large sizes and lower V_t levels, as we have discussed already in Section 9.5.1. This observation is reinforced here by the fact that the heuristic modes LRH, RSH and APPR return solutions with significantly smaller power on almost all designs.

It also implies that several iterations are often necessary until gates with a large size and low V_t level in the starting solution get a smaller size or higher V_t level.

From that point of view, it is preferable to use small sizes and high V_t levels as starting solutions. However, this requires a better handling of the power weight, which will be discussed in more detail Section 9.7.
Figure 9.5, Figure 9.6 and Figure 9.7 show the convergence of our algorithms for design matrix_mult with fast clock period with respect to WS, static power consumption, and SNS. As for the microprocessor designs, the RS based modes (RS, RSH and APPR) reveal the better and more stable convergence behavior.

9.7 Conclusion

Our experiments illustrate that the RS algorithm exhibits better and more stable convergence behavior than the LR algorithm, and improves over the LR algorithm with respect to the timing metrics, and often also with respect to power consumption. Nonetheless we see a gap between theory and practice, and there is room left for improvement. This is illustrated by the fact that heuristic sizing oracles considering the timing metrics more directly (mode LRH, RSH and APPR) improved both algorithms with respect to power and timing metrics. On the microprocessor designs, mode APPR that directly targets the timing metrics achieved the best tradeoff between power consumption and timing optimization. Recall that no approximation guarantees are known for our oracle algorithms. Optimizing the local refine function for each gate as it is done in the local refine algorithm for the continuous relaxation (cf. Section 5.1) appears to be the best way to minimize the weighted sum of edge delays and power in the timing graph. In practice, this does not achieve the best results, and we discussed some possible reasons in Section 9.5 and Section 9.6. Careful tuning of the algorithmic components and parameters can further improve the results.
One issue that arises in the RS algorithm is the power budget. Usually, there is no budget available or known for a design, and binary search is expensive. Additionally, we observed on the ISPD 2013 benchmarks that a budget which is large compared to the initial solution can cause an extensive power increase during the first iterations because the power resource weight is relatively small. The power increase then needs to be recovered. Similar to global routing in VLSI design, a heuristic solution to this problem could be to specify a budget that is iteratively adapted during the algorithm (Müller et al. [MRV11]). Alternatively, the power weight can be updated after each local sizing step in the oracle algorithm such that it grows faster, or be computed based on the timing criticality of the design as proposed by Tennakoon and Sechen [TS08]. We deem the latter approach most promising based on practical observations, as it directly sets power and timing metrics into relation.
Lower bounds on resource usages that introduce more stability into the algorithm can help avoiding degradations of resources with low usage due to small weights (see also Müller et al. [MRV11]). For path resources, it is not clear how to specify these lower bounds, as path weights are only computed implicitly. This would be clear for edge delay resources in combination with arrival time customers as proposed in

		Timing metrics			Power	Usages		∅ Usages		
Design	Run	WS [ps]	SNS [ns]	SLS [ns]	P_{static} [μW]	P_{static}	WS	P_{static}	WS	RT [min]
usb_phy	LR	-75.8	-0.7	-1.5	19.2	11.85	1.25			1.2
usb_phy	RS	-314.0	-5.8	-29.1	9.7	5.99	2.05	6.91	1.58	1.5
usb_phy	LRM	-94.5	-0.9	-2.1	25.8	15.92	1.32			1.3
usb_phy	LRH	0.0	0.0	0.0	1.7	1.04	1.00			1.1
usb_phy	RSH	0.1	0.0	0.0	1.7	1.04	1.00	1.05	1.00	1.1
usb_phy	APPR	0.1	0.0	0.0	1.7	1.05	1.00	1.05	1.00	0.9
pci_bridge32	LR	-373.2	-134.2	-422.1	578.1	6.25	1.50			23.5
pci_bridge32	RS	-96.2	-2.2	-3.6	251.4	2.72	1.13	2.68	1.08	24.9
pci_bridge32	LRM	-258.8	-158.7	-423.8	541.8	5.85	1.35			24.1
pci_bridge32	LRH	-12.4	-0.5	-1.1	150.9	1.63	1.02			22.2
pci_bridge32	RSH	0.0	0.0	0.0	123.0	1.31	1.00	1.38	1.00	23.7
pci_bridge32	APPR	0.0	0.0	0.0	122.3	1.32	1.00	1.36	1.00	16.4
fft	LR	-868.9	-190.5	-3219.1	947.0	4.42	1.62			32.5
fft	RS	-68.4	-0.8	-4.0	828.3	3.87	1.05	3.75	1.04	37.2
fft	LRM	-731.2	-199.9	-3879.1	792.1	3.70	1.52			32.8
fft	LRH	0.0	0.0	0.0	330.9	1.54	1.00			33.5
fft	RSH	0.0	0.0	0.0	351.0	1.64	1.00	1.67	1.01	35.6
fft	APPR	0.0	0.0	0.0	355.2	1.66	1.00	1.68	1.01	30.8
cordic	LR	-1572.9	-544.9	-9537.6	3117.7	1.78	1.60			34.0
cordic	RS	-112.2	-7.9	-69.4	2257.3	1.29	1.04	1.26	1.04	33.5
cordic	LRM	-1388.2	-377.0	-7445.2	2940.4	1.68	1.53			33.7
cordic	LRH	-28.9	-1.7	-14.5	1267.5	0.73	1.01			29.4
cordic	RSH	-38.4	-0.6	-9.8	1312.2	0.75	1.01	0.77	1.02	32.6
cordic	APPR	-91.2	-3.8	-56.9	1368.5	0.78	1.03	0.75	1.03	24.7
des_perf	LR	-377.8	-119.1	-1190.5	6845.0	8.69	1.33			86.1
des_perf	RS	-84.7	-3.1	-6.2	2445.6	3.11	1.07	3.55	1.06	86.7
des_perf	LRM	-272.9	-124.2	-1157.7	6267.1	7.96	1.24			86.8
des_perf	LRH	-315.6	-56.1	-384.6	6223.9	7.90	1.28			83.3
des_perf	RSH*	-384.6	-363.4	-7976.4	418.7	0.53	1.34	1.88	1.15	86.3
des_perf	APPR	-7.9	-0.3	-0.5	1328.4	1.69	1.01	2.20	1.03	59.2
edit_dist	LR	-566.2	-138.1	-647.1	3823.2	6.36	1.19			102.5
edit_dist	RS	-172.0	-61.0	-179.0	2773.6	4.61	1.06	4.32	1.05	110.3
edit_dist	LRM	-478.2	-142.7	-693.6	3212.8	5.34	1.16			106.9
edit_dist	LRH	-153.0	-65.3	-202.0	2302.1	3.83	1.05			98.2
edit_dist	RSH*	-1726.5	-2161.6	-8934.3	1478.8	2.46	1.58	3.17	1.15	109.3
edit_dist	APPR	-176.1	-107.4	-376.6	2297.2	3.82	1.06	3.62	1.06	82.0
matrix_mult	LR	-650.1	-45.2	-6565.5	7047.9	3.30	1.30			117.9
matrix_mult	RS	-58.4	-3.1	-56.0	5042.4	2.36	1.03	2.44	1.03	126.2
matrix_mult	LRM	-847.9	-63.1	-7752.1	6748.9	3.16	1.39			117.6
matrix_mult	LRH	-39.6	-1.2	-30.0	3232.8	1.51	1.02			108.1
matrix_mult	RSH	-54.0	-0.6	-12.9	2790.0	1.31	1.02	1.47	1.02	120.1
matrix_mult	APPR	-24.8	-0.6	-8.2	3312.0	1.55	1.01	1.60	1.03	96.1
netcard	LR	-142.3	-24.7	-35.2	7067.6	1.31	1.07			753.4
netcard	RS	-29.0	-0.4	-0.8	6110.3	1.13	1.01	1.13	1.02	806.6
netcard	LRM	-51.2	-3.3	-6.2	7270.5	1.35	1.03			751.8
netcard	LRH	0.0	0.0	0.0	5371.6	0.99	1.00			735.7
netcard	RSH	0.0	0.0	0.0	5395.4	1.00	1.00	1.01	1.01	851.4
netcard	APPR	-9.1	-0.1	-0.4	5416.6	1.00	1.00	1.01	1.02	569.4

Table 9.5: 25 iterations of all optimization modes with sizing and V_t optimization on the ISPD 2013 benchmarks with faster clock period. Modes marked with a star indicate a possible numerical overflow.

		Timing metrics			Power	Usages		∅ Usages		
Design	Run	WS [ps]	SNS [ns]	SLS [ns]	P_{static} [μW]	P_{static}	WS	P_{static}	WS	RT [min]
usb_phy	LR	-41.6	-0.1	-1.3	4.2	3.72	1.09			1.1
usb_phy	RS	-189.0	-1.4	-6.4	8.7	7.70	1.42	5.52	1.27	1.2
usb_phy	LRM	-43.9	-0.1	-0.1	2.7	2.39	1.10			1.0
usb_phy	LRH	0.7	0.0	0.0	1.1	0.97	1.00			1.0
usb_phy	RSH	0.4	0.0	0.0	1.1	0.97	1.00	0.97	1.00	1.0
usb_phy	APPR	1.6	0.0	0.0	1.1	0.97	1.00	0.97	1.00	0.8
pci_bridge32	LR	-199.8	-12.3	-632.7	274.3	4.57	1.20			22.5
pci_bridge32	RS	-6.2	-0.0	-0.1	104.5	1.74	1.01	1.67	1.03	23.5
pci_bridge32	LRM	-139.3	-51.2	-117.2	212.4	3.54	1.14			22.8
pci_bridge32	LRH	0.1	0.0	0.0	61.1	1.02	1.00			21.1
pci_bridge32	RSH	0.2	0.0	0.0	62.0	1.03	1.00	1.04	1.00	23.0
pci_bridge32	APPR	0.0	0.0	0.0	60.0	1.00	1.00	1.00	1.00	15.6
fft	LR	-532.3	-31.5	-498.9	540.0	5.89	1.30			31.2
fft	RS	-46.5	-0.4	-1.0	297.7	3.25	1.03	2.98	1.03	34.3
fft	LRM	-517.8	-35.2	-604.3	524.6	5.73	1.29			31.2
fft	LRH	0.1	0.0	0.0	104.3	1.14	1.10			31.3
fft	RSH	0.0	0.0	0.0	106.2	1.16	1.00	1.18	1.00	33.2
fft	APPR	0.0	0.0	0.0	106.5	1.16	1.00	1.17	1.01	28.7
cordic	LR	-1422.1	-279.2	-4083.9	2343.0	7.21	1.47			32.5
cordic	RS	-200.6	-5.9	-114.8	1209.7	3.73	1.07	3.37	1.04	33.6
cordic	LRM	-987.8	-178.7	-2773.6	2167.2	6.68	1.33			32.5
cordic	LRH	-102.6	-3.8	-61.7	723.5	2.23	1.04			29.1
cordic	RSH	-123.6	-3.2	-57.4	762.0	2.35	1.04	2.06	1.04	31.4
cordic	APPR	-108.5	-7.9	-10.6	761.3	2.35	1.04	1.87	1.04	24.3
des_perf	LR	-201.2	-34.2	-142.1	3980.0	11.18	1.15			80.7
des_perf	RS	-54.3	-2.5	-4.4	1407.1	3.95	1.04	4.45	1.04	80.9
des_perf	LRM	-243.6	-97.3	-667.0	2934.2	8.25	1.19			83.4
des_perf	LRH	-182.6	-30.2	-117.9	2249.4	6.32	1.14			69.5
des_perf	RSH*	-314.6	-360.1	-8812.4	566.5	1.59	1.24	2.57	1.08	75.8
des_perf	APPR	-19.7	-0.4	-0.6	664.9	1.87	1.02	2.34	1.03	55.3
edit_dist	LR	-372.9	-178.8	-599.5	1698.0	3.77	1.10			94.9
edit_dist	RS	-189.6	-56.2	-133.7	1378.1	3.06	1.05	3.08	1.05	102.6
edit_dist	LRM	-397.4	-98.0	-303.6	1599.2	3.55	1.11			96.4
edit_dist	LRH	-237.6	-93.6	-266.5	1252.1	2.78	1.07			92.4
edit_dist	RSH*	-501.8	-547.0	-5236.6	1316.6	2.92	1.14	2.44	1.05	100.9
edit_dist	APPR	-230.4	-76.9	-208.5	1223.6	2.71	1.06	2.62	1.05	78.0
matrix_mult	LR	-353.0	-12.8	-1156.6	2347.4	4.83	1.13			103.7
matrix_mult	RS	-83.2	-2.5	-26.2	1711.9	3.52	1.03	3.41	1.04	111.6
matrix_mult	LRM	-381.6	-16.2	-1500.0	1892.8	3.89	1.13			105.4
matrix_mult	LRH	-102.3	-2.1	-54.5	1190.3	2.45	1.04			96.1
matrix_mult	RSH	-94.7	-1.6	-23.6	1030.8	2.12	1.03	2.10	1.03	108.7
matrix_mult	APPR	-108.1	-2.8	-58.5	1135.8	2.34	1.04	2.06	1.04	87.4
netcard	LR	-53.5	-1.8	-1.8	5525.5	1.03	1.02			721.8
netcard	RS	-39.8	-0.3	-0.6	5404.6	1.01	1.02	1.00	1.01	832.1
netcard	LRM	-42.8	-1.4	-1.8	5601.6	1.04	1.02			738.3
netcard	LRH	0.0	0.0	0.0	5212.8	0.97	1.00			740.1
netcard	RSH	0.0	0.0	0.0	5216.2	0.97	1.00	0.97	1.00	839.2
netcard	APPR	0.1	0.0	0.0	5213.2	0.97	1.00	0.97	1.00	571.6

Table 9.6: 25 iterations of all optimization modes with sizing and V_t optimization on the ISPD 2013 benchmarks with slower clock period. Modes marked with a star indicate a possible numerical overflow.

Section 8.2 - 8.5.
Another reason to implement the model with edge delay resources is that the number of path resources is exponential, and edge weights can be dominated by the number of paths passing through it. This issue can be avoided with arrival time customers.
Also a larger value of δ can address this problem, as it makes the differences between timing criticalities more pronounced. In our experiments we encountered numerical problems with larger values of δ on both testbeds, which need to be addressed. Resource weights became relatively large in later iterations such that the differences between timing criticalities were not really "seen" anymore. This requires careful tuning of the number of iterations and the value of δ.
An advantage of path resources certainly is that the weights can be transferred to pin weights, which is exploited in mode APPR. On the microprocessor designs, this mode finds the best tradeoff between power consumption and timing metrics on most designs.
On the ISPD 2013 benchmarks, the situation is not that clear, as on several designs RSH improves over APPR.
To improve accuracy of the timing computations, copies of the timing graph can be maintained to model different phases. Also the signal transitions (rise, fall) can be taken into account. However, it is not clear if this would improve results significantly.
As was also observed by Reimann et al. [RSR15], algorithms for sizing and V_t optimization need to address different issues when applied to real-life instances that cannot necessarily become timing clean instead of the ISPD 2013 benchmarks. For one thing, sizing tools need to be able to run incrementally. In an industrial design flow, other optimizations were performed in advance, and it makes no sense to "forget" about their results by resetting sizes and V_t levels to an arbitrary solution beforehand. In that case, it would take several iterations until the weights are more balanced and reflect the actual timing criticalities. From this point of view, the RS algorithm also improves over the LR algorithm because the initial weights we computed better reflect the status of the design. This is illustrated in Figure 9.2 and Figure 9.4 where worst slack and SNS degrade significantly in the first iteration on Unit6 for the LR algorithm, whereas in the RS algorithm WS is relatively stable and SNS improves. The situation is similar for the ISPD 2013 benchmarks (Figure 9.5 and Figure 9.7).
Another challenge is that on real-life instances, the critical paths cannot always be improved. This can lead to timing degradations of less critical paths, because the critical path weights increase faster and can dominate other paths.
In our discussion of the results, we usually referred to the solution returned in the last iteration of all algorithms, although for the RS algorithm convergence guarantees refer to the average of the solutions computed in each iteration. It is reasonable to store intermediate solutions, and return the "best" solution over all iterations, although this term is ambiguous in this context: The solution minimizing the maximum resource usage can incur a poor SNS or SLS.

For a practical application, we propose a stopping criterion that sets power and slack improvements into relation. This can also save running time, which is quite large in our experiments with 25 iterations despite our multi-threaded implementations. A stopping criterion would also improve the results of the LR algorithm.
An advantage of our implementation is that BonnRefine is already used successfully in a design flow, and multi-threading reveals high speedups in its general setting. For the RS algorithm, speedups are likely to be the same. Running time can also be improved by restricting the number of solutions to be evaluated in each oracle call, for example by only considering gates for which edge weights in the neighborhood graph have changed by more than a certain threshold since the last iteration. Also the solution candidates that are evaluated for each gate can be further restricted.
We conclude that our first evaluation of the resource sharing approach is promising as the weights better reflect the status of the design, and the algorithm improves over Lagrangian relaxation regarding results and convergence behavior. For a practical application, further improvements are necessary.

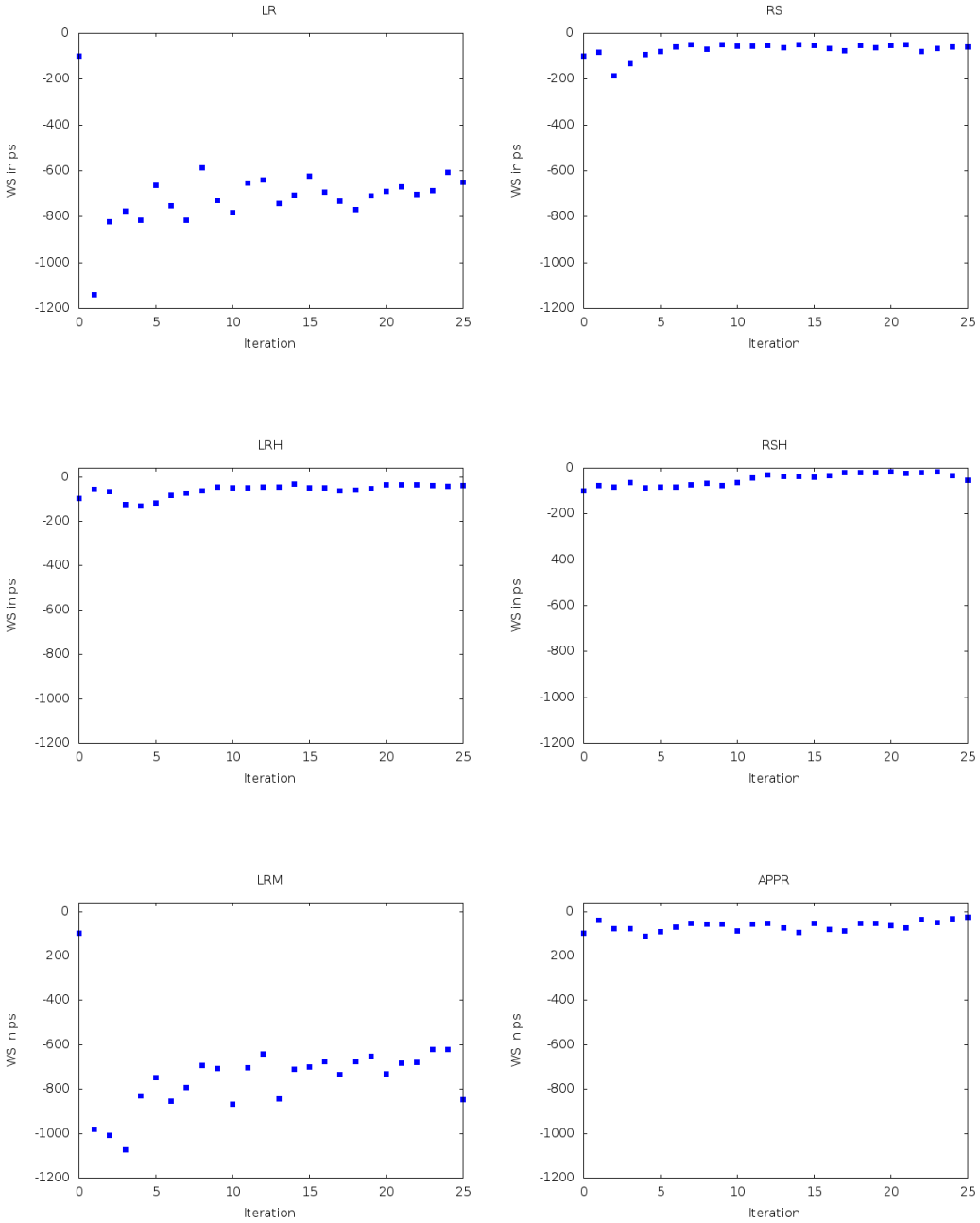


Figure 9.5: Convergence of WS for design matrix_mult with fast clock period and all optimization modes.

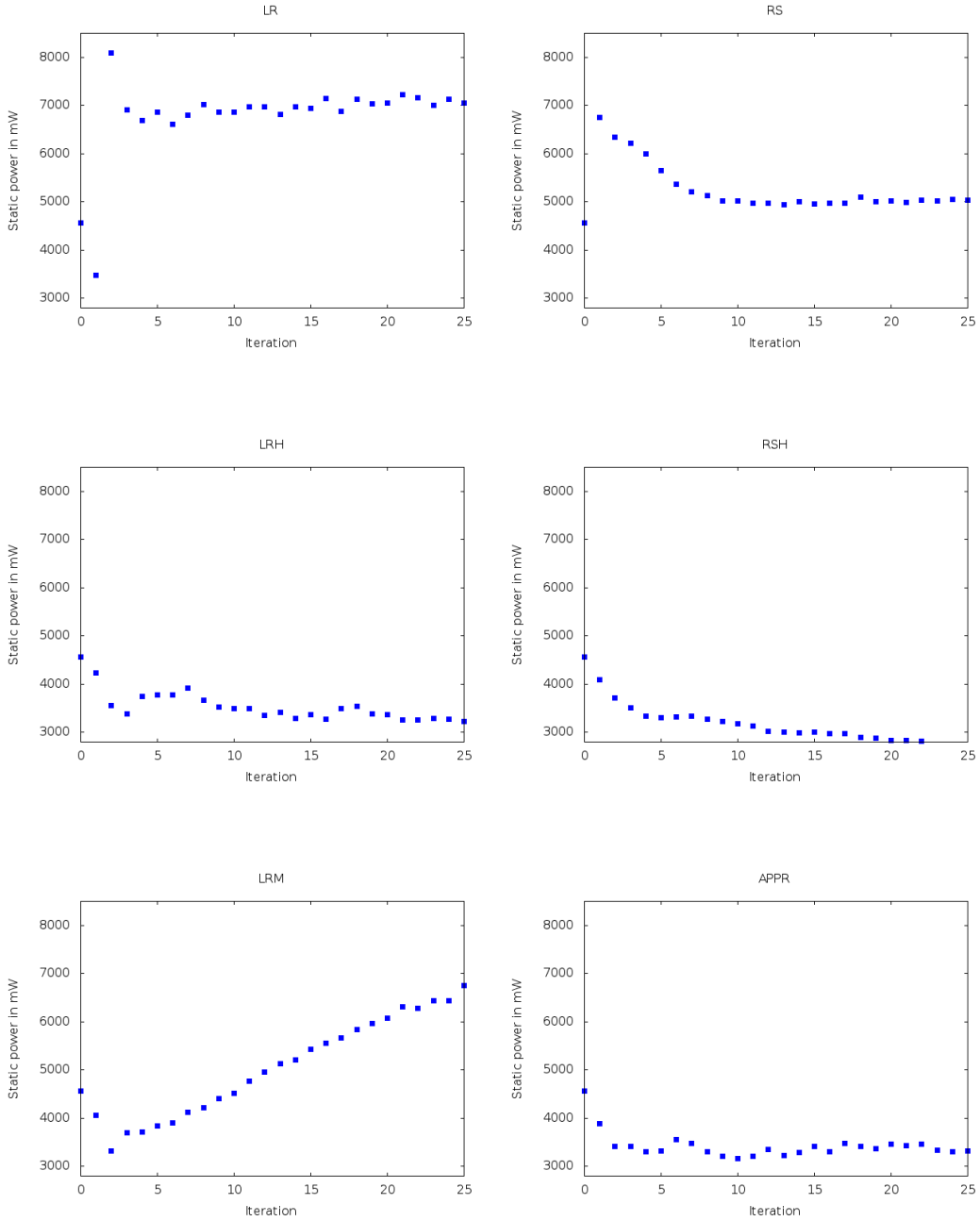


Figure 9.6: Convergence of static power consumption for design matrix_mult with fast clock period and all optimization modes.

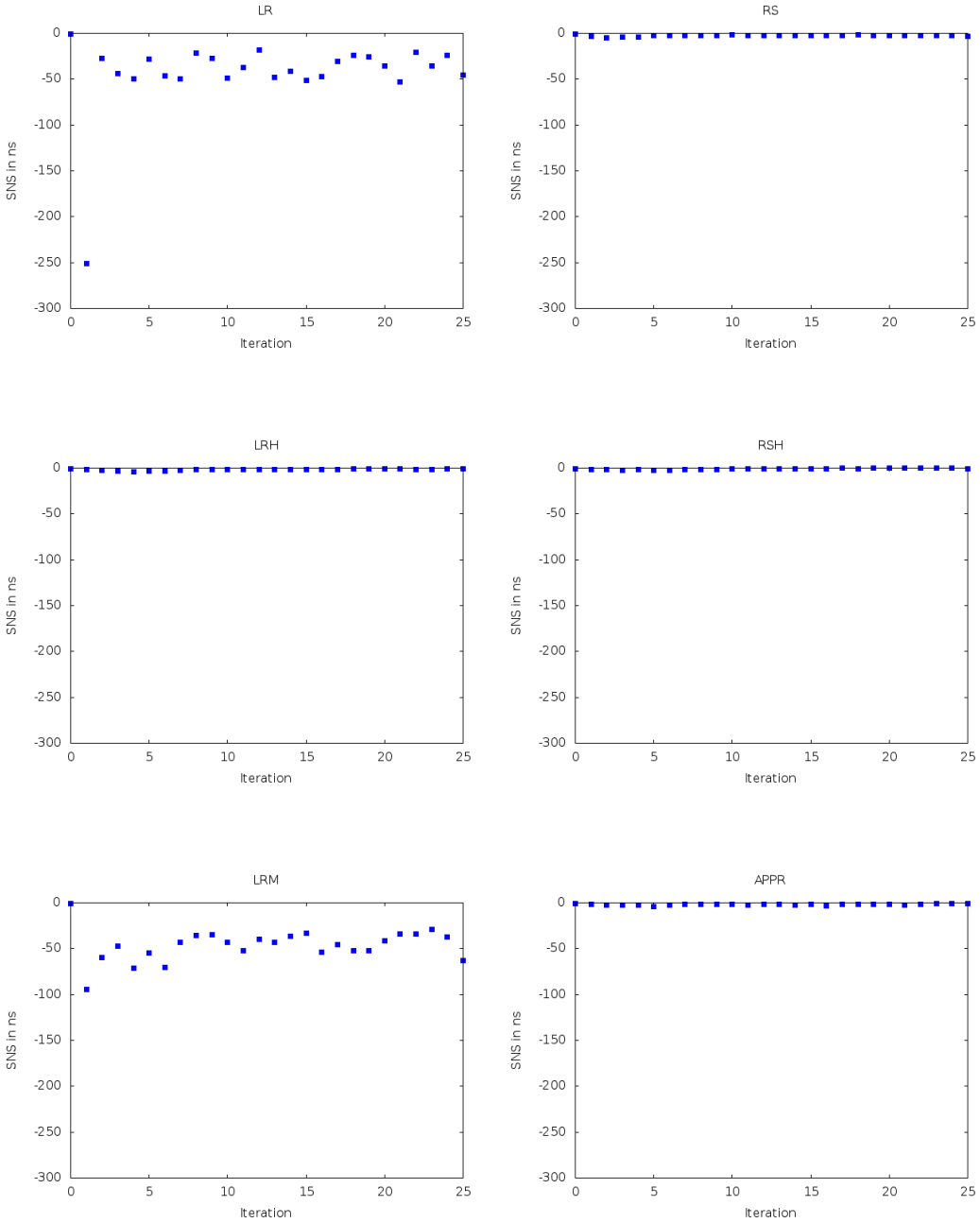


Figure 9.7: Convergence of SNS for design matrix_mult with fast clock period and all optimization modes.

10 Post-Routing Latch Optimization for Timing Closure

In the clock network design phase, timing constraints are often considered only indirectly during latch placement and sizing [Alp+07; Tre+04]. Afterwards, their placement and sizes remain mostly unchanged even if the criticalities of data signals change, as redesigning the clock network and clock routing is costly.
We present an algorithm for timing-driven optimization of latches that maintains the clock footprint and routing and can therefore be applied late in the design flow. The algorithm permutes latch positions and sizes in so-called *latch clusters* and finds an optimal solution under mild assumptions. Figure 10.1 shows an example of a latch cluster, where the latches are arranged around a clock buffer in a structured fashion.
We start with a motivation and related work, and provide a formal problem formulation in Section 10.2. It will become clear in Section 10.3 why a simple swap heuristic will not work in general. Then, in Section 10.4, we give a detailed description of our algorithm that is based on binary search and bipartite matchings which are fast in theory and practice. Extensions to more placement or sizing choices are given in Section 10.5, followed by implementation details in Section 10.6. Our experimental results demonstrate how the algorithm improves slacks on industrial microprocessors by up to 7.8% of cycle time in Section 10.7.
The results in this chapter are joint work with Stephan Held [HS14].

10.1 Motivation and Related Work

Clock network design for high performance microprocessors is one of the most challenging problems in VLSI design. The clock network distributes the clock signals that open and close the registers once per cycle, and is often realized by a clock tree or a clock grid with latches and flip-flops at the bottom level stage (cf. Chapter 2). In the following we use the term latch for both flip-flops and transparent latches. Local clock buffers (LCBs) dispense the clock signal. To bound clock skew and power consumption, latches are often clustered and placed next to a common local clock buffer (LCB) in a structured fashion (Chan et al. [Cha+03], Cho et al. [Cho+13], Papa et al. [Pap+11], Ward et al. [War+13]). Figure 10.1 shows an example of a latch cluster with an LCB in the center and latches arranged tightly around it in circuit columns. The clock signal is distributed by the (red) clock net with a fishbone structure having a wide horizontal backbone for minimum latency and skew.

LCB

Latch cluster

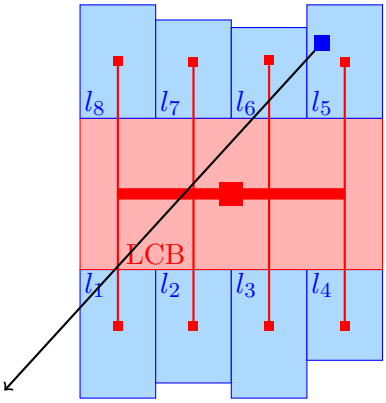


Figure 10.1: A local clock buffer (LCB) and net (red) with latches (blue). The black line indicates the worst slack data path starting at l_5 leading into the lower left.

Previous approaches for latch clustering and placement are mostly intended for global optimization and consider timing constraints indirectly using net weights or barriers, or with simplifying linear delay models.
The main goal of the structured latch placement in Chan et al. [Cha+03], Cho et al. [Cho+13], Papa et al. [Pap+11], Ward et al. [War+13] is to minimize clock power and skew, while trying to keep the changes to an initial global placement small.
There is a broad range of literature for timing-driven detailed placement that is intended for or can be applied to latch placement. In Papa et al. [Pap+08] and Luo et al. [Luo+08] timing-driven placement refinements based on linear delay models are proposed and particularly applied to latch placement. In addition to so-called activity based signal net weights, latches are clustered in Cheon et al. [Che+05] to reduce the power consumption of clock networks and the whole chip. Power reduction is also the main focus in the clustering and placement of pulsed latches in Chuang et al. [Chu+11a; Chu+11b].
Once the clock placement and routing is determined, data paths can be optimized given the precise knowledge of the clock signals Alpert et al. [Alp+07], Li et al. [Li+12b], Trevillyan et al. [Tre+04]. Thereby the latch clusters remain unchanged. However, during this process the initial timing criticalities on which the latch placement was based may change, and the given latch placement and sizes might not be favorable anymore.
Figure 10.1 shows an example of a latch cluster that was similarly observed in practice. The critical data path starts at the upper right latch l_5 aiming to a sink in the lower left (outside the figure). With this knowledge, the latch should rather be placed at the bottom left corner of the cluster. In addition, the latch l_5 would benefit from a higher drive strength, e.g. the size of l_1.
Our algorithm fills a gap in restructuring latch clusters late in the design flow. The algorithm permutes latch positions and sizes within a cluster to maximize

the worst slack. Subsequently, secondary objectives such as the sum of negative endpoint slacks or wire length are minimized. It preserves the clock footprint and routing, such that the top-level clock network does not need to be redesigned, and can therefore be applied late in the design flow after clock network design and analysis. It works for arbitrary circuit and wire delay models.

Remark 10.1 (Optimizing clock domains) A chip may have several clock networks with different frequencies. We say that memory elements which are fed by the same clock signal belong to the same clock domain. Our algorithm is intended for but not limited to optimizing local clusters. It could also be used to optimize a group of clusters or a whole clock domain at once, allowing swaps between different clusters.

10.2 Problem Formulation

An instance of the latch re-assignment problem consists of a set of $n \in \mathbb{N}$ latches $L = \{l_1, \ldots, l_n\}$ with logically equivalent implementation in a common clock domain. Each latch must be assigned to a placement location and a discrete size. We call such a pair a slot and denote by $S = \{s_1, \ldots, s_n\}$ the set of available slots. $L = \{l_1, \ldots, l_n\}$ $S = \{s_1, \ldots, s_n\}$
We will assume that the set of available slots are the initial latch positions, and the size of a slot is the size of the latch initially placed at this position and therefore fixed. Extensions of this formulation are considered in Section 10.5.
We are looking for an assignment of latches to slots. The assignment can be expressed by binary variables $x_{ij} \in \{0, 1\}$ that equal one if and only if l_i is assigned to slot s_j $(1 \leq i, j \leq n)$, and the following assignment constraints: $x_{ij} \in \{0, 1\}$

$$\begin{array}{rcll} \sum_i x_{ij} & = & 1 & \forall\ 1 \leq j \leq n, \\ \sum_j x_{ij} & = & 1 & \forall\ 1 \leq i \leq n, \\ x_{ij} & \in & \{0,1\} & \forall\ 1 \leq i, j \leq n. \end{array} \tag{10.1}$$

Note that the sizes of the slots are fixed, such that assigning latch l_i to slot s_j also implies that latch l_i gets the size of the latch initially placed in s_j.
We then denote by WS_{ij} the minimum of the worst (late) slack at the data pins of l_i when assigned to s_j and a slack target. We assume the slack target to be zero, thus $\mathrm{WS}_{ij} \leq 0$. Furthermore, by WL_{ij} we denote the total wire length of the nets attached to the data pins of l_i when assigned to s_j. WS_{ij} WL_{ij}
As before, we consider late mode slacks only, because late mode timing constraints are usually harder to meet than early mode constraints. In practice our method does hardly affect the final early mode padding.

10.2.1 Assumptions

We make the simplifying assumption that the values WS_{ij} can be computed for each latch l_i $(1 \leq i \leq n)$ independently from the other latches and their assignments. In reality, data paths may of course start in one latch in L and end in one or

several other latches in L. However, assuming that critical paths cover rather long distances it is unlikely that two latches in proximity affect their critical data paths mutually. Thus this assumption is reasonable when L contains latches from a local region, in particular from a structured cluster.
Similarly, one data net could be the input to several latches in L and therefore the wire lengths of the data nets attached to latches in L may not be considered independently. But this scenario will rarely occur, as latches in proximity storing the same bit could be merged.
Note that without these assumptions, the problem would become significantly harder. E.g. the problem of finding an assignment minimizing the wire length with two-terminal nets only would be as hard as the quadratic (unweighted) assignment problem, which is even hard to approximate, see Queyranne [Que86].

10.2.2 Primary Objective

The primary objective of our latch optimization problem is to maximize the worst slack of the cluster L, i.e.

$$\max_x\{\min_{i,j}\{\mathrm{WS}_{ij} \ : \ x_{ij} = 1\} \ : \ x \text{ satisfies (10.1)}\} \tag{10.2}$$

10.2.3 Secondary Objectives

Improving the worst slack of an instance involves changing positions and sizes of two or more latches and could degrade the total wire length or sum of negative slacks. Thus, we consider as a secondary objective the minimization of the sum of slacks at the cluster

SCS

$$SCS := \sum_{i,j} -\mathrm{WS}_{ij} x_{ij} \tag{10.3}$$

or the wire length

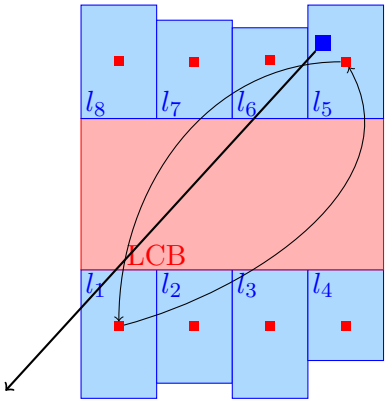
$$\sum_{i,j} \mathrm{WL}_{ij} x_{ij}. \tag{10.4}$$

Another objective could be the overall disruption:

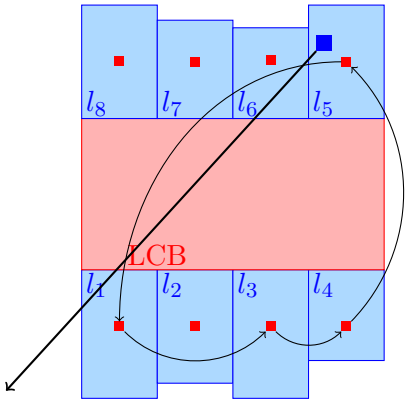
$$||x - \tilde{x}||_1 = \sum_{i,j} (1 - \tilde{x}_{ij}) x_{ij}, \tag{10.5}$$

where the values $\tilde{x}_{ij}$ refer to the initial assignment.
We assume the slack target to be zero, therefore $\mathrm{WS}_{ij} \leq 0$ holds for all i, j and (10.3) is non-negative. Note that a clock routing that is feasible for one assignment is feasible for any other assignment, because the set of slots and in particular their sizes remain the same, and each slot is used by a latch. Thus, clock nets are not considered in the objectives.

10.3 Greedy Algorithm



(a) When latches l_1 and l_5 are swapped, the new slack of l_1 might become even worse than the initial slack of l_5.



(b) A cyclic sequence might improve the worst slack at l_5 and could bound a possible slack degradation of l_1, l_3, and l_4 due to the short moves.

Figure 10.2: A cyclic sequence is necessary to improve the worst slack data path starting at l_5 and leading to the lower left without degrading the slacks at other latches.

A simple greedy method to optimize a latch cluster would be to iteratively swap pairs of latches if this improves the slacks. However, such a pair may not exist. E.g. when latches l_1 and l_5 are swapped in the example in Figure 10.2(a) to improve the worst slack data path starting at l_5 and leading to the lower left, the new slack of l_1 when moved to the upper right corner might become even worse than the initial slack of l_5. As the initial size of l_4 is smaller than the size of l_5, swapping these two latches might degrade the slack at l_5 even though it is moved in the direction of its predecessor on the critical data path. Therefore no improvement might be found by iteratively swapping pairs of latches only. Instead a cyclic sequence as in Figure 10.2(b) could bound a possible slack degradation of l_1, l_3, and l_4 due to the short moves. To overcome this limitation we propose a globally optimum algorithm in the next section.

10.4 Global Assignment Algorithm

Our algorithm for slack optimal latch assignments works in two phases. In the first phase the worst slack is maximized whereas in the second phase the secondary objective is optimized while preserving the best possible worst slack found in the first phase.
Algorithms to solve the arising matching respectively minimum-cost flow problems

providing the given running times can be found in text books such as by Korte and Vygen [KV12], which also contain references to the original papers.

10.4.1 Worst Slack Maximization

The idea of the worst slack maximization is to perform a binary search on the worst slack values WS_{ij} $(1 \leq i, j \leq n)$ in the cluster, as finally the worst slack of the cluster will be equal to one of these values.
The achievability of a guessed slack value can be checked with a maximum cardinality matching algorithm in a bipartite graph as shown in Figure 10.3(a): The vertex set V consists of vertices for all latches (blue vertices) and all slots (black vertices). The edge set E contains a directed (blue) edge from each latch l_i to each slot s_j $(1 \leq i, j \leq n)$.

Theorem 10.2 *There is an assignment x fulfilling (10.1) with worst slack at least $\Theta \in \mathbb{R}$ if and only if there is a matching of cardinality n between the set of latches L and the set of slots S that uses only edges (l_i, s_j) with $\mathrm{WS}_{ij} \geq \Theta$ $(1 \leq i, j \leq n)$.*

The proof of this theorem is straight-forward. A maximum cardinality matching in a bipartite graph with $2n$ vertices and at most n^2 edges can be found in $O(n^3)$ using a maximum-flow algorithm. The overall running time for maximizing the worst slack is given as follows:

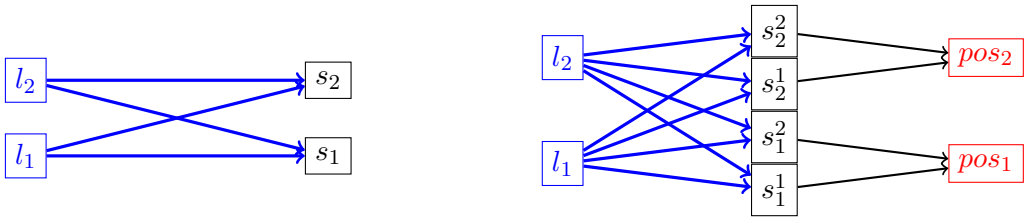
Theorem 10.3 *Given all values WS_{ij} $(1 \leq i, j \leq n)$, the maximum worst slack can be computed in $O(n^3 \log n)$ time.*

Proof. We carry out a binary search on the set $\{\mathrm{WS}_{ij} : 1 \leq i, j \leq n\}$ of cardinality $O(n^2)$. Sorting the set takes $O(n^2 \log n)$ time. The binary search tests $O(\log n)$ possible values for the best achievable worst slack. As each test requires $O(n^3)$ time the overall running time bound is $O(n^3 \log n)$. □

For implementation details, see Section 10.6.

10.4.2 Minimizing the Secondary Objective

When minimizing a secondary objective, we want to preserve the previously maximized worst slack $\Theta^\star$. Thus we maintain the final assignment instance from the worst slack maximization (Section 10.4.1) that contains only edges (l_i, s_j) with $\mathrm{WS}_{ij} \geq \Theta^\star$, implying that all latches are assigned to slots where their estimated slack is larger than or equal to $\Theta^\star$, such that the worst cluster slack is $\Theta^\star$.
An assignment that minimizes the linear objectives (10.3), (10.4), or (10.5) and guarantees that all slacks are at least $\Theta^\star$ can be found in $O(n^3)$ time by a minimum weight perfect matching algorithm in the pruned bipartite graph by choosing edge weights appropriately.
The wire length minimization is similar to the minimum-cost network optimization from Cho et al. [Cho+13] except that we prohibit assignments (i, j) that would



(a) Basic network (b) Extended network

Figure 10.3: The network model allowing (a) one latch and (b) multiple sizes at a placement position.

result in a poor worst slack, i.e. those with $\mathrm{WS}_{ij} < \Theta^\star$. In fact, the underlying algorithm for computing a minimum-weight perfect matching is a minimum-cost flow algorithm:
Each latch vertex l_i has a flow supply of $b(l_i) = 1$ and each slot vertex a flow demand of $b(s_j) = -1$. Each edge $e \in E$ is assigned a capacity $u(e) = 1$. An integral b-flow of value n corresponds to an assignment of the latches to slots, where a feasible b-flow f is defined by flow conservation and capacity constraints as follows:

$$\begin{aligned} \textstyle\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e) &= b(v) \quad \forall v \in V, \\ 0 \leq \quad f(e) &\leq u(e) \quad \forall e \in E. \end{aligned} \tag{10.6}$$

As all edge capacities and supply/demand values on vertices are integral, there exists an integral minimum cost flow if any feasible flow exists, regardless of the assigned edge costs.

10.5 Extensions

Our algorithm can be extended to allow additional latch positions or sizes with a greater potential for slack improvement.

Additional latch positions

If the structure of latches around their driving LCB is not fixed, it is also possible to allow additional latch positions for example at free spaces around the cluster. This results in additional slots in the assignment problem and $|S| \geq n = |L|$. The matching problems from Section 10.4 would turn into a maximum cardinality matching and a minimum weight matching problem of cardinality n, respectively, but could still be solved by a minimum-cost flow algorithm. The total running time bound for optimizing the primary and secondary objective would become $O(n^2|S|\log(|S|))$. The difficulty lies in finding reasonable additional positions.

Additional sizes

A slightly more involved extension is to allow different sizes for each available placement position. Assume that for each latch position $pos_k,\ k = 1, \ldots, n$, there are n_k available sizes respectively slots s_k^j $(1 \leq j \leq n_k)$. We can find an assignment of the latches $l_1, \ldots l_n$ to the slots $\{s_k^j : 1 \leq k \leq n; 1 \leq j \leq n_k\}$ such that no two latches use the same slot with the extended network flow model shown in Figure 10.3(b):

We introduce additional (red) vertices for all placement positions. The edge set E contains a directed (blue) edge from each latch l_i $(i = 1, \ldots, n)$ to each slot s_k^j $(1 \leq k \leq n; 1 \leq j \leq n_k)$ and a (black) edge from each slot to its unique placement position pos_k. All edge capacities equal one. Flow demands of latch vertices are as before, while all slot vertices should preserve the flow conservation $b(s_k^j) = 0$ and position vertices have flow demand of $b(pos_i) = -1$. Now an integral b-flow of value n corresponds to an assignment of the latches to placement slots.

Because the flow demand of the position vertices pos_i $(1 \leq i \leq n)$ equals one, exactly one latch is assigned to each position.

The worst slack can be maximized by binary search and edge pruning as in Section 10.4.1. Secondary objectives can be considered by choosing edge weights on the (blue) edges appropriately, while (black) edges would have weight zero. Similar to Theorem 10.2, Θ is achievable if and only if there is a b-flow for the corresponding "pruned" network.

Of course, it is also possible to consider additional positions and multiple sizes at once. However, in both scenarios the clock footprint can change and a re-synthesis of the lower levels of the clock tree might become necessary, i.e. a re-routing of the clock nets. Depending on how additional positions and sizes are chosen, overlaps with other circuits can occur that would have to be resolved as well.

Remark 10.4 (V_t optimization) V_t optimization can be incorporated in our algorithm using the same construction as in Figure 10.3(b). We did not implement this feature because changing the V_t level has no impact on the clock footprint, and can thus be part of any other algorithm for V_t optimization. Additionally, it requires significantly more running time to compute the values WS_{ij}, as will become clear in the next section.

10.6 Implementation Details

As we are focusing on late optimization, we implemented and tested only the algorithm described in Section 10.4 for the problem defined in Section 10.2, i.e. we compute new assignments of latches to slots within a latch cluster, thereby keeping the sizes of the slots fixed.

10.6.1 Calculating Assignments

Instead of using algorithms with the best worst-case running time to solve the assignment problems, we apply the network simplex algorithm that typically shows faster running times in practice. The network simplex algorithm (see for example Korte and Vygen [KV12], Chapter 9), as most prevalent minimum-cost flow algorithms, will find integral solutions or decide that no feasible b-flow exists.

10.6.2 Assignments for Less Critical Instances

We found that the worst slack maximization sometimes comes with a degradation of the sum of negative endpoint slacks in the design. Thus we apply the worst slack maximization only to clusters that are in a 10 ps window above the initial worst design slack. The value of 10 ps is an experimentally observed upper bound on the achievable design slack improvement. For the remaining instances we optimize only the secondary objectives preserving the initial worst slack of the cluster.

10.6.3 Calculating Slacks and Wire Lengths

We assume that a legal placement of the latches and the clock network is given and that data path optimization like gate sizing, V_t optimization, repeater insertion or timing-driven detailed placement have been performed extensively.

We place each latch l_i $(1 \leq i \leq n)$ tentatively in each slot s_j $(1 \leq j \leq n)$ and compute the resulting total wire length WL_{ij} and worst slack WS_{ij} at its data pins. The clock arrival times and slews are simply transferred from the latch initially assigned to s_j.

Data signals can be re-analyzed using arbitrarily accurate delay models. In our experiments we used the industrial timing engine IBM-Einstimer for all delay, slew, and slack calculations under the RICE delay model (Ratzlaff and Pillage [RP94]), which was also used in the preceding data path optimization. Wires are estimated as Steiner trees that are computed by a Prim-heuristic.

To speed up the calculations, the timing engine can restrict delay re-calculations to a bounded number of logic levels around a latch. This prevents delays, slews and arrival times from being propagated through the whole logic cone, but can introduce small inaccuracies.

Sometimes data inputs are feeding pass-gates with tight slew constraints. We prune assignments resulting in slew or capacitance violations except for the input assignment $\tilde{x}$ that we always allow.

Wire lengths of data nets are estimated by the Steiner trees that are used for the delay calculation, and WL_{ij} is simply the total length of all data nets attached to l_i.

10.6.4 Dealing with Inaccuracies and Violated Assumptions

The restricted timing updates in Section 10.6.3 and the assumptions we made in Section 10.2.1 introduce small inaccuracies in our algorithm, and the improvement predicted by our algorithm might not hold true after realizing the new assignment and a new timing analysis.
In such situations we revert our change and keep the initial solution. In Section 10.7 we will see that this does not occur too often.

10.7 Experimental Results

We implemented our latch assignment algorithm in C++. Experiments were made on a Linux cluster of Intel Xeon CPUs with clock frequencies between 2.9–3.4 GHz. Our testbed consists of a set of nine microprocessor units in 22nm technology with 733–50861 latches provided by our industrial partner IBM, and is summarized in Table 10.1.

Design	# latches	#cluster	cycle time (ps)
Unit 1	733	26	174
Unit 2	4526	213	176
Unit 3	4759	167	340
Unit 4	5137	255	174
Unit 5	8010	300	174
Unit 6	17089	834	174
Unit 7	36756	1796	208
Unit 8	45372	1870	340
Unit 9	50861	2222	208

Table 10.1: Microprocessor units in our experiments

The algorithm is integrated into a physical synthesis flow after the clock network has been fixed and latches are clustered and placed in a structured fashion in groups of up to 32 latches using algorithms from Chan et al. [Cha+03], Cho et al. [Cho+13], Papa et al. [Pap+11], Ward et al. [War+13]. After the latch placement, timing optimization such as size and V_t optimization, repeater insertion, detailed placement, and logic restructuring have been performed, but no signal routing has been done yet.
We say that a cluster is critical if the worst slack found at one of its latches is below the slack target. For each unit, we ran our algorithm on the 20% most critical latch clusters based on the worst cluster slack, but at least 50 clusters except for unit U1, which has only 26 clusters. The clusters were optimized one after another.
For running time reasons, we did not include optimization of whole clock domains in our final tests.
When looking at the final design slacks, we found that minimizing the sum of slacks

Mode	secondary objective	data path refinement
S	SNS (10.3)	no
SR	SNS (10.3)	yes
N	wire length (10.4)	no
NR	wire length (10.4)	yes
R	no latch optimization	yes

Table 10.2: Four experimental setups

(10.3) at the cluster (SCS) can lead to slight decreases in the sum of negative slacks at all endpoints in the design (SNS). An explanation could be that (10.3) captures the slacks in the cluster but not all slacks in the design: Experimentally we observed that preserving the initial worst slack instead of maximizing the worst slack in phase one improves the sum of slacks SCS locally, but can degrade the sum of negative endpoint slacks SNS globally. In our final experiments we minimized the sum of slacks (10.3) and wire length as secondary objectives. As data paths might again become improvable after changing latch positions and sizes, we also conducted experiments refining the placement and sizes of the immediately preceding and succeeding data gates of the latches in a cluster. This was done by a local search gate sizing and placement, maintaining a legal placement throughout. For comparison, we conducted experiments performing only these refinement steps. Table 10.2 shows the four combinations of secondary objective and potential subsequent data path refinement for which we conducted experiments.

The results for all designs and all modes are presented in Table 10.3. The first column shows the instance names, and the second column the optimization modes. The next 3 columns show the number of most critical clusters *"#Cl"* (which is the maximum of 50 and 20% of all clusters) that were optimized by our algorithm, the number of improved clusters *"Impr"* (either worst slack or secondary objective preserving the initial worst slack) and the number of clusters that were reverted after new timing or wire length analysis, respectively, *"Fail"* (as described in Section 10.6.4).

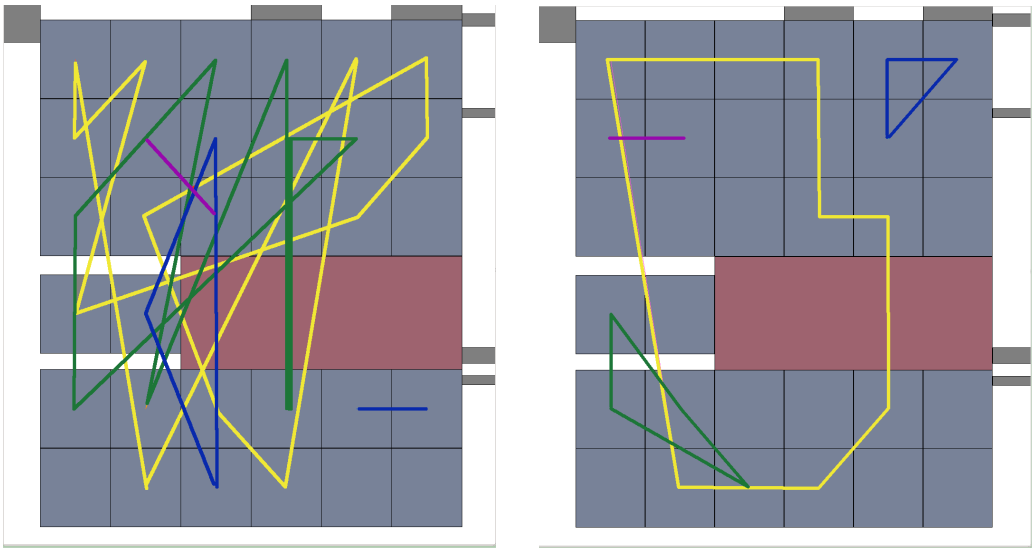
Columns 6-8 show the best worst slack improvement *"Best"* seen at a cluster, its percentage of the cycle time *"%cycle"* and the average worst slack improvement among all clusters where the aim was to improve worst slack *"∅WS"*.

The next columns show the worst design slack *"WS"* and worst design slack change *"ΔWS"*. *"SNS"* denotes the sum of all slacks, and *"WL"* the wire length of the whole unit. Finally, the last column shows the CPU running time *"RT"* of the latch assignment plus the data path refinement, if performed.

For each unit the first row shows the initial values before any optimization, and the following five rows show the results of the modes according to Table 10.2.

The number of fails, i.e. when solutions were reverted, is small for mode "S" ($< \%7$) and mode "N" ($< 15\%$). The number is larger for runs with data path refinement, as it can decrease cluster slacks and cluster wire length, respectively, to reduce elec-

trical capacitances and delays on more critical nets hidden from latch optimization. The maximum slack improvement seen at a cluster is between 1.1 and 14.2 ps and up to 7.8% of the cycle time for the plain latch assignment, which is in the expected range of a local optimization routine. The average worst slack improvement per cluster is between 0.3 and 3.0 ps.
Data path refinement increases the maximum slack improvement to 39.4 ps and up to 21% of cycle time. The average worst slack improvement per cluster is then between 0.4 and 4.8 ps.
Even for the most critical latch clusters relatively large improvements can be observed, e.g. the worst design slack increased for example by 9.9 ps on unit U9 or by 9.2 ps on unit U3. Particularly U3 demonstrates an application scenario where design closure is almost achieved and the latch assignment yields a substantial improvement to timing closure. These worst design slacks are not achieved by running data path refinement without preceding latch optimization (mode "R"), except for unit U1.
There are moderate changes in the sum of negative endpoint slacks *"SNS"* and wire length. A refinement step on the whole design instead of only the latch neighborhood will likely improve these numbers further.
On some units, e.g. U6, we observe a slight decrease in the sum of negative endpoint slacks, even when they are considered as a secondary objective. This can be a consequence of the worst slack maximization. Another reason is that we do not capture all endpoints in the design in our objective. In side experiments we observed that preserving the initial worst slack instead of maximizing it improves the sum of slacks SCS locally, but can degrade the sum of endpoint slacks globally. In contrast, the wire length, which is measured locally, often improves and never degrades when considered as a secondary objective.
The running times are fast, usually 1–3 seconds per cluster, allowing the algorithm to be applied several times in the design flow, potentially on a smaller set of most critical instances.
Figure 10.4 shows the re-assignment results for a cluster on unit U3. There is a line between slot s_i and s_j if either a latch was moved from s_i to s_j or the other way around. Solutions to the assignment problem can be partitioned into circuits, which are the basis for the line colors. When minimizing the sum of slacks SCS in Figure 10.4(a), most latches are assigned to another slot in this example. When minimizing wire length in Figure 10.4(b), the changes are less pronounced.



(a) Minimizing the sum of slacks

(b) Minimizing wire length

Figure 10.4: A cluster on unit U3 on which the worst slack improved from -29.8 to -21.4 ps. Latches are colored blue, the LCB red, and data logic is gray. The lines indicate the permutation of the latches. They are colored by circuits into which the global assignment is partitioned.

Unit	Mode	#Cl	Impr	Fail	Best (ps)	%cycle	⌀WS (ps)	WS (ps)	ΔWS (ps)	SNS (100ps)	WL (mm)	RT (sec)
U1	init							-182.5		-29.5	694	
	S	25	25	0	6.4	3.7	1.7	-180.7	1.8	-28.9	696	57
	SR	25	21	4	6.4	3.7	3.4	-176.9	5.6	-27.5	696	213
	N	25	13	0	1.1	0.6	0.5	-180.7	1.8	-29.3	693	72
	NR	25	8	1	4.8	2.8	1.7	-176.9	5.6	-28.2	693	114
	R	25	23	2	3.7	2.1	0.9	-176.2	6.3	-28.5	694	104
U2	init							-132.0		-101.6	1724	
	S	50	47	3	5.6	3.2	1.2	-130.5	1.5	-99.8	1732	130
	SR	50	43	7	7.9	4.5	1.5	-130.5	1.5	-101.7	1732	276
	N	50	8	7	1.4	0.8	0.5	-130.5	1.5	-101.8	1724	112
	NR	50	7	8	7.9	4.5	0.5	-130.5	1.5	-101.8	1724	153
	R	50	46	4	7.9	4.5	0.6	-132.0	0.0	-103.5	1725	126
U3	init							-29.8		-26.1	1125	
	S	50	45	0	8.4	2.5	2.5	-21.8	8.0	-23.0	1133	115
	SR	50	45	0	22.3	6.6	4.8	-20.6	9.2	-17.6	1133	395
	N	50	17	3	8.4	2.5	1.0	-26.1	3.7	-25.0	1125	111
	NR	50	17	1	22.1	6.5	2.7	-26.1	3.7	-22.0	1125	237
	R	50	47	3	22.1	6.5	3.4	-27.7	2.1	-19.5	1126	310
U4	init							-138.3		-197.4	2230	
	S	51	47	1	5.0	2.8	1.2	-137.2	1.1	-195.3	2247	182
	SR	51	44	4	8.8	5.0	1.9	-137.4	0.9	-193.5	2246	386
	N	51	11	2	2.8	1.6	0.4	-137.2	1.1	-197.3	2229	163
	NR	51	9	5	8.8	5.0	1.0	-137.4	0.9	-197.1	2230	210
	R	51	47	4	3.2	1.8	0.2	-138.4	-0.1	-195.0	2230	256
U5	init							-53.9		-156.5	2409	
	S	60	57	2	6.7	3.9	1.4	-50.3	3.6	-154.8	2415	160
	SR	60	53	6	36.5	21.0	2.6	-44.3	9.6	-153.3	2414	352
	N	60	31	0	6.3	3.6	1.4	-50.3	3.6	-156.4	2409	151
	NR	60	26	4	36.2	20.8	2.6	-44.3	9.6	-155.5	2409	259
	R	60	57	3	35.4	20.3	1.1	-47.8	6.1	-154.5	2409	215
U6	init							-109.9		-274.9	3827	
	S	167	157	5	13.5	7.8	1.5	-110.7	-0.8	-275.1	3838	237
	SR	167	126	36	14.4	8.3	1.3	-109.2	0.7	-274.9	3837	711
	N	167	67	20	12.4	7.1	0.7	-109.5	0.4	-275.4	3827	227
	NR	167	42	45	11.4	6.5	0.5	-109.5	0.4	-275.4	3828	424
	R	167	134	33	9.1	5.2	0.4	-109.9	0.0	-276.0	3828	388
U7	init							-101.8		-890.7	14451	
	S	360	347	5	8.2	3.9	1.0	-100.8	1.0	-895.0	14480	882
	SR	360	319	33	11.3	5.4	1.0	-100.8	1.0	-892.3	14479	2827
	N	360	125	53	7.6	3.6	0.3	-100.8	1.0	-894.6	14449	827
	NR	360	118	62	7.6	3.6	0.4	-100.8	1.0	-893.0	14450	1776
	R	360	339	21	10.5	5.0	0.5	-100.9	0.9	-886.5	14452	1395
U8	init							-256.1		-2733.8	12126	
	S	374	366	3	12.2	3.6	1.3	-253.8	2.3	-2718.7	12156	1021
	SR	374	297	72	14.8	4.4	1.2	-253.8	2.3	-2717.0	12118	2498
	N	374	151	46	4.9	1.5	1.0	-253.8	2.3	-2730.1	12124	991
	NR	374	115	84	9.0	2.6	1.0	-253.8	2.3	-2727.5	12125	2171
	R	374	345	29	14.8	4.4	0.6	-256.1	0.0	-2729.7	12129	1624
U9	init							-153.4		-14123.4	20095	
	S	445	425	20	14.2	6.8	2.9	-145.3	8.1	-14109.9	20129	1164
	SR	445	375	70	39.4	18.9	3.5	-143.5	9.9	-14099.8	20135	3523
	N	445	177	14	8.8	4.2	2.6	-145.5	7.9	-14125.9	20073	1154
	NR	445	166	25	10.6	5.1	2.6	-145.5	7.9	-14121.9	20074	2085
	R	445	403	42	31.1	14.9	0.8	-151.8	1.6	-14115.0	20097	2296

Table 10.3: Experimental results on 9 microprocessor units with optimization modes explained in Table 10.2.

11 Summary

One of the key problems in the physical design of a computer chip consists of choosing a physical realization for the logic gates and memory circuits on the chip from a discrete set of predefined layouts given by a library. Thereby the most common objective is to minimize total power consumption of the chip subject to constraints on the delay of signal paths. In this thesis we present new algorithms for the problem of choosing sizes for the circuits and its continuous relaxation, and we evaluate these in theory and practice.

In the continuous relaxation of the sizing problem, sizes are restricted to intervals. Under the Elmore delay model, it can be formulated as a convex program and solved in polynomial time, but it poses a challenge to researchers because of the huge instance sizes that can occur in practice. The discrete problem is *NP*-hard.

In Chapter 6 we consider an approach that is based on Lagrangian relaxation of the convex program. Thereby the constraints on the delays of signal paths (timing constraints) are relaxed using Lagrange multipliers and are incorporated into the objective function. We provide the first comprehensive discussion of this approach and fill gaps in the convergence analysis of the projected gradient method for this problem. The method iteratively computes new multipliers based on an additive update rule until a good solution has been found. In each iteration, an oracle that is guided by the multipliers computes intermediate solutions. We point out why the running time for the continuous relaxation is not necessarily polynomial, and highlight difficulties in obtaining convergence guarantees for the discrete problem.

In practice, variants of the projected gradient method are usually employed to find a good solution, among them a multiplicative multiplier update rule and an additional weight for the objective power consumption. We show in Chapter 7 that the well-known multiplicative weights algorithm applied to the feasibility version of the convex program returns a solution that approximately fulfills all constraints. The discretized algorithm essentially is the modified Lagrangian relaxation approach and justifies these modifications.

In Chapter 8 we consider gate sizing modeled as a min-max resource sharing problem, which consists of distributing a limited set of resources among a limited set of customers. An optimal solution distributes the resources in such a way that the maximum resource usage is minimized. In the resource sharing algorithm (Müller et al. [MRV11]), a weight is maintained for each resource and updated iteratively based on its usage and a multiplicative update rule. Customer oracle algorithms compute solutions that approximately minimize the weighted resource usages of the customers.

In our context, we have a power resource and resources for signal delays. We show how gate sizing fits into this framework with a single customer representing all gates. We obtain a fast approximation of the continuous relaxation that improves over the Lagrangian relaxation approach. Under the assumption that we are given a fixed library and reasonably long electrical wires, we obtain an $\eta(1+\epsilon)$ approximation in polynomial time for $\epsilon > 0$, where the error of the customer oracle algorithms is bounded by $\eta > 0$. Additionally, timing optimization objectives like worst slack maximization can be modeled more directly. The power resource weight allows to find a better tradeoff between power minimization and timing constraint optimization. We further show that constraints on local placement density and electrical constraints can be integrated without impairing convergence guarantees of the continuous relaxation.
For the discrete problem, a discrete oracle algorithm needs to be solved. The solution returned by the algorithm is then a convex combination of the intermediate solutions and not necessarily feasible.
In Chapter 5 we consider the subproblem that occurs in the Lagrangian relaxation and resource sharing algorithms, and which needs to be solved by the oracle algorithms. It consists of minimizing a weighted sum of power consumption and signal delays. While in the former algorithm, the task is to compute sizes close to the optimal solution, the aim in the latter is to find a good approximation on the value of the weighted sum. We show that the conditional gradient method is a pseudo-polynomial approximation algorithm for this problem and continuous sizes. It is polynomial under certain assumptions. For the discrete problem, we provide a fully polynomial approximation scheme for instances where the size of the antichains in the graph containing all gates is bounded by a constant.
In Chapter 9 we describe our implementations of the discrete Lagrangian relaxation and resource sharing algorithm as part of the BonnTools optimization suite for VLSI physical design, developed at the Research Institute for Discrete Mathematics in Bonn in an industrial cooperation with IBM. Both implementations are extended to incorporate V_t optimization. We compare the implementations on state-of-the-art microprocessor instances provided by IBM and the ISPD 2013 benchmarks (Ozdal et al. [Ozd+13]). Our results show that the resource sharing algorithm exhibits more stable convergence behavior and better timing on almost all instances. On several designs, power consumption was also improved. We further observed that oracle algorithms which considered timing objectives more directly often performed better than those which relied solely on the resource weights and the Lagrange multipliers, respectively. Similar observations have been reported in previous works on Lagrangian relaxation. We conclude this chapter with an outlook on future research. For example, a few challenges that arise by integrating the algorithm into an industrial environment remain.
In the clock network design phase, timing constraints are often considered only indirectly when latches are sized and placed on the chip area. Because redesigning the clock network and clock routing is costly, latch placement and sizes remain mostly unchanged afterwards. Often, latches are arranged around a common local

clock buffer in a structured fashion in clusters. The algorithm presented in Chapter 10 permutes latch positions and sizes in latch clusters and thereby maintains the clock footprint, such that it can be applied late in the design flow. Under mild assumptions, our algorithm efficiently maximizes the worst slack. Our experiments illustrate that it can improve slacks on industrial microprocessor instances effectively by up to 7.8% of design cycle time. As the algorithm is fast in theory and practice it has been integrated into an industrial design flow. It can be extended to optimize several clusters or whole clock domains simultaneously.

List of Figures

Notation

$\mathbb{R}$, $\mathbb{R}_{\geq 0}$, $\mathbb{R}_{>0}$	Set of real numbers, nonnegative real numbers and positive real numbers	
$\mathbb{N}$	Set of natural numbers	
$\mathbb{Z}$	Set of integers	
$\mathrm{V_{dd}}$	High voltage	13
$\mathrm{V_0}$	Ground/zero voltage	14
$\mathcal{I}$	Chip image	20
$\mathcal{C}$	Set of circuits	20
$\mathcal{G}$	Set of gates	20
$\mathcal{P}$	Set of pins	20
$\mathcal{N}$	Set of nets	20
$\gamma : \mathcal{P} \to \mathcal{C} \dot{\cup} \mathcal{I}$	Mapping of pins to circuits and $\mathcal{I}$	20
$\mathcal{B}$	Circuit library	20
$\phi : \mathcal{C} \to \mathcal{B}$	Mapping of circuits to books	21
$\mathcal{B}_c \subset \mathcal{B}$	Set of books available for circuit c	21
σ	Signal	21
$\tau(\sigma) \in \{rise, fall\}$	Transition of signal σ	21
$G = (V, E)$	Timing graph	22
$V_{start} \subset V$	Vertices corresp. to timing start points	22
$V_{end} \subset V$	Vertices corresp. to timing endpoints	22
V_{inner}	Vertices corresp. to $V \setminus \{V_{start} \cup V_{end}\}$	22
$\mathscr{G} = (\mathscr{V}, \mathscr{E})$	Gate graph	23
$G_g = (V_g, E_g)$	Neighborhood graph of $g \in \mathcal{G}$	24
$at_p(\sigma)$	Arrival time of σ at $p \in V$	24
$slew_p(\sigma)$	Slew of σ at $p \in V$	24
$delay_e^\tau$	Delay function for $e \in E$	25
$slew_e^\tau$	Slew function for $e \in E$	25
$cap(N)$	Total capacitance of net N	25
$rat_p(\sigma)$	Required arrival time of σ at $p \in V$	27
$slack_p(\sigma)$	Slack at $p \in V$ for signal σ	27
WS	Worst design slack	27
SNS	Sum of negative timing endpoint slacks	28
SLS	Sum of subpath slacks	28
$loadcap_p$	Load capacitance seen at pin $p \in \mathcal{P}$	28
$\mathcal{P}_{load}/V_{load}$	Set of pins with a load limit	28
$\mathcal{P}_{slew}/V_{slew}$	Set of pins with a slew limit	28
$loadlim_p$	Load capacitance limit at $p \in \mathcal{P}_{load}$	28

$slewlim_p$	Slew limit at $p \in \mathcal{P}_{slew}$	28
$area(o)$	Area usage of object $o \in \mathcal{C} \cup \mathcal{B}$	33
$size(o)$	Size of object $o \in \mathcal{C} \cup \mathcal{B}$	48
x_g	Size variable for $g \in \mathcal{G}$	49
X_{disc}	Set of feasible discrete gate size vectors	49
$a_v \in \mathbb{R}$	Arrival time variable for $v \in V$	52
X_{cont}	Set of feasible continuous gate sizes after variable transformation	53
$I_g = [l_g, u_g]$	Feasible interval for variable $x_g \in X_{cont}$	53
$cost(x)$	Objective function for gate sizing (convex for $x \in X_{cont}$)	53
$delay_e(x)$	Delay over $e \in E$ (convex for $x \in X_{cont}$)	53
$cost(x_i)$	Cost (power) of g_i induced by size x_i	53
$tr(x, \omega)$	Power-delay tradeoff function	65
$\omega \in \mathbb{R}^{m+1}_{\geq 0}$	Vector of delay and power weights	65
$tr_x(x_i, \omega)$	Local refine function for $g_i \in \mathcal{G}$	67
$lip(\omega)$	Lipschitz constant of $\nabla tr(x, \omega)$	67
$diam_X$	Diameter of set X_{cont}	70
tr_{ratio}	Running time of Algorithm 5.3 depends on this value	72
$\mathcal{F}$	Set of nonnegative network flows in G	87
$L(\lambda, x)$	Lagrange function	87
$D(\lambda)$	Lagrange dual objective function	87
$\mathcal{A} \subset \mathbb{R}^{\lvert V \rvert}$	Feasible arrival time assignments	114
D	Clock cycle time	116
$gatecus$	Gate customer	129
$atcus_v$	Arrival time customer for $v \in V$	128
$budget_{power} \in \mathbb{R}_{\geq 0}$	Power budget for all gates	128
$budget_e \in \mathbb{R}_{\geq 0}$	Delay budget for $e \in E$	128

Glossary

Arrival time (of a signal) The time when the voltage change of the signal reaches 50% (pages 22 and 24).

Book Blueprint of a circuit (page 20).

Chip → Integrated circuit.

Chip area A rectangle distributed over a placement plane and several routing planes (page 20).

Chip image Consists of the chip area, a set of blockages and I/O-ports (page 20).

Circuit area Rectangular area occupied by the circuit's shapes on the placement plane (page 21).

Circuit library Defines a set of logically equivalent books for each logic function and register type on the chip (page 15 and 20).

Clock domain Set of registers controlled by the same clock signal (page 17).

Clock network A network that distributes the clock signal to the registers and is often realized by a clock tree or a clock grid (page 17).

Delay (of a signal) Amount of time it takes a signal to traverse a certain distance between two timing points (page 25).

Conditional gradient method Descent method for constrained optimization problems that computes a series of descent directions such that the next iterate is feasible (page 42).

Convex Program Optimization problem over a convex set where the objective function and the inequality constraints are convex, and the equality constraints are affine functions (page 36).

Delay model Used to approximate the delay of electrical signals over wires (and circuits) in integrated circuits (page 25).

Dynamic power (of a circuit) The power consumed by a circuit due to switching (short circuit power) and charging and discharging capacitances (page 16).

EDA (Electronic design automation) Software tools to design electronic systems (page 19).

Elmore delay A widely-used delay model that approximates signal delay over wire segments in a net modeled as RC-tree. (page 28).

Fanin/Fanout → Neighborhood.

Feasibility Problem Consists of finding a solution that fulfills a given set of constraints, or decide that no such solution exists (page 108).

FPTAS (fully polynomial time approximation scheme) An algorithm that approximates the optimal solution within a factor of $(1 + \epsilon)$ in time polynomial in the input size and $1/\epsilon$.

Gate A circuit that computes a boolean function (page 15).

Gate graph A directed graph that arises from the timing graph by contracting the vertices that correspond to pins of the same gate to a single vertex (page 23).

Gate sizing problem Formal definition of the problem to choose a size for each gate on the chip (page 48).

Integrated Circuit An electrical circuit made from one piece of semiconductor material, more commonly known as chip (page 17).

I/O-ports Connection points of the chip with the outer world by which electrical signals enter and leave the chip. They are also referred to as primary input and output pins (page 20).

Lagrange relaxation A method for constrained optimization that relaxes difficult constraints in the objective function with Lagrange multipliers. The resulting function is called Lagrange function, and the Lagrange primal problem consists of minimizing this function. The Lagrange multipliers can be interpreted as variables of the Lagrange dual problem (page 37).

Latch → Register.

Latch cluster Latches arranged around an LCB in a structured fashion (page 177).

LCB (Local clock buffer) Dispenses clock signals to memory circuits (page 177).

Load capacitance The electrical capacitance driven by a pin or a circuit (page 28).

Load violation A load violation occurs if the load capacitance of a pin exceeds its prescribed load capacitance limit (page 28).

Local refine function (of a gate): Terms in the power-delay tradeoff function that depend on the size of this gate (page 67).

Min-max resource sharing problem Formal definition of the problem to distribute a finite set of resources to a finite set of customers such that the largest resource consumption is minimized.

Moore's law A prediction of Gordon Moore from 1965 (revised in 1975) that the number of components per integrated circuit will double every two years (page 18).

Multiplicative weights method Broader term for algorithms that maintain weights for elements of a certain set, and iteratively change these weights based on a multiplicative update rule (page 108).

Neighborhood (graph) The neighborhood of a pin/gate contains the immediate successor pins/gates (fanout), predecessor pins/gates (fanin), and the pins/gates that have a common driver. The neighborhood graph of a gate is a subgraph of the timing graph (page 24).

Net A set of pins connected by electrical wires (page 20).

Netlist (of a chip) Consists of finite sets of circuits, pins and nets, and a mapping of pins to circuits or the chip image (I/O ports) (page 20).

Pin Connection point of a circuit (page 14) or the chip itself $\rightarrow$ I/O ports.

Placement density (of a region) The ratio of the placement area covered by circuits and the placement area of the region itself (page 33).

Placement (location) A placement is a mapping of circuits to the placement plane. The placement location of a circuit is the location of its anchor point on the placement plane (page 21).

Physical design A step in VLSI design that maps a netlist to a chip image and circuits in the netlist to books (page 19).

Physical design instance Consists of a chip image, a netlist and a circuit library with an initial binding of circuits to books (page 21).

Power-delay tradeoff problem Problem to find gate sizes minimizing a weighted sum of power and signal delays (the power-delay tradeoff function) (page 65).

Primary input/output pin $\rightarrow$ I/O ports.

Projected gradient method Descent method for constrained optimization problems that computes a series of iterates and projects them to the constraint set in each iteration (page 41).

Register Memory element that can store one bit at a time (page 14).

Required arrival time (of a signal) The latest arrival time of a signal which ensures that the timing constraints are fulfilled (page 27).

Signal Voltage change over time (page 21). We distinguish between data signals that represent the logical computations of the chip (page 22) and clock signals that control the memory elements on the chip (page 22).

Slack (of a signal) The difference between the arrival time and the required arrival time of a signal at a timing point (page 27). For slack target see page 27.

Slew (of a signal) The slew is usually given as the time between 10% and 90% of the voltage change of the signal (page 22).

Slew violation A slew violation occurs if the slew exceeds its prescribed slew limit at a timing point (page 28).

SNS, SLS The sum of negative timing endpoint slacks (page 28) and the sum of subpath slacks in the timing graph (page 28).

Static power (leakage) (of a circuit) The power consumed when the circuit is not switching (page 16).

Static timing analysis (STA) Checks if conditions on the speed of electrical signals on a chip are fulfilled (page 21).

Timing closure A design has closed timing if all timing constraints on the signals are fulfilled (page 21).

Timing constraints Constraints on signal arrival times. In this thesis we are only interested in late mode timing constraints which demand that all signals arrive on time at primary output pins and register input pins (page 33).

Timing engine A computer program to compute approximate signal delays over wires and circuits (page 26).

Timing graph A directed acyclic graph whose vertices correspond to the timing points, which usually comprise the pins in the netlist (page 22).

Timing rules Provide information for each book in the circuit library about the behaviour towards a voltage change (page 16).

Total power (of a circuit) The sum of static and dynamic power consumption of the circuit (page 16).

Transistor An electronic switch with three external connections. The voltage applied to the control terminal determines when the transistor is conducting (page 13).

Transition (of a signal) The direction of the signal (rising or falling) (page 21).

VLSI design The process to design integrated circuits whose integration level is referred to as very large scale integration (VLSI) (page 18).

Voltage threshold ($\mathbf{V_t}$ level) (of a circuit) The voltage of the signal needed at the circuit's input pins so that the circuit switches (page 14).

$\mathbf{V_t}$ optimization problem Formal definition of the problem to choose a V_t level for each gate on the chip (page 54).

Bibliography

[AHK12] Sanjeev Arora, Elad Hazan, and Satyen Kale. The Multiplicative Weights Update Method: A Meta-Algorithm and Applications. In: *Theory of Computing* 8 (2012), pages 121–164. doi: 10.4086/toc.2012.v008a006.

[Alb01] Christoph Albrecht. Global routing by new approximation algorithms for multicommodity flow. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 20.5 (2001), pages 622–632. doi: 10.1109/43.920691.

[Alp+07] Charles J. Alpert, Shriran K. Karandikar, Zhuo Li, Gi-Joon Nam, Stephen T. Quay, Haoxing Ren, Cliff N. Sze, Paul Villarrubia, and Mehmet C. Yildiz. Techniques for Fast Physical Synthesis. In: *Proceedings of the IEEE* 95.3 (2007), pages 573–599. doi: 10.1109/JPROC.2006.890096.

[Bar14] Christoph Bartoschek. Fast Repeater Tree Construction. PhD thesis. University of Bonn, 2014.

[Ber99] Dimitri P. Bertsekas. *Nonlinear programming*. Athena Scientific, 1999. ISBN 9781886529007.

[BF85] Frank Brglez and Hideo Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortran. In: *Proceedings of the IEEE International Symposium on Circuits and Systems*, ISCAS 1985, pages 677–692.

[BJ90] Michel R. C. M. Berkelaar and Jochen A. G. Jess. Gate Sizing in MOS Digital Circuits with Linear Programming. In: *Proceedings of the Conference on European Design Automation*, EURO-DAC 1990, pages 217–221. doi: 10.1109/EDAC.1990.136648.

[Boy+05] Stephen B. Boyd, Seung-Jean Kim, Dinesh D. Patil, and Mark A. Horowitz. Digital Circuit Optimization via Geometric Programming. In: *Geometric Operations Research* 53.6 (2005), pages 899–932. doi: 10.1287/opre.1050.0254.

[Bre+15] Ulrich Brenner, Anna Hermann, Nils Hoppmann, and Philipp Ochsendorf. BonnPlace: A Self-Stabilizing Placement Framework. In: *Proceedings of the International Symposium on Physical Design*, ISPD 2015, pages 9–16. doi: 10.1145/2717764.2717778.

[BSS06] Mokhtar S. Bazaraa, Hanif D. Sherali, and C.M. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley, 2006. ISBN 978-0-471-48600-8.

[BV04] Stephen P. Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. ISBN 0521833787.

[CCW99] Chung-Ping Chen, Chris C. N. Chu, and D.F. Wong. Fast and Exact Simultaneous Gate and Wire Sizing by Lagrangian Relaxation. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.7 (1999), pages 1014–1025. doi: 10.1109/43.771182.

[CH96] Jason Cong and Lei He. An efficient approach to simultaneous transistor and interconnect sizing. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 1996, pages 181–186. doi: 10.1109/ICCAD.1996.569580.

[Cha+03] Yiu-Hing Chan, Prabhakar Kudva, Lisa Lacey, Greg Northrop, and Thomas Rosser. Physical Synthesis Methodology for High Performance Microprocessors. In: *Proceedings of the 40th IEEE/ACM Design Automation Conference*, DAC 2003, pages 696–701. doi: 10.1145/775832.776009.

[Cha90] Pak K. Chan. Algorithms for library-specific sizing of combinational logic. In: *Proceedings of the 27th IEEE/ACM Design Automation Conference*, DAC 1990, pages 353–356. doi: 10.1109/DAC.1990.114881.

[Che+05] Yongseok Cheon, Pei-Hsin Ho, Andrew B. Kahng, Sherief Reda, and Qinke Wang. Power-aware Placement. In: *Proceedings of the 42nd IEEE/ACM Design Automation Conference*, DAC 2005, pages 795–800. doi: 10.1145/1065579.1065791.

[Cho+13] Minsik Cho, Hua Xiang, Haoxing Ren, Matthew M. Ziegler, and Ruchir Puri. LatchPlanner: Latch Placement Algorithm for Datapath-oriented High-Performance VLSI Designs. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2013, pages 342–348. doi: 10.1109/ICCAD.2013.6691141.

[CHP00] Wei Chen, Cheng-Ta Hseih, and M. Pedram. Simultaneous Gate Sizing and Placement. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.2 (2000), pages 206–214. doi: 10.1109/43.828549.

[Chu+11a] Yi-Lin Chuang, Sangmin Kim, Youngsoo Shin, and Yao-Wen Chang. Pulsed-Latch Aware Placement for Timing-Integrity Optimization. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.12 (2011), pages 1856–1869. doi: 10.1109/TCAD.2011.2165717.

[Chu+11b] Yi-Lin Chuang, Hong-Ting Lin, Tsung-Yi Ho, Yao-Wen Chang, and D. Marculescu. PRICE: Power Reduction by Placement and Clock-Network Co-Synthesis for Pulsed-Latch Designs. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2011, pages 85–90. doi: 10.1109/ICCAD.2011.6105310.

[CK05] David Chinnery and Kurt Keutzer. Linear Programming for Sizing, V_{th} and V_{dd} Assignment. In: *Proceedings of the 2005 International Symposium on Low Power Electronics and Design*, ISLPED 2005, pages 149–154. doi: 10.1109/LPE.2005.195505.

[CLL11] Jason Cong, John Lee, and Guojie Luo. A Unified Optimization Framework for Simultaneous Gate Sizing and Placement under Density Constraints. In: *IEEE International Symposium on Circuits and Systems*, ISCAS 2011, pages 1207–1210. doi: 10.1109/ISCAS.2011.5937786.

[CMS07] Nicolò Cesa-Bianchi, Yishay Mansour, and Gilles Stoltz. Improved second-order bounds for prediction with expert advice. In: *Machine Learning* 66.2-3 (2007), pages 321–352. doi: 10.1007/s10994-006-5001-7.

[CR15] Tony Casagrande and Nagarajan Ranganathan. GTFUZZ: A Novel Algorithm for Robust Dynamic Power Optimization via Gate Sizing with Fuzzy Games. In: *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE 2015, pages 677–682. doi: 10.7873/DATE.2015.0560.

[CSH95] Weitong Chuang, Sachin S. Sapatnekar, and Ibrahim N. Hajj. Timing and Area Optimization for Standard-Cell VLSI Circuit Design. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 14.3 (1995), pages 308–320. doi: 10.1109/43.365122.

[CW01] Chris C. N. Chu and D.F. Wong. VLSI Circuit Performance Optimization by Geometric Programming. In: *Annals of Operations Research* 105.1-4 (2001), pages 37–60. doi: 10.1023/A:1013345330079.

[CW03] John F. Croix and D.F. Wong. Blade and Razor: Cell and interconnect delay analysis using current-based models. In: *Proceedings of the 40th IEEE/ACM Design Automation Conference*, DAC 2003, pages 386–389. doi: 10.1109/DAC.2003.1219030.

[CW99] Chris C. N. Chu and D. F. Wong. Greedy Wire-sizing is Linear Time. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 18.4 (1999), pages 398–405. doi: 10.1109/43.752924.

[CWC05] Hsinwei Chou, Yu-Hao Wang, and Charlie C.-P. Chen. Fast and effective gate-sizing with multiple-Vt assignment using generalized Lagrangian Relaxation. In: *Proceedings of the 13th Asia and South Pacific Design Automation Conference*, ASP-DAC 2005, pages 381–386. doi: 10.1109/ASPDAC.2005.1466193.

[DA89] Z.-J. Dai and K. Asada. MOSIZ: a two-step transistor sizing algorithm based on optimal timing assignment method for multi-stage complex gates. In: *Proceedings of the IEEE Custom Integrated Circuits Conference*, CICC 1989, pages 17.3.1–17.3.4. doi: 10.1109/CICC.1989.56775.

[Dab15] Siad Daboul. Algorithms for the gate sizing and V_t assignment problem. Master thesis. University of Bonn, 2015.

[De+97] Prabuddha De, E. James Dunne, Jay B. Ghosh, and Charles E. Wells. Complexity of the Discrete Time-Cost Tradeoff Problem for Project Networks. In: *Operations Research* 45.2 (1997), pages 302–306. doi: 10.1287/opre.45.2.302.

[Den+74] Robert H. Dennard, Fritz H. Gaensslein, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of Ion-Implanted MOSFET'S with Very Small Physical Dimensions. In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pages 256–268. doi: 10.1109/JSSC.1974.1050511.

[DS05] Irit Dinur and Samuel Safra. On the Hardness of Approximating Minimum Vertex Cover. In: *Annals of Mathematics*. Second Series 162.1 (2005), pages 439–485. doi: 10.4007/annals.2005.162.439.

[Dun87] J. C. Dunn. On the convergence of projected gradient processes to singular critical points. In: *Journal of Optimization Theory and Applications* 55.2 (1987), pages 203–216. doi: 10.1007/BF00939081.

[DW01] Vladimir G. Deineko and Gerhard J. Woeginger. Hardness of approximation of the discrete time-cost tradeoff problem. In: *Operations Research Letters* 29.5 (2001), pages 207–210. doi: 10.1016/S0167-6377(01)00102-X.

[Elm48] William C. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. In: *Journal of Applied Physics* 19.1 (1948), pages 55–63. doi: 10.1063/1.1697872.

[Erm66] Yuri M. Ermoliev. Methods for solving nonlinear extremal problems. In: *Kybernetika* 4 (1966), pages 1–17.

[ESZ02] Funda Ergun, Rakesh Sinha, and Lisa Zhang. An Improved FPTAS for Restricted Shortest Path. In: *Information Processing Letters* 83.5 (2002), pages 287–291. doi: 10.1016/S0020-0190(02)00205-3.

[Far+13] Amin Farshidi, Logan Rakai, Laleh Behjat, and David Westwick. Optimal Gate Sizing Using a Self-Tuning Multi-Objective Framework. In: *Integration, the VLSI Journal* 47.3 (2013), pages 347–355. doi: 10.1016/j.vlsi.2013.10.008.

[FD85] Jack P. Fishburn and Al E. Dunlop. TILOS: A posynomial programming approach to transistor sizing. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 1985, pages 326–328. doi: 10.1007/978-1-4615-0292-0˙23.

[FGW02] Anders Forsgren, Philip E. Gill, and Margaret H. Wright. Interior Methods for Nonlinear Optimization. In: *SIAM Review* 44.4 (2002), pages 525–597. doi: 10.1137/S0036144502414942.

[Fla+14] Guilherme Flach, Tiago Reimann, Gracieli Posser, Marcelo Johann, and Ricardo Reis. Effective Method for Simultaneous Gate Sizing and V_t Assignment Using Lagrangian Relaxation. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.4 (2014), pages 546–557. doi: 10.1109/TCAD.2014.2305847.

[FW56] Marguerite Frank and Philip Wolfe. An algorithm for quadratic programming. In: *Naval Research Logistics* 3 (1956), pages 95–110. doi: 10.1002/nav.3800030109.

[GJ77] Michael R. Garey and David S. Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. In: *SIAM Journal on Applied Mathematics* 32.4 (1977), pages 826–834. doi: 10.1137/0132071.

[GK07] Naveen Garg and Jochen Könemann. Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems. In: *SIAM Journal on Computing* 37.2 (2007), pages 630–652. doi: 10.1137/S0097539704446232.

[GK94] Michel D. Grigoriadis and Leonid G. Khachiyan. Fast Approximation Schemes for Convex Programs with Many Blocks and Coupling Constraints. In: *SIAM Journal on Optimization* 4.1 (1994), pages 86–107. doi: 10.1137/0804004.

[Gol64] A. A. Goldstein. Convex programming in Hilbert space. In: *Bulletin of the American Mathematical Society* 70 (1964), pages 709–710. doi: 10.1090/S0002-9904-1964-11178-2.

[Gup+10] Puneet Gupta, Andrew B. Kahng, Amarnath Kasibhatla, and Puneet Sharma. Eyecharts: Constructive benchmarking of gate sizing heuristics. In: *Proceedings of the 47th ACM/IEEE Design Automation Conference*, DAC 2010, pages 597–602. doi: 10.1145/1837274.1837421.

[GW04] Alexander Grigoriev and Gerhard J. Woeginger. Project scheduling with irregular costs: complexity, approximability, and algorithms. In: *Acta Informatica* 41.2-3 (2004), pages 83–97. doi: 10.1007/s00236-004-0150-2.

[Häh15] Nicolai Hähnle. *Time-Cost Tradeoff and Steiner Tree Packing with Multiplicative Weights*. Technical report no. 1511115. Research Institute for Discrete Mathematics, University of Bonn, 2015.

[Hel+15] Stephan Held, Dirk Müller, Daniel Rotter, Vera Traub, and Jens Vygen. Global Routing with Inherent Static Timing Constraints. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2015, pages 102–109.

[Hel08] Stephan Held. Timing Closure in Chip Design. PhD thesis. University of Bonn, 2008.

[Hel09] Stephan Held. Gate Sizing for Large Cell-Based Desings. In: *Proceedings of the Conference on Design, Automation Test in Europe*, DATE 2009, pages 827–832. doi: 10.1109/DATE.2009.5090777.

[HHS11] Yi-Le Huang, Jiang Hu, and Weiping Shi. Lagrangian Relaxation for Gate Implementation Selection. In: *Proceedings of the 2011 International Symposium on Physical Design*, ISPD 2011, pages 167–174. doi: 10.1145/1960397.1960436.

[HKH09] Shiyan Hu, Mahesh Ketkar, and Jiang Hu. Gate Sizing for Cell-Library-Based Designs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.6 (2009), pages 818–825. doi: 10.1109/TCAD.2009.2015735.

[HR06] Narender Hanchate and Nagarajan Ranganathan. Post-layout gate sizing for interconnect delay and crosstalk noise optimization. In: *7th International Symposium on Quality Electronic Design*, ISQED 2006, pages 92–97. doi: 10.1109/ISQED.2006.101.

[HS14] Stephan Held and Ulrike Schorr. Post-Routing Latch Optimization for Timing Closure. In: *Proceedings of the 51st IEEE/ACM Design Automation Conference*, DAC 2014, pages 1–6. doi: 10.1145/2593069.2593182.

[HSC82] Robert B. Hitchcock, Gordon L. Smith, and David D. Cheng. Timing Analysis of Computer Hardware. In: *IBM Journal of Research and Development* 26.1 (1982), pages 100–105. doi: 10.1147/rd.261.0100.

[Hu+12] Jin Hu, Andrew B. Kahng, Seokhyeong Kang, Myung-Chul Kim, and Igor L. Markov. Sensitivity-guided metaheuristics for accurate discrete gate sizing. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2012, pages 233–239. doi: 10.1145/2429384.2429428.

[IFI91] Satoru Ibaraki, Masao Fukushima, and Toshihide Ibaraki. Dual-based Newton methods for nonlinear minimum cost network flow problems. In: *Journal of the Operations Research Society of Japan* 34.3 (1991), pages 263–286.

[Jag13] Martin Jaggi. Revisiting Frank-Wolfe: Projection-free sparse convex optimization. In: *International Conference on Machine Learning*, ICML 2013, pages 427–435.

[JB08] Siddharth Joshi and Stephen P. Boyd. An Efficient Method for Large-Scale Gate Sizing. In: *IEEE Transactions on Circuits and Systems I* 55.9 (2008), pages 2760–2773. doi: 10.1109/TCSI.2008.920087.

[JW59] James E. Kelley Jr and Morgan R. Walker. Critical-path Planning and Scheduling. In: *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 160–173. doi: 10.1145/1460299.1460318.

[JZ08] Klaus Jansen and Hu Zhang. Approximation algorithms for general packing problems and their application to the multicast congestion problem. In: *Mathematical Programming* 114.1 (2008), pages 183–206. doi: 10.1007/s10107-007-0106-8.

[Kah+11] Andrew B. Kahng, Jens Lienig, Igor L. Markov, and Jin Hu. *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer, 2011. ISBN 9789048195909.

[Kah+13] Andrew B. Kahng, Seokhyeong Kang, Hyein Lee, Igor L. Markov, and Pankit Thapar. High-performance gate sizing with a signoff timer. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2013, pages 450–457. doi: 10.1109/ICCAD.2013.6691156.

[Kar84] Narendra Karmarkar. A New Polynomial-time Algorithm for Linear Programming. In: *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC 1984, pages 302–311. doi: 10.1145/800057.808695.

[Kha04] Rohit Khandekar. Lagrangian Relaxation Based Algorithms for Convex Programming Problems. PhD thesis. Indian Institute of Technology, Delhi, 2004.

[KK12] Andrew B. Kahng and Seokhyeong Kang. Construction of Realistic Gate Sizing Benchmarks with Known Optimal Solutions. In: *Proceedings of the 2012 ACM International Symposium on Physical Design*, ISPD 2012, pages 153–160. doi: 10.1145/2160916.2160949.

[KKS00] Kishore Kasamsetty, Mahesh Ketkar, and Sachin S. Sapatnekar. A new class of convex functions for delay modeling and its application to the transistor sizing problem [CMOS gates]. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 19.7 (2000), pages 779–788. doi: 10.1109/43.851993.

[KV12] Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. 5th edition. Springer Publishing Company, Incorporated, 2012. ISBN 9783642244889.

[Lan00] Katharina Langkau. Gate Sizing in VLSI Design (in German). Diploma thesis. University of Bonn, 2000.

[Lee+01] Jin-Fuw Lee, D.L. Ostapko, Jeffery Soreff, and C.K. Wong. On the signal bounding problem in timing analysis. In: *IEEE/ACM International Conference on Computer Aided Design*, ICCAD 2001, pages 507–514. doi: 10.1109/ICCAD.2001.968693.

[LG12] John Lee and Puneet Gupta. Discrete Circuit Optimization: Library Based Gate Sizing And Threshold Voltage Assignment. In: *Foundations and Trends in Electronic Design Automation* 6.1 (2012), pages 1–120. doi: 10.1561/1000000019.

[LH10] Yifang Liu and Jiang Hu. A New Algorithm for Simultaneous Gate Sizing and Threshold Voltage Assignment. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.2 (2010), pages 223–234. doi: 10.1109/TCAD.2009.2035575.

[LH11] Chen Liao and Shiyan Hu. Approximation scheme for restricted discrete gate sizing targeting delay minimization. In: *Journal of Combinatorial Optimization* 21.4 (2011), pages 497–510. doi: 10.1007/s10878-009-9267-0.

[Li+12a] Li Li, Peng Kang, Yinghai Lu, and Hai Zhou. An efficient algorithm for library-based cell-type selection in high-performance low-power designs. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2012, pages 226–232. doi: 10.1145/2429384.2429427.

[Li+12b] Zhuo Li, Charles J. Alpert, Gi-Joon Nam, Cliff N. Sze, Natarajan Viswanathan, and Nancy Y. Zhou. Guiding a physical design closure system to produce easier-to-route designs with more predictable timing. In: *Proceedings of the 49th IEEE/ACM Design Automation Conference*, DAC 2012, pages 465–470. doi: 10.1145/2228360.2228442.

[Li+93] Wing-Ning Li, Andrew Lim, Prathima Agrawal, and Sartaj Sahni. On the circuit implementation problem. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12.8 (1993), pages 1147–1156. doi: 10.1109/43.238607.

[Li94] Wing Ning Li. Strongly NP-hard discrete gate-sizing problems. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.8 (1994), pages 1045–1051. doi: 10.1109/43.298040.

[Liv+13] Vinicius S. Livramento, Chrystian Guth, Jose Luis Guntzel, and Marcelo O. Johann. Fast and efficient Lagrangian Relaxation-based Discrete Gate Sizing. In: *Proceedings of the Conference on Design, Automation Test in Europe*, DATE 2013, pages 1855–1860. doi: 10.7873/DATE.2013.370.

[Liv+14] Vinicius S. Livramento, Chrystian Guth, José Luís Güntzel, and Marcelo O. Johann. A Hybrid Technique for Discrete Gate Sizing Based on Lagrangian Relaxation. In: *ACM Transactions on Design Automation of Electronic Systems* 19.4 (2014), 40:1–40:25. doi: 10.1145/2647956.

[LP66] Evgeny S. Levitin and Boris T. Polyak. Constrained minimization methods. In: *USSR Computational Mathematics and Mathematical Physics (english translation)* 6.5 (1966), pages 1–50.

[LPT14] Dirk A. Lorenz, Marc E. Pfetsch, and Andreas M. Tillmann. An Infeasible-point Subgradient Method Using Adaptive Approximate Projections. In: *Computational Optimization and Applications* 57.2 (2014), pages 271–306. doi: 10.1007/s10589-013-9602-3.

[LSH08] Yifang Liu, Rupesh S. Shelar, and Jiang Hu. Delay-optimal simultaneous technology mapping and placement with applications to timing optimization. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2008, pages 101–106. doi: 10.1109/ICCAD.2008.4681558.

[LT92] Zhiquan Q. Luo and Paul Tseng. On the Convergence of the Coordinate Descent Method for Convex Differentiable Minimization. In: *Journal of Optimization Theory and Applications* 72.1 (1992), pages 7–35. doi: 10.1007/BF00939948.

[Luo+08] Tao Luo, David A. Papa, Zhuo Li, Cliff N. Sze, Charles J. Alpert, and David Z. Pan. Pyramids: An efficient computational geometry-based approach for timing-driven placement. In: *Proceedings of the IEEE/ACM International Conference onComputer-Aided Design*, ICCAD 2008, pages 204–211. doi: 10.1109/ICCAD.2008.4681575.

[Mar89] David Marple. Transistor Size Optimization in the Tailor Layout System. In: *Proceedings of the 26th IEEE/ACM Design Automation Conference*, DAC 1989, pages 43–48. doi: 10.1145/74382.74391.

[MBP97] Noel Menezes, Ross Baldick, and Lawrence T. Pileggi. A sequential quadratic programming approach to concurrent gate and wire sizing. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 16.8 (1997), pages 867–881. doi: 10.1109/43.644611.

[Min84] Michel Minoux. A polynomial algorithm for quadratic minimum-cost flow problems. In: *European Journal of Operational Research* 18.3 (1984), pages 377–387. doi: 10.1016/0377-2217(84)90160-7.

[Moo65] Gordon E. Moore. Cramming More Components onto Integrated Circuits. In: *Electronics* 38.8 (1965), pages 114–117. doi: 10.1109/JPROC.1998.658762.

[MR04] Ashok K. Murugavel and N. Ranganathan. Gate sizing and buffer insertion using economic models for power optimization. In: *Proceedings of the 17th International Conference on VLSI Design*, VLSID 2004, pages 195–200. doi: 10.1109/ICVD.2004.1260924.

[MRV11] Dirk Müller, Klaus Radke, and Jens Vygen. Faster min-max resource sharing in theory and practice. In: *Mathematical Programming Computation* 3.1 (2011), pages 1–35. doi: 10.1007/s12532-011-0023-y.

[NB10] Angelia Nedic and Dimitri P. Bertsekas. The Effect of Deterministic Noise in Subgradient Methods. In: *Math. Program.* 125.1 (2010), pages 75–99. doi: 10.1007/s10107-008-0262-5.

[Nem04] Arkadi Nemirovski. Interior point polynomial time methods in convex programming. In: *Lecture Notes* (2004). URL: `http://www2.isye.gatech.edu/~nemirovs/Lect_IPM.pdf`.

[Nes04] Yurii Nesterov. *Introductory Lectures on Convex Optimization. A Basic Course*. Kluwer, 2004.

[Ngu+03] 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: *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, ISLPED 2003, pages 158–163. doi: 10.1109/LPE.2003.1231853.

[NP73] Laurence W. Nagel and D.O. Pederson. *SPICE (Simulation Program With Integrated Circuit Emphasis)*. Technical Report Memorandum, ERL-M382. University of California, Berkeley, 1973.

[OBH12] Muhammet M. Ozdal, Steven Burns, and Jiang Hu. Algorithms for Gate Sizing and Device Parameter Selection for High-Performance Designs. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.10 (2012), pages 1558–1571. doi: 10.1109/TCAD.2012.2196279.

[Onn10] Shmuel Onn. *Nonlinear Discrete Optimization: An Algorithmic Theory*. European Mathematical Society Publishing House, 2010. ISBN 9783037190937.

[Ozd+12] Muhammet M. Ozdal, Chirayu Amin, Andrey Ayupov, Steven Burns, Gustavo Wilke, and Cheng Zhuo. The ISPD-2012 Discrete Cell Sizing Contest and Benchmark Suite. In: *Proceedings of the 2012 ACM International Symposium on International Symposium on Physical Design*, ISPD 2012, pages 161–164. doi: 10.1145/2160916.2160950.

[Ozd+13] Muhammet M. Ozdal, Chirayu Amin, Andrey Ayupov, Steven Burns, Gustavo Wilke, and Cheng Zhuo. An Improved Benchmark Suite for the ISPD-2013 Discrete Cell Sizing Contest. In: *Proceedings of the 2013 ACM international symposium on International symposium on physical design*, ISPD 2013, pages 168–170. doi: 10.1145/2451916.2451959.

[Pap+08] David A. Papa, Tao Luo, Michael D. Moffitt, Cliff N. Sze, Zhou Li, Gi-Joon Nam, Charles J. Alpert, and Igor L. Markov. RUMBLE: An Incremental, Timing-Driven, Physical-Synthesis Optimization Algorithm. In: *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.12 (2008), pages 2156–2168. doi: 10.1109/TCAD.2008.2006155.

[Pap+11] David A. Papa, Charles J. Alpert Cliff N. Sze, Zhuo Li, Natarajan Viswanathan, Gi-Joon Nam, and Igor L. Markov. Physical Synthesis with Clock-Network Optimization for Large Systems on Chips. In: *EEE Micro* 31.4 (2011), pages 51–62. doi: 10.1109/MM.2011.41.

[Pol67] Boris T. Polyak. A general method for solving extremum problems. In: *Doklady Akademii Nauk SSSR 174/1 (in Russian). Translated in Soviet Mathematics Doklady* 8.3 (1967), pages 593–597.

[PST95] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast Approximation Algorithms for Fractional Packing and Covering Problems. In: *Mathematics of Operations Research* 20 (1995), pages 257–301. doi: 10.1287/moor.20.2.257.

[Que86] Maurice Queyranne. Performance ratio of polynomial heuristics for triangle inequality quadratic assignment problems. In: *Operations Research Letters* 4.5 (1986), pages 231–234. doi: 10.1016/0167-6377(86)90007-6.

[Rag88] Prabhakar Raghavan. Probabilistic Construction of Deterministic Algorithms: Approximating Packing Integer Programs. In: *Journal of Computer and System Sciences* 37.2 (1988), pages 130–143. doi: 10.1016/0022-0000(88)90003-7.

[RD08] Huan Ren and Shantanu Dutt. A Network-Flow Based Cell Sizing Algorithm. In: *The International Workshop on Logic Synthesis*, 2008, pages 7–14.

[RD13] Huan Ren and Shantanu Dutt. Fast and Near-Optimal Timing-Driven Cell Sizing under Cell Area and Leakage Power Constraints Using a Simplified Discrete Network Flow Algorithm. In: *VLSI Design - Special issue on New Algorithmic Techniques for Complex EDA Problems* 2013 (2013). doi: 10.1155/2013/474601.

[Roc71] R. Tyrrell Rockafellar. Ordinary convex programs without a duality gap. In: *Journal of Optimization Theory and Applications* 7.3 (1971), pages 143–148. doi: 10.1007/BF00932472.

[Roy+15] 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: *Proceedings of the 52nd IEEE/ACM Design Automation Conference*, DAC 2015, 129:1–129:6.

[RP94] Curtis L. Ratzlaff and Lawrence T. Pillage. RICE: Rapid interconnect circuit evaluation using AWE. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.6 (1994), pages 763–776.

[RPH83] Jorge Rubinstein, Jr. Paul Penfield, and Mark A. Horowitz. Signal Delay in RC Tree Networks. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and System* 2.3 (1983), pages 202–211. doi: 10.1109/TCAD.1983.1270037.

[RS04] Dieter Rautenbach and Christian Szegedy. *A Subgradient Method using Alternating Projections*. Technical report no. 04940. Research Institute for Discrete Mathematics, University of Bonn, 2004.

[RSR15] Tiago Reimann, Cliff N. Sze, and Ricardo Reis. Gate sizing and threshold voltage assignment for high performance microprocessor designs. In: *20th Asia and South Pacific Design Automation Conference*, ASP-DAC 2015, pages 214–219. doi: 10.1109/ASPDAC.2015.7059007.

[RTS11] Mohammad Rahman, Hiran Tennakoon, and Carl Sechen. Power reduction via near-optimal library-based cell-size selection. In: *Proceedings of the Conference on Design, Automation Test in Europe*, DATE 2011, pages 1–4. doi: 10.1109/DATE.2011.5763293.

[Sap+93] Sachin S. Sapatnekar, Vasant B. Rao, Pravin M. Vaidya, and Sung-Mo Kang. An exact solution to the transistor sizing problem for CMOS circuits using convex optimization. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 12.11 (1993), pages 1621–1634. doi: 10.1109/43.248073.

[Sap04] Sachin S. Sapatnekar. *Timing*. Kluwer Academic Publishers, 2004.

[Sch14] Jan Schneider. Transistor-Level Layout of Integrated Circuits. PhD thesis. University of Bonn, 2014.

[Sch99] Jürgen Schietke. Timing-Optimierung beim physikalischen Layout von nicht-hierarchischen Designs hochintegrierter Logikchips. PhD thesis. University of Bonn, 1999.

[Sha+05] Saumil Shah, Ashish Srivastava, Dushyant Sharma, Dennis Sylvester, David Blaauw, and Vladimir Zolotov. Discrete Vt assignment and gate sizing using a self-snapping continuous formulation. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD-2005, pages 705–712. doi: 10.1109/ICCAD.2005.1560157.

[Sha+15] Ankur Sharma, David Chinnery, Sarvesh Bhardwaj, and Chris Chu. Fast Lagrangian Relaxation Based Gate Sizing Using Multi-Threading. In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '15, pages 426–433.

[She+87] Jang-ping Sheu, Don L. Scharfetter, Ping-keung Ko, and Min-Chie Jeng. BSIM: Berkeley short-channel IGFET model for MOS transistors. In: *IEEE Journal of Solid-State Circuits* 22.4 (1987), pages 558–566. doi: 10.1109/JSSC.1987.1052773.

[Shy+88] Jyao-Min Shyu, Jack P. Fishburn, Al E. Dunlop, and Alberto L. Sangiovanni-Vincentelli. Optimization-based transistor sizing. In: *IEEE Journal of Solid-State Circuits* 23.2 (1988), pages 400–409. doi: 10.1109/4.1000.

[SIA13] Semiconductor Industry Association (SIA). *International Technology Roadmap for Semiconductors*. 2013.

[Sku97] Martin Skutella. Approximation Algorithms for the Discrete Time-cost Tradeoff Problem. In: *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA 1997, pages 501–508. doi: 10.1287/moor.23.4.909.

[SN90] Takayasu Sakurai and A. Richard Newton. Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. In: *IEEE Journal of Solid-State Circuits* 25.2 (1990), pages 584–594. doi: 10.1109/4.52187.

[Sve13] Ola Svensson. Hardness of Vertex Deletion and Project Scheduling. In: *Theory of Computing* 9.24 (2013), pages 759–781. doi: 10.4086/toc.2013.v009a024.

[Sze05] Christian Szegedy. Some Applications of the weighted combinatorial Laplacian. PhD thesis. University of Bonn, 2005.

[Tim12] Alexander Timmermeister. Exakte Algorithmen für das Gate Sizing (in German). Master's thesis. University of Bonn, 2012.

[Tra15] Vera Traub. Global Routing mit Delay-Beschränkungen (in German). Master's thesis. University of Bonn, 2015.

[Tre+04] Louise Trevillyan, David Kung, Ruchir Puri, Lakshmi N. Reddy, and Michael A. Kazda. An integrated environment for technology closure of deep-submicron IC designs. In: *IEEE Design & Test of Computers* 21.1 (2004), pages 14–22. doi: 10.1109/MDT.2004.1261846.

[TS02] Hiran Tennakoon and Carl Sechen. Gate Sizing Using Lagrangian Relaxation Combined with a Fast Gradient-Based Pre-Processing Step. In: *IEEE/ACM International Conference on Computer Aided Design*, ICCAD 2002, pages 395–402. doi: 10.1109/ICCAD.2002.1167564.

[TS08] Hiran Tennakoon and Carl Sechen. Nonconvex Gate Delay Modeling and Delay Optimization. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.9 (2008), pages 1583–1594. doi: 10.1109/TCAD.2008.927758.

[Vég12] László A. Végh. Strongly Polynomial Algorithm for a Class of Minimum-cost Flow Problems with Separable Convex Objectives. In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC 2012, pages 27–40. doi: 10.1145/2213977.2213981.

[Vyg01] Jens Vygen. Theory of VLSI layout. Habilitation. University of Bonn, 2001.

[Vyg06] Jens Vygen. Slack in static timing analysis. In: *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems* 25.9 (2006), pages 1876–1885. doi: 10.1109/TCAD.2005.858348.

[War+13] Samuel I. Ward, Natarajan Viswanathan, Nancy Y. Zhou, Cliff N. Sze, Zhuo Li, Charles J. Alpert, and David Z. Pan. Clock Power Minimization Using Structured Latch Templates and Decision Tree Induction. In: *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 599–606.

[WD08] Tai-Hsuan Wu and Azadeh Davoodi. PaRS: Fast and near-optimal grid-based cell sizing for library-based design. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2008, pages 107–111. doi: 10.1109/ICCAD.2008.4681559.

[WDZ07] Jia Wang, D. Das, and Hai Zhou. Gate sizing by lagrangian relaxation revisited. In: *IEEE/ACM International Conference on Computer-Aided Design*, ICCAD 2007, pages 111–118. doi: 10.1109/ICCAD.2007.4397252.

[WH10] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. 4th edition. Addison-Wesley Publishing Company, 2010. ISBN 9780321547743.

[Wit13] Sonja Wittke. Discrete Time-Cost Tradeoff Problems in Timing Optimization. Master thesis. University of Bonn, 2013.

[Wri15] Stephen J. Wright. Coordinate descent algorithms. In: *Mathematical Programming* 151(1) (2015), pages 3–34. doi: 10.1007/s10107-015-0892-3.

[XC15] Jiani Xie and C.Y. Roger Chen. Lookup Table Based Discrete Gate Sizing for Delay Minimization with Modified Elmore Delay Model. In: *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 361–366. doi: 10.1145/2742060.2742094.

[You01] Neal E. Young. Sequential and parallel algorithms for mixed packing and covering. In: *In 42nd Annual IEEE Symposium on Foundations of Computer Science*, pages 538–546.