Fast Repeater Tree Construction

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Vorgelegt von

Christoph Bartoschek

aus

Peiskretscham, Polen

Bonn, Mai 2014

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

1. Gutachter: Professor Dr. Jens Vygen
2. Gutachter: Professor Dr. Stephan Held
Tag der Promotion: 11. Juli 2014
Erscheinungsjahr: 2014

Danksagung/Acknowledgments

At this point, I would like to express my gratitude to my supervisors Professor Dr. Jens Vygen and Professor Dr. Stephan Held. This work would not be possible without their extensive support, inspiration, and extraordinary patience.

A very special thanks goes to Professor Dr. Dr. h.c. Bernhard Korte for his encouraging support and for creating such an excellent working environment at the Research Institute for Discrete Mathematics at the University of Bonn.

I would further like to thank my past and present colleagues at the institute working with me on timing optimization, especially Dr. Jens Maßberg, Professor Dr. Dieter Rautenbach, Daniel Rotter, Dr. Christian Szegedy and Dr. Jürgen Werber. Their support and ideas proved to be invaluable.

Special thanks go to all the students for their collaboration, especially Laura Geisen, Nicolas Kämmerling and Philipp Ochsendorf.

I also thank all other colleagues at the institute for inspiring discussions, in particular Dr. Ulrich Brenner, Christian Panten, Jan Schneider and Dr. Markus Struzyna.

I am grateful that I have been able to work with all the people from IBM who shared their knowledge and hardest chip designs with us, especially Karsten Muuss, Dr. Matthias Ringe, and Alexander J. Suess.

But my biggest thanks go to my wife Kerstin and my little daughters Johanna and Barbara. Without their support and endless patience, I would have never finished this thesis.

Contents

1 Introduction **9**

2 Timing Optimization – Basic Concepts **13**

2.1 Basic Notation 13
2.2 Integrated Circuit Design 13
2.3 Static Timing Analysis 14
2.4 Repeater 15
2.5 Wire Extraction 18
2.5.1 Elmore Delay 18
2.5.2 Higher Order Delay Models 19
2.6 Slew Limit Propagation 20
2.7 Required Arrival Time Functions 20
2.7.1 Propagation of Required Arrival Times 21

3 Repeater Tree Problem **23**

3.1 Repeater Tree Instances 23
3.2 The Repeater Tree Problem 25
3.2.1 Repeater Tree Timing 25
3.2.2 Feasible solutions 27
3.2.3 Objectives 27
3.3 Our Repeater Tree Algorithm 28

4 Instance Preprocessing **29**

4.1 Analysis of Library and Wires 29
4.1.1 Estimating two-pin connections 29
4.1.2 Parameter d_{wire} 34
4.1.3 Buffering Modes 34
4.1.4 Slew Parameters 36
4.1.5 Sinkdelay 37
4.1.6 Further Preprocessing 38
4.2 Blockage Map and Congestion Map 39
4.2.1 Grid 39
4.2.2 Blockage Map 40
4.2.3 Blockage Grid 40
4.2.4 Congestion Map 42

5 Topology Generation **43**
5.1 A Simple Delay Model . . . 44
5.1.1 Time Tree . . . 47
5.2 Repeater Tree Topology Problem . . . 48
5.2.1 Topology Algorithm Overview . . . 49
5.3 Restricted Repeater Tree Problem . . . 49
5.4 Sink Criticality . . . 49
5.5 A Simple Topology Generation Algorithm . . . 50
5.5.1 Topology Generation Algorithm . . . 50
5.5.2 Theoretical Properties . . . 52
5.6 Topology Generation Algorithm . . . 56
5.6.1 Handling High Fanout Trees . . . 58
5.7 Blockages . . . 59
5.8 Plane Assignment . . . 60
5.9 Global Wires as Topologies . . . 60

6 Repeater Insertion **63**
6.1 Computing Required Arrival Time Targets . . . 64
6.1.1 Linear Time-cost Tradeoff . . . 66
6.1.2 Effort Assignment Algorithm . . . 68
6.2 Repeater Insertion Algorithm . . . 69
6.2.1 Cluster . . . 69
6.2.2 Initialization . . . 70
6.2.3 Timing Model during Repeater Insertion . . . 70
6.2.4 Finding a new Repeater . . . 73
6.2.5 Buffering Algorithm . . . 75
6.2.6 Merging operation . . . 75
6.2.7 Moving operation . . . 77
6.2.8 Arriving at the root . . . 79
6.2.9 Running Time . . . 80
6.2.10 Repeater Insertion - Summary . . . 81
6.3 Dynamic Programming . . . 82
6.3.1 Basic Dynamic Programming Approach . . . 83
6.3.2 Buffering Positions . . . 84
6.3.3 Extensions to Dynamic Programming . . . 84

7 BonnRepeaterTree **89**
7.1 Repeater Library . . . 89
7.1.1 Repeater and Wire Analysis . . . 90
7.1.2 RAT and Slew Backwards Propagation . . . 90
7.2 Blockages and Congestion Map . . . 91
7.3 Processing Repeater Tree Instances . . . 92
7.3.1 Identifying Repeater Tree Instances . . . 92
7.3.2 Constructing Repeater Trees . . . 98

7.3.3 Replacing Repeater Tree Instances 99
7.4 Implementation Overview 100
7.4.1 Repeater Tree Construction Framework 100
7.4.2 Repeater Tree API 100
7.4.3 Parallelization 101
7.5 BonnRepeaterTree in Global Timing Optimization 101
7.6 BonnRepeaterTree Utilities 102
7.6.1 Removing Existing Repeaters 102
7.6.2 Postprocessing Repeater Chains 103

8 Experimental Results 109
8.1 Comparison to an Industrial Tool 110
8.2 Comparison to Bounds 111
8.2.1 Running Time 113
8.2.2 Wirelength 114
8.2.3 Number of Inserted Inverters 114
8.2.4 Timing 115
8.3 Fast Buffering vs. Dynamic Programming 116
8.4 Varying η 118
8.5 Varying d_{node} 119
8.6 Disabling Effort Assignment 119
8.7 Disabling Parallel Mode 121
8.8 Choosing Tradeoff Parameters 121

A Detailed Comparison Tables 125

B Bibliography 141

1 Introduction

We live in a world where computer chips can be found in nearly every device we come into contact with. On the one hand, there is a huge demand for powerful but not power-consuming chips that can be added to our wearables, houses or household items. On the other hand, there is still a huge demand for fast processors that are able to crunch the enormous amounts of data we produce daily.

Chip designers create the small miracles driving all of this in a process called *physical design*. It is an area where one of our most advanced technologies meets mathematics and computer science.

Physical design contains a lot of fascinating problems that can be tackled with methods from combinatorial optimization. This thesis focuses on one of the problems that arise during the optimization of the timing behaviour of a chip, the optimization of interconnections or repeater trees. *Interconnections* distribute signals from a source to one or several sinks. Figure 1.1 shows the interconnection between a source (blue) and four sinks (red). If interconnections get too long, the speed of signals degrades and electrical constraints get violated. To improve speed and to avoid electrical violations, *repeaters* can be inserted into a chip.

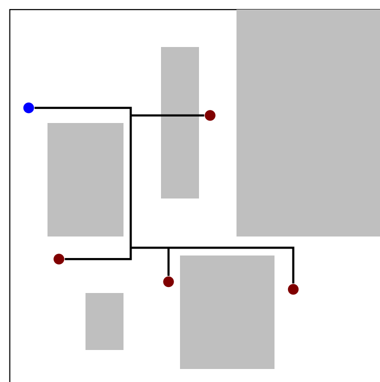


Figure 1.1: Example of an interconnection. A signal has to be distributed from a root (blue) to sinks (red).

There are two flavours of repeaters: buffers and inverters. Buffers are used to refresh a signal. Inverters have, in addition, the property that they change the polarity of a signal. A signal switching from a logical 0 to a 1 becomes a signal switching from 1 to 0 and vice-versa. The logical symbols of both repeater types and the schematic of an inverter are shown in Figure 1.2.

An interconnection distributes the signals in a tree-like fashion from a source to

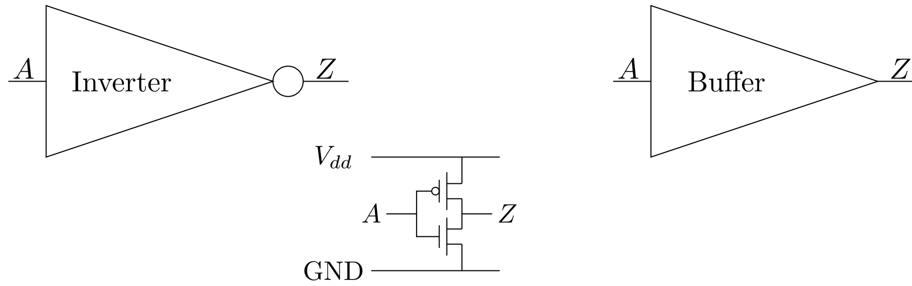


Figure 1.2: The symbols of inverters and buffers and a schematic of a simple inverter. It consists of two transistors with converse switching behaviour. GND (logical 0) and V_{dd} (logical 1) are power supplies. Both repeaters have a single input A and an output Z. If a 0 arrives, then only the gate to V_{dd} opens and vice-versa. A buffer is usually constructed by two inverters in series.

the sinks via wires. A Repeater is added into the interconnection by subdividing a wire segment and connecting the ends to the repeater's input and output. We call an interconnection that potentially has repeaters in between a *repeater tree*.

In the early years of physical design, interconnection optimization between gates was a minor task. Repeaters were only necessary for very long distances or very high fanouts. However, with the downscaling of technology, resulting in smaller gates and thinner wires, the ability of gates to drive long wires deteriorated and more repeaters became necessary to meet timing requirements. Saxena et al. (2003) predicted that for the 45 nm and 32 nm technology nodes 35 % and 70 % of all circuits on typical designs will be repeaters. In the meantime, the technology nodes have arrived and we see that the numbers are slightly better than predicted. We see that on 45 nm designs 25–40 %, on 32 nm designs 30–45 %, and on 22 nm designs 35–50 % of all circuits are repeaters.

Nevertheless, if 30 % of all gates are repeaters, repeater tree construction becomes an important task. It affects all aspects of physical design in addition to timing and electrical correctness. For example, repeaters represent a significant number of circuits that have to be placed and later connected by routing. A significant part of the power consumption can be attributed to repeaters. Modern designs have a wide range of instances with up to hundred thousands of sinks. We have seen designs with millions of instances. For large instances, running time becomes a crucial feature of a repeater tree construction algorithm.

The algorithms we present fit into all stages of physical design. A very fast algorithm that we call Fast Buffering can be used for rebuilding all instances globally in early and middle design stages. We can optimize around 5.7 million instances per hour on a single core and a couple of minutes in parallel. For later stages of physical design, a more accurate version of our algorithm can be enabled that is able to squeeze out the last tenth of a picosecond.

The structure of this thesis is as outlined in the following paragraphs. The basic concepts from timing optimization in chip design that we use are explained in

Chapter 2. We then define the Repeater Tree Problem in Chapter 3. Our problem formulation encapsulates most of the constraints that have been studied so far. To the best of our knowledge, we also consider several aspects for the first time, for example, slew dependent required arrival times at repeater tree sinks. These make our formulation more adequate to the challenges of real-world repeater tree construction.

For creating good repeater trees, one has to take the overall design environment into account. The employed technology, the properties of available repeaters and metal wires, the shape of the chip, the temperature, the voltages, and many other factors highly influence the results of repeater tree construction. To take all this into account, we first preprocess the environment to extract parameters for our algorithms. These parameters allow us to quickly and yet quite accurately estimate the timing of a tree before it has even been buffered. Chapter 4 shows how we extract them from the timing environment.

The next two chapters explain our algorithm to solve the Repeater Tree Problem. Chapter 5 shows how we construct an underlying Steiner tree for our solution. We prove that our algorithm is able to create timing-efficient as well as cost-efficient trees.

Chapter 6 deals with the problem of adding buffers to a given Steiner tree. The predominantly used algorithms to solve this problem use dynamic programming. However, they have several drawbacks. Firstly, potential repeater positions along the Steiner tree have to be chosen upfront. Secondly, the algorithms strictly follow the given Steiner tree and miss optimization opportunities. Finally, dynamic programming causes high running times. We present our new buffer insertion algorithm that overcomes these limitations. It is able to produce results with similar quality to a dynamic programming approach but a much better running time. In addition, we also present our improvements to the dynamic programming approach that allow us to push the quality at the expense of a high running time.

As part of this thesis, we implemented the discussed algorithms as a module within the BonnTools optimization suite which is developed at the Research Institute for Discrete Mathematics at the University of Bonn in an industrial cooperation with IBM. BonnTools are used by chip designers worldwide within IBM. Some implementation details will be described in Chapter 7. Our algorithms are used and help engineers dealing with some of the most complex chips in the world. In the same chapter, we also shortly describe the framework we have written that makes it easy to implement new repeater tree construction algorithms. As an example, we will show how routing congestion on a chip can be reduced by rerouting repeater chains.

Our cooperation partner IBM provided us with a large number of real world chip designs. For this thesis we have chosen a set of twelve challenging chips with a total number of more than 3.3 million different repeater tree instances. In Chapter 8, we present experimental results that show the quality and speed of our algorithms.

2 Timing Optimization – Basic Concepts

2.1 Basic Notation

We use the same notation for graphs as Korte and Vygen (2012).

For an arborescence (i.e. a directed tree where each node except for the root has exactly one entering edge) $T = (V, E)$ with nodes $v, w \in V$ we denote the edges on the path from v to w by $E_{[v,w]}$. The parent of node $v \in V$ is called $parent(v)$.

For a set $X \subset \mathbb{R}^n$, we define the componentwise maximum

$$\max_{x \in X} x := \left(\max_{x \in X} x_1, \max_{x \in X} x_2, \ldots, \max_{x \in X} x_n \right)$$

and the componentwise minimum

$$\min_{x \in X} x := \left(\min_{x \in X} x_1, \min_{x \in X} x_2, \ldots, \min_{x \in X} x_n \right).$$

As current technologies prefer to route wires only in horizontal and vertical direction, we almost exclusively use the ℓ_1-norm:

$$\|a\| := \|a\|_1$$

2.2 Integrated Circuit Design

The *netlist* of a chip design consists of *primary pins* (input or output), *gates* and *nets*. Gates are circuits computing small Boolean functions, for example, NOT, AND, or macros encapsulating larger functionality. Gates have pins, input pins or output pins, as external connection points. Pins, primary pins and gate pins, are connected by nets. Typically a net has a single source (gate output pin or primary input pin) and a set of sinks (gate input pin or primary output pin). If a gate's output pin is the source of a net, then we say that the gate is the *driver* of the net. *Physical design* is the phase in the process of creating a chip where a netlist that has just been compiled from a hardware description language is mapped to a *chip image*, typically a rectangular area with free space for gates. During physical design, gates get placed on the *chip image*. Then, optimizations are performed without changing the logical function of the design to improve the timing behaviour. One such operation is rebuilding repeater trees. Finally, the pins of each net get connected with wires by a routing tool. Modern chip designs and technologies are so challenging that placement, timing optimization, and routing have to be considered together most of the time. An introduction into the optimization steps of modern physical design is given by Held (2008).

2.3 Static Timing Analysis

We use the concepts of *static timing analysis* (Hitchcock et al., 1982) that is based on the critical path method (Kelley and Walker, 1959). A thorough introduction to timing analysis is given by Sapatnekar (2004). In this thesis only the following concepts are important.

The voltage at a given point of a chip compared to ground defines the logical state. The voltage V_{dd} represents a logical 1 and GND (ground) represents a 0.

A signal is defined as the change of the voltage over time. A *rising* signal describes a change from GND to V_{dd}. On the other hand, a *falling* signal describes the change from V_{dd} to GND. We call the direction the signal's *edge*. The possible edges are r (*rise*) and f (*fall*). We define the inversion of an edge as

$$f^{-1} := r, \qquad r^{-1} := f.$$

In static timing analysis signals are measured at certain points of the design that are also called *timing points*. For most of this discussion, it is sufficient to restrict ourselves to gate pins and primary pins as timing points. In addition, some gates have internal timing points. We only have to consider them when we work with real chip designs (Chapter 7). A signal is estimated by a piecewise linear function that is given by the *arrival time* and *slew* of the signal. Usually, the arrival time of a rising or falling signal is given as the time when the voltage change reaches 50 %. Similarly, the slew is given as the time between 10 % and 90 % of the voltage change.

Static timing analysis makes worst case assumptions and computes at each measurement point an *early* and a *late* signal. A real signal will arrive after the early signal and before the late signal.

At certain timing points the arrival times of signals are compared to the arrival times of other signals or design-specific constants. This imposes constraints on the signals. For example, signals are not allowed to be too fast (their early arrival time has to be high enough) or too slow (the late arrival time has to be small enough) when they arrive at registers[1] compared to clock signals. Repeaters are used to slow down signals that are too fast and they are also used to speed up signals. In this thesis, we only consider the problem of speeding up signals to meet requirements on the late arrival time. As we are not interested in the early arrival times of signals, we will ignore them for the remainder of this thesis and only work with late signals.

Transistors show different characteristics for rising and falling signals depending on their size or the technology. Gates show asymmetric behaviour depending on the edge of the incoming signal. Therefore, timing analysis computes the arrival times and slews for both signals separately and stores them in time pairs, one value for each signal edge.

Definition 1. A time pair is a tuple $(rise, fall) \in \mathbb{R}^2$ of time values. For a given time pair $t = (rise, fall)$ we define $t^r := rise$ and $t^f := fall$.

[1]Registers are gates that store information between clock cycles.

At each timing point, a signal is given by an arrival time time pair (we often write arrival time pair) and a slew time pair (we often write slew pair):

$$(at^r, at^f), (slew^r, slew^f).$$

The measurement points are connected by directed *propagation arcs*. Timing nodes and propagation arcs form the *timing graph*. For a net, there is a propagation arc for each sink pin p connecting the source of the net to p. For gates, there are technology dependent rules specifying which internal timing points and arcs are added to the timing graph. Propagation arcs are also called propagation *segments*. Repeaters normally have a single propagation arc from their input pin to their output pin. The propagation arc within an inverter is an inverting propagation arc; a rising (falling) signal edge at the input becomes a falling (rising) signal edge at the output. Buffers have, similar to nets, a non-inverting propagation arc; a rising (falling) signal edge at the input becomes a rising (falling) signal edge at the output.

The difference between the arrival time of a signal at the head of a propagation arc and the arrival time at the tail is called *delay*.

The computer program that computes arrival times and slews for all signals on all timing nodes is called *timing engine*.

2.4 Repeater

A repeater t is characterized by

- its logical function (buffers implement the identity function, inverters implement the negation),
- an input pin t_a and an output pin t_z with pin capacitances $cap^{\text{in}}(t)$ and $cap^{\text{out}}(t)$,
- its delay function $delay_t$,
- its slew function $slew_t$, and
- its leakage power consumption $pwr(t)$.

The logical function determines whether a signal propagating through a repeater is inverted or not. Inverters change, according to their propagation arcs, a rising signal into a falling signal and vice-versa. Buffers do not change the edge of a signal. Given a signal edge $* \in \{r, f\}$, we define

$$t(*) := \begin{cases} *^{-1} & \text{if } t \text{ is an inverter} \\ * & \text{if } t \text{ is a buffer.} \end{cases}$$

Each repeater is driver of the net connected to its output pin. The sum of pin and wire capacitances visible from the output pin, including $cap^{\text{out}}(t)$, is called *load capacitance* or *load*.

The functions $delay_t$ and $slew_t$ are called *timing functions*. The delay function computes the delay over the internal propagation arc of a repeater. It depends on

the load at the output pin and the slew pair at the input pin. The function is given by

$$\begin{aligned} delay_t &: [0, loadlim(t)] \times [0, slewlim(t)]^2 \to \mathbb{R}^2 \\ delay_t(l,s) &:= \left(delay_t^r(l,s^r), delay_t^f(l,s^f\right) \end{aligned}$$

where, for each signal edge $* \in \{r, f\}$, there is a function

$$delay_t^* : [0, loadlim(t)] \times [0, slewlim(t)] \to \mathbb{R}.$$

As inverters change the signal edge, we have to distinguish between inverters and buffers when we want to add delays to a given arrival time pair. For time pair a, load l and slew pair s, we use the function

$$update_t : \mathbb{R}^2 \times [0, loadlim(t)] \times [0, slewlim(t)]^2 \to \mathbb{R}^2$$

$$update_t(a,l,s) := \begin{cases} \left(a^f + delay_t^f(l,s^f), \quad a^r + delay_t^r(l,s^r)\right) & \text{if } t \text{ is an inverter} \\ \left(a^r + delay_t^r(l,s^r), \quad a^f + delay_t^f(l,s^f)\right) & \text{if } t \text{ is a buffer.} \end{cases}$$

Similarly, the slew function determines the slew at the output pin of the repeater depending on the slew pair at the slew pin and the load at the output pin. It is given by

$$slew_t : [0, loadlim(t)] \times [0, slewlim(t)]^2 \to \mathbb{R}^2_{\geq 0}$$

$$slew_t(l,s) = \begin{cases} \left(slew_t^f(l,s^f), \, slew_t^r(l,s^r)\right) & \text{if } t \text{ is an inverter} \\ \left(slew_t^r(l,s^r), \, slew_t^f(l,s^f)\right) & \text{if } t \text{ is a buffer,} \end{cases}$$

and, for each input signal edge $* \in \{r, f\}$, there is the function

$$slew_t^* : [0, loadlim(t)] \times [0, slewlim(t)] \to \mathbb{R}_{\geq 0}$$

computing the slew for the output signal edge $t(*)$.

Each repeater has a pair $slewlim(t)$ of limits on the maximum rising respectively falling slew associated with the input pin and a maximum output load limit $loadlim(t)$ associated with the output pin. Both define the domain of the timing functions. There are also tiny lower limits on the possible load and slews to the delay and slew functions. However, they are not relevant in practice. The minimum load limit can only be violated by unconnected pins and the lower slew limits are so steep that it is not possible to reach them within the corresponding technology. We therefore ignore them and use 0 for lower limits. The delay and slew functions as well as the pin capacitances and limits are called *timing rules*.

Especially in the early design stages, it is often not possible to keep the slews and loads within the domain of the load and slew functions. Such a condition is called

electrical violation. The timing engine extrapolates the timing functions in such a case to work with somehow reasonable values.

Generally, we do not make many restricting assumptions on the timing functions of a repeater. There might be small deviations from the following, but we can assume that both functions are strictly monotonically increasing for each input. For example, if for a fixed load the input slews are increased, then the delays and output slews also increase.

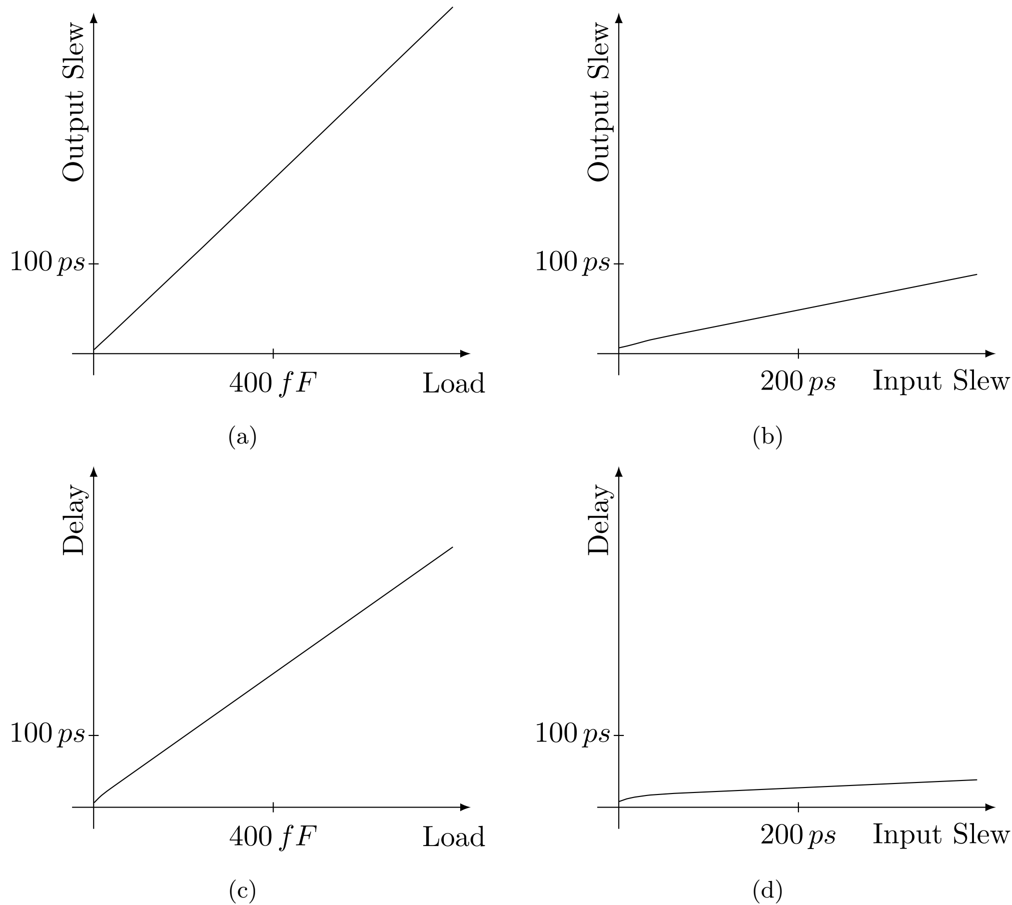


Figure 2.1: Delays and slews of an example repeater depending on the input slew and load. Figures (a) and (c) show the results of $slew_t^r$ and $delay_t^r$ for fixed input slew and different load capacitances. Figures (b) and (d) show the results of $slew_t^r$ and $delay_t^r$ for fixed load capacitance but different input slews. In all cases the rise-fall transition of an inverter is shown.

We use the leakage power consumption of repeaters as costs. Repeaters also cause dynamic power consumption that depends on the switching behaviour of the circuit. Circuits that switch often consume more power than circuits that keep their state. The power consumption also depends on the capacitance of the capacitors that change voltage. The dynamic power consumption is roughly linear to the total

capacitance of a circuit. As all circuits in a repeater tree show the same switching patterns, we can basically expect that shorter trees use less dynamic power.

2.5 Wire Extraction

Given a net, we are interested in the delay between the source of the net and each sink. The process of computing the delays and slew changes is called *wire extraction*. There are several different approaches to model the timing behaviour of nets. The most dominant one is shown in the next section.

2.5.1 Elmore Delay

The most commonly used delay model for nets in works on interconnection optimization is the Elmore delay because it is easy to calculate and gives good results compared to earlier approaches[2].

Elmore (1948) proposed to estimate the delay of a monotonic step response of a circuit by using the mean of the impulse response. It can be shown that the Elmore delay is an upper bound on the actual 50 % delay to a sink. Rubinstein et al. (1983) showed how to compute the Elmore delay on an RC-tree in linear time. An *RC-tree* describes the physical properties of the wires of a net. A wire segment e is modeled using the π-model, that is, a resistance segment r_e between two capacitors $c_e/2$. Here, r_e is the resistance of the wiring segment and c_e is its capacitance.

At the first design stages there are often no wires that can be used for RC-tree generation. To estimate the delay over a net, it is common to compute a Steiner tree S and use it as the RC-tree with default resistances and capacitances for the edges of the Steiner tree.

We assume that the tree is oriented away from the input pin of the net, that each vertex is assigned a position in the plane, and that the edges are embedded at the shortest paths between their nodes. On a path $E(S)_{[a,b]}$ from vertex a to vertex b, the Elmore delay rc is calculated by:

$$rc_S = \sum_{e \in E(S)_{[a,b]}} r_e \left(\frac{c_e}{2} + downcap(e) \right).$$

The downward capacitance $downcap(e)$ is the sum of all wiring and pin capacitances reachable from e in the oriented tree.

Both r_e and c_e are proportional to the length of edge e. The resulting rc value is therefore quadratic in the length.

The Elmore delay approximates the response of a net to a step excitation. It is used as a raw value, often called *RC-delay*, that is merged with environmental variables into a delay function and a slew function that compute the response of

[2]See Pileggi (1995) for a description of some earlier models.

the net to a skewed input signal. We assume that the timing engine provides a *wiredelay* function

$$wiredelay : \mathbb{R} \times \mathbb{R}_{\geq 0} \to \mathbb{R}$$

and a *wireslew* function

$$wireslew : \mathbb{R} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}.$$

Given a wire segment with Elmore delay rc and a slew s_{in} at the segment's start, $wiredelay(rc, s_{in})$ computes the delay over the wire and $wireslew(rc, s_{in})$ computes the slew at the end. Both functions are sometimes linear, but often they cannot be calculated by a simple expression. The functions are independent from the signal edge. To simplify the calculation of delays and slews for time pairs, we define the combinations *wiredelay* and *wireslew*:

$$\begin{aligned}
wiredelay &: \mathbb{R} \times \mathbb{R}^2_{\geq 0} \to \mathbb{R}^2 \\
wiredelay(rc, (slew^r, slew^f)) &= (wiredelay(rc, slew^r), wiredelay(rc, slew^f)) \\
wireslew &: \mathbb{R} \times \mathbb{R}^2_{\geq 0} \to \mathbb{R}^2 \\
wireslew(rc, (slew^r, slew^f)) &= (wireslew(rc, slew^r), wireslew(rc, slew^f)).
\end{aligned}$$

The Elmore delay has the property that splitting up arbitrary nodes in the tree does not change the result. For example, if we have an edge (a, b) and split it with node c then

$$rc_{(a,b)} = rc_{(a,c)} + rc_{(c,b)}.$$

However, we do not assume that the same holds for *wiredelay* or *wireslew*. The following equations are not necessarily true for an input slew s_{in}:

$$\begin{aligned}
wiredelay(rc_{(a,b)}, s_{in}) &= wiredelay(rc_{(a,c)}, s_{in}) \\
&\quad + wiredelay(rc_{(c,b)}, wireslew(rc_{(a,c)}, s_{in}))) \\
wireslew(rc_{(a,b)}, s_{in}) &= wireslew(rc_{(c,b)}, wireslew(rc_{(a,c)}, s_{in})).
\end{aligned}$$

This means that in general we cannot compute delays and slews segment by segment and just stitch them together. Instead, we have to calculate the Elmore delay for the whole net first before we ask for the total delays and output slews by using the black-box functions *wiredelay* and *wireslew*.

2.5.2 Higher Order Delay Models

The Elmore delay is popular for timing optimization because of its simplicity. However, it is too pessimistic for some applications. Timing engines typically support more accurate delay models in addition to Elmore delay.

While we focus on the Elmore delay in our discussion, the operations where we extract a net and compute the delay and slew degradation are independent from the delay model. With small modifications and by using black-box functions that

return delays and slews for a given net, it would be possible to create versions of our algorithm that work on more accurate delay models. Until now, we refrained from doing so because we expect only a small improvement in quality to our solutions that would be paid by a drastically increased running time.

A description of higher order delay models can be found in Sapatnekar (2004) or Alpert et al. (2008), p. 546ff.

2.6 Slew Limit Propagation

In the context of repeater trees, only input pins of gates have slew limits. However, if one has to choose a gate that drives a net, one often wants to know the maximum slew that may arrive at the source of the net such that the limit is not violated for each sink of the net. We assume that there exists a function

$$slewinv : \mathbb{R} \times \mathbb{R}^2_{\geq 0} \to \mathbb{R}^2_{\geq 0}$$

that computes the maximum slew limit at net sources. For a sink s with a slew limit pair $slewlim(s)$ and RC-delay rc_s the pair of maximum slews that are allowed at the source of the net such that the sink's limits are not violated is $slewinv(rc_s, slewlim(s))$. Sometimes it is not possible to obey a sink's slew limit because it is too tight or the net is too long. In such cases, we assume that $slewinv$ returns 0 for the corresponding signal edge.

For a net with sink set S, the maximum allowable slews at the source are the componentwise minimum values over all slew limits:

$$slewlim := \min_{s \in S}\{slewinv(rc_s, slewlim(s))\}.$$

We ask the same question for each repeater: What is the highest slew that may arrive at the input pin such that the output slew is below a certain limit for a given load capacitance? We assume that for each repeater t a function

$$slewinv_t : \mathbb{R}_{\geq 0} \times \mathbb{R}^2_{\geq 0} \to \mathbb{R}^2_{\geq 0}$$

is given. For a slew limit pair $slewlim$ at the output pin and a load capacitance $load$, the pair of highest allowable slews at the input pin is $slewinv_t(load, slewlim)$.

2.7 Required Arrival Time Functions

As indicated above, there are constraints on the arrival times. If a signal is not allowed to arrive too late at a timing point, then the latest feasible arrival time is called *required arrival time* (RAT). We define required arrival times for all timing nodes even if they have no direct arrival time constraints. For a timing point v, the required arrival time for a signal is the latest arrival time such that for all timing nodes reachable from v the arrival time constraints are met. As the delays

of subsequent propagation segments depend on the slew at v, the required arrival time is a function of slews.

For each timing point, there is a *RAT function*

$$\begin{aligned} rat &: \mathbb{R}^2_{\geq 0} \to \mathbb{R}^2 \\ rat(slew) &= \left(rat^r(slew^r), rat^f(slew^f)\right) \end{aligned}$$

with rat^r, rat^f being edge-specific RAT functions. For a signal edge $* \in \{r, f\}$, we have

$$rat^* : \mathbb{R}_{\geq 0} \to \mathbb{R}.$$

Given arrival time a^* and slew s^* for signal edge $* \in \{r, f\}$, the required arrival time constraint at a point is feasible if

$$a^* \leq rat^*(s^*).$$

The *slack* at the point is defined as

$$\sigma^* := rat^*(s^*) - a^*.$$

The timing of a netlist is clean if the slacks are non-negative for all constraints on timing points.

2.7.1 Propagation of Required Arrival Times

Given a net, the RAT function at its source r depends on the net topology and the RAT functions at the sinks. Given a sink s with RAT function rat_s, we can ask for a function rat_r such that arrival times and slews are feasible at s if they are feasible at r. Delay and slew over a wire segment only depend on the Elmore delay and the input slew. We therefore assume there is function

$$\begin{aligned} ratinv &: \mathbb{R} \times (\mathbb{R}^{\mathbb{R}_{\geq 0}} \times \mathbb{R}^{\mathbb{R}_{\geq 0}}) \to \mathbb{R}^{\mathbb{R}_{\geq 0}} \times \mathbb{R}^{\mathbb{R}_{\geq 0}} \\ ratinv\Big(rc, (rat^r, rat^f)\Big) &= \Big(ratinv^r(rc, rat^r), ratinv^f(rc, rat^f)\Big) \end{aligned}$$

and for each edge $* \in \{r, f\}$ a function

$$ratinv^* : \mathbb{R} \times \mathbb{R}^{\mathbb{R}_{\geq 0}} \to \mathbb{R}^{\mathbb{R}_{\geq 0}}$$

such that, for every slew $slew$ and Elmore delay rc_s,

$$ratinv^*(rc_s, rat^*_s)(slew^*) = rat^*_s(wireslew(rc_s, slew^*)) - wiredelay(rc_s, slew^*).$$

At the source of multi-pin nets, we have to take the minimum over all RAT functions coming from the sinks to be sure that arrival times are feasible at all sinks. The minimum is the function

$$rat_r := \min_{s \in S} ratinv(rc_s, rat_s)$$

such that for each edge $* \in \{r, f\}$ and all slews $slew \in \mathbb{R}_{\geq 0}$

$$rat_r^*(slew) = \min_{s \in S} ratinv^*(rc_s, rat_s)(slew).$$

If we have to compute the minimum over a set of RAT functions, the result is the lower contour of input RAT functions. In practice, however, we always approximate the lower contour by a linear function.

Similarly, we assume that for each repeater t there is a function $ratinv_t$ giving us the RAT function at the input pin for a RAT function at the output pin. Assume that t drives a net with load capacitance $load$ and that the RAT function at the output pin is rat. We then have a function

$$ratinv_t : \mathbb{R}_{\geq 0} \times (\mathbb{R}^{\mathbb{R}_{\geq 0}} \times \mathbb{R}^{\mathbb{R}_{\geq 0}}) \to \mathbb{R}^{\mathbb{R}_{\geq 0}} \times \mathbb{R}^{\mathbb{R}_{\geq 0}}$$
$$ratinv_t\Big(load, (rat^r, rat^f)\Big) = \Big(ratinv_t^r(load, rat^{t(r)}), ratinv_t^f(load, rat^{t(f)})\Big)$$

and for each edge $* \in \{r, f\}$ a function

$$ratinv_t^* : \mathbb{R} \times \mathbb{R}^{\mathbb{R}_{\geq 0}} \to \mathbb{R}^{\mathbb{R}_{\geq 0}}$$

such that, for every slew pair $slew$ and load capacitance $load$,

$$ratinv_t^*(load, rat_s^*)(slew^*) = rat_s^{t(*)}(slew_t^*(load, slew^*)) - delay_t^*(load, slew^*).$$

Note that for inverters the rise RAT function at the input pin is determined by the fall RAT function at the output pin and vice-versa.

3 Repeater Tree Problem

3.1 Repeater Tree Instances

An instance of the Repeater Tree Problem consists of

- a root r, its location in the plane $Pl(r) \in \mathbb{R}^2$, a root arrival time function $at_r : \mathbb{R}_+ \to \mathbb{R}^2$, a root slew function $slew_r : \mathbb{R}_+ \to \mathbb{R}^2$, a pin capacitance cap^{out}, and a capacitance limit $loadlim(r)$,
- a set S of sinks, and for each sink $s \in S$ its parity $par(s) \in \{+, -\}$, its location in the plane $Pl(s) \in \mathbb{R}^2$, input capacitance $cap^{\text{in}}(s)$, a RAT function rat_s, and a pair of slew limits $slewlim(s)$,
- a set L of repeaters with timing rules $delay_t$, $slew_t$, $loadlim(t)$, $slewlim(t)$, $cap^{\text{in}}(t)$ and $cap^{\text{out}}(t)$ for each repeater $t \in L$,
- a set A of rectangles defining blocked areas,
- a global routing graph,
- a set W of wiring modes, and
- timing functions $wiredelay$ and $wireslew$.

Root

The root r is typically an output pin of a circuit or a primary input pin of the netlist. The root arrival time (resp. slew) function computes a time pair of arrival times (resp. slews) at the root pin for a given load capacitance.

Sinks

Sinks are usually primary outputs or input pins of circuits that are not repeaters. The parity determines how many inverting repeaters are required on root-sink paths. The number of inversions on the path from the root to the sink must be even (odd) if the sink has parity $+$ ($-$). We say a sink is positive (negative) if it has parity $+$ ($-$).

Most formulations of the Repeater Tree Problem assume fixed required arrival times at the sinks. In practice, required arrival times depend on the slews that arrive at the pins[1]. Higher slews cause higher delays in following stages and

[1] See Section 2.7.

reduce the required arrival times. The first propagation segment after a sink has the highest impact on the delays introduced by higher slews. If the sink is a circuit, its delay function often shows nearly linear behaviour that can be captured by a linear function. Effects on subsequent propagation segments are much smaller and can be neglected.

For sinks that are primary outputs or other nodes where required arrival times are created, the RAT function is often constant.

The slew limit is determined by the timing rules of the sink and global parameters.

Blockages and Congestion

Information about areas of the design that are blocked for repeater insertion is given in a blockage map. Basically, the blockage map is a set of rectangles.

Similarly, congestion on the wiring layers is passed to the repeater tree routine via a global routing graph. Both data structures are described in Section 4.2.

Wiring Modes

We assume to have a fixed number of wiring modes, each of which corresponds to a type of wire we can route on a plane. A wiring mode w is a 4-tuple $(p, width(w), wirecap(w), wireres(w))$ consisting of

- a routing plane p,
- routing space consumption $width(w)$,
- capacitance per unit length $wirecap(w)$, and
- resistance per unit length $wireres(w)$.

The routing space consumption depends on the wire width and the necessary spacing to neighboring wires and is used to update the congestion map. Usually, there is one wiring mode per usable plane of a design.

Given a wire segment ws with mode w, the total capacitance and resistance of the wire are linear in its length $l(ws)$:

$$
\begin{aligned}
cap(ws) &:= wirecap(w) \cdot l(ws) \\
res(ws) &:= wireres(w) \cdot l(ws).
\end{aligned}
$$

There are two default wiring modes, one on a horizontal layer w_h^* and one on a vertical layer w_v^*, that are used for the bulk of the wires in the design. Typically, they are the least expensive wiring modes in terms of routing space consumption and most expensive in terms of delay. As routers are free to choose higher planes than the plane of the assigned wiring mode and as higher planes typically mean better timing, the assignment of the default wiring modes is a pessimistic choice.

Timing Rules

For each repeater t, the functions $delay_t$ and $slew_t$ are given with the according electrical limits $loadlim(t)$ and $slewlim(t)$. The capacitance of its input pin is $cap^{\text{in}}(t)$ and it is $cap^{\text{out}}(t)$ for its output pin.

For nets, *wiredelay* is the timing rule that computes the delay of the timing engine for a given RC-delay and input slew. Similarly, *wireslew* computes the slew for a given RC-delay and input slew.

3.2 The Repeater Tree Problem

The Repeater Tree Problem is the task of computing a repeater tree for a given repeater tree instance. We first define what a repeater tree is:

Definition 2. A *repeater tree* R for a Repeater Tree Problem instance is a tuple (T, Pl, R_t, R_W). It consists of

- an arborescence $T = (V(T), E(T))$ with $V(T) = \{r\} \dot\cup S \dot\cup I_r \dot\cup I_s$ rooted at r, leaves S, and inner nodes $I_r \cup I_s$ (nodes corresponding to repeaters and steiner nodes),
- an embedding of the nodes into the plane $Pl : V(T) \to \mathbb{R}^2$,
- a repeater assignment function $R_t : V(T) \to L \cup \{\emptyset\}$ with $R_t(v) \in L$ iff $v \in I_r$, and
- a wiring mode assignment function $R_W : E(T) \to W$.

The root and leaves of a repeater tree are the root and sinks of the corresponding repeater tree instance. When a repeater tree is inserted into a chip design, for each node $v \in I_r$, a new repeater gate with type $R_t(v)$ is created and placed at $Pl(v)$. Root, sinks, and new repeaters are connected with nets that consist of the edges between the corresponding nodes. The nodes I_s and their incident edges determine the topology of the nets. There is exactly one net for each node in $\{r\} \cup I_r$.

3.2.1 Repeater Tree Timing

Given a repeater tree $R = (T, Pl, R_t, R_W)$ for a repeater tree instance, we have to compute the arrival times and slews at the sinks to compare them with required arrival times. To achieve this, we have to extract the nets between the involved pins and compute delays over repeaters and nets.

For each node $v \in V(T)$, let T_v be the maximal subtree rooted at v such that all its inner nodes are in I_s. We say T_v is a net iff $v \in \{r\} \cup I_r$. The sinks of the net are the leaves of T_v.

Given a sink $a \in V(T_v)$ for a net rooted at v, let $p(a) := v$ be the root node of the net. On the one hand, each repeater node $v \in I_r$ is root of the net T_v, and on the other hand, it is a sink in the net $T_{p(v)}$.

We first compute the Elmore delay to each net's sink. Given an edge (v, w), its capacitance and resistance are

$$\begin{aligned} cap((v,w)) &:= wirecap(R_W((v,w)))||Pl(v) - Pl(w)|| \\ res((v,w)) &:= wireres(R_W((v,w)))||Pl(v) - Pl(w)||. \end{aligned}$$

Given an edge $(w, y) \in E(T_v)$, the capacitance visible downwards is the sum of all edge capacitances in the subtree rooted at y and the input pin capacitances of reachable sinks:

$$downcap((w,y)) := \sum_{(a,b)\in E(T_y)} cap((a,b)) + \sum_{\substack{a\in V(T_y)\\ \delta^+(a)=0}} cap^{\text{in}}(R(a)).$$

For the root node and the internal repeaters, the load capacitance is the sum of visible capacitances:

$$\begin{aligned} &load : I_r \cup \{r\} \to \mathbb{R} \\ &\quad load(x) = cap^{\text{out}}(x) + \sum_{e\in\delta^+(x)} downcap(e). \end{aligned}$$

Now we can compute the Elmore delay rc for each sink:

$$\begin{aligned} &rc : I_r \cup S \to \mathbb{R} \\ &rc(x) = \sum_{e\in E(T_{p(x)})_{[p(x),x]}} res(e)\left(\frac{cap(e)}{2} + downcap(e)\right). \end{aligned}$$

The next step is to propagate the slews and the arrival times from the root to the instance sinks. We define the slew recursively distinguishing between the input slew, $slew_i$, for input pins and output slew, $slew_o$, for output pins:

$$\begin{aligned} &slew_o : I_r \cup \{r\} \to \mathbb{R}^2 \\ &\quad slew_o(v) = \begin{cases} slew_r(load(v)) & v = r \\ slew_{R(v)}(load(v), slew_i(v)) & v \neq r \end{cases} \\ &slew_i : I_r \cup S \to \mathbb{R}^2 \\ &\quad slew_i(v) = wireslew(rc(v), slew_o(p(v))). \end{aligned}$$

In a similar way, the arrival times at inputs (at_i) and output pin (at_o) are defined as

$$\begin{aligned} &at_o : I_r \cup \{r\} \to \mathbb{R}^2 \\ &\quad at_o(v) = \begin{cases} at_r(load(v)) & v = r \\ update_{R(v)}(at_i(v), load(v), slew_i(v)) & v \neq r \end{cases} \\ &at_i : I_r \cup S \to \mathbb{R}^2 \\ &\quad at_i(v) = at_o(p(v)) + wiredelay(rc(v), slew_o(p(v))). \end{aligned}$$

The *slack* of the repeater tree is now

$$slack(T) := \min_{s \in S} \min\{rat^r_s(slew^r_i(s)) - at^r_i(s),\ rat^f_s(slew^f_i(s)) - at^f_i(s)\}.$$

We also define the static power consumption of the tree

$$power(T) := \sum_{v \in I_r} pwr(R(v))$$

and its length

$$length(T) := \sum_{(v,w) \in E(T)} ||Pl(v) - Pl(w)||.$$

The length roughly correlates with the dynamic power consumption of a repeater tree.

3.2.2 Feasible solutions

A repeater tree is feasible if internal nodes I_r are legally placed and connected in such a way that the signals arrive at each sink with the correct parity. Repeaters are placed legally if their position is not marked as blocked in the blockage map:

$$Pl(v) \notin A \qquad \forall v \in I_r.$$

Note that we ignore overlaps between repeaters and other gates.

Furthermore, we have to obey capacitance and slew limits everywhere:

$$\begin{aligned} load(r) &\leq loadlim(r) & \\ load(v) &\leq loadlim(R(v)) & \forall v \in I_r \\ slew_i(v) &\leq slewlim(R(v)) & \forall v \in I_r \\ slew_i(s) &\leq slewlim(s) & \forall s \in S. \end{aligned}$$

The timing of the repeater tree is feasible if $slack(T) \geq 0$.

3.2.3 Objectives

Among the repeater trees satisfying the above conditions, one typically searches a tree with the smallest power consumption.

In practice, however, it is often not possible to achieve a positive slack. It might also not be possible to get a solution that has no electrical violations. In such cases, our first objective is to minimize the sum of electrical violations. The second objective is to maximize $\min\{0, slack\}$ followed by minimizing power. In addition, we seek solutions minimizing the use of wiring resources.

It is often desirable to balance between the two main objectives, timing and wirelength. To this end, we introduce a parameter $\xi \in [0, 1]$, indicating how timing-critical a given instance is. For $\xi = 1$, we primarily optimize the worst slack, and we

optimize wirelength for $\xi = 0$. We consider the other objective only in case of ties. In practice, however, we mainly use values of ξ that are strictly between 0 and 1.

The Repeater Tree Problem is NP-hard because it contains the Steiner Minimum Tree Problem if one just wants to minimize ℓ_1-netlength (see Garey and Johnson (1977)). In addition, delay and slew functions are non-linear leading to further difficulties.

A good overview of existing approaches to solve the Repeater Tree Problem can be found in Alpert et al. (2008), Chapter 24–28. For an older discussion see Cong et al. (1996).

3.3 Our Repeater Tree Algorithm

We present our algorithm for the Repeater Tree Problem, that we call Fast Buffering, in the next chapters. Most repeater tree instances are build in the same environment with the same repeater library, blockages, global routing, wiring modes, and timing functions. We therefore spend some time to preprocess the environment and to compute parameters that allow us to perform further steps efficiently. The preprocessing step is explained in Chapter 4.

A common approach to the Repeater Tree Problem is to divide it into two steps, Steiner tree generation (also called *topology generation*) and repeater insertion (also called *buffering*). Our algorithm takes the same route. A main reason is that a) for some applications the topology is fixed and we only have to do repeater insertion, and b) other applications only need a timing-aware topology. Thus, our topology generation algorithm can be used for different applications, for example, the optimization of symmetric fan-in trees[2]. Similarly, our buffering algorithm can work on topologies from other sources (e.g. routing). The division allows us to exchange one algorithm without touching the other. The algorithms are independent from each other, but, on the one hand, we already consider expected results from repeater insertion during topology creation by using a delay model for estimation that tightly matches buffering results[3], and, on the other hand, we allow our buffering algorithm to modify the input topology if it is suitable.

Chapter 5 explains how we create topologies, and in Chapter 6 we show how we buffer them.

[2]Symmetric fan-in trees compute symmetrical functions with n inputs. They are reverse to repeater trees. Signals from n sources are merged into a single output.

[3]See Figure 5.3 and the surrounding discussion.

4 Instance Preprocessing

4.1 Analysis of Library and Wires

We compute some auxiliary data and parameters in advance, which are then used for all instances of a design. It is possible to precompute parameters for a single global optimization run because the environment does not change and because technology, library, and wire types are the same for a lot of instances.

On the other hand, it is not possible to precompute useful data between several runs because the environment changes too often. Due to different timing rules, voltages, and temperatures between different optimization runs, it is necessary to recompute the parameters even if the basic technology or library did not change.

The main goal is to identify some parameters that allow us to estimate the timing of a repeater tree based on the Steiner tree. In addition, we compute some values that guide us in the buffering step of a particular instance.

4.1.1 Estimating two-pin connections

Figure 4.1 shows the delay over a two-pin connection depending on its length after buffering it in an approximatively delay-minimal way (see Section 6.3). The experiment was done with a 22 nm chip design using default planes and wire widths.

We see how repeater insertion linearizes the delay that would be quadratic otherwise. The red line in the figure shows a linear approximation of the delay function between the two inverters. The slope of the approximation is d_{wire}. Given this approximation, we can predict the delay for two-pin nets after buffering:

$$delay = d_{wire} * length. \tag{4.1}$$

While there are closed-form solutions for buffering two-pin nets[1], we do not use them for approximating d_{wire} because they rely on simplifications like Elmore delay and do not capture all environmental parameters. Instead, we search for the best way of buffering long two-pin nets by implementing it in the design and using the timing engine to calculate delays and slews. This way, we capture all effects that affect timing.

We now show how we compute the constant d_{wire}. To bridge large distances of wire in wiring mode w using repeater t, we partition the wire equidistantly by adding a repeater after l units of wire.

[1] See for example Alpert et al. (2008), p. 536f

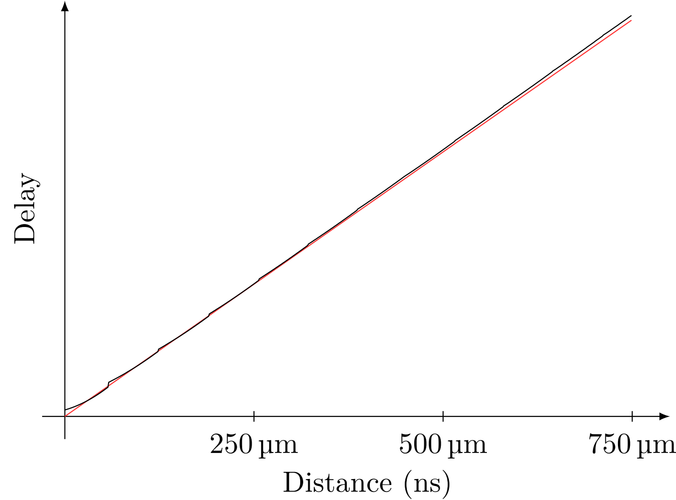


Figure 4.1: Two medium-sized inverters are placed at a given distance. The net between them is then buffered with the highest effort. The graph shows the resulting delay depending on the distance (black). The red line shows the linear approximation that we compute for the delay function.

To measure the delay over the line, we add two repeaters of type t into the design and connect them by a net such that the length of the net is l. We modify the pin capacitance at the end of the line to be c. At the input of the first repeater, the slew pair s_{in} is asserted. The whole setup guarantees that most global timing parameters are considered. Local timing parameters (e.g. coupling capacitance) are ignored.

Let $d(t, s_{in}, l, w, c)$, $s_{out}(t, s_{in}, l, w, c)$ and $p(t, s_{in}, l, w, c)$ be the total delays over the stage (through repeater and wire), the slews at the other end of the wire, and the power consumption, respectively. We assume that all values are infinite if a load limit or a slew limit is violated.

Let now s_0 be a reasonable slew pair (we just use the minimum allowed slews for t). We define

$$s_{i+1} := s_{out}(t, s_i, l, w, c) \qquad (i \geq 0). \tag{4.2}$$

The sequence $(s_i)_{i=1,2,3,...}$ typically converges very fast to a fixpoint or quickly becomes ∞ due to an electrical violation. We call $s_\infty(t, l, w, c) := \lim_{i\to\infty} s_i$ the *stationary slews* of (t, l, w, c). In practice, we iterate over s_i until we reach a fixpoint due to the limited precision of the floating-point numbers used to represent slews. Typically, the fixpoint is reached within ten iterations. We then use the computed value as an approximation of $s_\infty(t, l, w, c)$. There is only a single fixpoint because the slew function is typically contractive over a whole stage. Thus, the choice of the initial slews does not matter as long as it is within the domain of the slew and delay functions.

One might think that inverters need special treatment, because the rising (falling) output slew does not depend on the rising (falling) input slew. Instead, the signal is

inverted internally. The stationary slews might only be achieved after propagating over two stages in the inverter chain. There might also be different stationary slews for odd and even numbers of stages respectively. Fortunately, one can easily show that one has only to deal with a single stationary slew pair.

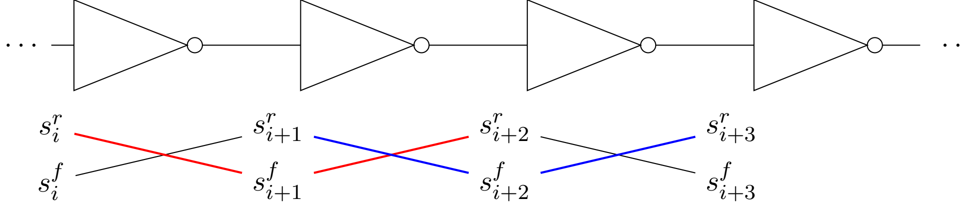


Figure 4.2: An endless chain of equidistantly distributed inverters. The slews alternate between rise and fall. Although a given rise value is not used in the computation of the following one, like, for example, s_i^r and s_{i+1}^r, consecutive rise values converge to a fixpoint.

Figure 4.2 shows an infinite line of equidistantly distributed inverters and the slews at the input pins. The lines below show how slews are propagated. For example, the rising slew s_1^r is propagated to the falling slew s_2^f. Let rf be the rise-fall slew function for a whole stage[2] of the chain and fr the fall-rise slew function. As both functions are contractive, their compositions are also contractive.

Within the chain of inverters the even and odd slews form separate sequences with

$$s_{i+2} = \Big(fr(rf(s_i^r)), rf(fr(s_i^f))\Big).$$

The blue and red lines indicate both sequences in the figure. The sequences only differ by the starting point as the even sequence starts with minimum allowed slews and the odd sequence starts with s_1. Because contractive functions only have a single fixpoint, both sequences converge to it. However, this means that the combined sequence also converges to the fixpoint. Therefore, it suffices to consider a single stage, not only for buffers but also for inverters.

Due to the asymmetric nature of the delay and slew functions for the rising and falling slews, we take the average for further processing and abbreviate:

$$\begin{aligned}
d(t,l,w,c) &:= \frac{1}{2}\left[d(t,s_\infty(t,l,w,c),l,w,c)^r + d(t,s_\infty(t,l,w,c),l,w,c)^f\right]\\
s(t,l,w,c) &:= \frac{1}{2}\left[s_\infty(t,l,w,c)^r + s_\infty(t,l,w,c)^f\right]\\
p(t,l,w,c) &:= p(t,s_\infty(t,l,w,c),l,w,c).
\end{aligned}$$

For a given wiring mode, a repeater, and a length, we can now compute the delay

[2]The slew function for a whole stage combines the slew calculation from the input of a repeater through the repeater and the following net up to the input of the next repeater.

per unit distance and power consumption per unit distance

$$\bar{d}(t,l,w) := \frac{d(t,l,w,cap^{\text{in}}(t))}{l}$$
$$\bar{p}(t,l,w) := \frac{p(t,l,w,cap^{\text{in}}(t))}{l}$$

and the according stationary slews

$$\bar{s}(t,l,w) := s_\infty(t,l,w,cap^{\text{in}}(t)).$$

Figure 4.3 shows how the delay per unit distance typically behaves depending on the length between two consecutive repeaters. The delay is dominated by the repeater delay for small distances. The overall delay decreases until a delay-optimal distance is reached. For larger distances the wire delay begins to dominate. The curve ends as soon as the slews or loads create electrical violations. It is not shown in the figure, but stationary slews increase monotonically with the length.

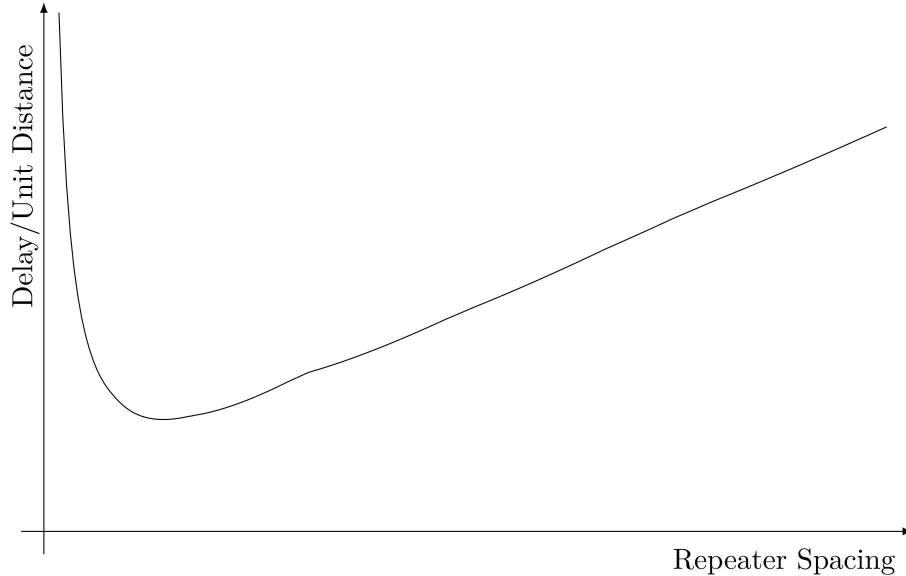


Figure 4.3: A typical curve showing $\bar{d}(t,l,w)$ for a given repeater t and wiring mode w over the range of valid lengths l. The curve shown is from a medium-sized inverter from a 22 nm design on the third metal layer using the smallest wiring mode.

Using the same repeater and wiring mode as in the previous figure, we can see in Figure 4.4 how power per unit distance and delay per unit distance relate to each other. For small distances (upper right endpoint of the curve) the power consumption is high due to the high amount of repeaters needed. Power consumption and delay decrease with larger distances until we reach the optimal distance. Further power reductions cause higher delays. The red points show possible stage lengths that

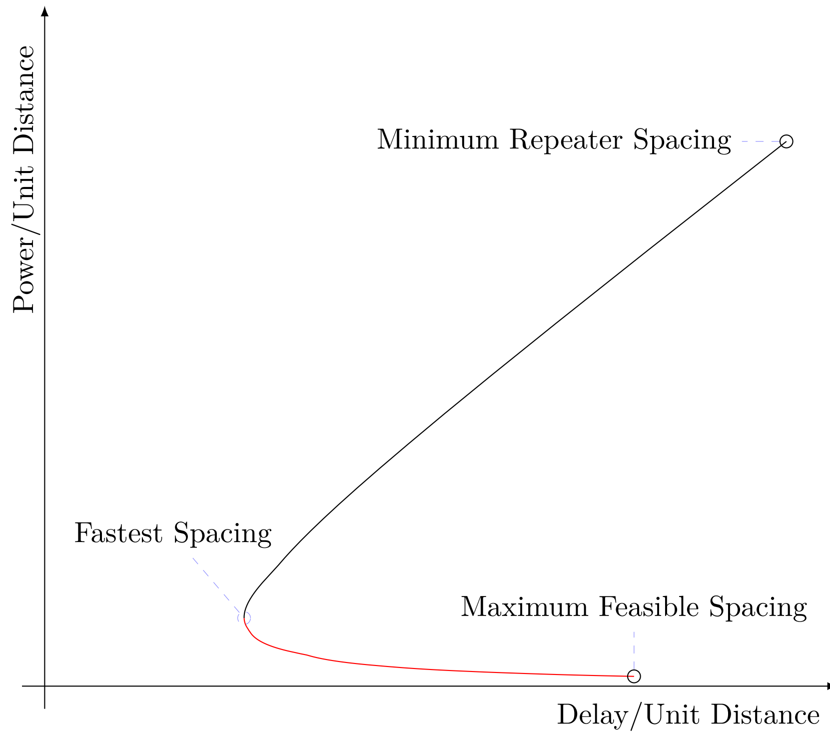


Figure 4.4: A typical curve showing $(\bar{d}(t,l,w), \bar{p}(t,l,w))$ for a given repeater t and wiring mode w parametrized over the range of valid lengths l. The curve is generated for the same repeater as in Figure 4.3.

are not dominated by distances that result in cheaper configurations with the same delay or faster configurations with the same power consumption.

We choose for each wiring mode w a repeater $t^*_w \in L$ and a length $l^*_w \in \mathbb{R}_{>0}$ which minimize the linear combination

$$\xi \bar{d}(t^*_w, l^*_w, w) + (1-\xi)\bar{p}(t^*_w, l^*_w, w). \tag{4.3}$$

If two different choices for t^*_w and l^*_w minimize the equation, then we choose the fastest one. We call the parameter ξ *power-time-tradeoff*. For library analysis, we choose $\xi = 1$ to calculate a lower bound on the achievable wire delay. The minima can then be found by binary search over all lengths for each repeater type. The functions we have to minimize are similar to the one shown in Figure 4.3.

4.1.2 Parameter d_{wire}

For delay estimation in topology generation, we do not want to distinguish between horizontal and vertical wire segments, because we often do not want to fix the exact embedding of path segments. Therefore, we use the default wiring modes w^*_h and w^*_v to build an average delay value. Typically, both wiring modes have similar electrical properties such that the resulting value is not far away from both. We choose optimal repeater t^* and length l^* such that

$$\xi\Big(\bar{d}(t^*,l^*,w^*_h) + \bar{d}(t^*,l^*,w^*_v)\Big) + (1-\xi)(\bar{p}(t^*,l^*,w^*_h) + \bar{p}(t^*,l^*,w^*_v))$$

is minimized. The resulting delay per unit distance

$$d_{wire} := \frac{\bar{d}(t^*,l^*,w^*_h) + \bar{d}(t^*,l^*,w^*_v)}{2} \tag{4.4}$$

is the parameter we searched for. It will be used for delay estimation during topology generation (see Equation 4.1).

We call the stationary slew corresponding to the repeater and length choice *optslew*:

$$optslew := \frac{\bar{s}(t^*,l^*,w^*_h) + \bar{s}(t^*,l^*,w^*_h)}{2}.$$

The average capacitance over a stage is called *maxcap*:

$$maxcap := \frac{wirecap(w^*_h) + wirecap(w^*_v)}{2} l^* + cap^{\text{in}}(t^*).$$

4.1.3 Buffering Modes

As indicated in the previous section, we allow diagonal segments in our Steiner trees such that it is not clear where we will eventually use horizontal or vertical wiring segments. Thus, we assign a *buffering mode* that approximates the properties of a horizontal and a vertical wiring mode to each segment. Buffering modes also represent the effort we want to put into buffering of a segment.

A buffering mode m is a 3-tuple (m_h, m_v, m_ξ) that consists of

- a horizontal wiring mode $m_h \in W$,
- a vertical wiring mode $m_v \in W$, and
- a power-time tradeoff m_ξ.

During buffering of a wire segment, we will try to replicate long-distance chains. The distance between repeaters in a chain buffered with a given mode is determined by the target repeater and the slew targets. The stationary slews of the chain will be slew targets.

We have to determine the set of buffering modes that we want to work with. We assume that there is a set $W_p \subseteq W \times W$ of wiring mode pairs. Each pair consists of a horizontal wiring mode and a vertical wiring mode. We also restrict ourselves to a set Ξ of power-time-tradeoffs between 0.0 and 1.0.

Given a wiring mode pair (w_h, w_v) with horizontal wiring mode $w_h \in W$ and vertical wiring mode $w_v \in W$ and a power-time-tradeoff $\xi \in \Xi$, we define a buffering mode (w_h, w_v, ξ).

For each buffering mode, we find the optimal repeater $t \in L$ and distance $l \in \mathbb{R}_{>0}$ minimizing

$$\xi \left(\bar{d}(t,l,w_h) + \bar{d}(t,l,w_v)\right) + (1-\xi)\left(\bar{p}(t,l,w_h) + \bar{p}(t,l,w_v)\right).$$

The optimal repeater for buffering mode m is called m_t. The according slew targets are

$$m_s := \frac{\bar{s}(t,l,w_h) + \bar{s}(t,l,w_v)}{2}$$

The delay of a buffering mode m is defined as

$$m_d := \frac{\bar{d}(t,l,w_h) + \bar{d}(t,l,w_v)}{2}.$$

The power consumption per length unit is

$$m_p := \frac{\bar{p}(t,l,w_h) + \bar{p}(t,l,w_v)}{2}.$$

The capacitance of a stage is

$$m_{cap} := \frac{wirecap(w_h) + wirecap(w_v)}{2} l + cap^{\text{in}}(t).$$

The average capacitance per unit length wire is

$$m_{wirecap} := \frac{wirecap(w_h) + wirecap(w_v)}{2}.$$

We use d_m and p_m to estimate the delay of an edge that is buffered using mode m.

In practice, the user creates the set W_p of reasonable wiring mode pairs. Typically, the horizontal and vertical wiring mode have similar widths and spacings and lie on neighboring planes for each wiring mode pair in W_p. In such a case, the delay, power, and slew values do not differ significantly between the wiring modes of a pair such that using the averages is not too far off. The set Ξ is also defined by the user, but, in practice, we only use two tradeoffs: 0 and ξ. The default wiring modes are always in W_p. Thus, there is always a buffering mode available with delay d_{wire} (compare to Section 4.1.2):

$$m^* := (w_h^*, w_w^*, \xi)$$

Finally, given W_p, we can determine the set M of buffering modes that we will use for buffering:

$$M := \{(w_h, w_v, \xi) \,|\, (w_h, w_v) \in W_p, \xi \in \Xi\}.$$

For each buffering mode m, there is a set of alternative buffering modes M_m containing all buffering modes with the same horizontal and vertical wiring modes as m including m itself. The alternative buffering modes only differ by the ξ value. We assume that M_m only contains non-dominated buffering modes. A buffering mode dominates another one if it is at the same time not slower and not more expensive than the other one.

4.1.4 Slew Parameters

As described in Section 3.1, different slews at the input pins of repeater tree subtrees have different effects to the downstream delays. To account for this where we do not have an explicit RAT function, we introduce a parameter ν that translates slew differences to delay differences (see also Vygen (2006)).

Let $t^* \in L$ and $l^* \in \mathbb{R}_{>0}$ be the repeater and length that minimize Equation 4.4. We define the slew pair of this optimal chain using only one of the default wiring modes

$$s_{opt} := \bar{s}(t^*, l^*, w_h^*).$$

We now compute

$$d_1 := \sum_{i=0}^{N} d(t^*, s_i, l^*, w_h^*, cap^{\text{in}}(t^*))$$

using the following slews:

$$\begin{aligned} s_0 &:= s_{opt} \\ s_i &:= s_{out}(t^*, s_{i-1}, l^*, w_h^*, cap^{\text{in}}(t^*)) \end{aligned}$$

In a second step, we compute d_2 in an analog way to d_1 by starting with $s_0 := 2 \cdot s_{opt}$. In practice, using $2 \cdot s_{opt}$ will not lead to a violation. We set the desired parameter to

$$\nu := \frac{d_2 - d_1}{s_{opt}}.$$

The number N is chosen such that the stationary slew is reached for both computations. The parameter depends on the timing environment and technology. Typically, it lies between 0.10 and 0.25. We use it to define

$$slewdelay(s) := \nu \cdot (s - s_{target})$$

for a given target slew s_{target} The function *slewdelay* is used for two similar tasks:

1. As discussed earlier, required arrival times are associated with individual slew requirements at sinks. To better compare RATs, we normalize them to a target slew. For example, if a sink has the slew requirement s, then we translate the RAT to s_{target} by adding $slewdelay(s)$ to it.

2. If a signal with slew s arrives at a sink with a slew target s, then we add $slewdelay(s)$ to the arrival time before we compute the slack of the signal.

Both happens only in our buffering algorithm based on dynamic programming (See Section 6.3).

4.1.5 Sinkdelay

Consider two inverters connected by a net and placed far apart (as described in Section 4.1.1). We now add repeaters to the line such that the delay between the inputs of the inverters is minimized.

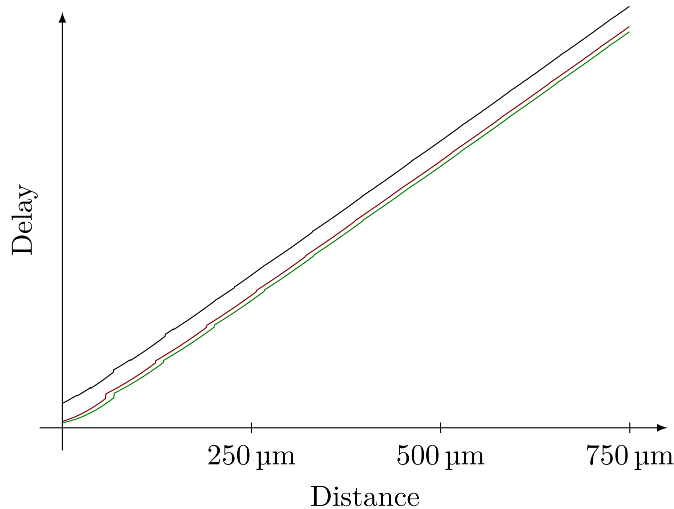


Figure 4.5: Two inverters are placed at a given distance. The net between them is buffered with the highest effort. The graph shows the resulting delays for the same source but three different sink capacitances at the end of the net: a small inverter (green), a medium sized inverter (red), and a huge inverter (black).

Figure 4.5 shows the resulting delays for different distances between the boundary inverters and for different capacitances at the end of the chain due to the different

inverter sizes. The difference between the delays remains nearly constant for longer distances. We choose a distance where the delay differences are significant enough and compute for different sink pin capacitances the resulting delay. Let d_0 be the delay of the chain if the capacitance at the sink is the input pin capacitance of t^*, the optimal repeater minimizing Equation 4.4. For a given capacitance c at the end of the chain and the resulting delay d_c, we define

$$sinkdelay(c) := d_c - d_0$$

This function is used to estimate the delay difference on repeater chains due to different sink capacitance compared to the optimal repeater.

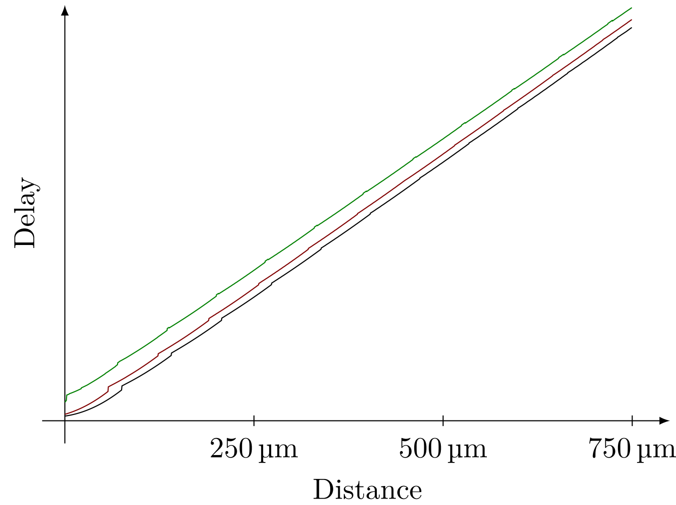


Figure 4.6: Two inverters are placed at a given distance. The net between them is buffered with the highest effort. The graph shows the resulting delays for the same sink capacitance but three different sources: a small inverter (green), a medium sized inverter (red), and a huge inverter (black).

Figure 4.6 shows the effects of changing the first gate instead of the last one in our setup. Stronger (weaker) gates result in smaller (larger) delays on the chain. The delay differences are also nearly constant at larger distances. In contrast to delays introduced by sink capacitances, we do not introduce a function to estimate the delays. Instead, we evaluate the root arrival time for the capacitance of the optimal chain.

4.1.6 Further Preprocessing

Next, we compute a parameter d_{node} that is used during topology generation to model the extra delay to be expected along a path due to additional capacitances induced by a side branch. We determine it by adding a small repeater to the repeater chain and measuring the additional delay.

Let $inv(c, s)$ denote the inverter with the smallest power consumption that still achieves a slew of at most s at its output pin if its input slew is s_{opt} and the load is c.

Let $t^1 := inv(maxcap, s_{opt})$, and let $t^2 := inv(cap^{\text{in}}(t^1), s_{opt})$. We compute d_1 as in Section 4.1.4. Now we modify the repeater chain by adding a capacitance load of $cap^{\text{in}}(t^1)$ in the middle of the first segment and consider it during the delay and slew computation of the first stage. We then get the new delay d_2 and set our branch penalty to

$$d_{node} := d_2 - d_1$$

4.2 Blockage Map and Congestion Map

Placement blockages and wiring congestion information is given to the repeater tree routine via a blockage map and a congestion map, respectively. The blockage map is used to check whether a given point is blocked. The congestion map holds the global routing information showing on which parts of the chip routing space is sparse.

4.2.1 Grid

Both the blockage map and the congestion map share the same grid. The grid partitions the chip area into tiles. The tiles are nodes of a grid graph.

The bounding box ca of the design area is given by

$$[ca_{minx}, ca_{maxx}] \times [ca_{miny}, ca_{maxy}].$$

Definition 3 (Grid)**.** A grid is a pair $(xlines, ylines)$ of cutlines $xlines = \{x_0, x_1, \ldots, x_m\}$ and $ylines = \{y_0, y_1, \ldots, y_n\}$ with $x_0 < x_1 < \ldots < x_m$ and $y_0 < x_1 < \ldots < x_m$ and $m > 1$ and $n > 1$. We call a grid feasible for a chip area ca if $x_0 \leq ca_{minx}$, $ca_{maxx} < x_m$, $y_0 \leq ca_{miny}$, and $ca_{maxy} < y_m$.

Most of the time, we use a grid with equidistant cutlines. An equidistant grid is accurate enough in the context of repeater tree insertion if it is not spaced too wide. Sometimes, however, one wants to use a Hanan grid given by the coordinates of all edges of significantly large blockages such that their positions are exactly captured in the blockage map.

Definition 4 (Tile)**.** Given a grid $(xlines, ylines)$ with $xlines = \{x_0, x_1, \ldots, x_m\}$ and $ylines = \{y_0, y_1, \ldots, y_n\}$, we call the rectangle $[x_i, x_{i+1}) \times [y_j, y_{j+1})$ for $0 \leq i < m$ and $0 \leq j < n$ the $tile(i, j)$ of the grid.

For a given point of the chip area, there is exactly one tile in a feasible grid that contains the point.

4.2.2 Blockage Map

The set of blocked regions for a repeater tree instance is stored in a data structure that we call blockage map.

The most important operation that is performed on the blockage map is searching for the nearest free location. Given a point in the plane, the blockage map is able to give us the nearest free location in a given direction rectilinear to the grid or the nearest free location in the whole plane with respect to ℓ_1-metric. Points on blockage boundaries that are next to free points are considered as free.

4.2.3 Blockage Grid

For an existing blockage map and a grid, we also construct a blockage grid. The blockage grid stores information whether a grid tile is blocked or not:

Definition 5 (Blockage Grid)**.** Given a grid $(xlines, ylines)$ with

$$xlines = \{x_0, x_1, \ldots, x_m\} \quad \text{and} \quad ylines = \{y_0, y_1, \ldots, y_n\}$$

and a blockage map, a *blockage grid* is a function

$$bg : \{0, \ldots, m-1\} \times \{0, \ldots, n-1\} \to \{0, 1\}$$

where $bg(x, y) = 1$ iff $tile(x, y)$ is completely blocked by the blockages of the map.

Shortest Path Searches

Typically, blockages do not block all wiring layers in a design. It is possible to cross them on higher layers. Repeater trees are also allowed to jump over blockages. However, the possible distance is limited by the slew and capacitance limits as it is not possible to place repeaters on blockages. Larger distances between repeaters caused by jumping over blockages also cost additional delay compared to an optimally spaced repeater chain.

At one step in our topology generation algorithm, we search for delay minimal paths between points in the design. We use a modified version of Dijkstra's shortest path algorithm on the blockage grid for this task.

Given two points, we first identify the tiles they belong to in the blockage grid and then compute a shortest path between both tiles. The costs of an edge between two neighboring tiles depends on whether the tiles are blocked or not. Crossing unblocked space costs proportional to d_{wire}. Costs over blocked area increase first linearly and after a threshold quadratically with the distance the path already went over blockages.

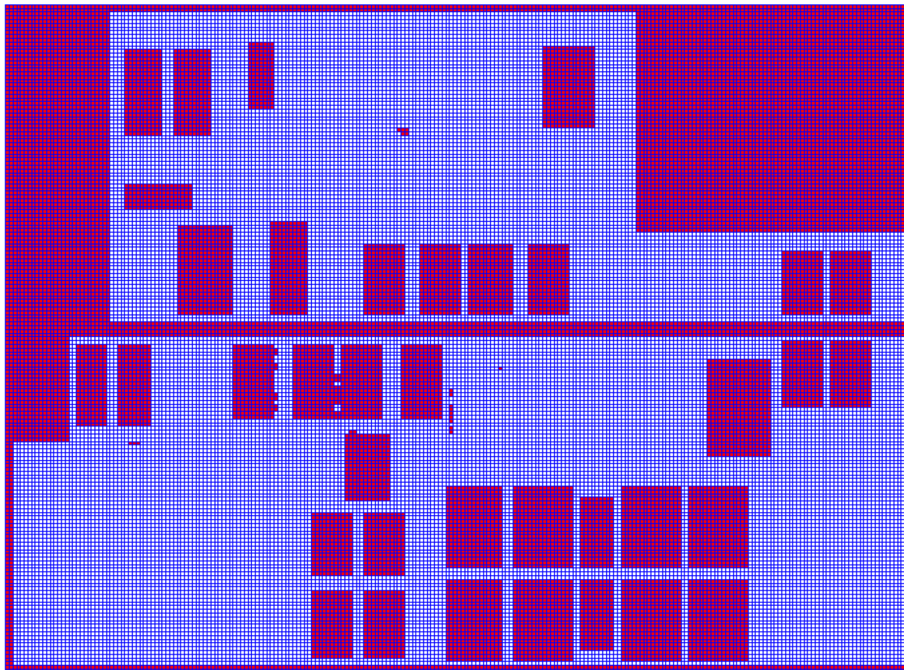


Figure 4.7: Blockage map (red) and grid (blue) on the design Julius.

4.2.4 Congestion Map

We implemented a rough global routing engine as congestion map. In contrast to a full-fledged global router, the congestion map does not try to find a congestion free global routing solution by all means. Instead, we embed a short ℓ_1-tree allowing only small detours. We also limit the number of iterations spent for improving the routing. The advantage is that we still see congestion that we then can try to avoid during repeater tree generation. As can be seen in Table 7.2, using a full global router would increase the running time of our algorithm significantly. Our algorithm has recently been integrated into BonnRouteGlobal as a fast mode.

5 Topology Generation

The first step in our repeater tree algorithm is the construction of a repeater topology. A repeater topology specifies the abstract geometric structure of the repeater tree. Given an instance of the REPEATER TREE PROBLEM with root r and sink set S, we can define:

Definition 6. A *topology* $T = (V(T), E(T))$ with $V(T) = \{r\} \dot{\cup} S \dot{\cup} I$ is an arborescence rooted at r with an embedding $Pl : V(T) \to \mathbb{R}^2$ of the nodes into the plane such that r has exactly one child, the internal nodes I have one or two children each, and the sinks S are the leaves.

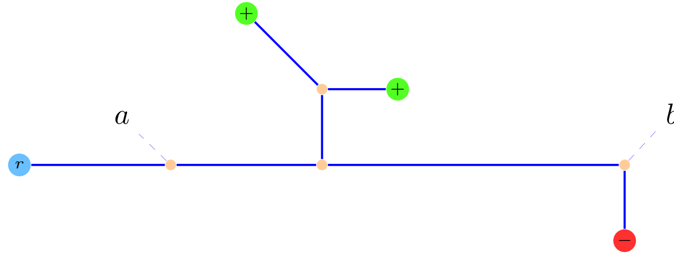


Figure 5.1: A topology for one root and three sinks. Steiner points like a and b are used to route the topology around obstacles. Although we do not use directed edges in our figures, the edges in a topology are always directed away from the root.

We often call the set I of internal points Steiner points. Internal points with only a single child are used to force the topology to pass a certain point in the plane. Figure 5.1 shows an example topology. We should clearly note that the internal nodes do not represent repeaters and that the topology does not specify details about the exact placement, routing, and types of repeaters used in the final tree.

The length of a topology is

$$\sum_{(v,w)\in E(T)} ||Pl(v) - Pl(w)||.$$

We have seen that after repeater insertion delays in a repeater tree are roughly linear in the length of the segments. Connecting a sink to the root via a long path results in a higher delay to a sink. The required arrival times at a sink then decide whether a path is fine or too long. Consider the example in Figure 5.2. Both topologies have the same length, but, for example, the distance to the root is 8 for the upper right sink in the first case and 2 in the second one. This example

Figure 5.2: Two topologies with the same shortest possible length but different timing behaviour. While all sinks are reached within 4 segments on the right side, it takes up to 8 segments to the furthest sink on the left side.

illustrates that topologies have a high influence on the timing of a repeater tree. It is therefore crucial to build timing-aware repeater trees.

Topologies for a root and a set of sinks that consider timing information do not only have an application in repeater tree construction. They can also prove to be useful in global routing. A global router internally often has to compute Steiner trees for the nets of a design. The routing result can be better with regard to timing if the Steiner trees are timing-aware topologies.

In this chapter, we first develop a way to estimate the timing of a topology and then state the Repeater Tree Topology Problem. We show how our algorithm solves the problem and prove some theoretical properties for restricted versions of our algorithm.

The results in this chapter are joint work with Stephan Held, Jens Maßberg, Dieter Rautenbach and Jens Vygen (Bartoschek et al., 2007a, 2010).

5.1 A Simple Delay Model

Since we want to evaluate the properties of our topologies with respect to timing, we somehow have to compute a slack at root and sinks. It would be prohibitively slow to insert repeaters into each topology we want to evaluate. Therefore, we propose a simple delay model that estimates the timing from the geometric structure of the topology. The delay model will compute arrival times and required arrival times for all nodes of a topology giving us a slack that can be used to evaluate the topology. The delay model mainly consists of two components: delay over wire segments and delay due to bifurcations.

We have seen in Section 4.1 how buffering a long net linearizes the delay. Given a buffering mode m, the estimated delay for a net between two points x and v is given by

$$delay := m_d||x - v||. \tag{5.1}$$

Every internal node of a topology with outdegree two is a bifurcation and thus an additional capacitance load for the circuit driving both of the two outgoing branches (compared to alternative direct connections). The real delay caused by bifurcations is hard to estimate beforehand. It will depend on the strength of the driver, the additional capacitance, and the position of the driver compared to the sinks. In

Section 4.1.6 we computed the parameter d_{node} estimating the average effect of a bifurcation. It is a very rough estimation, but we will show in Section 8.5 that the used value serves us well. To evaluate the delay through a topology, we will add the additional delay to each outgoing edge of a node with two children.

It is reasonable to assume that the additional load capacitance will be smaller for the less critical branch. Uncritical side path are more likely to be buffered by a small repeater with nearly neglectable capacitance. We therefore allow the distribution of d_{node} between both involved edges. We denote by $d_{node}(e)$ the amount assigned to edge e. We introduce a new parameter η controlling how uneven the distribution of d_{node} can be. If e is an outgoing edge of a node with outdegree 1, then we require $d_{node}(e) = 0$. Otherwise, we require that $d_{node}(e) \geq \eta d_{node}$. For two edges e, e' leaving the same internal node we require $d_{node}(e) + d_{node}(e') = d_{node}$. The parameter η has to be between 0 and 1/2 to be able to fulfill the requirements.

Next, we have to determine the arrival time at the root node. If the edge leaving the root has buffering mode m assigned, then we assume that the root will have to drive capacitance m_{cap}[1]. We set the arrival time at the root to

$$at_T(r) := \max\Big\{at_r^r(m_{cap}), at_r^f(m_{cap})\Big\}.$$

Note that for maximizing the worst slack, an accurate arrival time at the root is not important because each change affects the slack at all sinks in the same way.

Finally, we have to determine the required arrival time for each sink. In our simple delay model, we only want to handle a single RAT value and not a pair of functions. Therefore, we evaluate the RAT function at the slew target of the incoming edge. As shown in Section 4.1.5, the capacitance of a sink has to be taken into account when the delay is estimated. This is done by subtracting the appropriate *sinkdelay* from the resulting RAT.

Given a sink with required arrival time function *rat*, pin capacitance *cap*, and buffering mode m at the sink's incident edge, we define the RAT used in the delay model as

$$sinkrat(rat, cap, m) := \min\Big\{rat^r(m_s^r), rat^f(m_s^f)\Big\} - sinkdelay(cap). \tag{5.2}$$

Given a topology and a buffering mode assignment $F : E(T) \to M$, we can now estimate the slack at sink s to be

$$\sigma_s := sinkrat(rat_s, cap^{\text{in}}(s), m_s) - \sum_{e=(v,w)\in E(T)_{[r,s]}} (d_{node}(e) + F(e)_d||Pl(v) - Pl(w)||) - at_T(r)$$

with m being the buffering mode of the arc entering s and m_s the according slew target.

Figure 5.3 shows how our delay model correlates with the slacks that are achieved after buffering. For each instance of a 22 nm design we depict the difference between

[1] See Section 4.1.3.

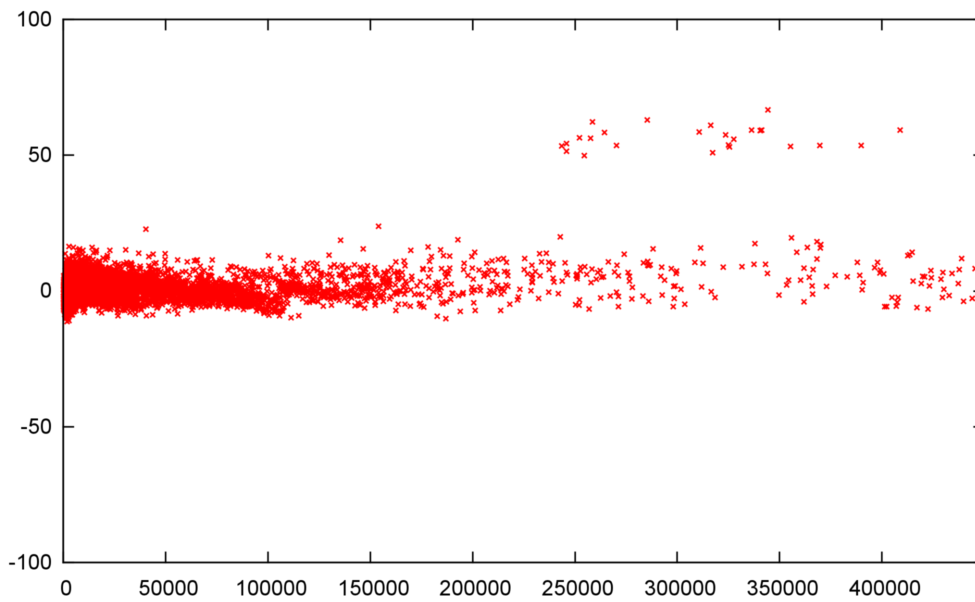


Figure 5.3: Correlation between estimated slacks and exact slacks. For each instance (slightly more than 300 000) of a middle-sized 22 nm design the difference (y-axis) between the slack in our delay model and the final slack after buffering is shown. The instances are sorted by the distance (x-axis) of the most critical sink to the root.

the slack of the topology used for repeater insertion and the slack of the final result. Although there are some outliers where we overestimate the strength of the root and are about 50 picoseconds too optimistic, the vast majority of instances are estimated up to 20 picoseconds correctly.

5.1.1 Time Tree

Algorithm 1 TIMETREE

Input: A topology T, an embedding Pl, a buffering mode assignment $F : E(T) \to M$, and parameters d_{node}, η

Output: Arrival time function at_T, RAT function rat_T and a d_{node} assignment

1: **for** $v \in V(T)$ traversed in postorder **do**
2: **if** v is a leaf **then**
3: Let e be the incoming edge to v if $v \neq r$
4: $rat_T(v) := sinkrat(rat_v, cap^{\text{in}}(v), F(e))$
5: **else**
6: **if** $|\delta^+(v)| = 1$ **then** ▷ v is root or Steiner point along a path
7: Let $a = \delta^+(v)$
8: $rat_T(v) := rat_T(a) - F((v,a))_d ||Pl(v) - Pl(a)||$
9: $d_{node}((v,a)) := 0$
10: **else**
11: Let $\{a, b\} = \delta^+(v)$ with $rat_T(a) \leq rat_T(b)$
12: $\alpha := rat(a) - F((v,a))_d ||Pl(v) - Pl(a)||$
13: $\beta := rat(b) - F((v,b))_d ||Pl(v) - Pl(b)||$
14: $rat_T(v) := \max_{\eta d_{node} \leq d \leq (1-\eta) d_{node}} \min\{\alpha - d, \beta - (d_{node} - d)\}$
15: $d_{node}((v,a)) := rat_T(a) - rat_T(v)$
16: $d_{node}((v,b)) := d_{node} - d_{node}((v,a))$
17: **end if**
18: **end if**
19: **end for**

20: Let e be the outgoing edge of r
21: $at_T(r) := \max\left\{at_r^r(F(e)_{cap}), at_r^f(F(e)_{cap})\right\}$
22: **for** $v \in V(T) \setminus r$ traversed in preorder **do**
23: Let w be the parent of v
24: $at_T(v) := at_T(w) + d_{node}((w,v)) + F((w,v))_d ||Pl(w) - Pl(v)||$
25: **end for**

During topology construction, we will only maintain the required arrival times and update them incrementally. Arrival times will not be explicitly calculated. However, it is often desirable to compute the delay model of a given topology. This can be done with Algorithm 1 (TIMETREE). It first traverses the topology bottom

up, computes required arrival times, and distributes d_{node}. Then, arrival times are computed in a second top-down traversal. Both traversals have a running time that is linear in the size of the topology as each update step can be done in constant time.

5.2 Repeater Tree Topology Problem

The Repeater Tree Topology Problem is the task of finding a topology for an instance of the Repeater Tree Problem, an embedding, and a buffering mode assignment. As for the Repeater Tree Problem, we allow several objectives like minimizing netlength or maximizing the delay model slack with minimal costs.

Minimizing the ℓ_1-length is an objective for topology generation that appears in early design stages or for timing-uncritical instances. This corresponds to computing shortest rectilinear Steiner trees. Garey and Johnson (1977) showed that already this problem is NP-hard.

Previous Work on Topology Generation

Alpert et al. (2008), Chapter 24–28, give a good overview of existing topology generation algorithms beginning with different flavours of Steiner trees and finishing with algorithms specific for repeater tree optimization.

Okamoto and Cong (1996) proposed a repeater tree procedure using a bottom-up clustering of the sinks and a top-down buffering of the obtained topology. Similarly, Lillis et al. (1996b) also integrated buffer insertion and topology generation. They introduced the P-tree algorithm, which takes the locality of sinks into account, and explored a large solution space via dynamic programming. Hrkić and Lillis (2002) considered the S-tree algorithm which makes better use of timing information, and integrated timing and placement information using so-called SP-trees (Hrkić and Lillis, 2003). In these approaches the sinks are typically partitioned according to criticality, and the initially given topology (e.g. a shortest Steiner tree) can be changed by partially separating critical and noncritical sinks. Whereas the results obtained by these procedures can be good, the running times tend to be prohibitive for realistic designs in which millions of instances have to be solved.

Alpert et al. (2002) create topologies in a two step approach. First, sinks are clustered based on parity and criticality. Second, clusters are merged by a Prim-Dijkstra heuristic that scales between shortest path trees and minimum spanning trees.

Further approaches for the generation or appropriate modification of topologies and their buffering were considered in Cong and Yuan (2000); Alpert et al. (2001b, 2004a); Müller-Hannemann and Zimmermann (2003); Dechu et al. (2005); Hentschke et al. (2007); Pan et al. (2007).

Repeater topology generation loosely overlaps with the design of delay constraint multicast networks where network traffic has to be distributed to clients. A survey can be found in Oliveira and Pardalos (2005).

5.2.1 Topology Algorithm Overview

We solve the Repeater Tree Topology Problem by splitting it into three steps. In the first step, we restrict ourselves to the default buffering mode m^* and compute an initial topology ignoring blockages. During the second step, we navigate around blockages if blocked segments in the initial topology get too long. In the final step, buffering modes from higher layers are assigned to topology edges if their slack is infeasible.

We start our explanation by describing a simplified version of the Repeater Tree Topology Problem.

5.3 Restricted Repeater Tree Problem

The first step of our topology generation algorithm uses the default buffering mode m^*. This is the fastest mode using the default wiring modes. To simplify notation we set $d := m^*_d$ and $c := d_{node}/2$. The bifurcation delay assigned to edge e is $c(e)$.

We evaluate topologies with our delay model that does not distinguish between signal edges and does not know RAT functions. We adapt the Repeater Tree Topology Problem to this simplification. We set for each sink $s \in S$ the required arrival time to

$$a_s := sinkrat(rat_s, cap^{\text{in}}(s), m^*) - \max\Big\{at^r(m^*_{cap}), at^f(m^*_{cap})\Big\}.$$

Given a topology T, the slack for sink $s \in S$ becomes

$$\sigma_s := a_s - \sum_{(v,w)\in E(T)_{[r,s]}} (d||Pl(v) - Pl(w)|| - c((v,w))).$$

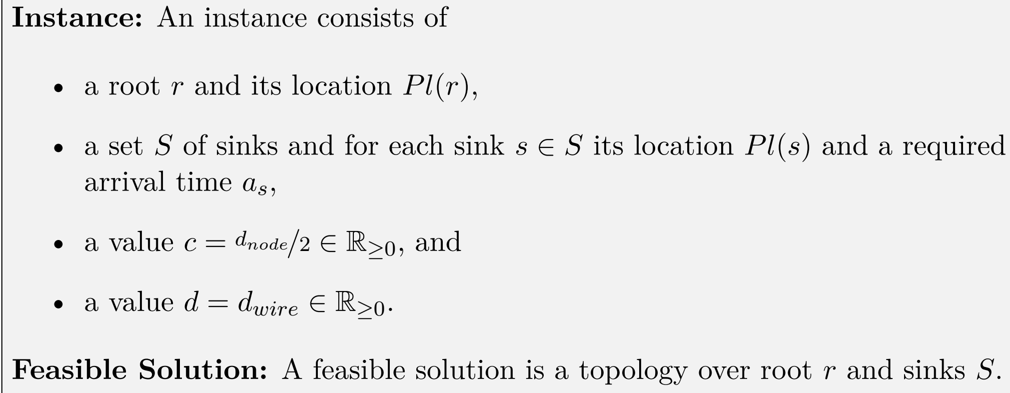
The slack of the whole topology is

$$\sigma(T) := \min_{s\in S} \sigma_s.$$

We call the simplified version of the topology problem Restricted Repeater Tree Topology Problem. It is shown in Figure 5.4.

5.4 Sink Criticality

Our topology generation algorithm will insert sinks into the topology one by one, and the resulting structure depends on the order in which the sinks are considered. Sinks that are inserted first are favored because they will potentially be connected shorter to the root. We thus want to prefer sinks that are more timing-critical. In order to quantify correctly how critical a sink s is, it is crucial to take its reqiured arrival time a_s as well as its location $Pl(s)$ into account. A sink that is further away from the root will, other things being equal, result in worse slack because the signal has to traverse the distance which costs delay. Similarly, if two sinks have the same



Instance: An instance consists of

- a root r and its location $Pl(r)$,
- a set S of sinks and for each sink $s \in S$ its location $Pl(s)$ and a required arrival time a_s,
- a value $c = {}^{d_{node}}/_2 \in \mathbb{R}_{\geq 0}$, and
- a value $d = d_{wire} \in \mathbb{R}_{\geq 0}$.

Feasible Solution: A feasible solution is a topology over root r and sinks S.

Figure 5.4: Restricted Repeater Tree Topology Problem

distance to the root, then both will pay approximately the same delay to reach the root but the sink with lower required arrival time will be more critical.

A good measure for the criticality of a sink s is the slack that would result from connecting s optimally to r and disregarding all other sinks. We can estimate the optimal connection using our delay model. The resulting slack equals:

$$\sigma_s = a_s - d||Pl(r) - Pl(s)||. \tag{5.3}$$

The smaller this number is, the more critical we will consider the sink to be.

5.5 A Simple Topology Generation Algorithm

Before we explain the topology generation algorithm that we use to solve the Restricted Repeater Tree Topology Problem in Section 5.6, we first look at an algorithm that has the basic structure of our final algorithm. We show the algorithm in the next section before we discuss some of its theoretical properties.

Our first algorithm creates topologies where each internal vertex is a bifurcation. We use $\eta = {}^1/_2$ in addition such that each arc but the one leaving the root have a node delay of c.

The slack of a topology T is then given by

$$\sigma(T) := \min_{s \in S} \left[a_s - c\Big(|E(T)_{[r,s]}| - 1\Big) - \sum_{(v,w) \in E(T)_{[r,s]}} d||Pl(v) - Pl(w)|| \right].$$

The properties we show for the simple topology generation algorithm were first published in Bartoschek et al. (2010).

5.5.1 Topology Generation Algorithm

Algorithm 2 inserts sinks into a topology one by one according to some order $s_1, s_2, \ldots, s_n$ starting with a tree containing only the root r and the first sink s_1.

Algorithm 2 Simple Topology Generation Algorithm

1: Choose a sink $s_1 \in S$
2: $V(T_1) \leftarrow \{r,s_1\}$
3: $E(T_1) \leftarrow \{(r,s_1)\}$
4: $T_1 \leftarrow (V(T_1),E(T_1))$
5: $n \leftarrow |S|$
6: **for** $i = 2, \dots, n$ **do**
7: Choose a sink $s_i \in S \setminus \{s_1,s_2,\dots,s_{i-1}\}$,
8: an edge $e_i = (u,v) \in E(T_{i-1})$,
9: and an internal vertex x_i with $Pl(x_i) \in \mathbb{R}^2$.

10: $V(T_i) \leftarrow V(T_{i-1}) \dot\cup \{x_i\} \dot\cup \{s_i\}$
11: $E(T_i) \leftarrow (E(T_{i-1}) \setminus \{(u,v)\}) \cup \{(u,x_i),(x_i,v),(x_i,s_i)\}$
12: $T_i \leftarrow (V(T_i),E(T_i))$
13: **end for**

The sinks s_i for $i \geq 2$ are inserted by subdividing an edge e_i with a new internal vertex x_i located at $Pl(x_i)$ and connecting x_i to s_i. The behaviour of the procedure clearly depends on the choice of the order, the choice of the edge e_i, and the choice of the placement $Pl(x_i) \in \mathbb{R}^2$.

In view of the large number of instances which have to be solved in an acceptable time, the simplicity of the above procedure is an important advantage for its practical application. Furthermore, implementing suitable rules for the choice of s_i, e_i, and x_i allows to pursue and balance various practical optimization goals.

We look at two variants (P1) and (P2) of the procedure corresponding to optimizing the worst slack (P1) or minimizing the length of the topology (P2), respectively.

(P1) The sinks are inserted in an order of non-increasing criticality, where the criticality of a sink $s \in S$ is quantified by $-\sigma_s$ as shown above.

During the i-th execution of the **for**-loop, the new internal vertex x_i is always chosen at the same position as r, and the edge e_i is chosen such that $\sigma(T_i)$ is maximized.

(Note that placing internal vertices at the same position means placing bifurcations at the same position. It does not mean placing several repeaters at the same position during repeater insertion.)

(P2) The sink s_1 is chosen such that $||Pl(r) - Pl(s_1)|| = \min\{||Pl(r) - Pl(s)|| \mid s \in S\}$ and during the i-th execution of the **for**-loop, s_i, $e_i = (u,v)$, and $Pl(x_i)$ are chosen such that

$$l(T_i) = l(T_{i-1}) + ||Pl(u) - Pl(x_i)|| + ||Pl(x_i) - Pl(v)|| + ||Pl(x_i) - Pl(s_i)|| - ||Pl(u) - Pl(v)||$$

is minimized.

5.5.2 Theoretical Properties

Theorem 1. Given an instance of the RESTRICTED REPEATER TREE TOPOLOGY PROBLEM with $\eta = 1/2$, the largest achievable worst slack σ_{opt} equals

$$\sigma^*(S) := \max\left\{\sigma \in \mathbb{R} \,\middle|\, \sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a_s - d||r-s|| - \sigma)\right\rfloor} \leq 1\right\},$$

and (P1) generates a repeater tree topology $T_{(P1)}$ with $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}}$.

Proof: Let $a'_s = a_s - d||r-s||$ for $s \in S$. Let T be an arbitrary repeater tree topology. By the definition of $\sigma(T)$ and the triangle-inequality for $||\cdot||$, we obtain

$$|E_{[r,s]}| - 1 \leq \left\lfloor \frac{1}{c}\left(a_s - \sum_{(u,v)\in E_{[r,s]}} d||u-v|| - \sigma(T)\right)\right\rfloor \leq \left\lfloor \frac{1}{c}(a'_s - \sigma(T))\right\rfloor$$

for every $s \in S$. Since the unique child of the root r is itself the root of a binary subtree of T in which each sink $s \in S$ has depth exactly $|E_{[r,s]}| - 1$, Kraft's inequality (Kraft, 1949) implies

$$\sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma(T))\right\rfloor} \leq \sum_{s\in S} 2^{-|E_{[r,s]}|+1} \leq 1.$$

By the definition of $\sigma^*(S)$, this implies $\sigma(T) \leq \sigma^*(S)$. Since T was arbitrary, we obtain $\sigma_{\text{opt}} \leq \sigma^*(S)$.

It remains to prove that $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}} = \sigma^*(S)$, which we will do by induction on $n = |S|$. For $n = 1$, the statement is trivial. Now let $n \geq 2$. Let s_n be the last sink inserted by (P1), which means that $a'_{s_n} = \max\{a'_s \mid s \in S\}$. Let $S' = S \setminus \{s_n\}$.

Claim

$$\text{frac}\left(\frac{\sigma^*(S)}{c}\right) \in \left\{\text{frac}\left(\frac{a'_s}{c}\right) \,\middle|\, s \in S'\right\} \tag{5.4}$$

where $\text{frac}(x) := x - \lfloor x \rfloor$ denotes the fractional part of $x \in \mathbb{R}$.

Proof of the claim: If $\frac{1}{c}(a'_s - \sigma^*(S)) \notin \mathbb{Z}$ for every $s \in S$, then there is some $\epsilon > 0$ such that $\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S))\right\rfloor = \left\lfloor \frac{1}{c}(a'_s - (\sigma^*(S) + \epsilon))\right\rfloor$ for every $s \in S$, which immediately implies a contradiction to the definition of $\sigma^*(S)$ as in the statement of the theorem. Therefore, $\frac{1}{c}(a'_s - \sigma^*(S))$ is an integer for at least one $s \in S$. If $\frac{1}{c}(a'_s - \sigma^*(S))$ is an integer for some $s \in S'$, then (5.4) holds. Hence, if the claim is false, then $\frac{1}{c}(a'_{s_n} - \sigma^*(S)) \in \mathbb{Z}$ and $\frac{1}{c}(a'_s - \sigma^*(S)) \notin \mathbb{Z}$ for every $s \in S'$. Since $a'_{s_n} - \sigma^*(S) \geq a'_s - \sigma^*(S)$ for every $s \in S'$, this implies

$$\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor > \max\left\{\left\lfloor \frac{1}{c}\left(a'_s - \sigma^*(S)\right)\right\rfloor \,\middle|\, s \in S'\right\}. \tag{5.5}$$

By the definition of $\sigma^*(S)$, we have

$$\Sigma := \sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S))\right\rfloor} \leq 1.$$

Considering the least significant non-zero bit in the binary representation of Σ, the strict inequality (5.5) implies that this bit corresponds to $2^{-\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor}$. This implies that

$$\sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S))\right\rfloor} \leq 1 - 2^{-\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor}.$$

Now, for some sufficiently small $\epsilon > 0$, we obtain

$$\sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - (\sigma^*(S)+\epsilon))\right\rfloor} = 2^{-\left\lfloor \frac{1}{c}\left(a'_{s_n} - \sigma^*(S)\right)\right\rfloor+1} + \sum_{s\in S'} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S))\right\rfloor} \leq 1$$

which contradicts the definition of $\sigma^*(S)$ and completes the proof of the claim. □

Let $T'_{(P1)}$ denote the tree produced by (P1) just before the insertion of the last sink s_n. By induction, $\sigma\left(T'_{(P1)}\right) = \sigma^*(S')$.

First, we assume that there is some sink $s' \in S'$ such that within $T'_{(P1)}$

$$|E_{[r,s']}| - 1 < \left\lfloor \tfrac{1}{c}\left(a'_{s'} - \sigma^*(S')\right)\right\rfloor.$$

Choosing e_n as the edge of $T'_{(P1)}$ leading to s', results in a tree T such that

$$\sigma^*(S) \geq \sigma_{\text{opt}} \geq \sigma\left(T_{(P1)}\right) \geq \sigma(T) = \sigma^*(S') \geq \sigma^*(S),$$

which implies $\sigma\left(T_{(P1)}\right) = \sigma_{\text{opt}} = \sigma^*(S)$.

Next, we assume that within $T'_{(P1)}$

$$|E_{[r,s]}| - 1 = \left\lfloor \tfrac{1}{c}\left(a'_s - \sigma^*(S')\right)\right\rfloor$$

for every $s \in S'$. This implies

$$\sum_{s\in S} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S'))\right\rfloor} > \sum_{s\in S'} 2^{-\left\lfloor \frac{1}{c}(a'_s - \sigma^*(S'))\right\rfloor} = 1$$

and hence $\sigma^*(S) < \sigma^*(S')$. By (5.4), we obtain

$$\begin{aligned}
\sigma^*(S) &\leq \max\left\{\sigma \,\middle|\, \sigma < \sigma^*(S'),\ \text{frac}\left(\tfrac{\sigma}{c}\right) \in \left\{\text{frac}\left(\tfrac{a'_s}{c}\right) \,\middle|\, s\in S'\right\}\right\} \\
&= \max\left\{\sigma \,\middle|\, \sigma < \sigma^*(S'),\ \text{frac}\left(\tfrac{\sigma - \sigma^*(S')}{c}\right) \in \left\{\text{frac}\left(\tfrac{a'_s - \sigma^*(S')}{c}\right) \,\middle|\, s\in S'\right\}\right\} \\
&= c\max\left\{x \,\middle|\, x < \tfrac{\sigma^*(S')}{c},\ \text{frac}\left(x - \tfrac{\sigma^*(S')}{c}\right) \in \left\{\text{frac}\left(\tfrac{a'_s - \sigma^*(S')}{c}\right) \,\middle|\, s\in S'\right\}\right\} \\
&= c\left(\tfrac{\sigma^*(S')}{c} - 1 + \max\left\{\text{frac}\left(\tfrac{a'_s - \sigma^*(S')}{c}\right) \,\middle|\, s\in S'\right\}\right) \\
&= \sigma^*(S') - c(1-\delta)
\end{aligned}$$

for
$$\delta = \max\left\{\text{frac}\left(\tfrac{a'_s - \sigma^*(S')}{c}\right) \;\middle|\; s \in S'\right\}.$$
If $s' \in S'$ is such that
$$\delta = \text{frac}\left(\tfrac{a'_{s'} - \sigma^*(S')}{c}\right),$$
then choosing e_n as the edge of $T'_{(P1)}$ leading to s', results in a tree T such that
$$\sigma^*(S) \geq \sigma_{\text{opt}} \geq \sigma\Big(T_{(P1)}\Big) \geq \sigma(T) = \sigma^*(S') - c(1-\delta) \geq \sigma^*(S),$$
which implies $\sigma\Big(T_{(P1)}\Big) = \sigma_{\text{opt}} = \sigma^*(S)$ and completes the proof. □

Theorem 2. (P2) generates a repeater tree topology T for which $l(T)$ is at most the total length of a minimum spanning tree on $\{r\} \cup S$ with respect to $||\cdot||$.

Proof: Let $n = |S|$ and for $i = 0, 1, \ldots, n$, let $\overline{T}_i$ denote the forest which is the union of the tree produced by (P2) after the insertion of the first i sinks and the remaining $n - i$ sinks as isolated vertices. Note that $\overline{T}_0$ has vertex set $\{r\} \cup S$ and no edge, while for $1 \leq i \leq n$, $\overline{T}_i$ has vertex set $\{r\} \cup S \cup \{x_j \mid 2 \leq j \leq i\}$ and $2i - 1$ edges.

Let $F_0 = (V(F_0), E(F_0))$ be a spanning tree on $V(F_0) = \{r\} \cup S$ such that
$$l(F_0) = \sum_{(u,v)\in E(F_0)} ||u - v||$$
is minimum. For $i = 1, 2, \ldots, n$, let $F_i = (V(F_i), E(F_i))$ arise from
$$\Big(V\Big(\overline{T}_i\Big), E(F_{i-1}) \cup E\Big(\overline{T}_i\Big)\Big)$$
by deleting an edge $e \in E(F_{i-1}) \cap E(F_0)$ which has exactly one end vertex in $V(T_{i-1})$ such that F_i is a tree. (Note that this uniquely determines F_i.)

Since (P2) has the freedom to use the edges of F_0, the specification of the insertion order and the locations of the internal vertices in (P2) imply that
$$l(F_0) \geq l(F_1) \geq l(F_2) \geq \ldots \geq l(F_n).$$
Since $F_n = T_n$ the proof is complete. □

For the ℓ_1-norm, the well-known result of Hwang (1976) together with Theorem 2 imply that (P2) is an approximation algorithm for the ℓ_1-minimum Steiner tree on the set $\{r\} \cup S$ with approximation guarantee 3/2.

We have seen in Theorem 1 and Theorem 2 that different insertion orders are favourable for different optimization scenarios such as optimizing for worst slack or minimum netlength.

Alon and Azar (1993) gave an example showing that for the online rectilinear Steiner tree problem the best achievable approximation ratio is $\Theta(\log n / \log\log n)$,

where n is the number of terminals. Hence, inserting the sinks in an order disregarding the locations, like in (P1), can lead to long Steiner trees, no matter how we decide where to insert the sinks.

The next example shows that inserting the sinks in an order different from the one considered in (P1) but still choosing the edge e_i as in (P1) results in a repeater tree topology whose worst slack can be much smaller than the largest achievable worst slack.

Example 1. Let $c = 1$, $d = 0$ and $a \in \mathbb{N}$. We consider the following sequences of $-a$'s and 0's

$$\begin{aligned} A(1) &= (-a, 0), \\ A(2) &= (A(1), -a, 0), \\ A(3) &= (A(2), -a, \underbrace{0, \ldots\ldots, 0}_{1+(2^1-1)(a+2)}), \\ A(4) &= (A(3), -a, \underbrace{0, \ldots\ldots\ldots, 0}_{1+(2^2-1)(a+2)}), \ldots, \end{aligned}$$

i.e. for $l \geq 2$, the sequence $A(l)$ is the concatenation of $A(l-1)$, one $-a$, and a sequence of 0's of length $1 + \left(2^{l-2} - 1\right)(a+2)$.

If the entries of $A(l)$ are considered as the required arrival times of an instance of the Restricted Repeater Tree Topology Problem, then Theorem 1 together with the choice of c and d imply that the largest achievable worst slack for this instance equals

$$\left\lfloor -\log_2 \left(l2^a + \left(1 + \sum_{i=2}^{l} \left(1 + (2^{i-2} - 1)(a+2) \right) \right) 2^0 \right) \right\rfloor .$$

For $l = a + 1$ this is at least $-2 - a - \log_2(a + 2)$.

If we insert the sinks in the order as specified by the sequences $A(l)$, and always choose the edge into which we insert the next internal vertex such that the worst slack is maximized, then the following sequence of topologies can arise: $T(1)$ is the topology with exactly two sinks at depth 2. The worst slack of $T(1)$ is $-(a + 1)$. For $l \geq 2$, $T(l)$ arises from $T(l-1)$ by (a) subdividing the edge of $T(l-1)$ incident with the root with a new vertex x, (b) appending an edge (x,y) to x, (c) attaching to y a complete binary tree B of depth $l-2$, (d) attaching to one leaf of B two new leaves corresponding to sinks with required arrival times $-a$ and 0, and (e) attaching to each of the remaining $2^{l-2} - 1$ many leaves of B a binary tree Δ which has $a + 2$ leaves, all corresponding to sinks of arrival times 0, whose depths in Δ are $1, 2, 3, \ldots, a-1, a, a+1, a+1$. Note that this uniquely determines $T(l)$.

Clearly, the worst slack in $T(l)$ equals $-a - l$. Hence for $l = a + 1$, the worst slack equals $-2a - 1$, which differs approximately by a factor of 2 from the largest achievable worst slack as calculated above.

This example, however, does not show that there is no online algorithm for approximately maximizing the worst slack, say up to an additive constant of c.

Recently, Held and Rotter (2013) presented an $O(n \log n)$ algorithm that given $\epsilon > 0$ and an initial topology T_0 solves the RESTRICTED REPEATER TREE TOPOLOGY PROBLEM for $\eta = 1/2$ such that if the worst slack of the instance is non-negative

$$\sigma(T) \geq -d_{node} - \epsilon \max_{s \in S}\{a_s\}$$
$$l(T) < \left(1 + \tfrac{2}{\epsilon}\right) l(T_0) + \frac{2n \cdot d_{node}}{\epsilon}$$

with $n := |S|$. The length of a topology $l(T)$ is the sum of edge lengths in T. Here, T_0 can be derived from any Steiner tree, for instance a minimum Steiner tree or an approximation of it.

5.6 Topology Generation Algorithm

We now extend our topology generation from Algorithm 2. The basic structure remains the same. However, we now allow the distribution of node delays using η between 0 and $1/2$.

Algorithm 3 shows the version of the topology generation that we use to construct repeater trees. We use the delay model but only required arrival times are updated during the process. In terms of Algorithm 2 the choice of s_i, $Pl(x_i)$, and e_i is as follows:

- Similar to (P1), the sinks are ordered by non-increasing criticality.
- $Pl(x_i)$ is chosen such that the netlength increase of the topology is minimized. In general, the new nodes do not lie on top of the root node as in (P1).
- The edge e_i is chosen such that the weighted sum of resulting topology slack and netlength increase is minimized.

We hope to reduce netlength by connecting the sinks as short as possible to the chosen candidate edge. However, by doing so, the slack is no longer guaranteed to be optimal as shown in Theorem 1 for (P1).

The algorithm first sorts the sinks according to criticality. Then, it connects the most critical sink directly to the root and initializes the topology and rat function accordingly (lines 1–4). The next step is to iterate over all sinks and to add them into the topology one after another. To determine e_i, we compute for each existing edge (v, w) the required arrival time at the root that we would get if we choose the edge (lines 9–26). The Steiner point is always in the bounding box between v and w due to the ℓ_1-norm used. The netlength increase is therefore $||z - Pl(s_i)||$. Given the resulting RAT rat at the root, we choose the edge that maximizes

$$\xi(\min\{rat, 0\}) - (1 - \xi)||z - Pl(s_i)||.$$

Algorithm 3 Topology Generation Algorithm

Input: An instance of the RESTRICTED REPEATER TREE TOPOLOGY PROBLEM
Output: A topology consisting of tree $T = (V, E)$ and embedding Pl

1: For each sink $s \in S$, compute the criticality σ_s
2: Sort S such that $\sigma_{s_1} \leq \sigma_{s_2} \leq \cdots \leq \sigma_{s_{|S|}}$
3: $V := \{r, s_1\}$ $E := \{(r, s_1)\}$
4: $rat_T(s_i) := a_{s_i}$ for $1 \leq i \leq |S|$
5: **for** $i := 2, \ldots, |S|$ **do**
6: $e_i := \emptyset$
7: $bval := -\infty$
8: $bdist := \infty$
9: **for all** $(v, w) \in E$ **do** ▷ Search for the best edge to connect s_i to
10: Choose $z \in \mathbb{R}^2$ minimizing $||z - Pl(v)|| + ||z - Pl(w)|| + ||z - Pl(s_i)||$
11: $\alpha_1 := rat_{s_i} - d_{wire}||z - Pl(s_i)||$
12: $\alpha_2 := rat_w - d_{wire}||z - Pl(w)||$
13: $rat := \max_{\eta d_{node} \leq b \leq (1-\eta) d_{node}} \min\{\alpha_1 - b, \alpha_2 - (d_{node} - b)\}$
14: **for** $(x, y) \in E_{[\delta^+(r), w]}$ traversed bottom-up **do**
15: Let u be the sibling of y
16: $\alpha_1 := rat_u - d_{wire}||Pl(x) - Pl(u)||$
17: $\alpha_2 := rat - d_{wire}||Pl(x) - Pl(y)||$
18: $rat := \max_{\eta d_{node} \leq b \leq (1-\eta) d_{node}} \min\{\alpha_1 - b, \alpha_2 - (d_{node} - b)\}$
19: **end for**
20: $val := \xi \min\{rat - d_{wire}||Pl(r) - Pl(\delta^+(r))||, 0\} - (1 - \xi)||z - Pl(s_i)||$
21: **if** $val > bval$ or $val = bval \wedge ||z - Pl(s_i)|| < bdist$ **then**
22: $bval := val$
23: $bdist := ||z - Pl(s_i)||$
24: $e_i := (v, w)$
25: **end if**
26: **end for**

27: Create a new Steiner node x_i
28: $V := V \cup \{s_i, x_i\}$
29: $E := E \setminus \{(v^*, w^*)\} \cup \{(v^*, x), (x, w^*), (x, s_i)\}$ with $e_i = (v^*, w^*)$
30: $Pl(x) := z \in \mathbb{R}^2$ minimizing $||z - Pl(v^*)|| + ||z - Pl(w^*)|| + ||z - Pl(s_i)||$
31: **for** $(v, w) \in E_{[\delta^+(r), s_i]}$ traversed bottom-up **do**
32: Let y be the sibling of w
33: $\alpha_1 := rat_y - d_{wire}||Pl(v) - Pl(y)||$
34: $\alpha_2 := rat_w - d_{wire}||Pl(v) - Pl(w)||$
35: $rat_T(x) := \max_{\eta d_{node} \leq b \leq (1-\eta) d_{node}} \min\{\alpha_1 - b, \alpha_2 - (d_{node} - b)\}$
36: **end for**
37: **end for**

If two solutions have the same value, we choose the shorter connection. The parameter ξ allows us to scale between topologies that optimize the worst slack and short topologies depending on the objective of the REPEATER TREE PROBLEM.

After choosing e_i, we add the new sink to the tree splitting e_i and update rat on all affected edges (lines 27–37).

Given an internal node v with children u_1 and u_2, let $\alpha_i := rat_T(u_i) - d||Pl(u_i) - Pl(v)||$ for $i \in \{1, 2\}$. By setting

$$rat_T(v) := \max_{\eta d_{node} \le b \le (1-\eta) d_{node}} \min\{\alpha_1 - b, \quad \alpha_2 - (d_{node} - b)\}$$

we implicitly maintain the node delay for each edge. If without loss of generality $\alpha_1 < \alpha_2$, then $rat_T(v)$ can be computed in constant time by

$$rat_T(v) := \alpha_1 - \eta d_{node} - \tfrac{1}{2} \max\{(1 - 2\eta) d_{node} - (\alpha_2 - \alpha_1), \quad 0\}.$$

The additional delays of the outgoing edges,

$$\begin{aligned} d_{node}((v, u_1)) &= \alpha_1 - rat_T(v) \\ d_{node}((v, u_2)) &= d_{node} - d_{node}((v, u_2)) \le \alpha_2 - rat_T(v), \end{aligned}$$

satisfy our requirements on the node delay.

Theorem 3. The worst case running time of the algorithm is $O(n^3)$ if $n = |S|$ and the best case running time is $\Omega(n^2)$.

Proof. There are $n - 1$ iterations of the outer loop of the algorithm (lines 5–37). Each iteration removes one edge from the tree and adds three new. When e_i is searched, there are $1 + 2(i - 2) = 2i - 3$ edges in the tree. The loop searching for e_i (lines 9–26) is called $\sum_{i=2}^{n}(2i - 3) \in \Theta(n^2)$ times. The loop computing the RAT at the root (lines 14–19) and the loop updating the RATs after sink insertion (lines 31-36) can be stopped if the RAT does not change at a node. In such a case, no updates will be done on the path to the root. In the best case, both loops only perform a constant number of updates resulting in an overall best case running time of $\Omega(n^2)$. In the worst case, the inner loop (lines 14–19) iterates linearly in the size of the graph resulting in an overall worst case running time of $O(n^3)$. □

5.6.1 Handling High Fanout Trees

As we will show in our experimental results, the running time of our algorithm is extremely small for instances up to 1 000 sinks. Nevertheless, our topology generation, as described above, has a cubic running time. There are instances with several hundred thousand sinks on actual designs, for which this would lead to intolerable running times.

One way to reduce the running time would be to consider only the nearest k edges while inserting a sink, where k is some positive integer. This would require to

store the edges as rectangles in a suitable geometric data structure (e.g. a k-d-tree). However, we chose a different approach.

For instances with more than 1 000 sinks, we first apply a clustering algorithm to all sinks, except for the 100 most critical ones if $\xi > 0$. More precisely, we find a partition $S' = S_1 \dot{\cup} \cdots \dot{\cup} S_k$ of the set S' of less critical sinks, and Steiner trees T_i for S_i ($i = 1, \ldots, k$), such that the total capacitance of T_i plus the input capacitances of S_i is at most *maxcap* (see Section 4.1.6). Among such solutions we try to minimize the total wire capacitance plus k times the input capacitance of the repeater t^*.

For this facility location problem, we use an approximation algorithm by Maßberg and Vygen (2008), which generates very good solutions in $O(|S| \log |S|)$ time. We introduce an appropriate repeater for each component; its input pin constitutes a new sink. This typically reduces the number of sinks by at least a factor of 10. If the number of sinks is still greater than 1 000, we iterate the clustering step. Finally, we run our topology generation algorithm as described above.

5.7 Blockages

The topology generation so far did not consider blockages or congestion. Nothing prevents Steiner points from being located on blockages or topology segments from having long intersections over them.

For the vast majority of instances, blockages do not play a role because there are none in the bounding box of the involved points. However, some instances can only be constructed properly if one navigates around blockages or considers congestion.

We handle blockages in the second step of our topology generation algorithm. First, we iterate over the topology bottom-up and move each Steiner point s to a free location. Our algorithm searches for the nearest free location in each direction that minimizes the sum of ℓ_1-distances between s and its neighbours. Then, we search a buffer-aware shortest path (see Section 4.2.3) within the blockage grid for each topology edge that crosses a blockage. A similar approach is shown by Zhang et al. (2012) and improved in Zhang and Pan (2014). They take an input topology and calculate slew degradations over blockages. In case of violations, they formulate an ILP and solve it to find a good replacement topology. Huang and Young (2012) propose a similar solution. Held and Spirkl (2014) propose a fast 2-approximation algorithm to create rectilinear Steiner trees that can cross obstacles for a limited distance.

After performing the shortest path search, some edges might still cross blockages that are small enough to pass over. These edges are splitted on blockage boundaries such that each topology segment is either completely blocked or free. Neighbouring blocked edges on a chain are merged into a single one.

5.8 Plane Assignment

So far, we only used one buffering mode and its choice is often appropriate for shorter connections. For bridging large connections, it might be better to use buffering modes on higher planes than the default ones. Such buffering modes are often faster and use less placement space due to larger repeater spacing.

The third and final step of our topology generation is plane assignment. We use a simple greedy routine to assign buffering modes to a computed topology. They are then used by our repeater insertion routine. We restrict ourselves to the fastest buffering mode for each wire mode. For each buffering mode, we know the repeater spacing.

The algorithm processes the nodes of the topology in a BFS traversal starting at the root. If the slack of the node is negative and if it is not too congested according to the congestion map, then we consider the edge between the node and its parent. The edge gets the fastest buffering mode assigned for which repeater spacing is smaller than half the length of the edge. After changing an assignment, the slack values are updated.

The algorithm runs in time $O(m \log b)$ where m is the number of edges in the topology and b is the number of buffering modes we might assign. The buffering modes can be sorted by the repeater spacing. If there is a buffering mode with higher spacing and worse timing than another one, then it can be removed because it will never be inserted. The resulting list is sorted increasingly by spacing and decreasingly by delay per length. Thus, for an edge, the best buffering mode can be found by binary search in $O(\log b)$ time. It is not necessary to recompute the whole timing after each assigment. It is sufficient to propagate an arrival time delta to the children of the node such that the timing updates are constant for each node.

The greedy routine is suitable at design stages where the placement of the circuits and the timing are premature. In later design stages, it is desirable to consider congestion better and to optimize plane assignment for best slack. We will present an extension to the dynamic program used for repeater insertion that maximizes slack and takes advantage of higher planes.

5.9 Global Wires as Topologies

As already mentioned, routing congestion is one of the biggest problems for optimizing current chip designs. The approach described in Section 5.7 works locally without considering the big picture. A global router (see for example Müller (2009)), on the other hand, optimizes the distribution of nets over the whole routing space. A recent trend in the industry is to use the result of global routing for repeater tree topologies.

The advantage of using global wires is a reduced expected congestion. However, global routing has to be adapted to yield results suitable for repeater insertion. One has to consider timing, placement space, blockages and instance sizes.

A global routing that will later be buffered should not use wires that cross blockages for too long because they cannot be processed in repeater insertion without electrical and timing violations. The available placement space has also to be considered. The number of necessary repeaters can be estimated using suitable buffering modes for the wires.

The global router should also consider the timing criticality of nets and sinks when it creates routes. Otherwise, uncritical nets might get short routes at the cost of detours in critical nets. One approach might be to use the topology algorithm presented here as a subroutine within the global router when it has to calculate Steiner trees for nets.

The input presented to the global router is often stripped from all buffers and unnecessary inverters. However, it is often not possible to remove some inverters if one wants to preserve logical correctness of the design. The global router will then use the placement of the inverter as a constraint to the routing. It will consider two nets for a single instance and connect the sink of the first net with the current inverter and the source of the second net. It is better to consider the inverter and both nets as a single net. This is also the case if existing repeaters are not stripped from the design[2].

[2] see also Section 7.6.1.

6 Repeater Insertion

Finding a topology for a repeater tree instance is the first step in our approach to solve the Repeater Tree Problem. The second step is to insert repeaters along the topology to create a feasible solution. For this we consider the Repeater Insertion Problem and describe how our algorithm solves it.

Instance: An instance (I, T, Bl, M, F) consists of
- an instance I of the Repeater Tree Problem,
- a topology T with embedding Pl connecting the root and sinks of I,
- a set $Bl \in E(T)$ of edges that are blocked,
- a set of buffering modes M, and
- buffering mode assignments $F : E(T) \to M$.

Task: Find a feasible solution of the Repeater Tree Problem for I minimizing costs such that each repeater lies on a shortest path between the endpoints of a topology edge and all sinks reachable from the repeater in the final tree are also reachable from the edge in the topology.

Figure 6.1: Repeater Insertion Problem

The dominant approach to solve the Repeater Insertion Problem is dynamic programming. An extensive survey of the dynamic programming approach can be found in Alpert et al. (2008), Sections 26.4 – 26.6. We give a short summary of existing work in Section 6.3.

Our main contribution is the repeater insertion of our Fast Buffering algorithm presented in Section 6.2 that, in practice, is considerably faster than the standard dynamic program. Our routine can be characterized as a version of the dynamic program that keeps only one solution at a time. Several heuristics are used to choose a solution that will lead to an overall good solution.

A substantial difference to the dynamic programming approach is that our algorithm is able to change the topology in order to reduce the number of repeaters inserted for preserving parity constraints. Figure 6.2 shows an extreme example where different topologies for the same sink set result in a huge difference in the minimum number of repeaters necessary to realise each of them. Given topology a), our algorithm will often create a solution that lies between both extremes depending on the criticality of the instance. However, our solution will still fulfill the constraints from the Repeater Insertion Problem.

Finally, the dynamic program depends on precomputed repeater positions. In

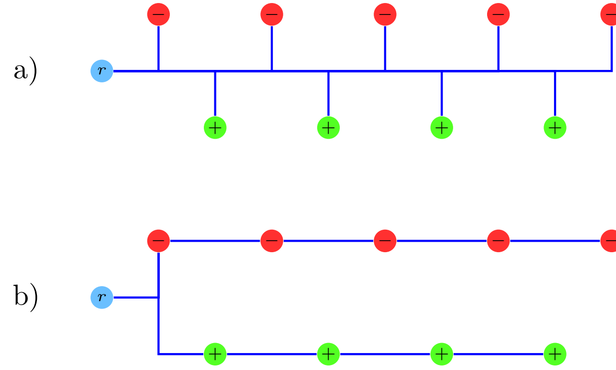


Figure 6.2: While topology b) requires only one inverter to realise the indicated sink parities, topology a) would require five.

contrast, our algorithm is free to choose any position along an unblocked edge of the input topology.

Our repeater insertion algorithm consists of two parts. In a first step, we assign delay efforts to the edges of the topology by solving a Deadline Problem. This allows us to buffer parts of the topology with less effort and leads to lower resource usage. In a second step, we replace the topology by a repeater tree in a bottom-up fashion.

We present in Section 6.3 how we use the standard dynamic programming technique to improve the solution found by the Fast Buffering algorithm.

The buffering algorithm we present here is an extension to joint work with Stephan Held, Dieter Rautenbach and Jens Vygen (Bartoschek et al., 2009, 2007b).

6.1 Computing Required Arrival Time Targets

As shown in Section 4.1.1, it is possible to buffer a long line with different delay and power consumption characteristics. For topology generation, we assumed that each edge is buffered such that the fastest delay using the default wire modes can be achieved.

After computing arrival times and required arrival times for all nodes of the topology using our delay model, there are sinks that have non-positive slack even if the fastest buffering mode is used on the path from the root. It is obvious that we want to buffer the paths as fast as possible to keep timing constraint violations small. However, other sinks and subtrees respectively might have positive slack using the fastest buffering mode. Each edge of such a subtree can potentially be slowed down to reduce the overall power consumption of the resulting repeater tree. As input to the Repeater Insertion Problem each edge has a buffering mode m assigned. We have the possibility to choose another buffering mode. At this point in time, we do not want to change the layer assignment of the edges. Therefore, we restrict ourselves to the alternative buffering modes in M_m (see Section 4.1.3).

Instance: An instance consists of

- an instance I of the REPEATER TREE PROBLEM,
- a topology T for I,
- a set of buffering modes M,
- buffering mode assignment for each edge $F : E(T) \to M$, and
- a maximal subtree T_z of T rooted at $z \in V(T)$ such that using our delay model $rat_T(z) - at_T(z) > 0$

.

Task: Let E' be the set of edges reachable from z ($E(T_z)$) including the edge leading to z if $z \neq r$.
Find an assignment $F' : E(T) \to M$ with $F'(e) = F(e)$ if $e \notin E'$ and $F'(e) \in M_{F(e)}$ and for all sinks reachable from z $rat_T(s) - at_T(s) \geq 0$ such that the total cost

$$\sum_{e=(v,w)\in E'} F'(e)_p ||Pl(v) - Pl(w)||$$

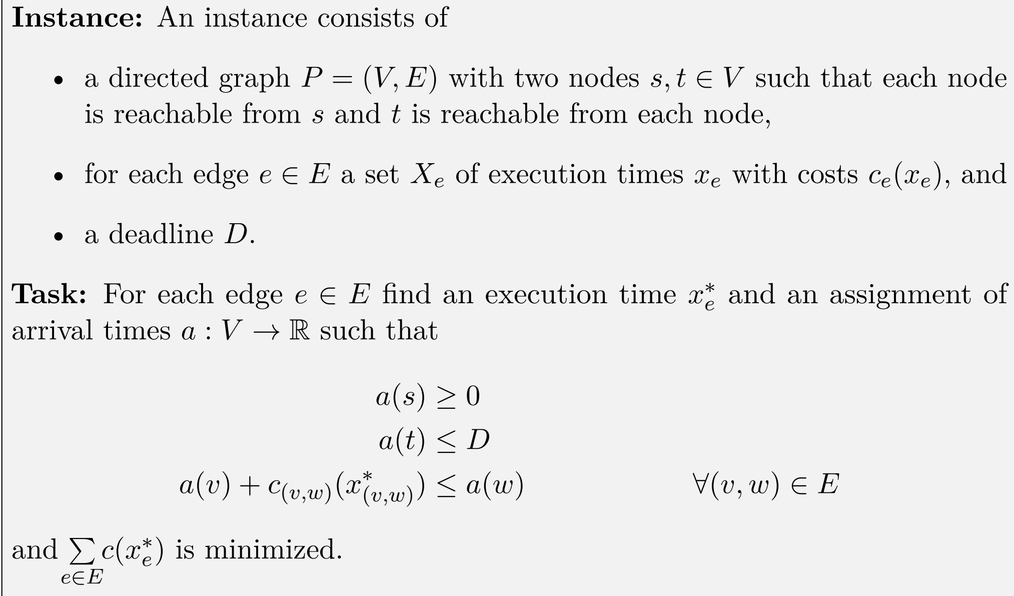
is minimized.

Figure 6.3: BUFFERING MODE ASSIGNMENT PROBLEM

We call the problem of assigning buffering modes to a subtree BUFFERING MODE ASSIGNMENT PROBLEM. It is shown in Figure 6.3. If the slack at the root of our topology is positive, then the whole tree is an instance to buffering mode assignment. Otherwise, each maximal subtree with positive slack is considered separately. The initial assignment for such a subtree is a feasible assignment but probably not the cheapest one. We find cheaper solutions, but we do not let the slack at the root become negative. Thus, the result of buffering mode assignment does not change RATs outside of the considered subtree. The problem can be solved independently for each subtree.

The BUFFERING MODE ASSIGNMENT PROBLEM is very similar to the DISCRETE DEADLINE PROBLEM (see for example Skutella (1998)) as a special case of the TIME-COST TRADEOFF PROBLEM (Kelley, 1961; Fulkerson, 1961). Figure 6.4 shows the DISCRETE DEADLINE PROBLEM. The graph P in an instance of the problem is called *project graph*. Each edge e corresponds to a task that has to be executed and X_e is the set of possible alternatives to finish the task with different execution times and costs.

An instance to the BUFFERING MODE ASSIGNMENT PROBLEM can be transformed to an instance of the DISCRETE DEADLINE PROBLEM. We show how to transform a subtree rooted at z with parent $parent(z)$. The transformation is done by (a) using the subtree induced by E' as a project graph, (b) adding a new node s and an edge $(s, parent(z))$ with the single execution time $at_T(parent(z))$ and costs 0, (c) adding a new node t and edges (s_i, t) for all sinks s_i reachable from z with single execution time $-rat_{s_i}$ and costs 0, (d) setting execution times according to valid buffering modes for all topology edges, and (e) setting the deadline to 0. Each topology edge



Instance: An instance consists of

- a directed graph $P = (V, E)$ with two nodes $s, t \in V$ such that each node is reachable from s and t is reachable from each node,
- for each edge $e \in E$ a set X_e of execution times x_e with costs $c_e(x_e)$, and
- a deadline D.

Task: For each edge $e \in E$ find an execution time x_e^* and an assignment of arrival times $a : V \to \mathbb{R}$ such that

$$\begin{aligned} a(s) &\geq 0 \\ a(t) &\leq D \\ a(v) + c_{(v,w)}(x^*_{(v,w)}) &\leq a(w) \qquad \forall (v,w) \in E \end{aligned}$$

and $\sum_{e \in E} c(x_e^*)$ is minimized.

Figure 6.4: Discrete Deadline Problem

(v, w) has the initial buffering mode $F((v, w))$ assigned. We set

$$X_{(v,w)} := \left\{ x_m \;\middle|\; m \in M_{F((v,w))} \right\}$$

with

$$\begin{aligned} x_m &:= m_d ||Pl(v) - Pl(w)|| \\ c_{(v,w)}(x_m) &:= m_p ||Pl(v) - Pl(w)||. \end{aligned}$$

If z is the root of the whole topology, then we just connect s to z during project graph construction. As mentioned earlier, each of our instances to the Buffering Mode Assignment Problem has a feasible solution. It follows that using the arrival times of the delay model as arrival times a in the Discrete Deadline Problem is a feasible solution. On the other hand, any feasible solution to the Discrete Deadline Problem results in a feasible buffering mode assignment if we choose for each the buffering mode m if the edge has execution time x_m.

The Discrete Deadline Problem is NP-hard, but Halman et al. (2008) have shown that there exists a FPTAS on series-parallel networks like repeater tree topologies.

6.1.1 Linear Time-cost Tradeoff

As our delay model is only a rough approximation of the reality after buffering, it makes no sense to spend too much effort into solving the Buffering Mode Assignment Problem exactly. Furthermore, it turns out that for our purpose, it is sufficient to solve a linear relaxation of the problem.

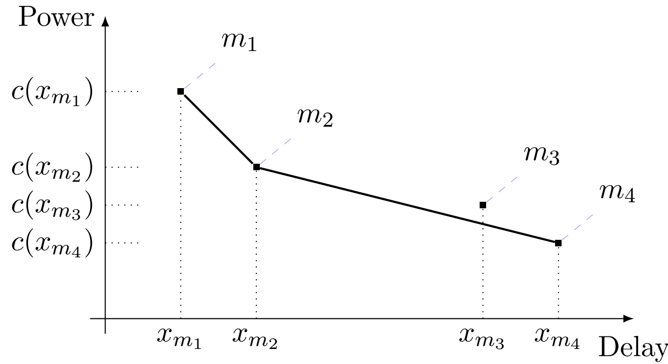


Figure 6.5: Piecewise-linear Relaxation of Buffering Modes. The buffering mode m_3 is dominated by a linear combination of m_2 and m_3 and can be cancelled.

If there is an edge $e = (v, w)$ with execution time x_e^* between execution times x_a and x_b for buffering modes a, b such that

$$x_e^* = \alpha x_a + (1 - \alpha)x_b,$$

then e can be divided into two edges such that the first edge has length $\alpha||Pl(v) - Pl(w)||$ and buffering mode a and the second edge has length $(1-\alpha)||Pl(v)-Pl(w)||$ with buffering mode b.

The linear relaxation of the problem uses piecewise linear time-cost functions for each edge. For example, Figure 6.5 shows how delays and costs are relaxed for an edge with four alternative buffering modes $\{m_1, m_2, m_3, m_4\}$ that are sorted by increasing delay. After removing buffering modes (m_3 in the example) that are dominated by linear combinations of others, the result is a convex piecewise-linear time-cost function. It approximates the non-dominated part of the delay-power tradeoff curve of a wire mode. One example is the red part of the curve shown in Figure 4.4.

Now the Time-Cost Tradeoff Problem can be solved efficiently by solving a Minimum-Cost Flow Problem (MCF). The construction is described in Fulkerson (1961) or Lawler (2001). Algorithms to solve the MCF problems can be found in Korte and Vygen (2012) Chapter 9. The input to the MCF algorithm is the project graph where each edge is replaced by a chain of at most $|M|$, the number of buffering modes and the maximum number of sampling points for a time-cost function, edges. Then, each edge in the chain is doubled. After solving the MCF, we get a node potential that corresponds to an arrival time assignment a.

Theorem 4. The effort assignment adds at most $|S|$ new vertices and edges into the topology.

Proof. The tree T in the spanning tree structure of any basic spanning tree solution of the MCF spans all vertices in our original topology. For any r-s-path with $s \in S$,

it omits at most one edge. By complementary slackness, all edges in T define integral buffering modes. Therefore there are at most $|S|$ fractional edges which are divided. □

Note that the Network Simplex Algorithm always maintains basic solutions. Furthermore, one can transform non-basic optimum solutions into basic ones in at most $|E|$ pivots.

6.1.2 Effort Assignment Algorithm

The algorithm we use to solve the Buffering Mode Assignment Problem is outlined in Algorithm 4 and called AssignEffort. The input is a topology and allowed buffering modes for every edge.

Algorithm 4 Buffering Mode Assignment Problem

1: **procedure** AssignEffort(T)
2: Compute timing using the fastest buffering mode.
3: **for all** $n \in |V(T)|$ with $slack(n) > 0$ and $slack(parent(n)) \leq 0$ **do**
4: Create MCF instance I for subtree rooted at n
5: Solve I
6: Assign buffering modes according to node potentials in I
7: **end for**
8: **end procedure**

First, we assign the fastest buffering mode to each edge of the topology and recompute the timing using Algorithm 1 (TimeTree). For all sinks with slack smaller or equal to 0 the fastest buffering mode is kept.

Then, we identify instances to the Buffering Mode Assignment Problem using a DFS search and process each subtree separately[1] as a Deadline Problem.

We solve the min-cost-flow formulation and compute node potentials. After having filtered out the sinks with non-positive slacks, we know that the problem is feasible and that the potentials are feasible.

Given potentials $\pi : V(T_n) \to \mathbb{R}$, the delay we want to assign to an edge (v, w) is $\pi(v) - \pi(w)$. For non-fractional edges the according buffering mode is used. For fractional edges we do not subdivide the edge as outlined in the previous section. Instead, we just round to the cheapest buffering mode faster than the fractional solution.

After rounding, the delays on all edges correspond to a buffering mode. We update for each edge $e \in E(T_n)$ the buffering mode assignment F accordingly.

[1] Note that it is possible to merge all deadline problems in a single one because they correspond to disjunct subtrees of the topology.

6.2 Repeater Insertion Algorithm

We now describe the main part of our repeater insertion algorithm. The input is an instance of the REPEATER INSERTION PROBLEM (I, T^{in}, Bl, M, F), for example, as it has been computed by our topology generation algorithm. The result will be a repeater tree $R = (T, Pl, R_t, R_W)$[2].

We first update the buffering assignment F of the input topology T^{in} using ASSIGNEFFORT. Then, the algorithm traverses the topology in post-order fashion. We create a pair of so-called clusters at each node of the input topology. During topology traversal, leaf nodes are moved with their clusters towards their parents (MOVE operation). Eventually, the clusters are merged with the clusters at the parent of their node (MERGE operation). The node and the clusters get removed from the topology. At the same time we insert repeaters (mostly inverters) and build up T. Thus, the topology is successively replaced by the final repeater tree.

First, we will explain clusters. Then, we explain the timing model that we use in our algorithm. Finally, we describe the main parts of the algorithm.

6.2.1 Cluster

A cluster C is a triple $(S(C), M(C), P(C))$ which is assigned to a node $V(C)$ in the topology and consists of

- a set of sinks $S(C)$ containing pins corresponding to sinks of the original repeater tree instance as well as input pins of repeaters that have already been inserted earlier,
- a buffering mode $M(C) \in M$ or the empty set, and
- a so-called merge point $P(C) \in \mathbb{R}^2$.

By an empty cluster we mean $(\emptyset, \emptyset, (0, 0))$. The position of a cluster is always the same as the position of the node to which it is assigned $Pl(C) = Pl(V(C))$.

Definition 7. We say that a pair of clusters (C^+, C^-) at a node is in *parallel mode* if the sink sets $S(C^+)$ and $S(C^-)$ are both non-empty.

For a cluster pair (C^+, C^-) in parallel mode, the merge points $P(C^+)$ and $P(C^-)$ are both defined. They store the last location of a cluster where the sink set was changed. It is the location where the cluster pair entered parallel mode if the sink set did not change since then.

Figure 6.6 shows a cluster pair in parallel mode and the merge points for both parities. We depict cluster pairs as two stacked rectangles, a green (above) for positive sinks and a red (below) for negative sinks in all pictures showing clusters. As both clusters are always assigned to a node at the same position, we do not show the node explicitly.

[2]See Section 3.2

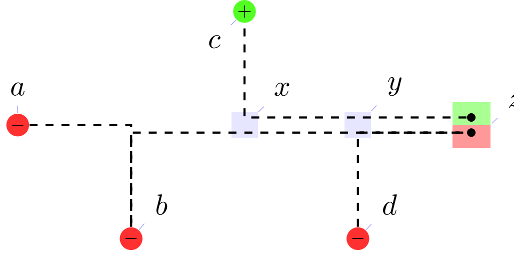


Figure 6.6: Example for a cluster pair (C^+, C^-) in parallel mode. Their current position is $Pl(C^+) = Pl(C^-) = z$. We have $S(C^+) = \{c\}$, $S(C^-) = \{a,b,d\}$, $P(C^+) = x$ and $P(C^-) = y$. Parallel mode was entered at point x; the last negative sink entered at point y.

By moving a cluster and adding repeaters, we want to realize a repeater chain that corresponds to the buffering mode $m := M(C)$ of the cluster. We say a cluster has target slews m_s and a target repeater m_t[3]. The resulting wires either use m_h or m_w as wiring mode depending on the direction.

6.2.2 Initialization

We want to move the nodes of input topology T^{in} but keep the instance sinks at their place. Therefore, we start initialization by replacing each sink $s \in S$ in T^{in} by a new node v_s at the same place. Then, we assign a pair of empty clusters, one for each parity, to each node in the modified topology. Finally, each sink is added to the sink set of the cluster at node v_s with the same parity resulting in cluster $(\{s\}, F(e), (0,0))$ with e being the edge incident to v_s.

We initialize the resulting tree by $T := (S, \emptyset)$.

6.2.3 Timing Model during Repeater Insertion

Our algorithm depends on the timing model during repeater insertion to guide the decisions. In the process of the algorithm, there are three different structures we maintain:

- At the top, there are the remaining parts of the initial topology T^{in} with cluster pairs at all nodes.
- The bottom T is a set of subtrees that will be part of the final repeater tree. At the beginning, the bottom consists of the instance sinks. Each inserted repeater extends T possibly merging subtrees and the result is a tree rooted at the new repeater which is then part of T.
- Clusters connect the topology with the final tree. While each cluster is associated with a node in the topology, its sink set consists of roots in T.

[3]See Section 4.1.3

For the resulting repeater tree we also maintain the node placement Pl, the repeater assignment R_t and the wiring mode assignment R_w.

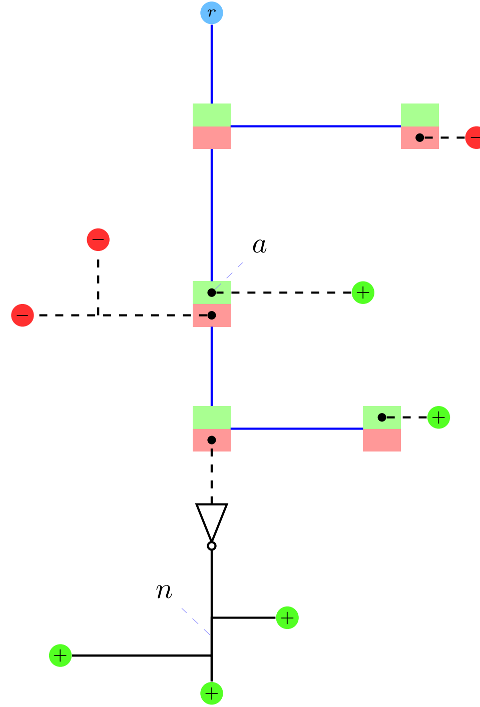


Figure 6.7: Topology, clusters, and resulting tree during repeater insertion. So far, the final tree forest consists of all sinks in clusters and the net n with three sinks and a driving inverter.

Figure 6.7 shows a possible intermediate state of our buffering algorithm. At the top, we have the remaining parts of the topology (blue edges) with cluster pairs at all nodes. The cluster pair at the root is not shown. Dashed black lines show for each cluster a Steiner tree between the cluster and its sinks. Solid lines show the final repeater tree. In this example, the set of final trees consists of net n with its driving repeater and all sinks.

We maintain additional information during the course of the algorithm:

- For each cluster sink s, we know a pair of slew limits $Sl(s)$, a required arrival time function $rat(s)$, and the pin capacitance $cap^{\text{in}}(s)$.
- For each cluster C, we maintain a pair of slew limits $Sl(C)$, a pair of slew targets $St(C)$, the load capacitance $cap(C)$, and a required arrival time function $rat(C)$.
- For each node v in the topology, we have an arrival time $at_{T^{in}}(v)$ and a required arrival time value $rat_{T^{in}}(v)$ coming from our delay model.

Note that during buffering *rat* is a function that, given a cluster sink or a cluster, returns us a required arrival time function which has to be evaluated for a slew. We now explain the data structures in more detail:

Cluster Sinks

Cluster sinks are either instance sinks or input pins of inserted repeaters. For instance sinks, the required arrival time function and slew limit pair are given with the input.

Each inserted repeater (see below) drives a set of cluster sinks. A Steiner tree is created between the repeater and the sinks forming a new net. The new net is then extracted as described in Section 3.2.1 using a minimum Steiner tree.

Given the Elmore delay for each sink, the required arrival times and slew limits can be propagated backwards to the source of the net (see also Section 2.6f) where they are merged. The results can then be propagated to the repeater's input pin. The input pin is then treated as a new cluster sink with slew limits and required arrival times.

Clusters

Each time a cluster is modified, for example, the cluster is moved or a sink is added, we recompute the timing of the cluster.

The cluster and its sinks are treated as a net. A Steiner tree connecting all pins is computed and extracted using the wiring modes of the cluster's buffering mode. Then, similar to the previous section, the rat functions and slew limits are propagated backwards to the root of the Steiner tree.

In addition, the slew target of the cluster's buffering mode is treated as a separate slew limit for each cluster sink and propagated backwards resulting in $St(C)$. The capacitance of a cluster $cap(C)$ is the sum of sink pin capacitances and the wire capacitances of the segments in the Steiner tree.

Topology

Delays in the unbuffered topology are estimated using the delay model introduced in Section 5.1 with parameters d_{node} and η. For each edge e, we have a wiring delay $F(e)_d$ as it is given by the edges buffering mode $F(e)$.

The timing of the topology is calculated by treating each non-empty cluster as a virtual node of the topology connected to its associated node via a zero-length edge. For a cluster C at node v the RAT in the topology $rat_{T^{in}}$ is given by

$$rat_{T^{in}}(C) := \min\{rat^r(C)(F(e)_s^r), rat^f(C)(F(e)_s^f)\} \tag{6.1}$$

with e being the edge pointing to v. We assume that the topology will try to reach the cluster with the edge's target slew $F(e)_s$.

For a node with two non-empty clusters (e.g. node a in Figure 6.7), we assume that there is an additional virtual node with both virtual cluster nodes as children.

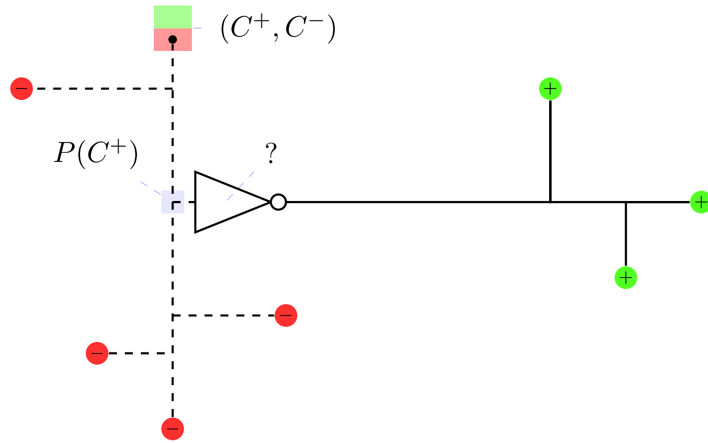


Figure 6.8: An inverter is searched for the positive cluster of pair (C^+, C^-). Its input pin will become a sink in $S(C^-)$. The position of the inverter is $M(C^+)$.

The virtual node is then connected via a zero-length edge to the original node. In the resulting virtual tree, each node has at most two outgoing edges. The required arrival time can then be computed for each node discarding values for virtual nodes.

6.2.4 Finding a new Repeater

At certain stages of the buffering algorithm, we insert a repeater that drives the sinks of a cluster or test the effect of such an insertion. We describe this operation for a cluster C^+, which is part of a cluster pair (C^+, C^-). The operation is completely analogous (exchanging $+$ and $-$) for cluster C^-. It will be applied only to non-empty clusters.

After inserting a new repeater for C^+, its input pin is a new sink that is inserted into an existing cluster C'. This cluster can be

1. C^+ itself (if we insert a buffer along a path),

2. C^- (if we insert an inverter along a path), or

3. a cluster from a different cluster pair (during a Merge operation).

The new repeater is going to drive all sinks in $S(C^+)$. The location of the new repeater depends on the mode of the cluster pair (C^+, C^-). If the cluster pair is in parallel mode, we insert a repeater at position $P(C^+)$. If the cluster pair is not in parallel mode, the location of the new repeater is the current position of the cluster $Pl(C^+)$.

The operation is called InsertRepeater. It takes three parameters: the cluster for which a repeater is searched, the cluster to which the new sink should be added, and the type of repeater we want to insert (buffer or inverter).

Figure 6.8 shows the situation when an inverter is searched for positive cluster C^+ in parallel mode. The new inverter should be inserted into the negative cluster $C' = C^-$ of the cluster pair.

The routine first computes a Steiner tree for cluster C' containing the new sink position. Then, for a repeater t of the requested type, the required arrival time function and slew limits are computed at the input pin using the load is has to drive

$$\begin{aligned} rat(t) &:= ratinv_t(cap(C^+), rat(C)) \\ Sl(t) &:= slewinv_t(cap(C^+), Sl(C)). \end{aligned}$$

A Steiner tree is extracted using the wiring modes from buffering mode $M(C')$ such that we have an Elmore delay rc_i for each sink $i \in S(C') \cup \{t\}$. Finally, the RAT function at C' is computed

$$rat := \min_{i \in S(C') \cup \{t\}} ratinv(rc_i, rat(i)).$$

Using the resulting capacitance $cap(C')$ of cluster C', we can compute a new required arrival time for C' (see Equation 5.2) and propagate it towards the root (we have to rebalance d_{node} with side branches) where we can calculate a slack σ_t. The weighted slack using the power-time tradeoff ξ is

$$\sigma_t^* := \xi \min\{\sigma_t, 0\} - (1 - \xi) pwr(t).$$

Finally, we assume that the resulting cluster C' is driven by repeater $M(C')_t$ with input slews $M(C')_s$. For each sink $i \in S(C') \cup \{t\}$, e propagate the slews through the Steiner tree resulting in slew pair s_i. We then compute the sum of slew violations:

$$s_t^{\text{vio}} := \min_{i \in S(C') \cup \{t\}} \max\{s_i - Sl(i), 0\}.$$

We also add possible load violations at both repeaters:

$$l_t^{\text{vio}} := \max\{cap' - loadlim(M(C')_t), 0\} + \max\{cap(C^+) - loadlim(t), 0\}.$$

After processing all repeaters, we have for each of them an estimated weighted slack σ_t^*, its power consumption $pwr(t)$, the load violation c_t^{vio}, and the slew violation s_t^{vio}. We choose the repeater that lexicographically minimizes

$$\Big(c_t^{\text{vio}}, s_t^{\text{vio}}, -\sigma_t^*, pwr(t)\Big).$$

After having chosen a repeater, we update the resulting tree. The Steiner tree behind the new repeater and the repeater are merged into T and Pl, the function R_t is updated to reflect the chosen repeater, and R_W is updated to use the wiring modes from $M(C^+)$ for the new edges.

Algorithm 5 Buffering Algorithm

```
1: procedure BUFFERING(T^in)
2:     ASSIGNEFFORT(T^in)
3:     Initialize the topology for buffering T^in.          ▷ See Section 6.2.2
4:     Initialize result (T, Pl, R_t, R_W).
5:     while |V(T)| > 0 do
6:         Choose leaf v ∈ V(T^in)
7:         if Pl(v) ≠ Pl(parent(v)) then
8:             MOVE(v)
9:         else
10:            MERGE(v)                                      ▷ Results in the removal of v
11:        end if
12:    end while
13:    CONNECTROOT
14: end procedure
```

6.2.5 Buffering Algorithm

The overall structure of the BUFFERING algorithm is described in Algorithm 5. Input to the algorithm is a topology. After having assigned new buffering modes to the topology, the data structures are initialized as described above. The topology is then successively modified to create a repeater tree. Leaves are moved towards their parent nodes and merged with them until only the root node is left. In a final step, the last remaining sinks are connected to the root.

In the next section, we describe the MERGE operation because it is also used in the MOVE operation which we explain later.

6.2.6 Merging operation

When some node l and its cluster pair has been moved to the position of another cluster pair, they will be merged. Let (C_l^+,C_l^-) be the cluster pair that arrives along arc e at the cluster pair (C_r^+,C_r^-) which is at the tail of arc e. We compute the merged cluster pair (C^+, C^-) using the MERGE operation.

If $|S(C_l^+)| \cdot |S(C_r^+)| = 0$ and $|S(C_l^-)| \cdot |S(C_r^-)| = 0$, the merging operation is straightforward. We set $C^+ := C_l^+$ if $|S(C_l^+)| = 0$ and $C^+ := C_r^+$ otherwise. The same is done for C^-. If the resulting cluster is not in parallel mode, the merge point is not updated. Otherwise, the merge point is set to the current cluster position for both parities.

In other cases, we give us five options: inserting an inverter driving one of the four clusters C_l^+, C_l^-, C_r^+, C_r^-, or merging clusters of the same parity without inserting any inverter. Note that this does not exclude the possibility of inserting an inverter later, as merge points are (re)defined if there are sinks of both parities after merging. Therefore, we do not evaluate possibilities that can be realised by resolving parallel

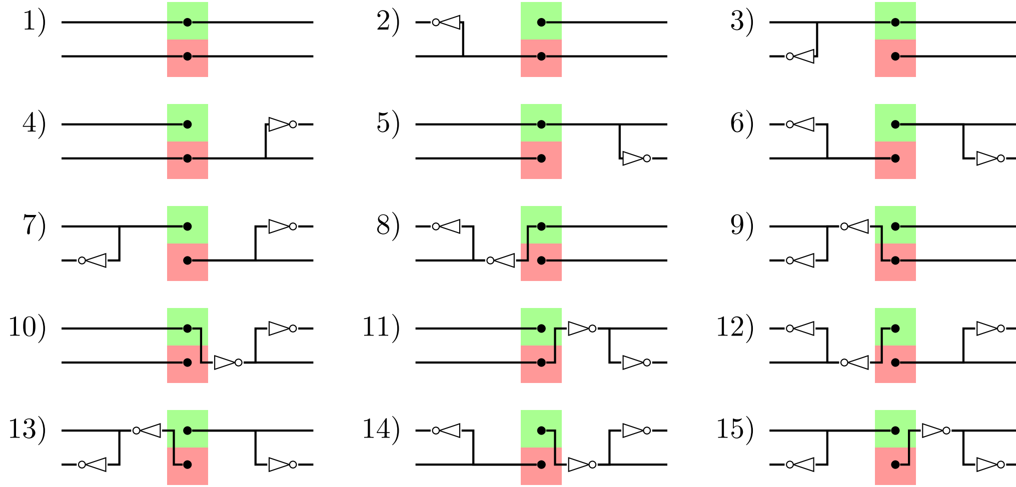


Figure 6.9: The possible merge configurations that are tested during a MERGE operation. In each case, the clusters (C_l^+, C_l^-) arrive from the left and the clusters (C_r^+, C_r^-) arrive from the right side of the resulting cluster pair.

mode later. We evaluate the remaining fifteen possibilities of inserting inverters in front of one or more clusters as shown in Figure 6.9.

For example, in Case 12 we would first put an inverter in front of C_l^+ and put its input pin as sink into C_l^-. Then, we would add another inverter in front of C_l^-. The result would be C^+. We would also put an inverter in front of C_r^+ and add its input as sink into C_r^- resulting in C^-.

Possibilities are only evaluated if we do not have to insert a repeater in front of an empty cluster. Table 6.1 shows which of the fifteen cases are evaluated if a set of clusters is empty. For example, if C_l^+ and C_r^+ are empty (first row in the table), then only cases 1, 3, 5, 9, 11, 13, 15 are evaluated. In all other cases, we would try to insert one or two repeaters either in front of C_l^+ or C_r^+.

C_l^+	C_l^-	C_r^+	C_r^-	Cases
$\emptyset$		$\emptyset$		1, 3, 5, 9, 11, 13, 15
	$\emptyset$		$\emptyset$	1, 2, 4, 8, 10, 12, 14
$\emptyset$				1, 3, 4, 5, 7, 9, 10, 11, 13, 15
	$\emptyset$			1, 2, 4, 5, 6, 8, 10, 11, 12, 14
		$\emptyset$		1, 2, 3, 5, 6, 8, 9, 11, 13, 15
			$\emptyset$	1, 2, 3, 4, 7, 8, 9, 10, 12, 14
				all

Table 6.1: Possible cases which of the four clusters are empty. The table shows which of the fifteen cases of Figure 6.9 are considered.

To evaluate a case, repeaters are tentatively inserted using the InsertRepeater function. Inverters are used if they are available. In case that we would be adding two inverters in front of an input cluster without additional side sinks, we add a single buffer instead. For example, this can happen if in Case 12 as explained above cluster C_l^- is empty at the beginning. Then only a single buffer is added to drive C_l^+ resulting in C^+.

Similarly to the InsertRepeater function, the resulting cluster pair has required arrival time functions at the current position and load capacitances. We use them to compute a slack in our delay model. We evaluate power consumption, slew violations, and load violations for both clusters in the same way as in InsertRepeater and add the values up as well as the values for each InsertRepeater invocation.

As a result, we get for each case a slack σ, a sum of slew violations s^{vio}, a sum of load violations s^{vio}, and a sum of power consumption. We compute the weighted slack using the power-time tradeoff ξ

$$\sigma^* := \xi \min\{\sigma, 0\} - (1 - \xi)pwr.$$

Among all cases, we choose the solution that lexicographically minimizes

$$\left(c^{\mathrm{vio}}, s^{\mathrm{vio}}, -\sigma^*, pwr\right).$$

We realize the chosen case and update our data structures accordingly. At the end, we replace (C_r^+,C_r^-) by (C^+, C^-) and remove node l together with its clusters and incoming edge from the topology.

Handling Different Buffering Modes

During the Merge operation, it can happen that we have to merge clusters with different buffering modes. We treat both clusters as if they have the wiring mode with the better wire delay in such a case. This prevents us from arbitrarily worsening the delay to the sinks that are currently in the cluster with the better wire delay. For example, we would not set a cluster that drives a long distance on higher planes to the default planes.

6.2.7 Moving operation

We now describe how to move cluster pairs within the topology towards the root. A leaf node of the current topology is moved together with its associated cluster pair (C^+, C^-) along the incoming edge. Algorithm 6 shows the Move operation.

First, we have to decide how far both clusters can be moved. Procedure RemainingDistance computes the maximum moving distance for a given cluster C. If the sink set $S(C)$ is empty, we return infinite distance. Otherwise, we assume that the repeater $M(C)_t$ that is optimal for the cluster's buffering mode $M(C)$ drives a wire segment and the cluster's sinks. We maximize the length of the wire segment such that slew targets and limits are not violated.

Algorithm 6 Move Procedure

```
1:  Let p = parent(v)
2:  l+ := RemainingDistance(C+(v))
3:  l- := RemainingDistance(C-(v))
4:  l := min{l+, l-}
5:  if l ≥ ||Pl(v) − Pl(p)|| then
6:      Pl(v) := Pl(p)
7:  else
8:      if S(C+(v)) > 0 and S(C-(v)) > 0 then
9:          ResolveParallel(v)
10:     else
11:         if Bl((p, v)) = 1 then
12:             InsertRepeater(v)
13:             Pl(v) := Pl(p)
14:         else
15:             Choose z with ||Pl(v) − z|| = l minimizing ||z − Pl(p)||
16:             Pl(v) := z
17:             InsertRepeater(v)
18:         end if
19:     end if
20: end if
```

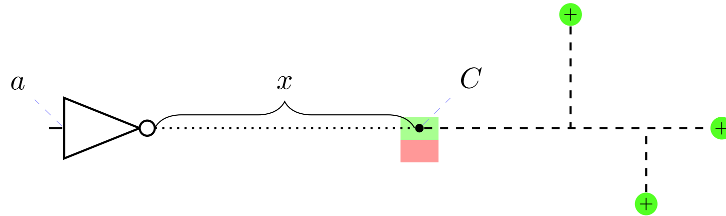


Figure 6.10: The maximum distance x is searched by which cluster C can be moved such that, given slew targets at the input a of an optimal repeater, slew limits and slew targets at C are not violated.

Figure 6.10 shows the situation for positive cluster C. A wire segment with length x is added in front of the cluster. Let $m := M(C)$ be the buffering mode stored at cluster C. Repeater m_t drives the resulting wire and we assume slew pair m_s at input pin a. Let rc be the Elmore delay of the wire segment. We can now compute the slew pair arriving at C:

$$s_{out} = wireslew(rc, slew_{m_t}(xm_{wirecap} + cap(C), m_s)).$$

We then search the maximum x such that $s_{out} \leq Sl(C)$ and $s_{out} \leq St(C)$ using binary search.

After computing the remaining distance for both clusters, we take the minimum for l. If a node and both clusters can be moved to the parent's place, the move is performed and both clusters are updated.

Otherwise, we have to insert a repeater. In the non-parallel mode one of the two clusters, say C^-, is empty and we insert a repeater for C^+ using InsertRepeater. We use a buffer if m_t is a buffer and we use an inverter otherwise.

The position of the new repeater depends on whether the edge is blocked or not. For a blocked edge, the repeater is added at the head of the edge. The resulting cluster is moved to the parent node, and the cluster timing is updated. For an unblocked edge we search a point along the path to the parent node such that the distance to the current position is l and the distance to the parent is minimized. The cluster is then moved and the solution from InsertRepeater is realised.

Resolving Parallel Mode

Both clusters are non-empty in parallel mode, and merge points are defined for both. We resolve such a situation by treating the cluster pair (C^+, C^-) as two cluster pairs, (C^+, C') and (C'', C^-), with empty dummy clusters C', C'' and using the Merge operation. However, we restrict the procedure to choose from Case 2 and Case 5 in Figure 6.9, the two valid operations that directly resolve parallel mode.

Running time

We stop the binary search as soon as the difference between the upper and lower bound gets smaller then the width of the smallest repeater $l_{t_{min}}$. The running time of a single invocation of RemainingDistance is in $O(\log(\frac{l_{max}}{l_{t_{min}}}))$ with l_{max} being the length of the longest edge in the topology.

6.2.8 Arriving at the root

When the last leaf node arrives at the root, we have to connect the remaining cluster pair to the root pin. As we get the load dependent arrival times from the root, we no longer depend on the topology delay model. Instead, we enumerate all possible solutions for connecting the root via a sequence of zero, one, or two repeaters.

In non-parallel mode all possible sequences that result in a correct parity are connected to the cluster with non-empty sink set. If the clusters are in parallel mode, we have to resolve it. In contrast to resolving a parallel mode in the MOVE operation, we search for an inverter for the positive cluster and an inverter for the negative cluster using INSERTREPEATER.

If the clusters are not in parallel mode, only one cluster has a non-empty sink set. We then create a chain of zero, one or two repeaters at the root for all combinations of repeaters that have the correct parity. The chain connects the root to the cluster sinks. Arrival times and slews are propagated from the root to the cluster.

For each combination that we evaluate, we get an estimated slack σ, the sum of slew violations s^{vio}, the sum of load violations c^{vio}, and the total power consumption pwr. Similar to the INSERTREPEATER and MOVE operations, we compute the weighted slack

$$\sigma^* = \xi \min\{\sigma, 0\} - (1 - \xi)pwr$$

and lexicographically minimize

$$\Big(c^{\text{vio}}, s^{\text{vio}}, -\sigma^*, pwr\Big).$$

If the clusters are in parallel mode, then we search for an inverter for both clusters respectively and then try all combinations in the same way as we do in the non-parallel case. The overall best solution is then chosen using the same criteria as above and all data structures are updated accordingly creating the final repeater tree.

6.2.9 Running Time

It is possible that the buffering algorithm that we presented does not terminate if it gets stuck by making no progress in the MOVE operation. This could happen, for example, if the cluster is not allowed to move but inserting any repeater in front of the cluster creates the same cluster or one with worse constraints. To prevent this problem, we move at least the width of the smallest repeater before we insert a new one. This is no limitation because after legalization of the repeaters' placement no two repeaters are allowed to overlap.

We also limit the number of sinks allowed in a cluster by a constant as this is often done in practice by designers. This allows us to also bound the running time necessary to evaluate a possibility in INSERTREPEATER, MERGE or for root connection by a constant as Steiner trees are computed over a bounded set of terminals.

In practice, we run the NETWORK SIMPLEX ALGORITHM to solve the BUFFERING MODE ASSIGNMENT PROBLEM. However, it is not a polynomial-time algorithm. For running time considerations, we use Orlin's algorithm (Orlin, 1993) which solves the MINIMUM COST FLOW PROBLEM in $O(m \log m(m + n \log n))$ with m being the number of edges and n the number of nodes. As we work on a series-parallel graph with roughly twice as many edges as nodes, the running time becomes $O(m^2(\log m)^2)$.

Converting a solution into a basic tree solution takes at most m iterations with linear running time. Finding the AssignEffort solution therefore has a worst case running time of $O(m^2(\log m)^2)$.

Theorem 5. Given an instance of the Repeater Insertion Problem with input topology T^{in}, set M of buffering modes, and repeater library L, the worst case running time of the algorithm is

$$O(Cm^2|L| + (r+m)(\log(\frac{l_{max}}{l_{t_{min}}}) + Cm|L|) + C|L|^2 + (|M|m)^2(\log(|M|m))^2)$$

with $m = E(T^{in})$ and r being the number of inserted repeaters in the output. The computation of each Steiner tree during the algorithm is bounded by C.

Proof. To solve the Minimum Cost Flow Problem an input graph is constructed with at most $2|M|m$ edges. The assignment itself runs in linear time. In total, AssignEffort runs in $O((|M|m)^2(\log(|M|m))^2)$.

A single invocation of InsertRepeater runs in $O(C|L|)$ time in the best case and $O(Cm|L|)$ in the worst case. The worst case arises, if, for calculating a slack for a solution, we have to traverse a significant number of edges of the topology.

A call to Merge makes a bounded number of calls to InsertRepeater and is therefore also in $O(Cm|L|)$. There are $O(m)$ calls to Merge resulting in a total worst case running time of $O(Cm^2|L|)$.

Move executes the binary search in a worst case running time of $O(\log(\frac{l_{max}}{l_{t_{min}}}))$, where l_{max} is the longest topology edge and $l_{t_{min}}$ is the smallest repeater width. The binary search is followed by at most one call to Merge or InsertRepeater. The number of calls to Move is bounded by $r + m$. The total worst case running time is in $O((r+m)(\log(\frac{l_{max}}{l_{t_{min}}}) + Cm|L|))$.

Connecting to the root is in $O(C|L|^2)$ as at most two repeaters are tried and each combination can be evaluated in constant time because there is only one edge left to the root. Putting everything together, we get the claimed running time. □

6.2.10 Repeater Insertion - Summary

As we show in the experimental results, our repeater insertion algorithm produces good results very quickly. For repeater libraries as they appear in practice, it generally finds a solution without capacitance violations if such a solution exists. Slew violations are also avoided most of the time but they appear slightly more often than capacitance violation. This is mainly due to tighter slew limits and the fact that the Steiner trees used for net extraction can change their topology if a sink or the driver is slightly moved.

Strictly Following the Topology

The repeater insertion algorithm presented here does not create repeater trees that follow the input topology strictly. This is one of its main features. Instead, some

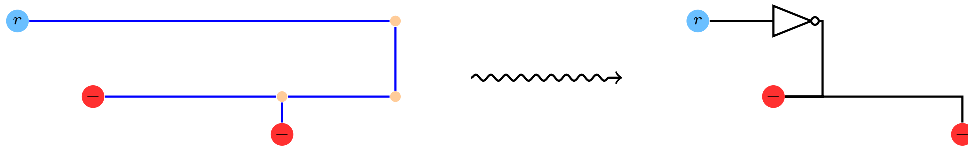


Figure 6.11: A topology (left) and the resulting repeater tree (right). Topology detours are removed by recomputing Steiner trees.

parts of the topology are used twice while moving clusters in parallel. Other parts are discarded due to recomputed Steiner trees. Figure 6.11 shows an example of how a detour is discarded during repeater insertion.

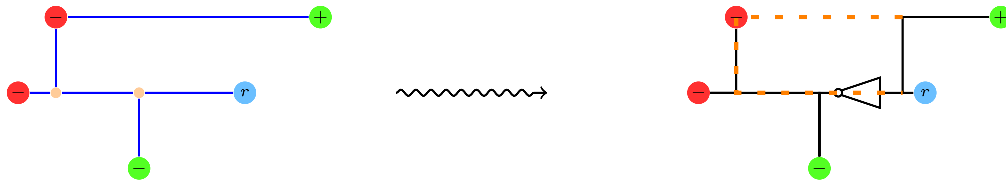


Figure 6.12: A topology (left) and the resulting repeater tree (right). By using a Steiner tree instead of following the topology, a detour for the sink with positive parity is avoided (orange dashed line).

In practice, recomputing Steiner trees gives better results than keeping the input topology because delay calculations are more close to the final (pre-routing) timing. Figure 6.12 shows how detours are avoided that would be induced by following topologies in parallel.

There are cases where it is desirable to strictly follow a topology, for example, if an existing routing should be buffered such that the result can use the same routes. This can be done by changing clusters to also store a subtree of the topology connecting the cluster to its sinks. Delay calculations are then performed on the stored tree.

6.3 Dynamic Programming

In his groundbreaking paper van Ginneken (1990) proposed a dynamic programming algorithm for buffering of repeater tree topologies maximizing the slack at the root. The algorithm worked only for a single buffer type and a single wiring mode. In addition to the input of the REPEATER INSERTION PROBLEM, the algorithm needs buffering positions along the topology. The canonical approach is to add repeater positions equidistantly along the topology. However, it makes sense to choose buffer positions based on library and input characteristics as shown by Alpert et al. (2004b). The running time of which was $O(n^2)$ where n is the number of buffer positions.

Later, Lillis et al. (1996a) extended the approach to handle a library consisting of b buffers or inverters with a running time of $O(n^2b^2)$. They also proposed a way to

handle power consumption. However, the algorithm is not polynomial.

The running time of the algorithm was later improved (Shi and Li, 2005; Li and Shi, 2006; Li et al., 2012) by using clever data structures and better pruning techniques than in the previous papers. For instances with only a single sink, they achieve a running time of $O(b^2 n)$. For nets with m sinks, the running time becomes $O(b^2 n + bmn)$.

There are a lot of extensions to the basic version of Lillis et al. (1996a). There are works considering higher-order delay models (Alpert et al., 1999; Chen and Menezes, 1999), simultaneous buffer insertion and tree construction (Okamoto and Cong, 1996; Hrkić and Lillis, 2002, 2003; Hu et al., 2003), segmenting wires (Alpert and Devgan, 1997), or minimum buffer insertion under slew constraints (Hu et al., 2007).

For an instance of the Repeater Insertion Problem with given repeater positions, the task of finding a repeater tree with maximum slack can be solved efficiently as we have just seen. If one wants to get the cheapest solution that satisfies the slack targets, then the problem becomes NP-complete even if one ignores load limits at the source and repeaters as shown by Shi et al. (2004). An FPTAS for the problem was presented by Hu et al. (2009).

6.3.1 Basic Dynamic Programming Approach

We have implemented a version of the dynamic program as introduced by Lillis et al. and added some improvements. We did not use the running time improvements by Li, Zhou and Shi for several reasons that we give below.

The dynamic program algorithm works with sets of candidates that are characterized by a required arrival time, a downstream capacitance, and a solution subtree. Candidates are propagated bottom up by adding wire segments and adding repeaters at buffering positions. A new candidate for each repeater type is created at each buffering position as long as capacitance limits are not violated. At inner nodes of the topology, the candidates of the left and right branch are merged together by adding all combinations to the candidate list.

An explosion of candidates is prevented by only keeping candidates that are not dominated (i.e. there is no other candidate with better or equal RAT and lower or equal capacitance). At each node, we have two candidate lists for subtrees that need a positive or negative signal to preserve parity.

Candidates are created at sinks. As the dynamic program algorithm does not work with RAT functions, we collapse the RAT to a single value and evaluate the functions for *optslew*.

We compare the quality of the Fast Buffering algorithm with the dynamic program algorithm in Section 8.3.

6.3.2 Buffering Positions

The running time and quality of a van Ginneken style algorithm highly depends on the choice of repeater positions. The result of our repeater insertion algorithm (Algorithm 5) is a complete repeater tree. Repeater nodes and Steiner nodes are used as buffering positions. If there is a node v such that

- v is a sink and has outdegree higher than 0,
- v is a root and has outdegree higher than 1, or
- v is an inner node and has outdegree higher than 2,

then we add additional nodes at the same positions and reconnect children to them until no node satisfies any of above conditions. We create at each sink an additional buffering position.

Optionally, we also split long edges in the resulting tree such that there is a potential buffer position at most after a given length. As the result of Fast Buffering already has quite optimal distances between consecutive repeaters on long lines, it does not make much sense to split the edges further.

The repeater tree from Fast Buffering already navigates around big blockages. Thus, it is not necessary to worry much about them. However, the nodes of the created Steiner trees can lie above blockages. Thus, we add repeater positions at edges that cross a blockage boundary. Finally, all nodes in the interior of blockages are marked as blocked. They are not used as repeater positions.

6.3.3 Extensions to Dynamic Programming

We describe in this section our changes to the basic dynamic program algorithm. Our changes are motivated by timing properties as observed in practice. By using some more accurate calculations the algorithm achieves better slacks than the basic version by Lillis et al. (1996a). This makes the algorithm suitable as postprocessing to Fast Buffering.

Black Box Timing Rules

As we have already discussed in Section 2.5.1, we do not exploit some properties of the Elmore delay and work with black box *wiredelay* and *wireslew* functions. This prevents us from using most of the techniques suggested by Shi, Li, and others to speed up the dynamic program. Each candidate knows the position of the last inserted repeater or the last merge of several topology branches. We call the position the sink of a candidate. For the sink, we always keep the required arrival time and the capacitance up-to-date. Instead of updating the sink when wire segments are added, we just accumulate the RC-delay and compute the required arrival time on demand. This makes the algorithm slower than assuming pure Elmore delays but improves the result quality, because we are nearer to the final timing over wire segments. Because of running time reasons, we only want to have one sink for each candidate. Thus, we update the sink on merge points.

Slew Effects

While computing the required arrival time for a candidate after adding wire or a repeater, we do not know the slew that will arrive at the candidate. This makes the calculations inaccurate and can lead to pruning of otherwise optimal candidates. One solution to mitigate the problem is the use of buckets of discrete slew values as for example proposed by Hu et al. (2007). For minimizing power consumption, this is a viable solution. However, for optimizing slack, this leads to an explosion of candidates. The resulting running times make such a solution impractical if one wants to handle millions of instances.

While we still assume a prototype slew at the inputs of our candidates, we see the resulting slew at the sink of each candidate. We use the difference between the slew that we assumed for the required arrival time calculation at the sink and the arriving slew to estimate the real required arrival time. Given a RAT at a candidate's sink and slew s, we can compute the required arrival time that we use for further calculations:

$$rat = RAT - slewdelay(s).$$

See Section 4.1.4 for the *slewdelay* function.

Wiring Mode Assignment

An important extension to the dynamic program is the handling of different wiring modes. For this, each candidate has a buffering mode assigned of which only the horizontal and vertical wiring modes are used. All candidates with the same buffering mode and parity are kept in a candidate list. Only candidates with the same buffering mode are merged during the merging step.

When we realise the net behind a candidate, all horizontal (vertical) wiring segments of the net will get the same horizontal (vertical) wiring mode. We use the wiring modes stored in the buffering mode of the candidate. Physically, multiple wiring modes per net would be possible, but most industrial routers tolerate only a single wiring mode per net and dimension. After a repeater has been inserted into a candidate, a buffering mode change can occur. Thus, the resulting candidate is copied into the lists of all modes. Such an extension was first proposed by Alpert et al. (2001a). They also showed that wire tapering, that is, the continuous assignment of widths to wire segments, only has marginal advantages compared to assigning a single wiring mode for each dimension to the whole net if there are enough modes available. Following this, we disabled changing of buffering modes at repeater positions if no repeater was inserted. With this, the number of buffering modes changes the overall running time only linearly.

Our layer assignment routine will not use the same set of buffering modes uniformly. Instead, each edge of the input topology gets a set of possible buffering modes. Candidates that arrive at an edge with a not assigned buffering mode are just discarded. To decide which buffering modes are available at an edge the blockage

and congestion maps are used. We remove modes that would cause too high congestion or are blocked. It can happen that the incident edges of a buffering position that lies on top of a blockage have an empty set respectively. As we cannot add a repeater and are not allowed to switch the buffering mode within a net, this would lead to empty candidate sets on all layers. We ignore the congestion map in such cases and allow the lowest buffering mode on all edges incident to the repeater position.

Distinguishing Rise/Fall

A typical instance to the Repeater Insertion Problem has different required arrival times for rise and fall as well as different arrival times at the root. Most implementations of the dynamic program do not distinguish between both values. Instead, they settle to a single value like the average or the worst value of both. To compute the delay over a repeater, also a single value is used. We have seen that several instances can be build better if one considers both values separately.

The candidates in our algorithm have a time pair as required arrival time. When a wire is added or a buffer is inserted, the values can be updated separately by calculating the according delays. This causes twice the effort of only propagating a single number.

The pruning step uses only the worse value of both required arrival times when comparing candidates to prevent an explosion of candidates that we have to handle. Our experiments showed that the benefit of only pruning candidates that are dominated in both required arrival times was small but it was paid with high running times.

Slew Limits

In addition to the required arrival times, we also propagate a slew limit backwards. The slew limit of a candidate is the maximum slew that can arrive at the current node such that the slew limits are not violated in the whole subtree of the candidate. Candidates are pruned as soon as their slew limit is not reachable unless all candidates have to be pruned.

Candidate Selection

As soon as we arrive at the root, we have to choose the best candidate to return a solution. Instead of relying on the required arrival times of the candidates, we recompute the whole timing of each candidate propagating the slew accurately from root to sinks. The sinks are then also evaluated using their required arrival time functions instead of constant values.

Power-aware Dynamic Programming

Lillis et al. (1996a) showed how to extend the dynamic program to find the cheapest solution satisfying the required arrival time constraints. For their FPTAS for the Repeater Insertion Problem with buffering positions, Hu et al. (2009) discretized the power consumption values of repeaters into cost buckets. We have also extended our implementation to work with cost bins. Basically, each candidate is now characterized by three values: *cost*, *cap*, and *rat*. Every time a buffer is added or candidates get merged, the result has to be inserted into the correct bucket. There is a limited number of buckets. All candidates using more power than the highest-valued bucket will be merged together. Unfortunately, the running time of this version is prohibitive high when using a number of buckets that is sufficient for good results. In comparison to the Fast Buffering solution, the timing results are very good and the power consumption is smaller than for the basic dynamic program version (see Table 8.4). However, the running time prevents that this version will be used extensively in production.

We have taken random instances of different characteristics from a 22 nm design and compared the results of the power-aware version of the dynamic program to the basic one. The values are in Table 6.2. All runs of the power-aware version use 40 buckets between 0 and the power consumption of the Fast Buffering solution. We see that that the sum of negative slacks and the worst slack are similar for both runs. The basic version uses a lot of area for bigger instances while the running time of the power-aware version explodes. See Chapter 8 for details of the hardware setup and the instances. A comparison between Fast Buffering and the power-aware dynamic program on the same instance set can be found in Section 8.1.

		Dynamic Program				Dyn. Program + Buckets			
	Sinks	SNS	Slack	Area	Time	SNS	Slack	Area	Time
I01	1	−378	−378	154	11	−377	−377	160	374
I02	1	−91	−91	12	2	−91	−91	14	56
I03	2	−414	−207	138	13	−413	−206	150	446
I04	2	−161	−81	6	4	−161	−81	6	128
I05	3	−161	−61	16	5	−143	−61	11	214
I06	3	−177	−68	42	9	−162	−68	35	370
I07	4	−22	−11	25	8	−22	−11	25	351
I08	4	−137	−36	24	9	−133	−36	8	371
I09	5	−371	−111	48	10	−351	−113	40	487
I10	8	−112	−18	48	16	−117	−20	34	829
I11	10	−444	−69	51	20	−387	−68	22	886
I12	15	−694	−59	187	55	−698	−59	116	2820
I13	24	−71	−15	89	57	−60	−8	38	3660
I14	33	−3317	−188	309	70	−3167	−189	144	3711
I15	47	−2563	−105	310	96	−2922	−109	101	5372
I16	65	−1523	−96	310	181	−3852	−96	48	13125
I17	73	−3762	−75	355	144	−3424	−78	132	10078
I18	120	−10275	−104	510	345	−10701	−107	333	22624
I19	322	0	25	1427	777	0	25	851	46548

Table 6.2: Results of optimizing for slack and power on several instances from a 22 nm design using the basic version and the power-aware version of the dynamic program. All times are given in ps. SNS is the sum of negative slacks for all sinks. Slack is the worst slack of the instance. Area is the space consumed by the internal repeaters measured in placement grid steps. Time gives the running time of the dynamic program excluding other parts in milliseconds.

7 BonnRepeaterTree

The repeater tree algorithm that we described in Chapter 5 and Chapter 6 has been implemented together with a small framework for repeater tree optimization as part of the BonnTools (Korte et al., 2007; Held et al., 2011) suite of physical design optimization tools developed at the Research Institute for Discrete Mathematics, University of Bonn in an industrial cooperation with IBM. The main algorithm together with some utility tools and its APIs is called BonnRepeaterTree.

BonnTools are now part of the IBM electronic design automation tools. They have to work with the requirements of industrial physical design. Thus, a huge amount of development work is spent to cope with real-world designs.

This chapter describes the aspects one has to consider if one wants to implement our algorithms for an existing physical optimization environment. We start with details that are valid for a whole range of repeater tree instances, like the repeater library or blockage map. We then show how a single instance is processed. We take a brief look at the BonnRepeaterTree software architecture and finish with a description of two tools that use our framework.

7.1 Repeater Library

A typical standard gate library has a lot of different repeaters. There are repeaters for special purposes like repeaters for clock trees or repeaters that should just add delays to the signals. Then there are standard repeaters that are used within repeater trees. They are sorted into families of similar properties.

First, there are repeaters from different V_t-levels. The voltage threshold (V_t) is the voltage where the gate starts to switch. Gates with a lower V_t-level are faster because they switch earlier but their power consumption is much higher due to higher leakage power. The optimization flow or the designer set the currently active V_t-level. BonnRepeaterTree prefers repeaters from the active V_t-level.

Second, repeaters are distinguished by their beta ratio (i.e. the difference between the fall and rise delays). Repeaters can be build such that for similar inputs the rise and fall delays are either balanced or asymmetrical. Chains consisting of balanced repeaters are usually slower than chains with unbalanced ones. We perform long-distance calculations for each repeater (see Section 4.1.1) and then choose the family with the fastest repeaters.

There are repeaters of different sizes or *BHCs* (Block Hardware Codes) within each family. Smaller repeaters consume less leakage power. They have lower load limits and are very sensitive to the load. In general, they are also slower than larger

repeaters that can drive higher load capacitances. The largest repeater is often the gate type that can drive the highest capacitance over all gates in the whole library. For a given V_t-level and beta ratio, we often use the whole family of BHCs and only hide the smallest repeaters because they are too sensitive, so that small changes in routing can result in huge timing differences.

As the running time of all repeater tree algorithms depends on the size of the library, one might choose to work with a subset of a family. For example, Alpert et al. (2000) proposed an algorithm to select a proper set of repeaters such that the results do not deteriorate too much. While such an approach would certainly improve the running time of our algorithms, the times we see in practice are fast enough to keep all repeater sizes (except the smallest ones as described above).

The sizes of some example libraries are shown in Table 8.3. Typically, the library has between 15–25 inverters and buffers. There are libraries that consist only of buffers and libraries that consist only of inverters.

In practice, we distinguish between the repeaters that can be removed from the design and the repeaters that can be inserted. While we limit the number of buffers and inverters that are used for construction, we want to be able to remove as much of the existing repeater trees as possible.

7.1.1 Repeater and Wire Analysis

During repeater tree construction, the timing rules are called millions of times to evaluate intermediate solutions. Unfortunately, evaluating timing rules is quite slow. It is prohibitive for running time reasons to call the timing rules each time one wants to calculate delays or slews. To speed up the calculations, we approximate the timing rules of all repeaters. To this end, we sample the domain of both functions equidistantly and use bilinear approximation between the sampling points, which can be evaluated very quickly.

Using 64 sampling points in both dimensions of the rules (input slew and output load), limits the error to about 2 picoseconds for the technologies in our testbed.

Similarly, we approximate the delay rules over net segments. Given an input slew and an Elmore delay, the timing rules compute an output slew and a delay. For some technologies, the timing rule is just a linear scaling. Other technologies, however, use more complicated functions. As we want to work with all inputs, we sample the timing rules for nets with 256 sampling points in both directions and also use bilinear approximation.

7.1.2 RAT and Slew Backwards Propagation

We use bilinear approximations for all flavours of the *slewinv* function. During approximation a binary search is performed to find the highest slew that achieves a given output slew and Elmore delay for nets or output slew and load capacitance for repeaters.

We approximate required arrival times by linear functions. During repeater insertion we are usually interested in the required arrival time for a given target slew s_t. Thus, we approximate the tangent of the RAT function at s_t. To compute the RAT function at the source of a net for a given sink and signal edge with Elmore delay rc and sink RAT function rat, we compute the required arrival time for slews s_t and $s_t + \epsilon$ for an appropriate small ϵ:

$$\begin{aligned} rat_{s_t} &:= rat(wireslew(rc, s_t)) - wiredelay(rc, s_t) \\ rat_{s_t+\epsilon} &:= rat(wireslew(rc, s_t + \epsilon)) - wiredelay(rc, s_t + \epsilon). \end{aligned}$$

The resulting required arrival time at the source is then the linear function going through rat_{s_t} and $rat_{s_t+\epsilon}$. To merge required arrival time from several sinks, we do not compute the lower contour. Instead, we evaluate all required arrival times for slew s_t and only keep one RAT function that attains the minimum.

Required arrival times are propagated backwards over repeaters in an analogue way. Additionally, we have to take care about signal edge inversions through inverters.

7.2 Blockages and Congestion Map

The blockage map contains the regions of the design that are blocked for repeater insertion. The bounding box of the chip area is an enclosing rectangle of all free space in the design. Everything outside is considered blocked. In addition, we also consider as blocked

- regions that belong to the bounding box of the chip area but not to the chip area itself,
- regions that are not free for gate placement within the design,
- regions that are blocked by the user,
- gates that are fixed in their location, and
- large gates that are usually difficult to legalize.

Given all blockages, we first block all free regions that are too small for placing a repeater within. In a second step, an overlap-free set of rectangles covering the blocked areas is computed. The rectangles are then stored in a quadtree that supports fast nearest free location searches.

Using the blockages, an equidistant blockage grid is created. The blockage map and blockage grid are then used to initialize free routing capacities between neighbouring tiles for a congestion map. Finally, all nets are added into the congestion map.

7.3 Processing Repeater Tree Instances

The basic steps of optimizing a single instance are extracting the input data from the netlist, building a new repeater tree, and replacing the original netlist with the new one. We describe the steps in the following sections.

As our algorithms are either heuristics to the problem or simplify it, it is possible that the new solution we computed is worse than the original one. To prevent degradation of the design, we evaluate the original and new solutions and compute some metrics like slack, length and number of electrical violations. If the new solution is better, then it will be inserted into the design. Otherwise, it will be discarded.

7.3.1 Identifying Repeater Tree Instances

The BonnRepeaterTree tool is designed to optimize all repeater tree instances of a design. However, the runtime environment does not give us all instances and the corresponding information directly. Instead, the tool has to find instances in the netlist.

After identifying instances, we have to extract all data relevant to the instance. We can assume that basic data like the repeater library, the wiring modes, the blockage map, and potentially a congestion map are already given. It remains to extract

- all nets and pins belonging to the instance and their placement,
- the arrival time functions of the rules and the the timing rules necessary to compute them,
- required arrival time functions at the sinks, and
- if wiring already exists and it is requested, the existing wiring topology.

The following sections describe the steps mentioned in the list.

Identifying Roots and Inner Circuits

Generally, all nets of a design are part of repeater trees. Nets incident to repeaters belong to the same repeater tree. However, some nets are not part of any repeater tree because they are protected from optimization by the designer. There are several sources of such hides:

- Nets are hidden if they are already optimized and if the designer does not want a tool to mess up with the current result.
- Nets are hidden by other tools because they depend on the current solution.

- Nets that carry clock signals have to be buffered but there are special requirements, for example, the signal should arrive at the same time at all sinks, such that special clock tree tools buffer them.
- Nets with analog signals should not be buffered.
- Nets can have multiple inputs. In such cases, special care has to be taken that no short-circuit is created.

It can easily be queried from the runtime environment whether a net is hidden. BonnRepeaterTree works on a set of nets from a design. To identify the instances corresponding to a set of nets, the following steps are performed:

1. Nets that are hidden are filtered out. The remaining nets, either a root net or a net at a deeper level, are part of a repeater tree.
2. For each net's source we identify its root pin and collect it. The source pin of a net is a repeater tree root if it has no gate or its gate is not a repeater or if it is not possible to remove the repeater and its incident nets without violating a hide or similar restriction. If a source pin is not a root, then it is an output pin of a repeater that is part of the tree. We recursively continue the search with the net connected to its input pin.
3. For each collected root, we start a forward search to include all repeaters and sinks that belong to the instance. A sink pin is reached if it does not belong to a gate or if its gate is not a repeater that can be removed.

After running the routine we have a set of instances. Each instance consists of a root pin, a set of inner repeaters, a set of sink pins, and a set of connections between them that are stored in a tree data structure.

Arrival Time Functions

After we have identified the root pin of an instance, we have to extract some characteristics of the root. We are interested in

- the arrival time functions for all signals at the root and
- the output load limit that the root can drive.

It is possible that different signals are going through a repeater tree instance. The timing engine creates so-called *phases* for different signal sources. Phases are propagated separately such that there are several independent arrival times and slews as well as required arrival times at the timing nodes. Slew limits are also phase-specific.

The output load limit is used in ConnectRoot to check for electrical feasibility of the solutions. For each phase of the instance, there is a pair of arrival time

functions, one for each transition respectively. Given the load at the root pin, we can compute the arrival time we are interested in.

There are several types of roots. For each type, we extract the arrival time functions in a different manner. Within the BonnRepeater framework, we might see root pins that are

- output pins of circuits,
- primary inputs of the netlist,
- pins on hierarchy boundaries, or
- output pins of circuits that are fed by transparent segments.

We describe each type of roots in the following sections.

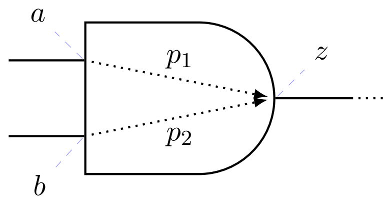


Figure 7.1: A repeater tree root z at the output pin of a standard AND-gate with two input pins. The arrival times and slews at the root are computed using arrival times and slews at the input pins a, b and the propagation segments p_1, p_2.

Outputs of Circuits The most prominent type of root is an output pin of a standard circuit or a macro. There are propagation segments heading towards the output pin within the circuit coming from its inputs or internal timing points. To estimate the timing at the root, we have to extract the timing rules of the propagation segments together with the arrival times and slews at their tails. The load limit of the root is given by the timing rule of the circuit's pin.

Figure 7.2: A repeater tree root at a primary input of the design. Arrival times and slews are constant.

Primary Inputs Primary input pins are starting points for signals coming from outside of the design. We distinguish two different types designs and on each type primary inputs are treated differently. Some designs are macros that are later included into larger designs forming a hierarchy of designs. The primary inputs and outputs of macro designs communicate with gates at higher hierarchy levels. Incident nets can be optimized by the repeater tree routine.

Top-level designs are the second type. They communicate with components outside of the chip image. Here, it is often not possible to optimize nets incident to primary inputs because they need special treatment due to electrical constraints. In

both cases, information about timing coming from the outside is given in form of arrival time and slew assertions. Normally, the values are constant, independent from the load at the pins. In addition, there is a load limit asserted to the pin.

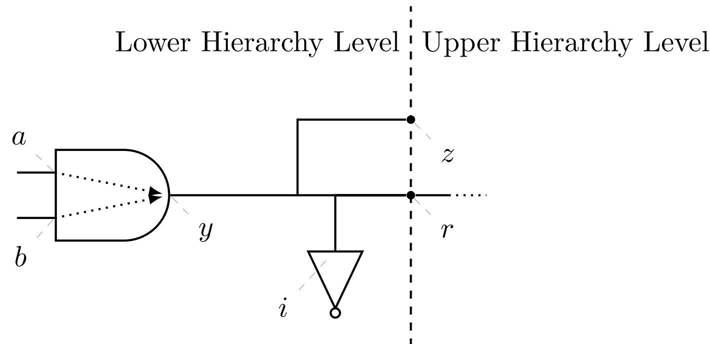


Figure 7.3: A repeater tree root (r) at a hierarchy boundary. It is in the middle of a net crossing the boundary. To compute arrival times and slews one has to fetch them at the driving gate's inputs and propagate over the gate and net. Changing the load at pin r has effects on the timing at side pins i and z.

Pins on Hierarchy Boundaries It is possible to load hierarchical designs such that the contents of all hierarchy levels are visible to optimization tools. Nets crossing hierarchy boundaries could be optimized in a single step. However, tools are often only allowed to work at a single level at a time. Thus, instances stop at hierarchy boundaries and it is possible to get roots at virtual pins that mark the hierarchy boundaries.

For roots at hierarchy boundaries, it is possible to take information about the previous level into account. If one changes the load capacitance at such a root this has an impact on the driver in the preceding net and the net itself. To calculate the timing at the root one has to extract the driver's timing as described in the previous case and also the timing behaviour of the net between the driver and the root.

In our tool, we first extract the driver's timing rules and then calculate a Steiner tree for the preceding net. When we want to get the timing at the root, we first calculate the final load at the driver and then compute the timing at the drivers output pin. In a second step, we recompute the Elmore delay for the root and propagate the timing over the net.

The load capacitance limit of the boundary pin is the highest capacitance we can connect to the pin without violating the capacitance limit of the preceding driver.

Changing a repeater tree at a boundary hierarchy also changes the timing on all pins that are siblings to the root in the preceding net. Figure 7.3 shows the situation with sibling sinks i and z. The slack can turn negative or electrical violations can appear. However, we ignore the timing at sibling pins because we did not see any problems in the past. If it turns out to be a problem in the future, the timing of sibling pins can be considered by limiting the capacitance load limit of the root such

that the sibling's timing remains feasible even in the worst case.

A different problem with siblings in the preceding net appears if they are hierarchy boundary pins (pin z in Figure 7.3). Working on one of the pins can significantly change the timing behaviour of the others. This can be a problem when we optimize both instances in parallel where we first extract the timing constraints and then optimize. Here, we do not see the changes we do during optimization of the first root when we start to work on a subsequent root. A better solution would recognize that all siblings belong together and optimize them in a single repeater tree instance. However, the problem appears rarely in practice and did not impose a problem. Thus, we ignore the situation so far.

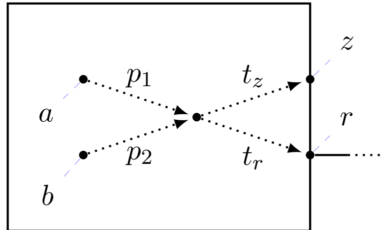


Figure 7.4: A root at a macro boundary with transparent segments. The load capacitance at r is visible over transparent segment t_r to propagation segments p_1 and p_2. Arrival times and slews are calculated by propagating them first over p_1 and p_2 and then over t_r. Timing at sibling pin z depends on the load capacitance at r and vice-versa.

Transparent Segments There is a fourth type of roots similar to roots at hierarchy boundaries that one can find at the output pins of macros. Macros that were processed as a separate hierarchy level are later finalized and treated as a single block. The timing of the whole block is catched in macro-specific timing rules. The behaviour at hierarchy boundaries is modelled using *transparent segments*. Transparent segments mimic preceding nets in this situation.

Normally, propagation segments within gates shield the load capacitance at their head such that it is not visible at preceding segments. In contrast, the capacitance at the head of transparent segments is visible at the tail and therefore influences previous propagation segments. This is analogue to a net where the load at the sink is visible at the source.

Transparent segments are preceded by propagation segments that correspond to the driving circuit in the hierarchy boundary case. To fully catch the timing behaviour, one has to extract the timing at the tails of the segments heading to a transparent segment and the timing rules of all involved segments.

The timing rules of the macro give us the load capacitance limit of the root.

Transparent segments have similar problems as pins on hierarchy boundaries. It is possible that there are sibling segments heading to other outgoing pins of the macro. Here, both instances should be considered at once, too. However, we have

not seen such a problem in practice yet.

RAT Functions at Sinks

For each phase that arrives at a sink, we approximate the rise and fall RAT functions by linear functions. This is a rough estimate compared to the effort we spend on accurately estimating arrival times at the root. In practice, adding more effort into getting better RAT functions improves the results only by a small margin. The influence of the root is much higher.

We know the current slews and required arrival times at the sinks from the timing engine. We then set the RAT functions for sinks at primary outputs, where required arrival times are asserted, to constant functions. For pins that are gate inputs, we set the RAT functions to the tangent around the current slews by evaluating the outgoing propagation segments.

Phase shifts

Ideally, different phases are handled separately during repeater tree construction because different sinks can be critical for different phases. Due to different slews as well as slew limits, propagating only a single phase can be too pessimistic. Because of running time reasons, we only handle a single phase during repeater tree construction. The different phases have to be merged into a single one. This is done by normalizing arrival times and RATs.

First, we assume that the root has a load capacitance of 0 and compute arrival times. Using our delay model, we determine the criticality for each phase and signal edge separately. The most critical phase and signal edge is used as reference. Then, all other signals are shifted by a constant such that their arrival times match the reference arrival time. The shift is also performed for the RAT functions at all sinks.

The arrival times and RATs used during repeater tree construction are the worst ones after shifting. We also use the tightest limits over all phases. When it comes to evaluate a solution, we propagate each phase independently again to avoid pessimism.

Unplaced Pins

Especially in early design stages, it is possible that sink pins or the root have no proper placement. A reason might be that the corresponding gate is not yet placed or the gate's design is not yet finished such that the pin's position within the gate is unclear. If there is at least one placed pin in an instance, then the unplaced pins are positioned at the center of gravity of the placed ones. If all pins are unplaced, we treat them as if they all lie at the origin of the coordinate system.

Identifying Existing Wiring

As mentioned in Section 5.9, existing wiring can be used as a topology for the buffering algorithm. Because wires are stored as a list of segments between two coordinates in our optimization environment, we first have to reconstruct a graph out of the segments. It can happen that the existing wiring is not connected or it does not cover all pins. In such a case, we just discard the whole graph and use our topology algorithm.

Slew Limits

In addition to the slew limits at input pins imposed by the timing rules, there might be additional design specific slew limits. First, there is a global slew limit that should not be violated at any pin. Second, there are phase-specific slew limits that are only valid for arrival times belonging to the according phase.

Given an instance, we know which slew limits apply. We then modify the instance such that the slew limits of the sinks and all repeaters in the library are not higher than the instance-specific limits. In case of phase-specific slew limits, this means that we choose the minimum over all slew limits for construction. This can be too pessimistic. For example, we sometimes compare slews from uncritical phases having higher slew limits with smaller slew limits from critical phases.

Lowering the slew limits at insertable repeaters can make some buffering modes invalid if their slew target is above the limit. We just remove invalid buffering modes before an instance is processed.

Capacitance Limits

Similar to phase-specific slew limits, we also see capacitance limits at output pins that depend on the phases propagating to gates. They are imposed to lessen the effects of electromigration. Electromigration is the movement of material in a conductor caused by current. It decreases the reliability of integrated circuits over time. One strategy used to cope with this is reducing the capacitance that gates are allowed to drive. As the strength of the effect depends on several factors including the frequency of the signals, the countermeasures also depend on the signals.

For a given instance, the load limits of all repeaters that can possibly be inserted have to be lowered according to the signals going through the instance. This can make some buffering modes invalid that depend on higher loads. They are just removed for the instance.

7.3.2 Constructing Repeater Trees

For repeater tree construction, we always first construct a topology and then add repeaters using our algorithm. The result can optionally be post-processed by the dynamic programming repeater insertion that treats the result of Fast Buffering as

an input topology. The initial topology can be constructed using existing wiring or by our topology algorithm.

Parameter ξ

The algorithms are controlled by the preprocessing that we presented in Chapter 4 and the ξ parameter. For our implementation, we have split the parameter into three different ones: ξ_m for buffering mode creation, ξ_t for topology generation, and ξ_r for the repeater insertion step. The parameters can be controlled by the user independently. In Section 8.8, we give hints which values work best for the parameters.

Currently, we only work with two buffering modes for each wire mode. The first buffering mode is extracted using $\xi = 0.0$ and for the second one we use ξ_r. The faster buffering mode is then used for topology generation and AssignEffort chooses between both. As described earlier, lowering slew or capacitance limits can render buffering modes invalid. This can only happen for the slower buffering mode as it has higher limits. In such a case, we increase ξ until all limits are met. Thus, we always have a choice between a slower and a faster buffering mode.

The parameter ξ_t is used within topology generation when we have to decide at which edge we want to connect a sink.

Finally, ξ_r is used during repeater insertion when we have to decide which solution to choose in InsertRepeater, Merge, and ConnectRoot.

7.3.3 Replacing Repeater Tree Instances

After a repeater tree has been constructed, we have to evaluate it. Our algorithm is only a heuristic that cannot guarantee a good solution. We therefore compare our solution to the existing repeater tree that was identified during instance collection. For this, we have the choice to

- evaluate the result using our approximations of the timing rules or
- to use the timing engine for evaluation.

Evaluation using the approximations has the disadvantage that it is slightly inaccurate and can miss effects influencing timing. However, it is much faster than the timing engine and it can run on different instances at the same time (see Section 7.4.3). If the new solution is not good enough to be kept, it is possible to discard it without modifying the netlist. If we want to evaluate using the timing engine, then we have to insert the result into the netlist first. Currently, the quick evaluation mode is used most of the time. Only if we want to be 100 % accurate, the timing engine is used.

The criteria used to evaluate a solution are ordered from the most important ones to the least important: electrical violations, slack, power consumption, and length.

7.4 Implementation Overview

BonnRepeaterTree is a module in the BonnTools suite. It has been implemented using the C++ programming language. The module consists of

- a repeater tree API,
- a framework that can be used to implement repeater tree construction algorithms,
- implementations of our repeater tree construction algorithms, and
- a layer translating between the framework and the IBM physical design tools.

All our algorithms have been implemented using the framework. To migrate them to a different physical design tool suite one only has to reimplement the translation layer. It will not be necessary to touch the algorithms.

7.4.1 Repeater Tree Construction Framework

Our framework provides an algorithm with all information that belongs to an instance as described in Chapter 3. In addition, the existing implementation of an instance is available. For example, this is used to do post optimization of instances or to compare new solutions with older ones.

To build a new tree, an algorithm only has to create a tree data structure that consists of nodes for roots, sinks, repeaters, and the connections between them. Evaluation and implementation into the design is done by the framework.

7.4.2 Repeater Tree API

The BonnRepeaterTrees are not only used as a standalone utility but also as a subroutine for other tools, for example, BonnLogic (Werber, 2007; Werber et al., 2007), a tool to restructure logic on the critical path.

There are also programs that need information about repeater tree instances. For example, determining whether a pin is part of a repeater tree at all is a functionality that is not provided by the timing engine.

We provide a small API to work on repeater trees using our algorithms consisting of

- utility functions to determine whether a pin is in a repeater tree, whether a pin is a root, and for a pin in a repeater tree the corresponding root,
- a function returning a whole repeater tree instance, and
- a function to construct a repeater tree using one of the algorithms.

Instances expose the original repeater tree. The user has direct access to root and sinks and can traverse the tree to get inner repeaters and nets.

For example, the RerouteChains tool uses the interface to fetch all repeater trees and traverses the original tree to identify chains. Most tools, however, have a set of pins and want to optimize the pins' repeater trees. Such tools just iterate over all pins, fetch an instance, and construct it.

7.4.3 Parallelization

About ten years ago the increase of CPU speed with each new generation slowed down significantly. Instead, multi-core CPUs appeared with the number of cores increasing with each generation. To fully utilize the power of modern CPUs, it is necessary to distribute work on the cores.

It rarely happens that repeater tree optimization is started on a single instance. Typically, hundreds or thousands of instances should be calculated at the same time. Because of this and the small running time of optimizing a single instance, it makes little sense to parallelize parts of our algorithm. Instead, we choose the simpler approach of parallelizing the computation of different instances.

The optimization environment does not allow to modify or even query the netlist or timing engine at the same time from different threads because this would lead to race-conditions. Therefore, we choose to protect all calls to the environment by a single mutex. To reduce congestion on the mutex the framework first fetches all information necessary to compute a repeater tree while the mutex is held. Then, during the whole computation, the mutex is never acquired. Only after the decision to insert a repeater tree has been made, the mutex is locked again to modify the netlist.

Testing Multithreading Running Times

We have tested the parallel execution on our testbed of chip designs using an Intel Xeon machine with 4 processors having 8 cores each. Table 7.1 shows the running times with different numbers of parallel threads. We achieve a speedup around 4 using 8 cores which is our default when we optimize all instances of a design. The speedup is limited by the serial parts, instance identification and instance insertion.

7.5 BonnRepeaterTree in Global Timing Optimization

Our algorithm is able to process millions of instances in reasonable time. It is therefore suitable to be used in global timing optimization where all instances are processed. BonnRepeaterTree offers two parameter sets by default that have proven useful for global optimization (see Held (2008)).

Design	1	2	4	8	Factor	12	16	24	32
Baldassare	20 s	11 s	7 s	6 s	3.34×	6 s	6 s	6 s	6 s
Beate	35 s	18 s	11 s	8 s	4.09×	8 s	8 s	8 s	8 s
Gerben	40 s	21 s	12 s	9 s	4.46×	9 s	8 s	8 s	10 s
Wolfram	45 s	23 s	14 s	11 s	4.08×	10 s	10 s	10 s	11 s
Luciano	102 s	55 s	35 s	29 s	3.48×	28 s	27 s	28 s	28 s
Benedikt	250 s	130 s	78 s	63 s	3.95×	59 s	58 s	59 s	59 s
Renaud	259 s	140 s	89 s	74 s	3.52×	69 s	69 s	69 s	70 s
Julius	309 s	158 s	92 s	72 s	4.38×	69 s	66 s	69 s	68 s
Franziska	358 s	188 s	113 s	89 s	4.02×	84 s	82 s	84 s	85 s
Meinolf	449 s	235 s	138 s	108 s	4.16×	103 s	101 s	101 s	102 s
Iris	501 s	269 s	170 s	142 s	3.53×	135 s	135 s	136 s	138 s
Gautier	1420 s	765 s	473 s	383 s	3.70×	372 s	363 s	378 s	390 s

Table 7.1: Running times with different numbers of used threads. As the running time gains from going further than 8 threads are rather small we use 8 threads by default. The speedup factor is then around 4.

Power Trees

The first parameter set aims to build all instances with minimal power consumption while electrical violations are avoided. We achieve this by relaxing all required arrival time constraints to infinity. We assume that the root gate is replaced by the strongest version from its BHC family. We mainly build short topologies, but to prevent too long daisy-chains, the parameter ξ_t is slightly above 0.0.

Slack Trees

The second parameter sets tries to improve the slack above the slack target for each instance. The ξ parameters are set around 0.8 to prevent a too high resource usage. Uncritical instance parts are built with parameters similar to the power tree ones due to the AssignEffort subroutine.

7.6 BonnRepeaterTree Utilities

Besides optimization, there are other tasks that are related to repeater trees. The most common tasks are implemented as separate tools that can be called directly by the user. The two existing tools are a rip out routine and repeater chain optimization.

7.6.1 Removing Existing Repeaters

It is desirable to remove all existing repeater trees to get into a clean initial state for optimization. Our RipOut routine removes all inverters and buffers from a design or specified instances such that at most one parity preserving inverter is left.

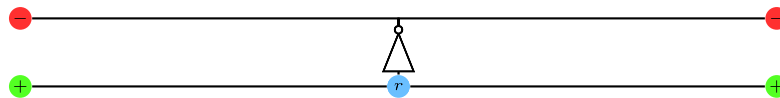


Figure 7.5: The netlength of this instance can nearly be halved by adding a second inverter and placing one of both at each negative sink.

The inverter is placed within the bounding box of all negative sinks such that it is nearest to the root.

Figure 7.5 shows how removing as many repeaters as possible increases the netlength of the design. This has to be considered if one uses a physical design flow where all repeater trees are built along topologies coming from a global router. Typical global routers are not capable of inserting inverters on their own. The topologies that they can generate are limited by the input, and the example above shows that removing as many repeaters as possible is no satisfactory solution because it can lead to unnecessary high netlength.

We offer a mode in our RipOut routine that tries to circumvent the problem. In this mode the routine will not insert the inverting repeater but modify logic by connecting all sinks directly to the root. The original parity is stored for each sink pin. As soon as the instance is touched again by one of our tools, the sink parities are restored. The routine is dangerous because the whole design might get broken if another tool changes the logic between rip out and restore. For global routers, however, each repeater tree instance is presented as a single net.

7.6.2 Postprocessing Repeater Chains

Larger distances are often covered by chains of repeaters. A chain is a set of subsequent nets with fanout one. For larger distances, it is more probable that it is better to use higher layers with wider wires for the nets than the default layers. It is often not possible to reach the slack targets without using wider wires and a lot of short nets on lower layers increase placement congestion due to a lot of additional repeaters. On the other hand, it makes no sense to assign high fanout nets to higher layers. High fanout nets often connect a lot of sinks locally due to the pin capacitance. The benefits of wider wires are not high in such a situation. Thus, we restrict ourselves to repeater chains.

We offer two routines for improving the placement and layer assignment of repeater chains.

Postprocessing

The first routine tries to improve the layer assignment of repeater tree chains by ripping them out and rebuilding them on higher layers. The routine iterates over all repeater chains and then builds a new solution using each configured layer assignment with the Fast Buffering routine. The best solution according to our

Figure 7.6: All repeater chains longer than 4 mm on a large design. Distributing the chains more evenly can give us less congestion while preserving the timing.

criteria, electrical violations, slack, power consumption, and length is kept. The routine either uses a shortest path computation in the blockage grid or uses the path search of the congestion map if available. In the later case, assignments are only performed if they do not violate congestion targets on the probed layers.

Congestion-aware Rerouting

The second routine, RerouteChains, does not try to improve the layer assignment of repeater tree chains. Instead, it tries to improve congestion by moving the repeaters into less congested areas. Figure 7.6 shows all repeater chains longer than 4 mm on a design that has a lot of congestion. By distributing the chains evenly and by avoiding congested areas, it is possible to reduce overall congestion. We have a simple ripup-and-reroute heuristic that reroutes the chains.

This routine only works in presence of a congestion map. In several iterations all chains are collected that use an edge in the map above a certain congestion level. Then, for each chain, the following steps are performed:

1. Collect the nets of the chain and the costs of their routing in the congestion map.
2. Search for a new path from the start point to the end point of the chain in the congestion map.
3. Distribute the internal repeaters of the chain along the new path such that relative distances are preserved. The repeaters are then placed legally at the free position next to their target position.
4. New routes for all incident nets are computed.

5. If the costs of the new nets are smaller than the old costs, then the new solution is kept. Otherwise, it is reverted.

The search for the new path does not take existing layer assignments into account because it is not supported by the congestion map. The assignments are only used when new routes for the nets are recomputed. Similarly, the path search ignores timing. We restrict the congestion map to the bounding box between the start and end of the path plus some tiles for detour. Thus, we expect that the routes do not get too long and timing is not degraded too much. Instead, the number of nets with long detours generated by the global router should become smaller.

The routine also does not consider placement congestion directly apart from searching for free positions. In practice, placing gates densely such that no gaps occur results in unroutable designs. Thus, if we have a design that is considered routable by the global router, we expect that placing the circuits legally in the corresponding global routing tiles should be feasible without much placement distortion. This cannot be guaranteed but the experiments did not show problems with placement legalization. If we skip placing the repeaters legally after movement, then the results are similar to runs where the circuits are legalized.

A potentially better solution to the problem has been proposed by Janzen (2012). He integrates the placement of repeater chains into the global router. Whole chains are considered as a single net by the global router, and, during path search, timing as well as placement area are considered. Compared to the method presented here, the running time of the global router increased significantly due to the additional work during path search. This approach is not suitable for application in practice so far.

Experimental Results

We test the RerouteChains routine on our testbed of chip designs (see Section 8). The test cases are output of a standard BonnTools optimization flow (see Section 7.5). We compare the effects of our routine on routability and timing.

Table 7.2 and Table 7.3 summarize the experiments. BonnRouteGlobal[1] is used to measure routability. We have one run of the global router before and one run after RerouteChains. The global router and the congestion map have nearly the same number of global routing tiles in both dimensions.

Table 7.2 first shows the number of repeater chains considered on the test instances. Our tool runs in five iterations over all chains and tries to reroute a chain if it uses an edge in the congestion map with more than 85 % utilization. If a cheaper solution is found, then it is kept. The second column shows the number of improvements found over all five iterations. The following columns show the running times of both global router runs (1st GR, 2nd GR), congestion map creation (CM), and RerouteChains (RC). The congestion map proves to be very fast. The RerouteChains running

[1] see (Gester et al., 2011; Müller et al., 2011; Müller, 2009)

			Running Times			
Design	Chains	Reroutes	1st GR	CM	RC	2nd GR
Baldassare	7613	381	9 s	2 s	3 s	10 s
Beate	5043	251	14 s	2 s	5 s	14 s
Wolfram	3842	1473	18 s	2 s	6 s	19 s
Gerben	3493	10	14 s	1 s	5 s	15 s
Luciano	27922	4200	46 s	5 s	20 s	50 s
Benedikt	37893	3209	106 s	14 s	38 s	111 s
Renaud	14908	12379	295 s	13 s	33 s	289 s
Julius	41677	7315	178 s	24 s	185 s	150 s
Franziska	56697	3108	151 s	22 s	65 s	160 s
Meinolf	128023	68221	165 s	23 s	103 s	170 s
Iris	76153	141865	750 s	30 s	193 s	797 s
Gautier	165491	214135	6454 s	205 s	1344 s	5247 s

Table 7.2: Testbed for the RerouteChains tool. The running times are given for the global routing (GR), congestion map creation (CM), and RerouteChains (RC).

times also contain the legalization of moved repeaters. Overall, the running times are acceptable.

Table 7.3 first shows the total overflow over all global routing edges as reported by BonnRouteGlobal. Then, the number of nets is reported with length longer than two times the length of a Steiner minimum tree and the highest relative detour. Finally, timing quality is measured with the sum of negative slacks (SNS). Most instances are uncritical and the tool has little effect. On Benedikt and Franziska the routability decreases slightly due to differences in the free edge capacities seen between the global router and the congestion map. On designs Gautier and Renaud the congestion was reduced significantly resulting in less detours. Due to the reduction of detours, also the sum of negative slacks was improved on these instances. In summary, there are instances where running the tool improves routability, while on uncritical instances no harm is done.

Figure 7.7 and Figure 7.8 show congestion maps as reported by BonnRouteGlobal for designs Julius and Gautier. The colors show the maximum relative edge utilization over all layers. Especially on Gautier, we see the huge reduction of overflow by a factor more than 100×.

Design	Step	Overflow	Nets > 100 %	Max detour	SNS
Baldassare	before	0	0	8 %	−140049 ps
	after	0	0	8 %	−140039 ps
Beate	before	0	0	78 %	−151997 ps
	after	0	0	78 %	−151990 ps
Wolfram	before	0	0	25 %	−75187 ps
	after	0	0	25 %	−75210 ps
Gerben	before	0	0	30 %	−109589 ps
	after	0	0	30 %	−109626 ps
Luciano	before	0	52	524 %	−464277 ps
	after	0	24	342 %	−481492 ps
Benedikt	before	1446	0	72 %	−207397 ps
	after	2906	0	103 %	−210871 ps
Renaud	before	4140245	913	747 %	−6557941 ps
	after	3444979	825	513 %	−5822497 ps
Julius	before	0	0	99 %	−18787024 ps
	after	0	0	89 %	−18766033 ps
Franziska	before	1307	0	78 %	−1694730 ps
	after	10982	0	78 %	−1694434 ps
Meinolf	before	0	0	98 %	−4622849 ps
	after	0	0	89 %	−4715832 ps
Iris	before	2843214	4057	1662 %	−19704061 ps
	after	1891486	3032	1662 %	−15602736 ps
Gautier	before	12682825	4609	306 %	−9647137 ps
	after	123511	111	131 %	−7058513 ps

Table 7.3: Difference in routability and timing before and after REROUTECHAINS.

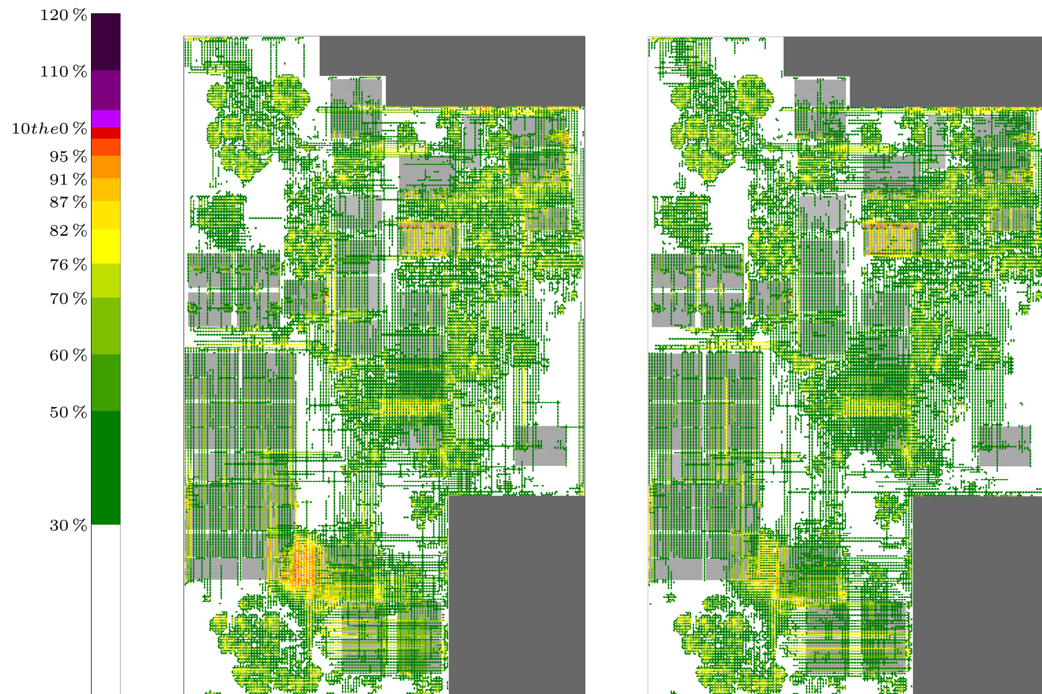


Figure 7.7: Congestion on design Julius before (left) and after (right) REROUTECHAINS.

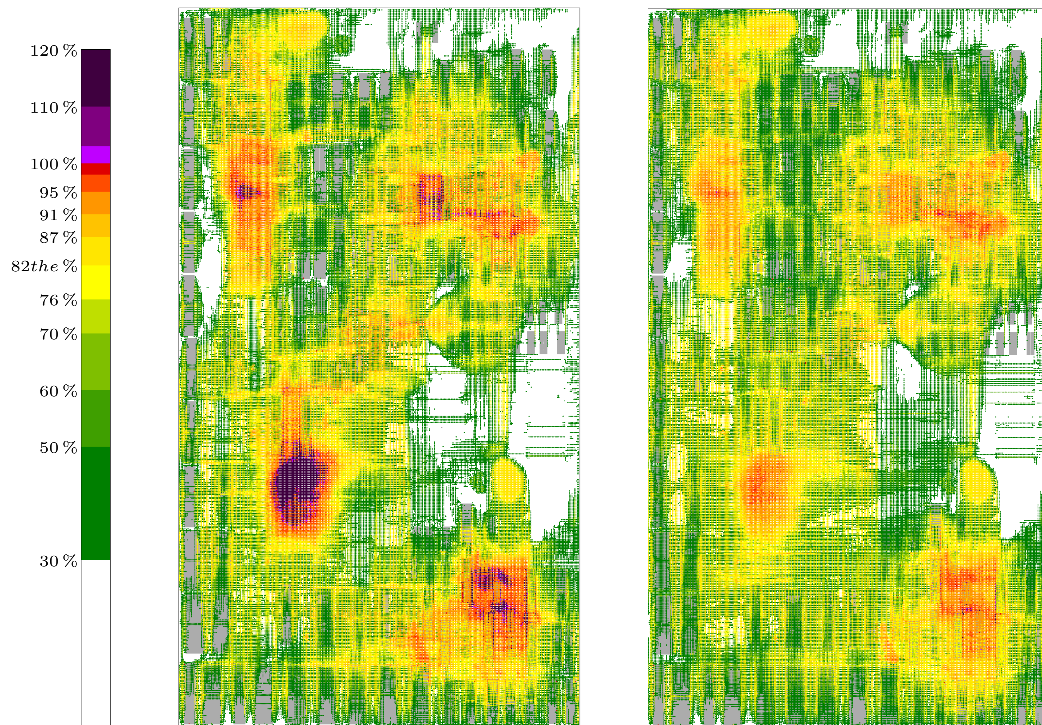


Figure 7.8: Congestion on design Gautier before (left) and after (right) REROUTECHAINS.

8 Experimental Results

As part of the IBM optimization tool suite, the BonnRepeaterTree tool is in daily use to optimize the physical design of current chip designs. It proved to be useful for current ASIC (application-specific integrated circuit) designs as well as for bleeding-edge processor units. Designers choose it as part of the BonnTools optimization tools to achieve timing-closure.

In this chapter, we want to show performance metrics of our tool. We compare it to another tool used by IBM for repeater tree optimization. We also compare against lower bounds for slack, netlength and repeater counts. We then show how parallel walk and effort assignment affect the results and give hints on proper selection of parameters ξ, η, and d_{node}.

Design	Technology	Instances	Max. Sinks
Baldassare	22 nm	20552	152
Beate	22 nm	34382	166
Wolfram	32 nm	44413	274
Gerben	32 nm	44677	194
Luciano	22 nm	101625	263
Benedikt	22 nm	241465	12061
Renaud	45 nm	268775	389
Julius	22 nm	284934	1589
Franziska	22 nm	328600	740
Meinolf	22 nm	364969	1539
Iris	22 nm	393457	3264
Gautier	45 nm	1275731	6859

Table 8.1: Designs used for experimental results. For each design, all repeater tree instances were build.

We have chosen twelve current chip designs from our industrial partner IBM for our experiments. For each design, we build all repeater tree instances regardless of difficulty. In total, we have more than 3.3 million instances of varying sizes with up to 12061 sinks. Table 8.1 shows the designs we used, their codename, technology, the number of instances, and the number of sinks in the biggest instance.

The distribution of instance sizes is very uneven as shown by Table 8.2, which shows it subdivided into the different technologies. The designs are dominated by single sink instances. Instances with up to four sinks already make up for more than 90 % of all instances. It is important to produce good repeater trees for

Sinks	45 nm	32 nm	22 nm	Total	
1	958307	58920	1307808	2325035	68.39 %
2	191709	11422	179836	382967	11.26 %
3	144054	9077	96330	249461	7.34 %
4	115818	2219	42237	160274	4.71 %
5	27687	2384	24882	54953	1.62 %
6	25420	968	20435	46823	1.38 %
7	16523	652	19334	36509	1.07 %
8	11518	521	18883	30922	0.91 %
9–20	27348	2336	44070	73754	2.17 %
21–50	14842	759	12434	28035	0.82 %
51–100	1667	239	5192	7098	0.21 %
101–250	775	46	2311	3132	0.09 %
251–500	233	1	269	503	0.01 %
501–1000	75	0	30	105	0.00 %
> 1000	110	0	28	138	0.00 %
Total	1536086	89544	1774079	3399709	100.00 %

Table 8.2: Test instances grouped by number of sinks and technologies.

small instances to get overall acceptable results. However, it is also important to optimize instances with more sinks well because, in practice, they are often among the timing-critical instances.

Table 8.3 shows the size of the repeater families used as libraries. The designs from 32 nm and 22 nm technologies do not have any buffers. Instead, two consecutive inverters are used if necessary.

We have performed all of our experiments on a machine with two Intel Xeon X5690 processors with 6 cores each. The machine runs with a base clock speed of 3.46 GHz. The Intel® Turbo Boost Technology is enabled with a maximum clock speed of 3.73 GHz. Hyperthreading was disabled. All experiments were run single-threaded, but up to 12 experiments were run in parallel if not noted otherwise. The running times reported might be higher than necessary because the machine was fully loaded with experiments. However, in practice, designers often have to share computing resources with others, too. The machine has 192 GiB of main memory.

The code was compiled with GCC version 4.1.2 under Red Hat Enterprise Linux Server 5.6 at optimization level O2.

8.1 Comparison to an Industrial Tool

As a first experiment, we compare our algorithm to a repeater tree construction tool that is used by IBM for most repeater trees. It uses a van Ginneken-style approach with the running time improvements by Li et al. (2012). It is the default tool in

	45 nm	32 nm	22 nm
Inverter	18	20	22
Buffer	18	0	0

Table 8.3: Repeater Library Sizes

the IBM optimization suite due to its good results and tight integration with the placement tool of the suite.

The integration makes it hard to compare both tools on all instances, because it is not possible to run the industrial tool on a single repeater tree instance without huge overhead. We have chosen a random sample of 19 instances from the Franziska design with different characteristics. Instances have different numbers of sinks and diameters. While it took seconds to run both tools on all instances, testing took one hour due to the overhead.

Table 8.4 shows the results of running the industrial tool, our BonnRepeaterTree Fast Buffering routine, and our BonnRepeaterTree routine with dynamic programming using 40 power buckets. All tools are configured to maximize slack. We have, however, reduced ξ to 0.8 for our repeater insertion. The reason is that higher values would not improve the slack anymore but cause higher area consumption. In practice, we also seldomly use higher ξ values. Both tools are configured to obey blockages and they are only allowed to use the default wiring modes. The instances are already optimized to give all tools reasonable arrival times at the root and required arrival times at the sinks. In practice, the IBM tool would prune the size of the repeater library to improve running time. We configured the tool to use the whole library because this leads to significantly better slacks. Our tool does not prune the library so far.

The overall result is that our dynamic program produces the best slack followed by Fast Buffering even though we reduced the ξ parameter. Better slack, on the other hand, costs more area. In general, we see that the industrial tool uses less area. While the Fast Buffering algorithm finds solutions without any electrical violation, both other tools create violations.

The experiment was performed on an otherwise empty machine to make the running times comparable. The running time of our Fast Buffering version is 1.26 seconds. The IBM tool uses 1.78 seconds. The dynamic program needs 113.8 seconds with 40 buckets and 3.2 seconds without buckets. The results of the version without buckets using the same setup are shown in Table 6.2. All running times include identifying and replacing instances and not only the core algorithm.

8.2 Comparison to Bounds

It is hard to show the quality of our algorithm for the Repeater Tree Problem because optimal solutions are not known. Despite that, for some aspects of the

		Industrial Tool				BonnRepeaterTrees				BonnRepeaterTrees + DP			
	Sinks	SNS	Slack	Area	Vio	SNS	Slack	Area	Vio	SNS	Slack	Area	Vio
I01	1	−400	−400	106	0/0	−379	−379	160	0/0	−377	−377	160	0/0
I02	1	−91	−91	12	0/0	−91	−91	10	0/0	−91	−91	14	0/0
I03	2	−475	−243	112	0/0	−417	−213	144	0/0	−413	−206	150	0/0
I04	2	−156	−82	6	0/0	−153	−85	8	0/0	−161	−81	6	0/0
I05	3	−157	−66	3	0/0	−140	−63	11	0/0	−143	−61	11	0/0
I06	3	−167	−71	25	0/0	−183	−71	43	0/0	−162	−68	35	0/0
I07	4	−25	−16	13	0/0	−26	−14	20	0/0	−22	−11	25	0/0
I08	4	−141	−40	6	0/0	−136	−40	8	0/0	−133	−36	8	0/0
I09	5	−368	−118	23	0/0	−349	−116	38	0/0	−351	−113	40	0/0
I10	8	−190	−30	14	0/0	−127	−20	28	0/0	−117	−20	34	0/0
I11	10	−368	−74	25	0/0	−428	−68	20	0/0	−387	−68	22	0/0
I12	15	−761	−72	68	0/0	−690	−62	107	0/0	−698	−59	116	0/2
I13	24	−74	−16	31	0/0	−79	−16	23	0/0	−60	−8	38	0/0
I14	33	−3799	−238	123	0/1	−3327	−197	228	0/0	−3167	−189	144	0/0
I15	47	−2998	−140	69	0/0	−2833	−108	113	0/0	−2922	−109	101	0/0
I16	65	−3354	−99	40	0/0	−2784	−96	127	0/0	−3852	−96	48	0/0
I17	73	−3208	−102	86	0/0	−4358	−86	165	0/0	−3424	−78	132	0/0
I18	120	−10988	−147	162	0/0	−9431	−107	256	0/0	−10701	−107	333	0/0
I19	322	−273	−29	145	0/25	−6	−3	342	0/0	0	25	851	0/0

Table 8.4: Results of optimizing for slack on several instances from a 22 nm design for the industrial tool, our Fast Buffering routine and our dynamic program implementation that considers power consumption. All times are given in ps. SNS is the sum of negative slacks for all sinks. Slack is the worst slack of the instance. Area is the space consumed by the internal repeaters measured in placement grid steps. Vio gives us the number of load and slew violations in the result.

solutions like netlength, number of inserted repeaters, and slack, we can compare the results to their respective bounds.

ξ	Power	Inversions	Length		Slack Dev.		Wall Time
			Avg.	Max.	Avg.	Max.	
0.0	739.174	1504726	2.0 %	837.7 %	12.07 ps	1419.46 ps	2301 s
0.1	797.945	1587022	1.8 %	833.1 %	9.23 ps	1037.46 ps	2347 s
0.2	882.063	1746005	1.8 %	832.0 %	7.48 ps	781.36 ps	2302 s
0.3	967.900	1920838	1.8 %	841.3 %	6.15 ps	666.23 ps	2291 s
0.4	1069.397	2076479	1.8 %	852.5 %	5.16 ps	534.88 ps	2280 s
0.5	1177.334	2265218	1.8 %	874.0 %	4.36 ps	376.63 ps	2257 s
0.6	1274.543	2547997	1.8 %	896.3 %	3.52 ps	352.00 ps	2213 s
0.7	1479.679	2914892	1.9 %	877.7 %	2.79 ps	250.13 ps	2194 s
0.8	1689.111	3412856	2.3 %	904.3 %	2.16 ps	212.43 ps	2156 s
0.9	2027.596	4163780	3.6 %	926.3 %	1.60 ps	174.89 ps	2116 s
1.0	2970.202	6254108	14.1 %	3321.0 %	1.14 ps	178.36 ps	2119 s

Table 8.5: Results of our repeater tree algorithm for different ξ values.

We ran our algorithm on all instances for ξ values ranging from 0.0 to 1.0. The results are shown in Table 8.5. In general we see that power consumption and netlength increase with higher ξ values. The slack deviation (see below) gets better and even the running time reduces.

Table 8.5 is a summary of all runs over all technologies and all numbers of sinks. We have added detailed tables in Appendix A where the data is separated by technology and number of sinks.

8.2.1 Running Time

We did not focus on running time during testing. The results show that the algorithm runs very fast. The average running time for a single instance is about 0.6 milliseconds, which means that we can solve about 5.7 million instances per hour. The wall time reported in Table 8.5 only contains the time used to build topologies and to insert repeaters. Overhead like identifying instances and adding the result into the design is not reported. Depending on the design, the overhead is between 100 % and 150 % of the running time used to solve the Repeater Tree Problem.

The running time decreases slightly with higher ξ values. The more we try to optimize slack the earlier repeaters are inserted to shield uncritical side paths from the critical ones. In addition more repeaters are added along paths due to timing reasons. The result is that the algorithm works with nets that have a smaller number of sinks. Due to the smaller instances the, subroutine that computes Steiner trees runs significantly faster.

8.2.2 Wirelength

A lower bound on the total wire length of a repeater tree instance is the length of a Steiner minimum tree spanning the root and all sinks. We computed one for all instances with less than 36 sinks. For bigger instances we used the minimum of a Steiner tree heuristic guaranteeing a $3/2$-approximation and the result of our routine over all test runs.

Table 8.5 shows how many percent we are away from the optimal repeater tree length. The optimal length can be 0 if the root is a primary input pin and a single sink is directly below it. Let I be the set of instances with non-zero optimal length. Given for each $i \in I$ the length of our tree $length(i)$ and the optimal length $opt(i)$, Table 8.5 shows the result of the following equation:

$$\frac{1}{|I|} \sum_{i \in I} \frac{length(i) - opt(i)}{opt(i)}.$$

The average length increase compared to an optimal Steiner tree is quite low. However, there are some instances with huge wirelength increases. The detailed tables in Appendix A show that this only happens on instances that use the clustering preprocessing or with high ξ values where we accept detours to keep bifurcations from the critical path.

Due to parallel walk and instances with sinks of different parities, a deviation of almost 100 % can be optimal. Consider an instance with one negative sink close to the root and two other sinks, one negative and one positive, at some distance. One inverter is necessary to negate the inversions, but we have the choice between bridging the distance twice or adding an additional inverter at the negative sink away from the root.

8.2.3 Number of Inserted Inverters

[1]To obtain a lower bound on the number of inverters needed to legally buffer a repeater tree instance let Cap_{extra} arise from the sum of the wire capacitance of a minimum Steiner tree and the input capacitances of all sinks by subtracting the maximum capacitance that can be driven by the root with the given input slew, such that the output slew is at most $optslew$. Every inserted repeater of type t can drive a certain amount $loadlim(t)$ of this capacitance but also contributes its own input capacitance $cap^{\text{in}}(t)$. Let value $MaxCap(t)$ be the biggest load the repeater can drive with an input slew of $optslew$ such that the output slew is smaller or equal to $optslew$.

We may assume $MaxCap(t) > cap^{\text{in}}(t)$. Therefore, if there is a legal inverter tree using x_t inverters of type t, then

$$Cap_{extra} + \sum_{t \in L} cap^{\text{in}}(t) x_t \leq \sum_{t \in L} MaxCap(t) x_t \tag{8.1}$$

[1]Part of this section is from Bartoschek et al. (2007b).

has to be satisfied.

Depending on whether we are interested in the number of inserted inverters, in their total area, or in power consumption, we can assign a cost $c_t \geq 0$ to each inverter type $t \in L$. We ask how well our algorithm minimizes this cost.

To obtain a lower bound on the cost that any inverter tree must have, we consider the problem of minimizing the total cost

$$\sum_{t \in L} c_t x_t$$

subject to (8.1) and $x_t \geq 0$ for all $t \in L$. This is a very simple linear program (LP). The dual LP is

$$\begin{aligned} &\text{maximize } Cap_{extra} y \\ &\text{subject to } (MaxCap(t) - cap^{\text{in}}(t))y \leq c_t \text{ for all } t \in L \\ &\qquad\qquad\quad y \geq 0 \end{aligned}$$

If $Cap_{extra} \leq 0$, then $y^* = 0$ is the optimum solution of this LP. If $Cap_{extra} > 0$, then

$$y^* = \min \left\{ \frac{c_t}{MaxCap(t) - cap^{\text{in}}(t)} \middle| t \in L \right\}$$

is optimum. By the LP duality theorem, the optimum value of the original (primal) LP is

$$\sum_t c_t x_t^* = \min \left\{ \frac{c_t Cap_{extra}}{MaxCap(t) - cap^{\text{in}}(t)} \middle| t \in L \right\}.$$

Of course, if we consider the number of inverters (i.e. $c_t = 1$ for all t), we can round up this lower bound to the next integer.

Three further modifications are possible to improve this bound in some cases: First, if the lower bound is 0 but there is a sink of negative parity, we clearly need at least one inverter. Moreover, if our lower bound is 1 but all sinks have positive parity, we need at least two inverters. Finally, if there is only one sink, we can round up the lower bound to the next even or odd integer, depending on the sink's parity, $+$ or $-$.

The resulting minimum number of inverters that have to be inserted over all of our instances is 1150712. Table 8.5 shows the number of inversions we have added for different ξ values. Each buffer added on the 45 nm instances is counted as two inversions. With $\xi = 0$, we are only 31 % above the bound that does not take slew propagation over wire segments into account. Getting better timing results increases the number of used inversions significantly such that for $\xi = 1.0$ we are 540 % over the bound.

8.2.4 Timing

It is hard to obtain an upper bound on the slack that one can achieve for a single repeater tree instance. We have chosen the following approach.

Figure 8.1: Test setup used for computing slack upper bounds.

For instances with a single sink, we build repeater trees with the highest effort and different parameter sets. We use our algorithm and also the dynamic program algorithm for post optimization. We choose the maximum slack we get over all different runs on the same instance as an upper bound. Obviously, it is not proven that the bound is indeed an upper bound. However, we are confident that the real upper bound is not far away. Any provable upper bound we can conceive is too far away from the actual achievable results because it has to be based on too optimistic assumptions.

For instances with two or more sinks, we construct a test instance for each sink. Each test instance consists of the original sink and an additional sink that corresponds to the input pin of the repeater with the smallest input capacitance. The additional sink is located at the root pin and has infinite required arrival time. The setup models the smallest impact that shielding off all other sinks can have on a critical sink. Figure 8.1 shows the setup.

Similar to instances with a single sink, we then build the best possible repeater tree we can achieve for each test instance. The maximum slack achievable over all runs is used as the upper bound for the according sink.

The minimum slack bound over all sinks is then used as the upper bound for the achievable slack of the repeater tree instance. As in the single sink case, it is not guaranteed that we get a real upper bound. However, it is not possible to achieve the best possible slack at all sinks simultaneously. The more sinks with similar criticality an instance has the worse the achievable slack is compared to the upper bound.

Table 8.5 shows the effect of ξ on the resulting slack deviation from the upper bound. While going to the extreme might be undesirable due to the high increase in netlength and power consumption, using a ξ of up to 0.9 can be justified by the slack improvements.

8.3 Fast Buffering vs. Dynamic Programming

We use our version of the dynamic program algorithm as a post optimization step for the most critical instances. Table 8.6 shows how the Fast Buffering algorithm compares to the dynamic program. The Fast Buffering algorithm is run with $\xi = 1.0$. The table summarizes the results on our testbed for all technologies but separated by number of sinks.

Slack deviation and length deviation are computed in the same way as in Section 8.2. The power columns sum up the power consumption and number of

	Slack Deviation			Power			Length			Running Time		
# Sinks		No DP	DP		No DP	DP		No DP	DP		No DP	DP
1	Avg.	0.32	0.16	Pwr.	500.412	545.606	Avg.	0.0	0.0	Top.	48.28	52.36
	Max.	34.23	43.72	Rpt.	1318262	1472938	Max.	0.0	0.0	Buf.	420.46	9870.21
2	Avg.	1.55	1.08	Pwr.	136.984	148.558	Avg.	0.0	0.0	Top.	11.12	11.91
	Max.	31.78	31.27	Rpt.	507048	625075	Max.	0.0	0.0	Buf.	115.85	2805.00
3	Avg.	2.50	1.69	Pwr.	128.043	145.060	Avg.	14.2	14.2	Top.	8.99	9.66
	Max.	37.42	36.92	Rpt.	462522	580626	Max.	99.3	99.3	Buf.	102.82	2393.74
4	Avg.	2.98	2.14	Pwr.	100.239	124.410	Avg.	22.8	22.8	Top.	6.96	7.36
	Max.	35.41	32.18	Rpt.	303944	410043	Max.	183.5	183.5	Buf.	83.89	1864.14
5	Avg.	3.71	2.55	Pwr.	53.277	62.546	Avg.	23.4	23.3	Top.	4.04	4.26
	Max.	41.38	36.78	Rpt.	190835	266861	Max.	176.7	176.7	Buf.	51.98	1031.52
6	Avg.	4.14	2.73	Pwr.	71.950	82.125	Avg.	12.6	12.6	Top.	3.56	3.79
	Max.	57.69	57.49	Rpt.	189147	268407	Max.	192.0	192.0	Buf.	48.01	1328.70
7	Avg.	4.98	3.34	Pwr.	95.207	106.697	Avg.	32.8	32.7	Top.	3.56	3.75
	Max.	92.07	84.38	Rpt.	194208	284789	Max.	314.4	296.6	Buf.	51.15	1623.05
8	Avg.	7.42	4.82	Pwr.	152.175	167.032	Avg.	31.8	31.8	Top.	4.17	4.34
	Max.	101.25	83.78	Rpt.	268542	369478	Max.	255.8	255.8	Buf.	59.68	2496.50
9–20	Avg.	7.74	5.25	Pwr.	118.038	131.693	Avg.	30.7	30.6	Top.	16.74	17.24
	Max.	73.60	61.00	Rpt.	739550	1055801	Max.	585.8	585.8	Buf.	225.67	3419.13
21–50	Avg.	13.92	9.32	Pwr.	143.395	156.515	Avg.	51.1	50.8	Top.	18.61	19.04
	Max.	92.16	71.64	Rpt.	560739	776370	Max.	1352.1	1337.8	Buf.	192.43	3187.63
51–100	Avg.	20.55	13.62	Pwr.	36.407	36.224	Avg.	75.4	74.6	Top.	18.43	19.00
	Max.	109.36	82.43	Rpt.	363539	518275	Max.	1755.5	1747.7	Buf.	133.07	1542.92
101–250	Avg.	21.22	14.07	Pwr.	25.143	24.841	Avg.	75.7	74.8	Top.	27.78	28.51
	Max.	86.98	80.86	Rpt.	326046	474345	Max.	2110.0	2042.0	Buf.	117.25	1256.83
251–500	Avg.	25.57	18.49	Pwr.	10.403	11.826	Avg.	82.9	81.5	Top.	20.37	20.79
	Max.	148.72	109.02	Rpt.	78538	116911	Max.	926.1	915.7	Buf.	36.41	394.31
501–1000	Avg.	51.36	35.06	Pwr.	14.425	15.619	Avg.	221.6	219.6	Top.	18.83	19.35
	Max.	178.36	132.57	Rpt.	28869	44360	Max.	1643.9	1637.6	Buf.	13.99	285.59
> 1000	Avg.	56.81	30.47	Pwr.	18.945	98.136	Avg.	387.0	376.9	Top.	5.22	5.41
	Max.	163.90	99.67	Rpt.	50864	273687	Max.	1066.1	1065.5	Buf.	7.55	1315.06
Total	Avg.	1.25	0.81	Pwr.	1605.045	1856.887	Avg.	16.4	16.3	Top.	216.65	226.75
	Max.	178.36	132.57	Rpt.	5582653	7537966	Max.	2110.0	2042.0	Buf.	1660.21	34814.33
Total >2	Avg.	4.87	3.30	Pwr.	967.648	1162.723	Avg.	33.4	33.2	Top.	157.26	162.49
	Max.	178.36	132.57	Rpt.	3757343	5439953	Max.	2110.0	2042.0	Buf.	1123.89	22139.11

Table 8.6: Comparison between Fast Buffering and our dynamic program.

repeaters inserted. Buffers are not counted twice here. The running times are given for topology generation and buffering separately.

As expected, the average slack deviation of the dynamic program is smaller. It should be used to get the last tenth of a picosecond from most critical instances. In general, the running time is more than 10× compared to Fast Buffering. In addition, the power consumption increases significantly. However, the additional power might be necessary to get better slacks. Netlengths are very similar for both algorithms.

8.4 Varying η

			Length		Slack Dev.	
η	Power	Repeaters	Avg.	Max.	Avg.	Max.
0.00	1835.132	7415542	16.0 %	2042.0 %	0.81 ps	132.57 ps
0.05	1611.987	5700476	15.0 %	1301.6 %	1.24 ps	148.93 ps
0.10	1598.349	5644917	15.4 %	1491.2 %	1.23 ps	224.05 ps
0.15	1596.091	5596761	15.7 %	1688.2 %	1.24 ps	201.43 ps
0.20	1595.128	5555896	16.0 %	1759.2 %	1.24 ps	216.41 ps
0.25	1592.330	5519986	16.1 %	2110.0 %	1.25 ps	178.36 ps
0.30	1593.975	5505832	16.4 %	2367.6 %	1.26 ps	221.72 ps
0.35	1592.487	5499882	16.4 %	2481.0 %	1.26 ps	195.71 ps
0.40	1591.764	5497906	16.4 %	2534.0 %	1.26 ps	204.04 ps
0.45	1590.603	5492262	16.4 %	2246.3 %	1.26 ps	203.28 ps
0.50	1589.791	5487923	16.3 %	2117.9 %	1.26 ps	196.78 ps

Table 8.7: Results of our repeater tree algorithm for different η values.

The next set of experiments compares the results of our algorithm optimizing for slack ($\xi = 1.0$) but with different η parameters. Table 8.7 shows the resulting power consumption, number of repeaters, length deviation and slack deviation.

In general, higher values of η consume less power but have higher netlength and higher slack deviation. Our preferred value of 0.25 seems to be a reasonable choice. Smaller values like 0.15 and 0.20 are also good candidates.

A choice of $\eta = 0.0$ is a special case. It allows to assign the whole d_{node} to a branch. Thus, all sinks whose criticality is better than d_{node} compared to the most critical sink can be connected to the critical sink without degrading the required arrival time. This leads to the degenerated case that it is favourable to add bifurcations to the most critical sink. During buffering a lot of repeaters are added to reduce the impact of the additional bifurcations. This explains the high power consumption of this run. Somewhat surprisingly, the average slack deviation is best with $\eta = 0.0$.

8.5 Varying d_{node}

			Length		Slack Dev.	
d_{node}-factor	Power	Repeaters	Avg.	Max.	Avg.	Max.
0.0	147.096	2412342	11.7 %	225.7 %	6.41 ps	97.72 ps
0.5	136.963	2296316	25.5 %	1181.6 %	4.71 ps	78.19 ps
1.0	135.092	2264650	32.9 %	2110.0 %	4.52 ps	84.31 ps
1.5	138.993	2321192	40.8 %	2154.6 %	4.52 ps	115.26 ps
2.0	144.373	2406360	48.9 %	2486.2 %	4.63 ps	112.52 ps
2.5	149.157	2489224	56.0 %	2560.3 %	4.73 ps	127.15 ps

Table 8.8: Results of our repeater tree algorithm for different d_{node} scaling factor values. We only consider instances with more than two sinks.

In Chapter 5 we claimed that it is necessary to add a bifurcation delay in our delay model to take additional capacitance on side paths into account. Table 8.8 shows the results of scaling the precomputed d_{node} value by certain factors optimizing our instances with $\xi = 1.0$. We only consider instances with more than two sinks, because d_{node} has little effect for two-sink instances and no effect on single-sink instances. Disabling d_{node} altogether has good effects on the repeater tree length because no detours are added to avoid additional delay on paths to critical sinks. On the other hand, the average slack deviation goes up significantly, and a lot of repeaters are added to shield off critical repeaters.

If d_{node} gets too big, the algorithm tends to build more balanced topologies. Detours are accepted to decrease the number of bifurcations on root-sink paths. The result is high netlength and a high repeater count.

Altogether, our current choice of d_{node} with factor 1.0 has the lowest repeater count and slack deviation. The additional netlength is acceptable because it leads to less repeaters.

8.6 Disabling Effort Assignment

All experiments so far used the assign effort step (see Section 6.1) to reduce the power consumption in tree parts that are above the slack target. To evaluate the effects of this step, we compared it to runs where the step was skipped using $\xi = 1.0$.

Table 8.9 shows how power consumption can be saved due to AssignEffort. We see that for some instances the power savings are significant. The potential depends on how critical the timing of the design is. The table shows that for some instances the power gets worse. This is due to the inexact repeater insertion. Most instances, however, are completely below the slack threshold and AssignEffort does not apply.

	Repeaters			pwr(A) - pwr(N)		
Design	Assign	No Assign	$pwr(A)/pwr(N)$	< 0	$= 0$	> 0
Baldassare	28151	30687	88.87 %	1849	18108	668
Beate	53198	55264	95.16 %	1588	31938	1104
Wolfram	40469	45411	89.68 %	2694	40816	1247
Gerben	44680	45852	98.47 %	1192	42300	1295
Luciano	165743	179219	89.38 %	8619	89374	3802
Benedikt	183350	289006	60.38 %	32021	201440	6887
Renaud	155931	152752	90.73 %	7492	256103	5080
Julius	604506	666291	89.18 %	25790	250624	9164
Franziska	631607	667934	92.86 %	14515	306362	9182
Meinolf	588236	614779	93.55 %	18623	335051	12974
Iris	963296	1026381	92.85 %	18016	365939	9861
Gautier	641085	1179108	49.22 %	186116	1079702	1593

Table 8.9: Repeaters used and power consumption in runs with and without AssignEffort. $pwr(A)$ ($pwr(N)$) is the power consumption of the run with (without) AssignEffort. The last three columns show on how many instances the power has improved (< 0), stayed the same ($= 0$), or degraded (> 0) due to AssignEffort.

	Slack Degradation		$slk(A) - slk(N)$		
Design	Assign	No Assign	< 0	$= 0$	> 0
Baldassare	0.54	0.53	594	19678	353
Beate	1.27	1.28	818	33189	623
Wolfram	0.43	0.46	717	43166	874
Gerben	1.79	1.79	832	42300	676
Luciano	0.56	0.56	2539	97663	1593
Benedikt	0.08	0.08	1041	238633	674
Renaud	0.20	0.18	1533	266437	705
Julius	0.77	0.76	7049	273785	4744
Franziska	0.98	0.97	6219	320887	3553
Meinolf	0.60	0.56	8225	355124	3299
Iris	0.51	0.49	5435	385838	2543
Gautier	0.02	0.00	989	1266301	121

Table 8.10: Slack degradation in runs with and without AssignEffort. The last three columns show for how many instances the slack below the slack target got worse, stayed the same, or got better due to AssignEffort.

Table 8.10 shows how the slack is affected by AssignEffort. The slack degradation is measured in the same way as in Section 8.2. The overall average degradation is shown. The effect of AssignEffort is that for a lot of instances the slack gets closer to the slack target. Due to the heuristic nature of our buffer insertion, more slacks can get worse than the slack target. This is also the reason for improvements on some nets. A different repeater insertion on an uncritical sidepath can accidentally lead to better buffering on the critical path. Overall, the vast majority of instances does not get worse.

8.7 Disabling Parallel Mode

A large part of the complexity of our repeater insertion routine lies in handling clusters in parallel mode. Buffering algorithms that use a variant of van Ginneken's algorithm, on the other hand, do not change the topology during processing and produce good results. We have to ask, whether using parallel clusters is worth the effort.

We processed the test instances with a variant of our algorithm that is not allowed to enter parallel mode. Merge solutions that would enter parallel mode are disabled. Table 8.11 shows the result of the comparison summed up for all technologies. The table compares the slack deviation from our upper bound, the power consumption and number of inserted inverter stages, and the deviation from the Minimum Steiner Tree for both runs. In general, the slack gets better if we are allowed to walk in parallel. The static power consumption is reduced by about 20 %. The average increase in netlength that corresponds to increase in dynamic power consumption is quite small. It is expected that netlength increases if we are allowed to go in parallel due to segments that are used twice. As we have disabled merge cases that result in parallel walk, the work done during merging decreases. Accordingly, the running time is smaller. Given the big improvements in slack and static power consumption, we accept the small degradations in netlength and running time.

8.8 Choosing Tradeoff Parameters

So far, we used our power-slack parameter ξ uniformly for all parts of the algorithm. However, as explained in Section 7.3.2 it is possible to use a different parameter for topology generation (ξ_t), repeater insertion (ξ_r), or buffering mode selection (ξ_m). We have varied all three parameters independently from 0.0 to 1.0 in 0.1 steps and optimized each design for each combination. The input netlists were placed, repeater trees were optimized for power, and gate sizing was performed. For each of the resulting $11 \times 11 \times 11$ runs we measured the resulting sum of negative endpoint slacks (SNS), area consumption (which correlates with power consumption), netlength, and worst slack. The measurements are not restricted to repeater trees. Instead, SNS is read at timing points where tests are performed. All nets are counted for netlength. Area consumption counts all gates in the design.

# Sinks		Slack Deviation			Power			Length			Running Time	
		Parallel	No Parallel		Parallel	No Parallel		Parallel	No Parallel		Parallel	No Parallel
1	Avg.	0.28	0.28	Pwr.	738.292	738.291	Avg.	0.0	0.0	Top.	58.77	60.57
	Max.	36.20	36.20	Rpt.	1545171	1545171	Max.	0.0	0.0	Buf.	513.68	521.59
2	Avg.	1.53	2.43	Pwr.	476.554	513.288	Avg.	0.0	0.0	Top.	16.21	16.21
	Max.	41.61	43.85	Rpt.	814674	852053	Max.	0.0	0.0	Buf.	176.20	167.21
3	Avg.	1.83	3.20	Pwr.	310.435	339.320	Avg.	14.7	14.7	Top.	14.50	14.34
	Max.	37.42	43.55	Rpt.	717013	764715	Max.	99.8	99.8	Buf.	166.36	154.34
4	Avg.	2.99	5.72	Pwr.	271.303	362.414	Avg.	15.9	15.8	Top.	12.06	11.78
	Max.	60.79	60.79	Rpt.	528539	574810	Max.	183.5	183.5	Buf.	150.73	133.12
5	Avg.	3.40	5.80	Pwr.	115.726	137.936	Avg.	30.1	29.8	Top.	5.14	5.06
	Max.	72.62	81.64	Rpt.	250945	277530	Max.	196.6	196.6	Buf.	66.42	57.58
6	Avg.	4.00	7.03	Pwr.	293.898	337.360	Avg.	39.2	39.0	Top.	5.21	5.16
	Max.	62.55	62.55	Rpt.	350513	381772	Max.	198.4	198.4	Buf.	71.96	62.54
7	Avg.	6.27	9.49	Pwr.	220.964	259.513	Avg.	31.8	31.4	Top.	4.81	4.78
	Max.	92.07	93.68	Rpt.	294115	329636	Max.	314.4	314.4	Buf.	68.37	56.56
8	Avg.	7.32	9.62	Pwr.	199.535	224.720	Avg.	27.4	27.0	Top.	4.75	4.71
	Max.	101.25	88.82	Rpt.	306462	329179	Max.	255.8	307.3	Buf.	67.55	58.32
9–20	Avg.	7.24	10.13	Pwr.	179.345	225.163	Avg.	25.6	24.0	Top.	19.69	19.78
	Max.	73.60	76.63	Rpt.	810857	912467	Max.	585.8	585.8	Buf.	263.99	218.87
21–50	Avg.	12.24	15.50	Pwr.	187.628	219.563	Avg.	44.5	41.7	Top.	23.55	23.39
	Max.	92.16	112.70	Rpt.	612392	682592	Max.	1352.1	1347.0	Buf.	246.82	204.03
51–100	Avg.	18.17	24.55	Pwr.	54.287	71.303	Avg.	65.5	60.4	Top.	20.62	20.93
	Max.	109.36	110.56	Rpt.	392700	469790	Max.	1755.5	1755.6	Buf.	150.97	118.91
101–250	Avg.	19.50	28.24	Pwr.	40.271	53.588	Avg.	70.2	63.8	Top.	34.20	34.46
	Max.	105.18	114.12	Rpt.	338506	420841	Max.	2110.0	2041.7	Buf.	146.51	114.15
251–500	Avg.	26.48	38.84	Pwr.	20.481	25.198	Avg.	110.3	99.5	Top.	26.52	26.31
	Max.	154.40	148.21	Rpt.	84870	112493	Max.	3321.0	3196.4	Buf.	51.69	38.03
501–1000	Avg.	73.19	89.59	Pwr.	32.716	38.808	Avg.	229.1	220.9	Top.	40.53	39.83
	Max.	232.30	198.38	Rpt.	39868	49175	Max.	1643.9	1636.7	Buf.	25.18	18.70
> 1000	Avg.	59.89	74.60	Pwr.	21.881	27.224	Avg.	380.9	372.2	Top.	5.34	5.37
	Max.	163.90	177.99	Rpt.	53972	60018	Max.	1096.2	1086.4	Buf.	8.04	6.05
Total	Avg.	1.21	1.79	Pwr.	3163.316	3573.690	Avg.	16.2	15.7	Top.	291.89	292.70
	Max.	232.30	198.38	Rpt.	7140597	7762242	Max.	3321.0	3196.4	Buf.	2174.45	1929.99
Total >2	Avg.	4.14	6.50	Pwr.	1948.470	2322.111	Avg.	31.5	30.6	Top.	216.92	215.92
	Max.	232.30	198.38	Rpt.	4780752	5365018	Max.	3321.0	3196.4	Buf.	1484.57	1241.19

Table 8.11: Testing the effects of parallel mode.

SNS	Worst Slack	Area	Netlength	ξ_t	ξ_r	ξ_m
-2460784 ps	-360.960 ps	4264514	11421392775 nm	1.0	0.7	1.0
-2503732 ps	-371.797 ps	4161722	11396585038 nm	1.0	0.6	1.0
-2526592 ps	-368.434 ps	4139336	11384182618 nm	1.0	0.7	0.9
-2529250 ps	-361.152 ps	4039524	11357719162 nm	1.0	0.6	0.9
-2581487 ps	-400.693 ps	4003001	10823042062 nm	0.9	0.6	0.9
-2587899 ps	-351.772 ps	3977094	11315124985 nm	1.0	0.6	0.8
-2598980 ps	-345.037 ps	3940266	11287051361 nm	1.0	0.6	0.7
-2661338 ps	-357.609 ps	3909961	11255902718 nm	1.0	0.6	0.6
-2692252 ps	-381.317 ps	3882043	11247554519 nm	1.0	0.5	0.7
-2721807 ps	-380.967 ps	3856202	11222038997 nm	1.0	0.5	0.6
-2774560 ps	-381.654 ps	3841977	11215744785 nm	1.0	0.4	0.7
-2812004 ps	-378.829 ps	3816806	11186602609 nm	1.0	0.4	0.6
-2909451 ps	-402.322 ps	3815549	10683834075 nm	0.7	0.5	0.5
-2912460 ps	-402.322 ps	3814486	10667397255 nm	0.6	0.5	0.5
-2914087 ps	-408.144 ps	3802287	10777654190 nm	0.9	0.4	0.6
-2933527 ps	-422.991 ps	3797866	11155820658 nm	1.0	0.4	0.5
-2963781 ps	-397.433 ps	3795544	11146478260 nm	1.0	0.3	0.6
-2975351 ps	-415.343 ps	3792524	11131954147 nm	1.0	0.4	0.4
-2985396 ps	-403.745 ps	3785308	10769355282 nm	0.9	0.4	0.5
-3006647 ps	-387.869 ps	3784393	10702389423 nm	0.8	0.4	0.5
-3022317 ps	-381.193 ps	3780095	10764354020 nm	0.9	0.4	0.4
-3052016 ps	-420.933 ps	3779169	11128458742 nm	1.0	0.3	0.5
-3055216 ps	-414.426 ps	3774567	11104510839 nm	1.0	0.3	0.4
-3096810 ps	-387.917 ps	3769622	10762057030 nm	0.9	0.3	0.5
-3136457 ps	-434.217 ps	3765553	10756910041 nm	0.9	0.3	0.4
-3154245 ps	-407.107 ps	3765336	10696614904 nm	0.8	0.3	0.4
-3157275 ps	-455.027 ps	3763362	11074617288 nm	1.0	0.3	0.0
-3200980 ps	-411.824 ps	3759277	10753998314 nm	0.9	0.3	0.3
-3273230 ps	-433.453 ps	3759111	10697382852 nm	0.8	0.3	0.3
-3282542 ps	-427.694 ps	3756085	10748223421 nm	0.9	0.3	0.0
-3419364 ps	-431.093 ps	3754110	10734650521 nm	0.9	0.2	0.0

Table 8.12: Non-dominated parameter sets on the Franziska design.

SNS	Worst Slack	Area	Netlength	ξ_t	ξ_r	ξ_m
-5196277 ps	-631.933 ps	4527479	8925535585 nm	0.9	0.8	0.9
-5257051 ps	-635.508 ps	4509205	8819283865 nm	0.8	0.8	0.9
-5269220 ps	-635.053 ps	4472708	8785736183 nm	0.8	0.7	1.0
-5365769 ps	-635.359 ps	4348123	8906582047 nm	0.9	0.7	0.9
-5383743 ps	-638.916 ps	4336342	8801360797 nm	0.8	0.7	0.9
-5449405 ps	-636.583 ps	4247059	8816022521 nm	0.8	0.7	0.8
-5503767 ps	-653.201 ps	4197911	8928168330 nm	0.9	0.7	0.7
-5546630 ps	-635.822 ps	4157311	8907015090 nm	0.9	0.6	0.8
-5621983 ps	-637.161 ps	4143570	8775993529 nm	0.7	0.6	0.8
-5653641 ps	-666.391 ps	4136134	8934564389 nm	0.9	0.7	0.5
-5666127 ps	-649.055 ps	4090594	8907969542 nm	0.9	0.6	0.7
-5725306 ps	-649.055 ps	4082070	8817992649 nm	0.8	0.6	0.7
-5780351 ps	-642.621 ps	4079167	8786841694 nm	0.7	0.6	0.7
-5807757 ps	-658.837 ps	4039493	8825447381 nm	0.8	0.6	0.6
-5860053 ps	-663.175 ps	4035272	8794310975 nm	0.7	0.6	0.6
-5919674 ps	-657.425 ps	4031406	8777209504 nm	0.6	0.6	0.6
-6007460 ps	-661.243 ps	4004925	8906715403 nm	0.9	0.5	0.7
-6071154 ps	-666.057 ps	3964178	8904311050 nm	0.9	0.5	0.6
-6130646 ps	-677.938 ps	3953669	9410850288 nm	1.0	0.5	0.5
-6189021 ps	-666.374 ps	3948620	8776106642 nm	0.6	0.5	0.6
-6306196 ps	-660.793 ps	3919318	9393366028 nm	1.0	0.5	0.4
-6434807 ps	-682.030 ps	3896112	8898249360 nm	0.9	0.5	0.4
-6512938 ps	-695.812 ps	3874760	8893895626 nm	0.9	0.5	0.3
-6635688 ps	-662.423 ps	3872070	9372604172 nm	1.0	0.4	0.4
-6639647 ps	-759.335 ps	3859165	9345686625 nm	1.0	0.5	0.0
-6656564 ps	-673.105 ps	3851146	9357824853 nm	1.0	0.4	0.3
-6674811 ps	-725.778 ps	3847456	8897120664 nm	0.9	0.5	0.1
-6694420 ps	-766.073 ps	3838560	8897072449 nm	0.9	0.5	0.0
-6745947 ps	-718.396 ps	3833113	8893540101 nm	0.9	0.4	0.3
-6836574 ps	-728.312 ps	3827277	8829467006 nm	0.8	0.4	0.3
-6864691 ps	-737.688 ps	3819098	8895643021 nm	0.9	0.4	0.2
-7008116 ps	-738.604 ps	3807331	8893223837 nm	0.9	0.4	0.1
-7052258 ps	-732.529 ps	3803516	8831553876 nm	0.8	0.4	0.1
-7104017 ps	-762.535 ps	3798011	8887622641 nm	0.9	0.4	0.0
-7165642 ps	-741.704 ps	3781045	9280930291 nm	1.0	0.3	0.0
-7306835 ps	-778.668 ps	3769847	8875640716 nm	0.9	0.3	0.0
-7430083 ps	-778.668 ps	3769836	8825179592 nm	0.8	0.3	0.0
-7819054 ps	-844.052 ps	3765793	8852407160 nm	0.9	0.2	0.1
-8071842 ps	-797.117 ps	3759992	8848223450 nm	0.9	0.2	0.0

Table 8.13: Non-dominated parameter sets on the Iris design.

Table 8.12 and Table 8.13 show non-dominated parameter sets (ξ_t, ξ_r, ξ_m) for two example designs. A set is not dominated if there is no other set with better or equal SNS and better or equal area. Area consumption is measured in units internal to the placement engine. It can only be compared relatively.

The tables indicate that choosing ξ_t above 0.7 is preferred, even if one does not care for timing. On the other hand, ξ_m and ξ_r scale nicely between power-aware and slack optimizing repeater trees.

A Detailed Comparison Tables

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	10.64	9.44	7.48	5.87	4.62	3.70	2.87	2.00	1.39	0.80	0.38	0.07
	Max.	247.71	213.35	190.29	167.15	148.34	132.12	126.82	88.91	66.38	40.60	23.81	43.72
2	Avg.	25.26	21.34	16.90	13.42	11.13	9.27	7.87	6.39	5.03	3.44	1.59	0.94
	Max.	256.22	193.40	155.58	137.14	124.63	106.57	90.57	68.01	53.13	30.26	31.78	31.27
3	Avg.	36.01	29.37	22.61	18.34	15.12	12.47	10.35	8.31	6.28	4.17	2.31	1.41
	Max.	220.36	171.63	141.15	131.27	125.30	93.83	78.46	63.28	46.90	31.32	32.90	36.92
4	Avg.	28.06	23.00	18.09	14.76	12.14	10.16	8.70	7.36	5.83	3.96	2.45	1.72
	Max.	235.76	165.66	131.52	112.36	103.72	97.22	67.57	48.54	52.09	35.72	35.41	32.18
5	Avg.	45.04	34.24	25.04	19.89	16.29	13.27	11.10	8.99	6.68	4.40	3.01	2.03
	Max.	320.20	292.29	253.06	160.18	124.47	92.84	77.20	70.00	55.70	46.37	41.38	27.92
6	Avg.	71.94	56.64	44.09	34.07	27.78	23.49	20.16	15.98	11.39	6.96	4.36	2.27
	Max.	269.21	184.60	168.54	149.68	134.47	121.58	103.94	83.45	64.64	49.41	57.69	57.49
7	Avg.	94.60	63.26	47.82	38.31	31.67	26.16	22.10	17.23	12.88	8.81	6.25	3.70
	Max.	374.31	262.16	214.23	230.03	156.98	146.81	138.78	112.25	104.10	104.44	92.07	84.38
8	Avg.	109.82	72.70	55.14	45.16	37.91	31.37	26.51	20.84	15.41	10.23	7.12	4.13
	Max.	584.53	584.53	273.53	273.53	185.21	180.72	149.05	120.49	108.67	93.22	101.25	83.78
9–20	Avg.	88.28	58.02	41.49	33.78	28.45	23.99	19.91	15.05	11.12	8.10	6.60	4.79
	Max.	557.50	557.50	227.28	227.28	188.57	159.22	107.55	82.63	71.42	50.14	73.60	61.00
21–50	Avg.	127.30	93.22	74.47	62.32	52.33	43.39	35.49	27.87	21.64	16.89	15.01	9.95
	Max.	480.46	266.21	237.06	203.11	167.58	147.85	136.01	105.62	77.62	71.61	92.16	71.64
51–100	Avg.	203.58	132.33	97.18	79.59	64.40	52.39	40.45	29.60	22.61	17.49	15.56	11.18
	Max.	693.86	350.25	339.29	284.41	266.51	208.37	182.17	153.52	135.76	120.09	109.36	77.88
101–250	Avg.	275.94	155.55	111.38	92.02	72.73	60.87	46.41	33.80	25.04	19.19	17.13	12.27
	Max.	569.89	363.89	259.88	222.04	201.31	188.48	135.90	122.70	87.87	87.92	81.82	69.03
251–500	Avg.	312.17	171.85	120.24	93.99	71.59	51.82	36.83	27.54	21.10	16.18	16.39	12.46
	Max.	701.41	414.25	266.72	273.57	229.18	186.29	131.03	116.98	106.49	113.01	148.72	95.53
501–1000	Avg.	670.98	357.14	271.76	216.80	181.17	147.32	120.71	97.80	83.03	69.00	65.18	46.11
	Max.	1329.86	696.60	567.61	511.95	418.86	355.99	352.00	236.72	207.64	174.89	178.36	132.57
> 1000	Avg.	367.51	271.57	219.91	204.83	166.20	146.71	133.56	111.10	100.55	81.25	66.35	36.37
	Max.	1082.28	849.02	736.17	666.23	364.46	300.75	294.24	232.69	212.43	141.56	117.80	83.70
Total	Avg.	21.01	16.91	13.16	10.51	8.54	7.00	5.71	4.37	3.23	2.11	1.28	0.70
	Max.	1329.86	849.02	736.17	666.23	418.86	355.99	352.00	236.72	212.43	174.89	178.36	132.57
Total >2	Avg.	53.64	39.88	30.45	24.70	20.46	17.01	14.24	11.37	8.61	5.93	4.12	2.69
	Max.	1329.86	849.02	736.17	666.23	418.86	355.99	352.00	236.72	212.43	174.89	178.36	132.57

Table A.1: Worst Slack Deviation on 45 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	Avg.	0.1	0.1	0.1	0.1	0.1	0.2	0.3	0.5	1.5	4.8	16.9	16.9
	Max.	89.5	92.1	92.1	92.1	92.1	92.1	92.1	92.1	92.1	96.2	99.3	99.3
4	Avg.	0.4	0.4	0.5	0.5	0.6	0.7	0.8	1.2	2.9	8.7	27.2	27.2
	Max.	94.9	96.1	96.1	96.1	96.1	96.1	94.1	87.3	94.1	109.3	167.2	167.2
5	Avg.	2.5	2.6	2.5	2.4	2.5	2.5	2.4	2.8	4.5	8.0	26.9	26.9
	Max.	96.2	97.1	97.1	96.8	96.8	96.8	96.8	96.8	96.8	118.4	125.8	125.8
6	Avg.	0.9	0.9	0.9	0.9	0.9	0.9	0.9	1.2	2.4	4.3	11.3	11.3
	Max.	97.6	97.6	97.6	97.6	97.6	97.6	93.9	106.6	125.3	128.2	151.0	151.0
7	Avg.	3.7	2.2	2.2	2.0	1.9	1.8	1.7	1.8	2.3	3.8	34.2	34.2
	Max.	97.6	98.5	98.5	98.5	96.0	98.9	98.9	128.1	134.7	160.4	221.4	221.4
8	Avg.	4.1	3.1	2.9	2.7	2.6	2.5	2.4	2.4	2.8	4.3	34.4	34.4
	Max.	99.4	96.7	96.7	94.9	94.5	94.3	93.9	112.9	137.0	112.9	247.7	247.7
9–20	Avg.	7.7	6.1	5.9	5.9	6.0	6.0	5.9	5.9	8.0	12.0	34.6	34.6
	Max.	97.9	97.9	96.1	95.8	95.8	94.9	93.2	107.1	112.3	123.2	319.4	319.4
21–50	Avg.	8.9	8.5	8.7	8.9	9.1	9.6	9.6	10.0	12.4	17.5	51.9	51.7
	Max.	86.2	86.2	86.2	86.2	86.2	89.4	89.4	81.4	78.3	102.0	1352.1	1337.8
51–100	Avg.	20.3	19.6	19.6	19.4	19.4	19.5	18.1	15.7	15.8	19.0	52.1	51.9
	Max.	84.9	83.0	83.0	83.0	86.5	78.5	84.3	62.1	58.8	61.7	313.1	310.7
101–250	Avg.	32.5	31.9	31.6	30.9	31.8	32.4	29.4	25.2	24.5	26.9	58.1	57.8
	Max.	92.1	92.1	85.2	85.2	85.2	86.6	76.2	64.5	56.7	64.5	638.6	638.0
251–500	Avg.	18.7	21.7	22.8	24.7	24.0	25.6	26.2	23.5	24.6	27.4	58.2	57.9
	Max.	49.5	54.8	51.7	51.5	49.0	49.0	54.9	36.7	34.4	38.2	609.6	609.6
501–1000	Avg.	24.0	26.8	27.8	28.6	28.4	27.6	26.5	27.0	28.7	34.8	286.7	284.3
	Max.	58.5	60.5	57.6	57.5	57.4	52.9	42.9	37.1	40.6	43.3	1026.0	1019.6
> 1000	Avg.	373.5	381.5	380.5	384.1	385.6	388.1	387.0	391.9	396.0	399.8	440.4	432.2
	Max.	837.7	833.1	832.0	841.3	852.5	874.0	896.3	877.7	904.3	926.3	947.4	950.2
Total	Avg.	2.0	1.8	1.7	1.7	1.8	1.8	1.8	1.8	2.4	4.0	14.8	14.8
	Max.	837.7	833.1	832.0	841.3	852.5	874.0	896.3	877.7	904.3	926.3	1352.1	1337.8
Total >2	Avg.	4.3	3.7	3.6	3.6	3.7	3.7	3.6	3.8	5.0	8.2	30.8	30.7
	Max.	837.7	833.1	832.0	841.3	852.5	874.0	896.3	877.7	904.3	926.3	1352.1	1337.8

Table A.2: Length Deviation on 45 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Pwr.	141.378	147.234	162.069	179.284	200.144	223.675	238.193	287.045	321.524	371.676	450.346	494.524
	Rpt.	80207	90849	111514	138754	165707	194912	231389	281488	322208	371683	437870	472559
2	Pwr.	35.435	37.535	41.529	46.462	51.158	56.820	59.704	70.059	78.370	90.067	115.505	126.756
	Rpt.	26199	28785	35294	42839	48828	54944	61542	69676	79428	92518	131093	151748
3	Pwr.	25.558	26.851	30.084	33.203	37.621	42.734	47.177	54.582	63.747	77.255	108.964	126.653
	Rpt.	28425	30466	36236	42246	47224	53043	59630	67186	78419	95953	138495	170192
4	Pwr.	12.538	13.908	16.832	19.660	23.012	26.202	29.267	33.483	40.067	52.097	89.763	113.599
	Rpt.	18647	20776	26546	31550	35162	39926	44840	50020	60109	78893	131243	172242
5	Pwr.	8.873	9.619	11.013	12.198	13.658	15.302	16.729	18.940	21.934	26.998	45.021	53.831
	Rpt.	11293	12938	15787	17741	19586	21759	24364	27716	33213	41799	66649	85141
6	Pwr.	21.505	22.933	24.317	25.338	26.962	29.007	29.875	34.028	38.036	45.235	63.878	73.540
	Rpt.	14298	15964	17974	19963	21838	24019	26542	29856	35317	43514	64404	85815
7	Pwr.	25.252	27.492	29.111	29.750	31.153	32.811	33.499	38.156	42.116	48.643	78.558	88.351
	Rpt.	15677	17828	20178	21990	23623	25673	28088	31656	36564	43244	65219	87882
8	Pwr.	41.883	44.377	45.992	46.531	49.262	51.807	52.388	59.482	65.767	75.756	121.124	134.596
	Rpt.	23675	26320	29053	31618	34004	37235	40672	45849	52848	62585	94480	126580
9–20	Pwr.	15.400	17.629	19.565	20.870	22.722	24.571	26.477	30.710	35.626	44.132	74.457	88.898
	Rpt.	19572	22702	26283	29546	33796	37720	43845	52812	65158	83375	145130	209131
21–50	Pwr.	13.661	16.881	19.065	21.014	23.858	26.962	30.415	35.743	43.316	57.159	112.488	128.392
	Rpt.	15082	17837	20688	23561	27241	31799	38133	45762	57477	77829	157571	234372
51–100	Pwr.	1.605	2.035	2.269	2.498	2.819	3.221	3.669	4.381	5.188	6.605	12.945	15.368
	Rpt.	1987	2289	2661	3078	3738	4451	5835	7906	10107	13217	26476	40344
101–250	Pwr.	0.362	0.507	0.601	0.718	0.854	1.037	1.262	1.530	1.862	2.450	5.080	6.056
	Rpt.	663	776	946	1100	1444	1786	2514	3341	4359	5773	11548	18294
251–500	Pwr.	0.314	0.408	0.476	0.540	0.647	0.765	0.897	1.021	1.268	1.644	3.796	4.628
	Rpt.	332	379	461	550	713	893	1106	1341	1840	2674	7010	10606
501–1000	Pwr.	0.454	0.616	0.714	0.797	0.952	1.103	1.334	1.516	1.914	2.532	7.938	8.589
	Rpt.	414	486	551	632	770	951	1264	1384	1783	2565	6913	10743
> 1000	Pwr.	1.042	1.154	1.235	1.315	1.450	1.585	1.706	1.984	2.287	2.836	4.362	23.114
	Rpt.	1515	1625	1727	1839	1997	2182	2405	2676	3013	3680	4633	40911
Total	Pwr.	345.260	369.176	404.873	440.179	486.271	537.601	572.593	672.660	763.023	905.085	1294.228	1486.895
	Rpt.	257986	290020	345899	407007	465671	531293	612169	718669	841843	1019302	1488734	1916560
Total >2	Pwr.	168.447	184.407	201.275	214.433	234.969	257.106	274.695	315.556	363.129	443.342	728.377	865.615
	Rpt.	151580	170386	199091	225414	251136	281437	319238	367505	440207	555101	919771	1292253

Table A.3: Power Consumption on 45 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Top.	12.72	12.91	12.46	12.84	13.01	12.81	12.21	12.43	12.53	12.37	12.07	13.36
	Buf.	107.79	109.17	104.63	109.14	110.92	108.70	103.09	108.09	109.71	109.07	107.74	6697.77
2	Top.	2.63	2.69	2.62	2.67	2.69	2.66	2.56	2.61	2.59	2.58	2.55	2.79
	Buf.	27.56	27.77	26.86	27.60	28.30	27.98	26.81	28.01	28.14	28.12	28.41	1755.45
3	Top.	2.48	2.55	2.49	2.53	2.56	2.53	2.45	2.48	2.47	2.47	2.44	2.63
	Buf.	29.40	29.63	28.52	29.15	29.82	29.60	28.20	28.64	28.25	28.23	28.76	1502.51
4	Top.	3.35	3.47	3.41	3.45	3.49	3.44	3.35	3.41	3.38	3.38	3.37	3.54
	Buf.	40.78	40.92	39.58	40.16	41.15	40.66	38.76	39.70	38.83	38.91	39.49	1353.03
5	Top.	1.31	1.37	1.35	1.36	1.37	1.36	1.33	1.34	1.33	1.34	1.33	1.40
	Buf.	18.13	18.21	17.60	17.88	18.19	18.02	16.93	16.80	16.36	16.23	16.42	640.97
6	Top.	1.01	1.07	1.06	1.06	1.07	1.06	1.04	1.05	1.04	1.05	1.04	1.10
	Buf.	14.98	14.95	14.44	14.61	14.98	14.86	14.08	14.03	13.60	13.54	13.73	957.22
7	Top.	0.74	0.80	0.79	0.79	0.80	0.80	0.78	0.79	0.78	0.78	0.78	0.82
	Buf.	11.69	11.70	11.37	11.53	11.70	11.64	11.03	10.71	10.35	10.35	10.68	1116.63
8	Top.	0.94	1.02	1.01	1.01	1.02	1.01	0.99	1.00	0.99	1.00	0.99	1.03
	Buf.	14.92	14.94	14.37	14.55	14.80	14.72	14.08	13.72	13.33	13.33	13.92	1726.36
9–20	Top.	3.39	3.96	3.91	3.92	3.95	3.96	3.85	3.89	3.85	3.87	3.82	3.87
	Buf.	59.87	59.32	57.80	58.00	58.17	57.92	55.40	50.04	47.85	46.59	45.98	1473.50
21–50	Top.	4.65	7.01	6.92	6.89	6.97	6.88	6.77	6.89	6.82	6.93	6.96	7.08
	Buf.	91.65	89.61	85.31	84.44	84.77	82.62	76.58	73.25	69.14	67.16	66.16	1782.41
51–100	Top.	0.78	1.61	1.59	1.58	1.58	1.58	1.56	1.57	1.57	1.59	1.62	1.64
	Buf.	20.29	19.91	18.93	18.42	17.76	17.11	15.53	13.25	12.14	11.40	10.91	260.35
101–250	Top.	0.42	1.32	1.28	1.27	1.27	1.26	1.24	1.25	1.24	1.26	1.32	1.34
	Buf.	9.61	9.89	9.35	8.91	8.63	8.31	7.32	6.43	5.75	5.38	5.08	108.55
251–500	Top.	0.45	3.02	2.87	2.85	2.81	2.75	2.68	2.73	2.68	2.70	2.75	2.80
	Buf.	6.22	6.76	6.45	6.38	6.12	5.84	5.41	4.90	4.42	4.16	3.85	71.73
501–1000	Top.	0.75	9.19	8.49	8.07	7.95	7.71	7.38	7.45	7.15	7.09	6.88	6.97
	Buf.	6.09	6.92	6.56	6.16	5.98	5.73	5.17	5.11	4.65	4.40	4.09	123.00
> 1000	Top.	0.03	0.05	0.05	0.06	0.07	0.07	0.08	0.09	0.11	0.12	0.15	0.15
	Buf.	0.40	0.44	0.45	0.48	0.52	0.54	0.52	0.58	0.60	0.64	0.69	320.03
Total	Top.	35.65	52.06	50.32	50.37	50.61	49.87	48.28	48.98	48.54	48.53	48.07	50.54
	Buf.	459.36	460.13	442.22	447.42	451.82	444.24	418.92	413.26	403.11	397.52	395.91	19889.51
Total >2	Top.	20.29	36.46	35.24	34.86	34.91	34.40	33.51	33.94	33.41	33.59	33.45	34.39
	Buf.	324.02	323.19	310.73	310.67	312.59	307.56	289.02	277.16	265.26	260.33	259.77	11436.29

Table A.4: Runtime on 45 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	0.61	0.18	0.18	0.17	0.15	0.14	0.12	0.10	0.07	0.06	0.13	0.05
	Max.	39.71	23.20	23.20	23.20	18.58	17.88	15.05	15.05	12.54	13.63	13.08	8.30
2	Avg.	4.44	3.35	3.34	3.27	3.13	2.98	2.83	2.73	2.65	2.53	2.25	1.77
	Max.	48.59	27.91	27.91	27.86	27.86	27.86	27.42	25.50	22.24	20.65	20.74	12.24
3	Avg.	8.40	7.01	6.83	6.49	6.23	5.82	5.52	5.28	5.11	4.85	4.55	3.43
	Max.	56.10	51.36	44.24	34.05	34.05	29.56	30.43	28.03	23.87	24.41	24.95	16.16
4	Avg.	11.41	9.57	9.42	8.93	8.38	7.92	7.53	7.14	6.69	5.91	5.61	4.02
	Max.	56.79	49.70	44.27	37.90	37.35	35.36	34.83	29.98	29.73	23.55	25.17	16.62
5	Avg.	14.06	11.40	11.11	10.59	10.05	9.60	9.26	8.95	8.71	8.44	9.53	6.40
	Max.	57.25	46.03	39.79	34.71	33.58	32.77	32.19	31.28	27.07	29.04	25.12	17.77
6	Avg.	15.70	13.48	13.21	12.02	10.92	10.16	9.36	8.82	8.03	6.86	6.98	4.63
	Max.	61.04	49.21	43.81	42.60	38.29	36.41	30.95	28.05	27.34	26.51	27.99	20.68
7	Avg.	17.08	14.62	14.39	12.92	11.89	10.92	10.04	9.31	8.07	6.68	6.85	4.31
	Max.	77.36	51.35	55.33	52.03	43.64	39.74	35.92	35.31	29.52	28.52	28.21	18.41
8	Avg.	23.26	19.91	19.24	16.71	14.77	13.39	12.39	11.43	10.10	9.30	9.73	6.43
	Max.	68.80	59.97	59.97	59.97	47.92	47.92	33.65	36.05	30.46	31.80	27.91	19.22
9–20	Avg.	35.80	31.12	29.26	26.42	23.18	20.41	18.26	16.04	13.90	11.89	12.05	7.77
	Max.	189.57	137.19	154.90	120.08	81.90	69.51	62.30	52.06	48.98	40.42	39.42	29.89
21–50	Avg.	60.57	50.96	44.91	39.92	33.74	29.95	25.94	22.79	19.47	17.13	17.53	11.02
	Max.	178.05	154.09	134.17	102.68	88.17	79.84	65.38	62.68	53.27	45.74	47.44	36.56
51–100	Avg.	94.56	77.28	66.69	60.80	52.95	46.68	41.26	35.77	33.00	30.30	30.53	20.56
	Max.	206.96	174.81	122.20	108.30	99.33	85.80	85.84	64.86	65.40	52.01	55.24	43.72
101–250	Avg.	220.78	159.80	109.71	104.54	89.92	68.87	60.84	58.81	45.66	39.90	42.28	28.54
	Max.	281.01	181.01	151.52	146.40	124.77	90.89	87.55	77.68	67.84	57.93	54.29	41.42
251–500	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
501–1000	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
> 1000	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
Total	Avg.	4.13	3.21	3.09	2.88	2.65	2.45	2.27	2.12	1.96	1.79	1.80	1.25
	Max.	281.01	181.01	154.90	146.40	124.77	90.89	87.55	77.68	67.84	57.93	55.24	43.72
Total >2	Avg.	15.31	12.92	12.33	11.39	10.43	9.57	8.87	8.24	7.61	6.93	6.94	4.81
	Max.	281.01	181.01	154.90	146.40	124.77	90.89	87.55	77.68	67.84	57.93	55.24	43.72

Table A.5: Worst Slack Deviation on 32 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	Avg.	0.8	0.8	0.8	0.9	0.9	1.0	1.1	1.4	1.8	2.7	4.9	4.8
	Max.	82.8	82.8	82.8	82.8	82.8	69.5	72.6	76.9	83.2	87.4	87.4	87.4
4	Avg.	4.2	3.8	3.4	2.9	2.7	2.6	2.8	2.9	4.0	6.4	11.7	11.6
	Max.	77.8	77.8	77.8	74.6	74.6	74.6	74.1	74.1	72.8	74.3	129.0	129.0
5	Avg.	3.1	2.7	2.7	2.6	2.7	2.6	2.8	3.0	4.0	6.1	11.1	10.5
	Max.	90.8	90.8	90.8	90.8	90.8	90.8	90.8	90.8	90.8	129.0	145.1	145.1
6	Avg.	6.8	6.4	6.2	5.6	5.5	5.6	5.7	6.2	7.5	10.0	15.3	14.9
	Max.	80.4	80.4	80.4	80.4	80.4	80.4	80.4	80.4	86.6	93.4	162.8	162.8
7	Avg.	7.4	6.9	6.7	6.7	6.4	6.3	6.9	8.2	9.5	12.9	20.7	20.2
	Max.	63.8	63.8	63.8	63.8	71.6	71.6	71.6	84.1	100.9	97.0	131.5	129.2
8	Avg.	11.6	10.4	9.3	8.8	9.1	9.1	9.3	10.1	11.7	14.5	22.3	21.4
	Max.	89.9	89.9	89.9	89.9	89.9	89.9	89.9	89.9	89.9	147.3	127.8	127.8
9–20	Avg.	19.4	18.9	18.3	17.7	18.1	17.8	18.3	18.8	19.2	20.7	36.4	35.5
	Max.	90.0	90.0	90.0	90.0	89.0	89.0	89.0	115.9	143.0	163.1	278.1	277.9
21–50	Avg.	23.3	21.6	21.0	20.3	20.3	20.4	21.2	22.3	22.1	25.0	58.4	57.3
	Max.	87.3	86.0	85.7	87.4	115.6	115.1	100.5	102.8	106.6	129.9	476.2	415.9
51–100	Avg.	37.4	33.3	31.5	31.9	31.5	30.7	32.3	33.9	36.3	38.2	112.4	109.6
	Max.	71.0	67.4	69.1	71.4	64.5	70.3	78.4	81.9	94.9	121.6	566.9	564.0
101–250	Avg.	71.7	71.8	65.5	68.2	74.7	72.7	72.4	70.8	73.6	68.9	214.8	207.8
	Max.	97.4	101.2	90.0	91.5	94.3	94.3	92.5	85.4	86.2	93.1	538.3	520.8
251–500	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
501–1000	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
> 1000	Avg.	–	–	–	–	–	–	–	–	–	–	–	–
	Max.	–	–	–	–	–	–	–	–	–	–	–	–
Total	Avg.	5.5	5.1	5.0	4.8	4.9	4.8	5.0	5.3	5.6	6.5	12.9	12.6
	Max.	97.4	101.2	90.8	91.5	115.6	115.1	100.5	115.9	143.0	163.1	566.9	564.0
Total >2	Avg.	11.1	10.5	10.1	9.8	9.9	9.8	10.2	10.7	11.4	13.3	26.4	25.7
	Max.	97.4	101.2	90.8	91.5	115.6	115.1	100.5	115.9	143.0	163.1	566.9	564.0

Table A.6: Length Deviation on 32 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Pwr.	0.397	0.397	0.397	0.397	0.401	0.404	0.411	0.423	0.470	0.637	1.005	0.939
	Rpt.	6557	6557	6557	6557	6557	6557	6557	6557	7055	9091	13057	12995
2	Pwr.	0.251	0.249	0.250	0.252	0.259	0.271	0.290	0.310	0.345	0.439	0.973	0.822
	Rpt.	4471	4420	4420	4441	4507	4655	4900	5110	5447	6425	12185	13818
3	Pwr.	0.355	0.348	0.343	0.351	0.371	0.391	0.424	0.468	0.559	0.735	1.438	1.175
	Rpt.	6272	6083	5928	5888	6146	6337	6637	7002	7629	9076	15642	19003
4	Pwr.	0.122	0.120	0.122	0.126	0.131	0.138	0.148	0.162	0.191	0.273	0.533	0.438
	Rpt.	2081	2050	2059	2093	2130	2203	2286	2423	2687	3513	5945	7366
5	Pwr.	0.139	0.127	0.129	0.133	0.139	0.148	0.158	0.173	0.215	0.315	0.680	0.459
	Rpt.	2127	1857	1851	1881	1937	2000	2072	2166	2489	3330	6327	7323
6	Pwr.	0.065	0.064	0.065	0.070	0.075	0.081	0.090	0.100	0.122	0.186	0.356	0.293
	Rpt.	1107	1087	1089	1121	1159	1203	1283	1357	1563	2178	3692	4972
7	Pwr.	0.050	0.049	0.050	0.053	0.057	0.062	0.067	0.076	0.099	0.150	0.290	0.247
	Rpt.	877	856	848	868	891	924	963	1024	1251	1754	2983	4279
8	Pwr.	0.049	0.049	0.051	0.055	0.059	0.063	0.071	0.082	0.106	0.153	0.279	0.209
	Rpt.	820	796	792	803	819	841	906	967	1174	1600	2728	3495
9–20	Pwr.	0.324	0.327	0.340	0.361	0.394	0.433	0.493	0.596	0.799	1.172	2.181	1.730
	Rpt.	5004	4792	4691	4752	4942	5153	5528	6133	7809	11391	20121	28034
21–50	Pwr.	0.154	0.158	0.166	0.176	0.190	0.204	0.234	0.284	0.364	0.517	1.037	0.776
	Rpt.	2159	2046	2031	2066	2118	2195	2362	2581	3209	4609	9111	11943
51–100	Pwr.	0.047	0.048	0.051	0.053	0.058	0.063	0.077	0.093	0.128	0.181	0.376	0.275
	Rpt.	703	670	654	639	651	668	734	790	959	1358	2793	3834
101–250	Pwr.	0.006	0.007	0.007	0.007	0.008	0.009	0.011	0.014	0.018	0.026	0.065	0.045
	Rpt.	97	97	92	80	91	90	101	107	126	199	486	626
251–500	Pwr.	–	–	–	–	–	–	–	–	–	–	–	–
	Rpt.	–	–	–	–	–	–	–	–	–	–	–	–
501–1000	Pwr.	–	–	–	–	–	–	–	–	–	–	–	–
	Rpt.	–	–	–	–	–	–	–	–	–	–	–	–
> 1000	Pwr.	–	–	–	–	–	–	–	–	–	–	–	–
	Rpt.	–	–	–	–	–	–	–	–	–	–	–	–
Total	Pwr.	1.958	1.943	1.970	2.034	2.142	2.267	2.474	2.780	3.417	4.784	9.214	7.407
	Rpt.	32275	31311	31012	31189	31948	32826	34329	36217	41398	54524	95070	117688
Total >2	Pwr.	1.310	1.297	1.324	1.385	1.482	1.592	1.772	2.048	2.602	3.708	7.237	5.647
	Rpt.	21247	20334	20035	20191	20884	21614	22872	24550	28896	39008	69828	90875

Table A.7: Power Consumption on 32 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Top.	1.61	1.57	1.52	1.57	1.54	1.56	1.53	1.55	1.51	1.55	1.53	1.59
	Buf.	11.55	11.42	11.35	11.43	11.35	11.39	11.38	11.37	11.37	11.40	11.35	88.99
2	Top.	0.50	0.49	0.48	0.49	0.48	0.49	0.48	0.49	0.48	0.49	0.48	0.50
	Buf.	4.65	4.57	4.53	4.58	4.54	4.54	4.54	4.54	4.53	4.56	4.59	43.07
3	Top.	0.54	0.54	0.53	0.54	0.53	0.54	0.53	0.53	0.52	0.54	0.53	0.55
	Buf.	6.08	5.96	5.90	5.95	5.90	5.89	5.90	5.90	5.83	5.70	5.70	57.93
4	Top.	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.17
	Buf.	2.10	2.06	2.04	2.06	2.04	2.04	2.04	2.05	2.00	1.90	1.87	18.94
5	Top.	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.22	0.21	0.22	0.22	0.22
	Buf.	2.98	2.91	2.88	2.91	2.88	2.87	2.88	2.88	2.82	2.73	2.66	30.43
6	Top.	0.10	0.11	0.10	0.11	0.10	0.11	0.10	0.10	0.10	0.10	0.10	0.11
	Buf.	1.60	1.58	1.55	1.57	1.56	1.57	1.57	1.57	1.51	1.41	1.37	13.17
7	Top.	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08	0.08
	Buf.	1.34	1.30	1.29	1.30	1.30	1.31	1.31	1.31	1.26	1.17	1.12	10.21
8	Top.	0.07	0.08	0.07	0.08	0.08	0.08	0.08	0.07	0.07	0.08	0.08	0.08
	Buf.	1.27	1.24	1.21	1.22	1.22	1.22	1.22	1.21	1.15	1.05	1.01	9.93
9–20	Top.	0.51	0.58	0.57	0.58	0.57	0.57	0.56	0.56	0.56	0.57	0.57	0.56
	Buf.	10.21	9.98	9.92	9.94	9.85	9.82	9.80	9.76	9.18	8.30	7.69	70.72
21–50	Top.	0.21	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.30	0.29
	Buf.	4.62	4.52	4.46	4.44	4.41	4.38	4.37	4.30	4.08	3.67	3.33	31.75
51–100	Top.	0.10	0.20	0.20	0.20	0.20	0.20	0.19	0.19	0.19	0.20	0.20	0.20
	Buf.	1.94	1.84	1.82	1.80	1.77	1.73	1.72	1.72	1.68	1.57	1.45	14.21
101–250	Top.	0.02	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05	0.05
	Buf.	0.35	0.34	0.35	0.35	0.35	0.34	0.34	0.34	0.33	0.29	0.22	2.28
251–500	Top.	–	–	–	–	–	–	–	–	–	–	–	–
	Buf.	–	–	–	–	–	–	–	–	–	–	–	–
501–1000	Top.	–	–	–	–	–	–	–	–	–	–	–	–
	Buf.	–	–	–	–	–	–	–	–	–	–	–	–
> 1000	Top.	–	–	–	–	–	–	–	–	–	–	–	–
	Buf.	–	–	–	–	–	–	–	–	–	–	–	–
Total	Top.	4.13	4.38	4.27	4.38	4.30	4.34	4.28	4.30	4.23	4.32	4.28	4.39
	Buf.	48.69	47.73	47.31	47.54	47.17	47.09	47.06	46.95	45.73	43.76	42.36	391.63
Total >2	Top.	2.02	2.32	2.27	2.32	2.28	2.29	2.26	2.27	2.24	2.29	2.27	2.31
	Buf.	32.49	31.74	31.43	31.52	31.28	31.16	31.15	31.04	29.83	27.80	26.42	259.57

Table A.8: Runtime on 32 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	2.48	0.91	0.87	0.82	0.76	0.65	0.50	0.34	0.23	0.21	0.31	0.20
	Max.	211.39	166.31	150.18	126.60	107.42	78.56	60.72	42.44	26.54	25.79	34.23	13.58
2	Avg.	6.66	4.10	3.99	3.77	3.47	3.11	2.73	2.36	2.05	1.85	1.49	1.08
	Max.	164.17	140.04	140.04	121.44	111.86	80.66	60.00	43.06	33.47	28.82	30.01	21.82
3	Avg.	10.98	6.96	6.71	6.17	5.57	4.96	4.40	3.89	3.43	3.01	2.39	1.66
	Max.	150.56	104.37	102.73	89.30	85.36	65.23	46.51	39.75	29.92	26.58	37.42	33.51
4	Avg.	12.46	8.79	8.46	7.87	7.25	6.62	5.94	5.22	4.51	3.81	3.42	2.52
	Max.	163.12	143.07	153.33	131.16	111.72	77.49	69.20	40.53	28.76	26.50	27.58	18.57
5	Avg.	14.06	10.74	10.38	9.63	8.76	7.87	6.96	5.86	4.84	4.03	3.56	2.50
	Max.	242.04	216.13	212.15	173.94	131.61	73.93	64.67	36.06	31.04	23.75	34.98	36.78
6	Avg.	18.55	14.61	14.00	12.87	11.32	9.77	8.53	7.28	5.77	4.31	3.87	2.86
	Max.	160.01	137.93	110.00	96.55	88.34	69.72	55.09	39.48	45.83	38.46	51.63	43.68
7	Avg.	19.10	15.51	15.02	14.06	12.80	11.33	9.77	8.31	6.75	4.93	4.43	3.18
	Max.	139.31	114.57	112.40	104.51	84.36	72.94	55.28	41.17	43.51	39.65	30.13	35.33
8	Avg.	27.90	20.73	19.75	18.26	16.31	14.42	12.71	10.92	9.06	7.50	7.35	5.02
	Max.	458.97	397.18	142.55	131.07	101.33	97.74	72.13	46.03	40.07	33.98	35.73	23.72
9–20	Avg.	36.80	27.60	25.65	22.62	19.74	16.93	14.47	11.91	9.45	7.81	7.85	5.27
	Max.	439.75	415.04	258.53	205.46	199.67	113.66	91.00	60.73	58.98	54.70	61.07	33.85
21–50	Avg.	63.71	49.90	45.36	38.42	31.76	26.84	22.50	18.59	14.85	12.75	13.08	8.82
	Max.	520.35	536.68	289.64	268.05	255.08	135.69	92.44	87.36	72.50	45.16	49.16	40.44
51–100	Avg.	103.29	80.71	70.05	57.46	46.73	38.75	31.97	26.04	20.97	18.19	19.16	12.61
	Max.	489.63	456.71	301.81	238.01	213.71	144.11	117.23	89.07	62.21	54.04	92.22	82.43
101–250	Avg.	159.81	122.84	97.75	74.09	57.90	46.89	36.64	28.72	21.85	18.21	18.52	11.65
	Max.	902.37	1037.46	346.02	216.88	188.42	142.03	127.45	102.30	79.88	74.20	86.98	80.86
251–500	Avg.	225.13	173.18	124.88	90.81	72.34	64.59	54.30	42.48	27.99	18.98	17.78	11.09
	Max.	642.05	543.15	416.40	239.67	222.11	204.07	181.67	159.31	94.92	55.61	52.01	32.02
501–1000	Avg.	58.25	70.75	36.96	16.57	19.44	10.15	8.96	5.70	5.04	5.26	6.31	3.80
	Max.	174.76	212.25	110.89	49.72	58.33	30.46	26.89	17.11	15.13	15.78	18.93	11.41
> 1000	Avg.	396.87	291.70	226.88	172.13	149.11	118.42	101.17	82.19	55.10	40.24	34.87	17.11
	Max.	658.76	523.64	406.77	314.23	219.34	191.18	151.56	134.10	117.96	70.47	69.79	40.18
Total	Avg.	5.74	3.46	3.28	3.00	2.68	2.34	1.98	1.63	1.32	1.12	1.10	0.76
	Max.	902.37	1037.46	416.40	314.23	255.08	204.07	181.67	159.31	117.96	74.20	92.22	82.43
Total >2	Avg.	20.69	15.23	14.30	12.83	11.31	9.88	8.58	7.29	6.01	4.99	4.62	3.20
	Max.	902.37	1037.46	416.40	314.23	255.08	204.07	181.67	159.31	117.96	74.20	92.22	82.43

Table A.9: Worst Slack Deviation on 22 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	Avg.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	Max.	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	Avg.	2.2	2.1	2.0	1.7	1.7	1.7	1.6	1.7	2.0	3.2	9.1	9.1
	Max.	96.8	96.8	96.8	96.8	96.8	96.8	96.8	96.8	96.8	95.9	96.9	96.9
4	Avg.	2.8	2.7	2.6	2.3	2.2	2.2	2.2	2.4	2.9	4.6	9.7	9.6
	Max.	96.1	96.1	96.1	96.1	96.1	96.1	96.1	96.1	107.9	173.7	183.5	183.5
5	Avg.	5.0	4.8	4.6	4.3	4.2	4.1	4.2	4.7	5.8	8.4	16.2	16.1
	Max.	97.3	97.3	97.3	97.3	97.3	97.3	97.3	100.8	123.2	156.2	176.7	176.7
6	Avg.	7.7	7.5	7.3	6.7	6.5	6.1	5.9	6.2	7.2	10.7	17.3	17.2
	Max.	91.9	91.9	91.9	91.9	91.9	91.9	93.3	101.7	139.3	138.0	192.0	192.0
7	Avg.	12.9	12.7	12.4	11.8	11.5	11.3	11.4	11.7	12.6	15.1	22.8	22.5
	Max.	98.4	98.4	95.6	95.6	94.5	100.0	103.8	122.8	122.8	140.6	314.4	296.6
8	Avg.	5.4	5.0	4.9	4.5	4.5	4.3	4.3	4.5	5.6	8.8	18.6	18.5
	Max.	95.0	95.0	101.6	95.0	95.0	95.0	95.0	98.8	117.8	150.9	255.8	255.8
9–20	Avg.	10.5	9.8	9.4	8.9	9.0	8.6	8.5	8.7	9.5	11.9	27.5	27.3
	Max.	97.1	97.1	97.1	97.1	97.1	97.1	104.4	128.5	166.8	200.7	491.0	469.6
21–50	Avg.	19.2	17.5	16.6	15.6	15.8	15.5	15.3	15.7	16.2	19.2	49.4	48.9
	Max.	108.7	108.7	108.7	108.7	111.4	111.4	111.4	119.7	159.7	170.2	946.9	920.3
51–100	Avg.	31.9	29.1	28.1	27.4	28.0	27.3	27.2	28.1	28.9	31.8	79.4	78.4
	Max.	105.3	103.2	100.1	98.1	104.5	112.2	117.4	125.3	161.2	207.4	893.4	882.4
101–250	Avg.	35.2	33.1	32.4	32.1	32.7	32.1	32.3	33.0	32.9	33.2	70.1	69.4
	Max.	122.6	108.9	106.1	102.7	108.7	104.1	127.9	161.8	226.1	377.3	2110.0	2042.0
251–500	Avg.	35.9	35.6	35.5	35.5	35.6	34.8	35.1	34.6	35.7	35.5	40.6	39.7
	Max.	96.8	93.1	92.6	93.1	92.9	91.7	94.1	94.1	91.2	87.8	138.2	133.8
501–1000	Avg.	37.4	37.2	36.7	37.1	36.9	36.7	36.6	36.5	36.6	35.5	35.2	35.2
	Max.	52.6	50.2	45.5	49.7	46.9	45.4	43.8	43.7	44.5	39.5	39.5	39.5
> 1000	Avg.	285.4	283.4	283.6	283.8	284.9	285.0	285.0	285.8	284.9	276.5	283.2	268.5
	Max.	500.0	509.4	501.8	512.8	497.6	509.1	510.3	507.9	516.4	488.0	523.3	499.6
Total	Avg.	4.2	3.9	3.7	3.6	3.6	3.5	3.4	3.5	3.8	4.8	10.5	10.4
	Max.	500.0	509.4	501.8	512.8	497.6	509.1	510.3	507.9	516.4	488.0	2110.0	2042.0
Total >2	Avg.	10.0	9.3	9.0	8.5	8.5	8.3	8.2	8.5	9.2	11.5	25.3	25.1
	Max.	500.0	509.4	501.8	512.8	497.6	509.1	510.3	507.9	516.4	488.0	2110.0	2042.0

Table A.10: Length Deviation on 22 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Pwr.	15.622	15.987	16.408	17.187	18.243	19.218	21.515	24.661	29.614	36.243	46.240	47.151
	Rpt.	317791	322463	328275	337855	351949	373721	413117	479471	573663	694817	860511	980717
2	Pwr.	6.226	6.298	6.422	6.679	7.044	7.358	8.081	8.980	10.458	12.541	19.214	19.646
	Rpt.	132993	133800	135656	139332	145101	152932	165521	183836	209308	243218	360371	455857
3	Pwr.	4.014	4.070	4.177	4.402	4.731	5.172	5.652	6.363	7.574	9.443	15.889	15.333
	Rpt.	98442	98936	100650	104543	110353	117789	127204	139980	160179	191201	302582	384809
4	Pwr.	2.264	2.286	2.339	2.456	2.623	2.800	3.123	3.543	4.338	5.557	8.928	9.200
	Rpt.	51317	51579	52433	54125	56579	59789	65062	72539	86223	107224	161459	224340
5	Pwr.	1.414	1.439	1.467	1.549	1.666	1.794	2.027	2.322	2.834	3.684	6.146	6.624
	Rpt.	32732	32984	33463	34701	36503	38912	42621	48331	57699	72600	113683	169283
6	Pwr.	1.215	1.234	1.277	1.376	1.514	1.650	1.868	2.141	2.682	3.606	6.191	6.549
	Rpt.	28368	28574	29257	30795	33019	35800	39391	44305	54614	72070	116926	172306
7	Pwr.	1.090	1.110	1.140	1.214	1.327	1.457	1.667	1.942	2.458	3.392	6.055	6.657
	Rpt.	26470	26629	27097	28246	30026	32460	36117	40996	50332	68619	115520	179128
8	Pwr.	1.905	1.955	2.030	2.176	2.405	2.602	2.979	3.413	4.206	5.565	9.164	8.788
	Rpt.	36573	36686	37547	39411	42543	45937	50818	57934	70653	90564	142253	204131
9–20	Pwr.	5.073	5.304	5.593	6.131	6.826	7.615	8.719	10.128	12.695	16.867	29.752	28.784
	Rpt.	108515	109091	113001	121988	131892	144057	160540	184148	227408	293898	505200	728076
21–50	Pwr.	1.978	2.098	2.274	2.542	2.924	3.286	3.864	4.627	6.007	8.283	16.388	14.304
	Rpt.	44819	44270	45914	49839	55030	60388	68617	79827	101952	136184	266374	364601
51–100	Pwr.	0.776	0.848	0.936	1.059	1.245	1.431	1.717	2.109	2.778	3.975	8.491	7.711
	Rpt.	19848	19477	20297	22264	24538	27257	31329	36767	46927	64787	138493	205950
101–250	Pwr.	0.300	0.318	0.359	0.414	0.491	0.569	0.692	0.853	1.145	1.676	3.637	3.739
	Rpt.	8232	8146	8591	9426	10350	11537	13187	15429	20089	28618	63291	104768
251–500	Pwr.	0.024	0.025	0.027	0.030	0.035	0.037	0.045	0.053	0.072	0.103	0.202	0.316
	Rpt.	733	723	731	771	825	857	933	1049	1255	1694	3696	8785
501–1000	Pwr.	0.010	0.010	0.011	0.011	0.011	0.011	0.012	0.013	0.014	0.017	0.029	0.096
	Rpt.	275	275	280	285	289	297	305	322	347	405	717	2133
> 1000	Pwr.	0.095	0.104	0.112	0.124	0.137	0.153	0.175	0.201	0.245	0.324	0.532	1.179
	Rpt.	3156	3364	3607	3947	4244	4669	5211	5837	6801	8564	12547	36717
Total	Pwr.	42.006	43.085	44.571	47.350	51.225	55.155	62.136	71.350	87.118	111.274	176.859	176.076
	Rpt.	910264	916997	936799	977528	1033241	1106402	1219973	1390771	1667450	2074463	3163623	4221601
Total >2	Pwr.	20.159	20.800	21.741	23.483	25.938	28.579	32.540	37.708	47.046	62.489	111.404	109.278
	Rpt.	459480	460734	472868	500341	536191	579749	641335	727464	884479	1136428	1942741	2785027

Table A.11: Power Consumption on 22 nm Instances

# Sinks		$\xi = 0.0$	$\xi = 0.1$	$\xi = 0.2$	$\xi = 0.3$	$\xi = 0.4$	$\xi = 0.5$	$\xi = 0.6$	$\xi = 0.7$	$\xi = 0.8$	$\xi = 0.9$	$\xi = 1.0$	DP
1	Top.	35.99	36.02	35.84	35.87	35.26	35.29	35.13	34.94	34.82	34.73	34.61	37.34
	Buf.	303.59	303.84	303.41	303.18	299.56	299.58	300.78	297.72	299.42	298.40	300.83	3035.93
2	Top.	8.20	8.25	8.22	8.20	8.17	8.14	8.12	8.10	8.06	8.06	8.06	8.58
	Buf.	80.87	80.78	80.76	80.66	80.30	80.20	80.54	80.16	80.42	80.65	82.51	985.77
3	Top.	6.02	6.12	6.10	6.09	6.06	6.04	6.02	6.01	5.97	5.97	5.97	6.42
	Buf.	70.55	70.32	70.26	70.15	69.64	69.29	69.29	68.91	67.41	66.47	67.69	803.75
4	Top.	3.34	3.43	3.42	3.42	3.41	3.40	3.39	3.39	3.37	3.37	3.37	3.59
	Buf.	45.26	45.04	44.97	44.77	44.56	44.43	44.51	44.29	42.90	41.77	41.84	473.42
5	Top.	2.38	2.48	2.47	2.47	2.46	2.46	2.45	2.46	2.43	2.44	2.45	2.58
	Buf.	35.47	35.27	35.21	34.99	34.86	34.84	34.95	34.85	33.56	32.53	32.32	336.41
6	Top.	2.33	2.44	2.44	2.44	2.42	2.39	2.39	2.39	2.37	2.37	2.37	2.53
	Buf.	37.12	36.84	36.80	36.65	36.17	35.87	36.00	35.82	34.54	32.73	32.22	331.57
7	Top.	2.50	2.65	2.66	2.66	2.64	2.63	2.63	2.63	2.60	2.61	2.62	2.76
	Buf.	44.34	43.97	43.91	43.50	43.22	43.00	43.11	42.90	41.64	39.68	38.10	351.92
8	Top.	2.70	2.91	2.90	2.90	2.89	2.89	2.88	2.89	2.87	2.87	2.87	3.00
	Buf.	45.05	44.57	44.33	44.01	43.76	43.72	43.75	43.37	42.21	41.18	41.30	445.96
9–20	Top.	10.07	11.54	11.50	11.51	11.38	11.34	11.33	11.33	11.23	11.19	11.26	11.69
	Buf.	188.46	185.45	184.04	181.94	179.29	177.56	176.85	173.80	166.23	159.22	157.32	1602.61
21–50	Top.	5.74	8.23	8.18	8.18	8.06	8.00	8.00	7.96	7.94	7.93	7.98	8.26
	Buf.	122.82	119.84	117.10	114.28	110.98	107.92	106.13	103.14	97.09	91.37	88.10	935.38
51–100	Top.	3.70	7.67	7.59	7.54	7.34	7.22	7.21	7.13	7.09	7.06	7.14	7.47
	Buf.	81.48	79.23	77.52	75.72	72.56	69.99	68.84	67.14	63.91	60.00	56.21	556.79
101–250	Top.	2.16	6.79	6.66	6.57	6.39	6.26	6.23	6.12	6.08	6.06	6.18	6.50
	Buf.	46.44	45.71	44.83	43.68	41.95	40.38	39.48	38.25	36.07	33.49	30.77	280.95
251–500	Top.	0.23	1.17	1.10	1.08	1.04	1.00	0.98	0.96	0.95	0.93	0.95	1.00
	Buf.	3.98	3.97	3.92	3.86	3.71	3.56	3.50	3.41	3.34	3.16	2.84	22.89
501–1000	Top.	0.10	0.33	0.33	0.33	0.33	0.32	0.32	0.32	0.32	0.31	0.32	0.32
	Buf.	0.97	0.96	0.95	0.95	0.93	0.89	0.89	0.88	0.87	0.85	0.80	6.21
> 1000	Top.	0.08	0.40	0.47	0.53	0.58	0.68	0.83	0.96	1.13	1.43	1.84	1.93
	Buf.	1.22	1.23	1.30	1.39	1.44	1.47	1.58	1.70	1.83	1.84	1.94	72.20
Total	Top.	85.54	100.41	99.89	99.77	98.42	98.05	97.91	97.57	97.25	97.32	97.99	103.96
	Buf.	1107.61	1097.04	1089.31	1079.73	1062.92	1052.70	1050.21	1036.34	1011.44	983.35	974.79	10241.77
Total >2	Top.	41.35	56.14	55.83	55.70	55.00	54.62	54.66	54.54	54.36	54.54	55.32	58.04
	Buf.	723.15	712.41	705.14	695.89	683.06	672.93	668.89	658.46	631.59	604.30	591.46	6220.07

Table A.12: Runtime on 22 nm Instances

Summary

Repeaters, inverters and buffers, are the logical gates that dominate modern chip designs. We see designs where up to 50 % of all gates are repeaters.

Repeaters are used during physical design of chips to improve the electrical and timing properties of interconnections. They are added along Steiner trees that connect root gates to sinks, creating *repeater trees*. Their construction became a crucial part of chip design. It has great impact on all other parts, for example, placement and routing.

We first present an extensive version of the Repeater Tree Problem. Our problem formulation encapsulates most of the constraints that have been studied so far. We also consider several aspects for the first time, for example, slew dependent required arrival times at repeater tree sinks. These make our formulation more adequate to the challenges of real-world repeater tree construction.

For creating good repeater trees, one has to take the overall design environment into account. The employed technology, the properties of available repeaters and metal wires, the shape of the chip, the temperature, the voltages, and many other factors highly influence the results of repeater tree construction. To take all this into account, we extensively preprocess the environment to extract parameters for our algorithms. These parameters allow us to quickly and yet quite accurately estimate the timing of a tree before it has even been buffered.

We present an algorithm for Steiner tree creation and prove that our algorithm is able to create timing-efficient as well as cost-efficient trees. Our algorithm is based on a delay model that accurately describes the timing that one can achieve after repeater insertion. This makes our algorithm suitable for creating good Steiner trees, the input for subsequent repeater insertion algorithms.

Next, we deal with the problem of adding repeaters to a given Steiner tree. The predominantly used algorithms to solve this problem use dynamic programming. However, they have several drawbacks. Firstly, potential repeater positions along the Steiner tree have to be chosen upfront. Secondly, the algorithms strictly follow the given Steiner tree and miss optimization opportunities. Finally, dynamic programming causes high running times. We present our new buffer insertion algorithm, Fast Buffering, that overcomes these limitations. It is able to produce results with similar quality to a dynamic programming approach but a much better running time. In addition, we also present improvements to the dynamic programming approach that allows us to push the quality at the expense of a high running time.

We have implemented our algorithms as part of the BonnTools physical design optimization suite developed at the Research Institute for Discrete Mathematics in

cooperation with IBM. Our algorithms are used and help engineers dealing with some of the most complex chips in the world. When we released the first version of our global optimization tools, for the first time it became possible to optimize chips with several million gates within reasonable running times. At the same time, designers were able to achieve compelling results. Our tools are not only used for global optimization in early design stages. For later stages of physical design, a more accurate version of our algorithm can be enabled that is able to squeeze out the last tenth of a picosecond.

Our implementation deals with all tedious details of a grown real-world chip optimization environment. At the same time, we offer a clean framework abstracting away the details such that new repeater tree construction algorithms can easily be implemented.

As a side project, we implemented a blockage map that helps managing the free/blocked information not only for our algorithms but also other optimization tools within BonnTools. Recently, the congestion map that we implemented has been added as a fast mode to BonnRouteGlobal, the global routing engine used throughout the whole IBM physical design.

We have created extensive experimental results on challenging real-world test cases provided by our cooperation partner. The testbed consists of more than 3.3 million different repeater tree instances. The average running time for a single instance is about 0.6 milliseconds, which means that we can solve about 5.7 million instances per hour. We also compare our implementation to an state-of-the-art industrial tool and show that our algorithm produces better results with less electrical violations.

B Bibliography

Noga Alon and Yossi Azar. On-Line Steiner Trees in the Euclidean Plane. *Discrete & Computational Geometry*, 10:113–121, 1993. doi: 10.1007/BF02573969.

Charles J. Alpert and Anirudh Devgan. Wire Segmenting for Improved Buffer Insertion. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 588–593, New York, NY, USA, 1997. ACM. doi: 10.1145/266021.266291.

Charles J. Alpert, Anirudh Devgan, and Stephen T. Quay. Buffer insertion with accurate gate and interconnect delay computation. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 479–484, 1999. doi: 10.1145/309847.309983.

Charles J. Alpert, R. Gopal Gandham, Jose L. Neves, and Stephen T. Quay. Buffer Library Selection. In *Proceedings of the International Conference on Computer Design*, pages 221–226, Los Alamitos, CA, USA, 2000. IEEE Computer Society. doi: 10.1109/ICCD.2000.878289.

Charles J. Alpert, Anirudh Devgan, John P. Fishburn, and Stephen T. Quay. Interconnect Synthesis Without Wire Tapering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(1):90–104, 2001a. doi: 10.1109/43.905678.

Charles J. Alpert, Gopal Gandham, Jiang Hu, Jose L. Neves, Stephen T. Quay, and Sachin S. Sapatnekar. Steiner Tree Optimization for Buffers, Blockages, and Bays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(4):556–562, 2001b. doi: 10.1109/43.918213.

Charles J. Alpert, Gopal Gandham, Miloš Hrkić, Jiang Hu, Andrew B. Kahng, John Lillis, Bao Liu, Stephen T. Quay, Sachin S. Sapatnekar, and A. J. Sullivan. Buffered Steiner Trees for Difficult Instances. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(1):3–14, 2002. doi: 10.1109/TCAD.2005.858348.

Charles J. Alpert, Gopal Gandham, Miloš Hrkić, Jiang Hu, Stephen T. Quay, and C. N. Sze. Porosity-Aware Buffered Steiner Tree Construction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(4):517–526, 2004a. doi: 10.1109/TCAD.2004.825864.

Charles J. Alpert, Miloš Hrkić, and Stephen T. Quay. A Fast Algorithm for Identifying Good Buffer Insertion Candidate Locations. In *Proceedings of the*

2004 International Symposium on Physical design, ISPD '04, pages 47–52, New York, NY, USA, 2004b. ACM. doi: 10.1145/981066.981076.

Charles J. Alpert, Dinesh P. Mehta, and Sachin S. Sapatnekar, editors. *Handbook of Algorithms for Physical Design Automation.* Auerbach Publications, Boston, MA, USA, 1st edition, 2008. ISBN 9780849372421.

Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. Efficient Generation of Short and Fast Repeater Tree Topologies. In *Proceedings of the 2006 International Symposium on Physical Design*, ISPD '06, pages 120–127, New York, NY, USA, 2006. ACM. doi: 10.1145/1123008.1123032.

Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. Efficient algorithms for short and fast repeater trees. I. Topology generation. Technical Report No. 07977, Research Institute for Discrete Mathematics, University of Bonn, 2007a.

Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. Efficient algorithms for short and fast repeater trees. II. Buffering. Technical Report No. 07978, Research Institute for Discrete Mathematics, University of Bonn, 2007b.

Christoph Bartoschek, Stephan Held, Dieter Rautenbach, and Jens Vygen. Fast Buffering for Optimizing Worst Slack and Resource Consumption in Repeater Trees. In *Proceedings of the 2009 International Symposium on Physical Design*, ISPD '09, pages 43–50, New York, NY, USA, 2009. ACM. doi: 10.1145/1514932.1514942.

Christoph Bartoschek, Stephan Held, Jens Maßberg, Dieter Rautenbach, and Jens Vygen. The repeater tree construction problem. *Information Processing Letters*, 110(24):1079–1083, 2010. doi: 10.1016/j.ipl.2010.08.016.

Chung-Ping Chen and Noel Menezes. Noise-aware Repeater Isertion and Wire Sizing for On-chip Interconnect Using Hierarchical Moment-Matching. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 502–506, 1999. doi: 10.1145/309847.309987.

Jason Cong and Xin Yuan. Routing Tee Construction Under Fixed Buffer Locations. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 379–384, New York, NY, USA, 2000. ACM. doi: 10.1145/337292.337502.

Jason Cong, Lei He, Cheng-Kok Koh, and Patrick H. Madden. Performance Optimization of Vlsi Interconnect Layout. *Integration, the VLSI Journal*, 21:1–94, 1996. doi: 10.1016/S0167-9260(96)00008-9.

Sampath Dechu, Cien Shen, and Chris Chu. An Efficient Routing Tree Construction Algorithm With Buffer Insertion, Wire Sizing, and Obstacle Considerations. *IEEE*

Transactions on Computer-Aided Design of Integrated Circuits and Systems, 24 (4):600–608, 2005. doi: 10.1109/TCAD.2005.844107.

William C. Elmore. The Transient Response of Damped Linear Networks with Particular Regard to Wideband Amplifiers. *Journal of Applied Physics*, 19(1): 55–63, 1948. doi: 10.1063/1.1697872.

Delbert R. Fulkerson. A Network Flow Computation for Project Cost Curves. *Management Science*, 7(2):167–178, 1961.

Michael R. Garey and David S. Johnson. The Rectilinear Steiner Tree Problem is NP-Complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977. doi: 10.1137/0132071.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Michael Gester, Dirk Müller, Tim Nieberg, Christian Panten, Christian Schulte, and Jens Vygen. BonnRoute: Algorithms and data structures for fast and good VLSI routing. Technical Report No. 111039, Research Institute for Discrete Mathematics, University of Bonn, 2011.

Nir Halman, Chung-Lun Li, and David Simchi-Levi. Fully polynomial time approximation schemes for time-cost tradeoff problems in series-parallel project networks. In *Proceedings of the 11th international workshop, APPROX 2008, and 12th international workshop, RANDOM 2008 on Approximation, Randomization and Combinatorial Optimization: Algorithms and Techniques*, APPROX '08 / RANDOM '08, pages 91–103, Berlin, Heidelberg, 2008. Springer-Verlag. doi: 10.1007/978-3-540-85363-3_8.

Stephan Held. *Timing Closure in Chip Design.* PhD thesis, University of Bonn, 2008.

Stephan Held and Daniel Rotter. Shallow-Light Steiner Arborescences with Vertex Delays. In *IPCO*, pages 229–241, 2013. doi: 10.1007/978-3-642-36694-9_20.

Stephan Held and Sophie Theresa Spirkl. A Fast Algorithm for Rectilinear Steiner Trees with Length Restrictions on Obstacles. In *Proceedings of the 2014 International Symposium on Physical Design*, ISPD '14, pages 37–44, New York, NY, USA, 2014. ACM. doi: 10.1145/2560519.2560529.

Stephan Held, Bernhard Korte, Dieter Rautenbach, and Jens Vygen. Combinatorial Optimization in VLSI Design. In Vasek Chvátal, editor, *Combinatorial Optimization: Methods and Applications*, volume 31 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 33–96. IOS Press, 2011. doi: 10.3233/978-1-60750-718-5-33.

Renato F. Hentschke, Jagannathan Narasimham, Marcelo O. Johann, and Ricardo L. Reis. Maze Routing Steiner Trees with Effective Critical Sink Optimization. In *Proceedings of the 2007 International Symposium on Physical Design*, ISPD '07, pages 135–142, New York, NY, USA, 2007. ACM. doi: 10.1145/1231996.1232024.

Robert B. Hitchcock, Gordon L. Smith, and David D. Cheng. Timing Analysis of Computer Hardware. *IBM Journal of Research and Development*, 26(1):100–105, 1982. doi: 10.1147/rd.261.0100.

Miloš Hrkić and John Lillis. S-Tree: A Technique for Buffered Routing Tree Synthesis. In *Proceedings of the 36th ACM/IEEE Annual Design Automation Conference*, pages 578–583, 2002. doi: 10.1145/513918.514066.

Miloš Hrkić and John Lillis. Buffer Tree Synthesis With Consideration of Temporal Locality, Sink Polarity Requirements, Solution Cost, Congestion, and Blockages. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(4):481–491, 2003. doi: 10.1109/TCAD.2003.809648.

Jiang Hu, Charles J. Alpert, Stephen T. Quay, and Gopal Gandham. Buffer Insertion With Adaptive Blockage Avoidance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(4):492–498, 2003. doi: 10.1109/TCAD.2003.809647.

Shiyan Hu, Charles J. Alpert, Jiang Hu, S.K. Karandikar, Zhuo Li, Weiping Shi, and C.N. Sze. Fast Algorithms for Slew-Constrained Minimum Cost Buffering. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(11):2009–2022, 2007. doi: 10.1109/TCAD.2007.906477.

Shiyan Hu, Zhuo Li, and Charles J. Alpert. A fully polynomial time approximation scheme for timing driven minimum cost buffer insertion. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 424–429, New York, NY, USA, 2009. ACM. doi: 10.1145/1629911.1630026.

Tao Huang and Evangeline F. Y. Young. Construction of rectilinear steiner minimum trees with slew constraints over obstacles. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design 2012*, ICCAD '12, pages 144–151, New York, NY, USA, 2012. ACM. doi: 10.1145/2429384.2429411.

Frank Hwang. On steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics*, 30(1):104–114, 1976. doi: 10.1137/0130013.

Maxim Janzen. *Buffer Aware Global Routing im Chip Design.* Diplomarbeit, University of Bonn, 2012.

James E. Kelley, Jr. Critical Path Planning and Scheduling: Mathematical Basis. *Operations Research*, 9:296–320, 1961.

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, New York, NY, USA, 1959. ACM. doi: 10.1145/1460299.1460318.

Bernhard Korte and Jens Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2012. ISBN 9783642244872.

Bernhard Korte, Dieter Rautenbach, and Jens Vygen. Bonntools: Mathematical Innovation for Layout and Timing Closure of Systems on a Chip. *Proceedings of the IEEE*, 95(3):555–572, 2007. doi: 10.1109/JPROC.2006.889373.

Leon G. Kraft, Jr. A Device for Quantizing, Grouping, and Coding Amplitude-Modulated Pulses. Master's thesis, Dept. of Electrical Engineering, M.I.T., Cambridge, Massachusets, 1949.

Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Dover Books on Mathematics Series. Dover Publications, 2001. ISBN 9780486414539.

Zhuo Li and Weiping Shi. An $O(bn^2)$ Time Algorithm for Optimal Buffer Insertion With b Buffer Types. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(3):484–489, 2006. doi: 10.1109/TCAD.2005.854631.

Zhuo Li, Ying Zhou, and Weiping Shi. $o(mn)$ Time Algorithm for Optimal Buffer Insertion of Nets With m Sinks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(3):437–441, 2012. doi: 10.1109/TCAD.2011.2174639.

John Lillis, Chung-Kuan Cheng, and Ting-Ting Y. Lin. Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model. *IEEE Journal of Solid-State Circuits*, 31(3):437–447, 1996a. doi: 10.1109/4.494206.

John Lillis, Chung-Kuan Cheng, Ting-Ting Y. Lin, and Ching-Yen Ho. New Performance Driven Routing Techniques With Explicit Area/Delay Tradeoff and Simultaneous Wire Sizing. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC '96, pages 395–400, New York, NY, USA, 1996b. ACM. doi: 10.1145/240518.240594.

Jens Maßberg and Jens Vygen. Approximation algorithms for a facility location problem with service capacities. *ACM Transactions on Algorithms*, 4(4):50:1–50:15, 2008. doi: 10.1145/1383369.1383381.

Dirk Müller. *Fast Resource Sharing in VLSI Routing*. PhD thesis, University of Bonn, 2009.

Dirk Müller, Klaus Radke, and Jens Vygen. Faster min-max resource sharing in theory and practice. *Mathematical Programming Computation*, 3:1–35, 2011. doi: 10.1007/s12532-011-0023-y.

Matthias Müller-Hannemann and Ute Zimmermann. Slack Optimization of Timing-Critical Nets. In *Algorithms - ESA 2003*, volume 2832 of *Lecture Notes in Computer Science*, pages 727–739. Springer Berlin Heidelberg, 2003. doi: 10.1007/978-3-540-39658-1_65.

Takumi Okamoto and Jason Cong. Buffered Steiner Tree Construction with Wire Sizing for Interconnect Layout Optimization. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, pages 44–49, Washington, DC, USA, 1996. IEEE Computer Society. doi: 10.1109/ICCAD.1996.568938.

Carlos A. S. Oliveira and Panos M. Pardalos. A Survey of Combinatorial Optimization Problems in Multicast Routing. *Computers & Operations Research*, 32(8): 1953–1981, 2005. doi: 10.1016/j.cor.2003.12.007.

James B. Orlin. A Faster Strongly Polynomial Minimum Cost Flow Algorithm. *Operations Research*, 41(2):338–350, 1993. doi: 10.1287/opre.41.2.338.

Min Pan, Chris Chu, and Priyadarshan Patra. A novel performance-driven topology design algorithm. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pages 244–249, Washington, DC, USA, 2007. IEEE Computer Society. doi: 10.1109/ASPDAC.2007.357993.

Lawrence Pileggi. Coping with RC(L) Interconnect Design Headaches. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '95, pages 246–253, Washington, DC, USA, 1995. IEEE Computer Society. doi: 10.1109/ICCAD.1995.480019.

Jorge Rubinstein, Jr. Paul Penfield, and Mark A. Horowitz. Signal Delay in *rc* Tree Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(3):202–211, 1983. doi: 10.1109/TCAD.1983.1270037.

Sachin S. Sapatnekar. *Timing*. Kluwer, 2004. ISBN 9781402076718.

Prashant Saxena, Noel Menezes, Pasquale Cocchini, and Desmond A. Kirkpatrick. The Scaling Challenge: Can Correct-by-Construction Design Help? In *Proceedings of the 2003 International Symposium on Physical Design*, ISPD '03, pages 51–58, New York, NY, USA, 2003. ACM. doi: 10.1145/640000.640014.

Weiping Shi and Zhuo Li. A Fast Algorithm for Optimal Buffer Insertion. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 24 (6):879–891, 2005. doi: 10.1109/TCAD.2005.847942.

Weiping Shi, Zhuo Li, and Charles J. Alpert. Complexity Analysis and Speedup Techniques for Optimal Buffer Insertion with Minimum Cost. In *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ASP-DAC '04, pages 609–614, 2004. doi: 10.1109/ASPDAC.2004.1337664.

Martin Skutella. Approximation Algorithms for the Discrete Time-Cost Tradeoff Problem. *Mathematics of Operations Research*, 23:909—-929, 1998.

Lukas P.P.P. van Ginneken. Buffer Placement in Distributed RC-Tree Networks for Minimal Elmore Delay. In *Proceedings of the 1990 IEEE International Symposium on Circuits and Systems*, volume 2, pages 865–868, 1990. doi: 10.1109/ISCAS.1990.112223.

Jens Vygen. Slack in Static Timing Analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(9):1876–1885, 2006. doi: 10.1109/TCAD.2005.858348.

Jürgen Werber. *Logic Restructuring for Timing Optimization in VLSI Design.* PhD thesis, University of Bonn, 2007.

Jürgen Werber, Dieter Rautenbach, and Christian Szegedy. Timing Optimization by Restructuring Long Combinatorial Paths. In *Proceedings of the 2007 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '07, pages 536–543, Piscataway, NJ, USA, 2007. IEEE Press. doi: 10.1109/ICCAD.2007.4397320.

Yilin Zhang and David Z. Pan. Timing-driven, over-the-block rectilinear steiner tree construction with pre-buffering and slew constraints. In *Proceedings of the 2014 International Symposium on Physical Design*, ISPD '14, pages 29–36, New York, NY, USA, 2014. ACM. doi: 10.1145/2560519.2560533.

Yilin Zhang, Ashutosh Chakraborty, Salim Chowdhury, and David Z. Pan. Reclaiming Over-the-IP-Block Routing Resources With Buffering-Aware Rectilinear Steiner Minimum Tree Construction. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design 2012*, ICCAD '12, pages 137–143, New York, NY, USA, 2012. ACM. doi: 10.1145/2429384.2429410.