Discrete Gate Sizing

John Lee\textsuperscript{1} and Puneet Gupta\textsuperscript{2}

\textsuperscript{1} Los Angeles, CA90095, Country, lee@ee.ucla.edu
\textsuperscript{2} Los Angeles, CA90095, USA, puneet@ee.ucla.edu

Abstract

Gate sizing methods are reviewed.
Contents

1 Introduction
  1.1 Other types of sizing problems 1
  1.2 Gate sizing and the physical design process 4
  1.3 The Standard Cell Library 7
  1.4 The gate sizing problem 9
  1.5 Notations and Acronyms 10

2 Static Timing Analysis
  2.1 Concepts and terminology 12
  2.2 Computing arrival times and slacks 16
  2.3 Interconnect delay 20
  2.4 Gate delay models 22
  2.5 Slew propagation 27
  2.6 Power and clock domain considerations 28
  2.7 Additional considerations 29
  2.8 Block-based delay formulation 29
  2.9 Difference between STA timers 31
Gate sizing is one of the most popular methods for optimizing digital circuits. It is very flexible as it can be used to manage trade-offs in power, timing, area, yield, crosstalk, statistical power, statistical delay and soft-errors. It can also be used incrementally, and is an ideal method for optimizing post-layout designs where the placement and interconnect have already been settled. After over three decades, research is still active in the area, as improved algorithms emerge and new applications are found.

This work will consider the case of gate sizing for standard library cell designs. In these designs, the gates are chosen from a library of precharacterized gates that act as its fundamental building blocks. These cell-based designs compose the majority of the digital designs today.

1.1 Other types of sizing problems

There are other variants of gate sizing that are not covered in this work:

---

1 Hereafter, the term gate sizing will refer to not only discrete gate sizing but $V_t$ assignment and gate-length biasing problems.
(1) Transistor sizing for analog design
(2) Transistor sizing for custom digital design

These variants are a minority of IC designs. Custom digital design is mainly limited to high-performance designs. However, analog designs are becoming increasingly important with the increase in systems on a chip (SoC) methodologies that integrate entire TV’s or radios on a single die [63].

1.1.1 Transistor sizing for analog designs

In analog design, designers must satisfy many different constraints and performance specifications, such as gain, accuracy, linearity, signal-to-noise, and impedance matching. For example, the pair of transistors forming a current mirror or differential pair must be matched, or have very similar electrical characteristics. Also, transistors that function as voltage-controlled resistors must be operating in the linear region, and an amplifier must have the proper signal-to-noise ratio for the system to work properly.

These constraints are related to the analog nature of the design. While standard cells mask much of the underlying electrical waveforms using logic states, analog designs work by utilizing these underlying characteristics to produce amplifiers, digital-to-analog converters, current sources, etc. This also means that the sizing methods must control many more facets of the design for it to work properly.

The main challenge in automating analog sizing is to input the design specifications and models into a form that can be used by the sizing method. This is a challenge because the range of analog designs is large. For instance, while the analytical performance models for a given op-amp topology might be well known, it is difficult to write down the equations that govern the sizing of an arbitrary design.

Early methods were knowledge-based which synthesized designs using templates [52, 71, 55]. As the templates are precharacterized, they would also carry information on a good initial sizing and how to optimize the given template. Sizing these designs was therefore equivalent to executing the design plans. In [52], the sizing proceeds by determining the bias current, then the $W/L$ ratios, followed by the $1/f$ noise
consideration and then finally the W and L of each device. In [71], the values are chosen using a fixed point method – the parameters are determined serially, where each parameter is chosen to best satisfy the design considerations, assuming the other parameters to be fixed.

The time required to construct these templates, however, was often much greater than the time needed to design the circuit directly [63]. The accumulation of the knowledge bases, and the codification of the expert knowledge was not practical. This, coupled with the limited range of circuits that the method could handle, led to the decline of these types of methods. However, there is recent interest in automating the knowledge-extraction process [104], and in identifying substructures in a design automatically [103].

Another branch of analog sizing is the optimization-based methods. These methods use optimization procedures, rather than codified design rules to size the design. The first subclass of optimization methods consists of equation based methods [85, 64, 74] that rely on the designer to provide the equations, but in contrast to the knowledge-based methods, the sizing process is automated using optimization methods, rather than rules. The optimization process may use simulated annealing [64], steepest-descent [85], or convex optimization [74]. This sub-class works well in certain contexts (such as in [74]), but they may be limited by their accuracy.

The second subclass of optimization based methods are the simulation based methods [110, 58, 54, 111, 115, 86] that use numerical simulations to measure the performance. The simulations provide these methods with greater accuracy, however they create a large overhead that makes these methods much slower than their equation-based counterparts.

### 1.1.2 Transistor sizing for custom digital designs

In the custom digital setting, every transistor in the design is available for optimization [45, 44, ?, 80, 9, 140]. Many of the early papers on gate sizing were based on transistor-level sizing (see [123, 60]), as the

---

2 See [63] for the taxonomy of analog sizing methods and a comprehensive review of methods prior to 2000.
standard cell abstraction was not yet in widespread usage. However, with the increasing complexity of designs, standard cell libraries have become almost universally used.

Custom digital design techniques are primarily used for high-performance parts of high-volume designs to recover the increased cost of designing at a transistor level. An example of this is the case of microprocessors [14]. However, these methods account for a minority of the digital designs today. Custom digital transistor sizing is still an active area of research today. Most current research is directed towards the statistical design of custom circuits [134, 43, 10].

1.2 Gate sizing and the physical design process

Gate sizing is a part of the larger Electronic Design Automation (EDA) ecosystem that transforms Register Transfer Language (RTL) descriptions into a physical layout. The process of creating the physical layout is appropriately called the physical design process, and consists of six steps:

1. **Logic synthesis.**
   - Input: RTL/HDL design description, library cell information, timing constraint information.
   - Output: Netlist mapped to the library.
   
   Transform the RTL/HDL design description into a gate level netlist, using a given cell library. Convert state machines, map arithmetic blocks, etc.

2. **Floorplanning.**
   - Input: Synthesized netlist, macro information, library information, standard cell row information, information on chip inputs and outputs.
   - Output: Floorplan with rows for standard cell placement, pads for input and output, locations for macros.

See, for example, [144].
Create a die for the design, and allocate space for input output ports, macros, and library gates.

(3) **Placement.**
- Input: Synthesized netlist, floorplan.
- Output: Locations for each of the cells in the design in a standard cell row.

Places, flips and rotates the cells into the rows created by the floorplanning algorithm. Minimizes the wirelength in the resulting layout, while meeting timing constraints (timing-driven placement).

(4) **Clock-Tree synthesis.**
- Input: Placed design, clock nets.
- Output: Clock tree to distribute the clock signal to the sequential elements (flip-flops, latches, etc.).

Creates a clock tree to distribute the clock signal across the design. The goals are to minimize the size of the clock tree (to minimize power), and the skew at each of the outputs of the clock tree.

(5) **Routing.**
- Input: Placed design, connection information, metal and via information from the library.
- Output: Design with cells connected with wires.

Connects cells in the library using metal layers and vias. The objective is to minimize the amount of interconnect needed, while observing design rules and meeting timing.

(6) **Physical Verification and Design for Manufacturability.**
- Input: Placed, routed design.
- Output: Design with improved yield.

Improves the yield of the design. Corrects design rules, inserts metal fill, doubles vias, checks connectivity.

A flowchart for the EDA process is shown in Figure [1.1].
Although gate sizing is not explicitly in the design flow above, it is used throughout the design flow to correct timing errors, and to optimize the design. For example, it is commonly used after placement to resize gates that violate maximum fanout rules or flip-flop setup time requirements. After clock tree synthesis, it is used to further fix rule violations and setup and hold violations, using the clock information from the clock-tree synthesis. After routing, it is again used to fix setup and hold violations, along with design rule violations. At this step, an incremental routing may be needed to account for changes in the cell sizes, and their corresponding pin locations.

It is also be used at different points in the design flow to reduce the power consumption. This optimization may be able to reduce the power \textit{with the same timing constraint}, but if the design is already heavily optimized, this will require the timing constraints to be relaxed, to trade-off the delay for power.

Gate sizing is a powerful tool for optimization, and is the most widely used incremental optimization tool. It is more powerful than adjusting the placement or routing of the design, and it is also less intrusive. For example, fixing setup time violations using placement
would be very intrusive – the gates on the violating path would need to be moved to decrease the resulting interconnects. Similarly, fixing setup time violations using routing would also require the connections between the cells to be rerouted. Generally, a significant portion of the design will need to be rerouted to provide benefits.

On the other hand, gate sizing is less disruptive. For example, the size of a buffer driving a large wire load could be increased to improve the timing. This may result in a local rerouting, to accommodate the different pin locations of the larger cell and if there is no space around the surrounding cell, then an incremental placement will be needed to create space for the cell. However, this is preferable to rerouting large sections of the design, or adjusting the placements of tens or hundreds of cells.

In some cases where only the gate lengths of threshold voltages change, the disruption is very minimal. In these cases, the cell dimensions and pin locations are the same. Thus, these cell alternatives can be swapped without any change in the routing or the placement. The only verification needed is to ensure that the crosstalk noise, power, and timing constraints are satisfied.

Another advantage of gate sizing is that it is versatile – it can be targeted to different optimization objectives. It has been used for power and timing optimization [46], to fix noise constraints due to crosstalk [94 154], to harden soft-errors due to radiation [158], to improve yield [50 87], and minimize statistical power [139 42].

1.3 The Standard Cell Library

The standard cell library contains logic cells such as inverters, ands, not-ands, and x-ors that implement basic logical functions. There are also sets of sequential cells such as flip-flops and latches, with variants that enable setting, resetting and reading in scan-chains. These cells provide memory, allowing pipe-lining, state machines, and a memory of past computations. Lastly, there are utility cells, such as filler cells, antenna cells, and buffer cells, which provide tools to help with the physical implementation of the design.

The library generally provides several gate options for each logic
function. Each of the options are logically equivalent, but have varying electrical characteristics, due to differences in their gate lengths, widths, PMOS-NMOS width ratios and threshold voltages $V_t$. These families provide alternative gates that can be used to optimize the design. For example, critical paths can be sped up by swapping high-$V_t$ cells by low-$V_t$ cells, and gates with fanout violations can be fixed using alternatives with larger-transistor widths and smaller effective resistances at the gate outputs.

Two library files that are used for sizing are:

1. Physical library information (usually expressed in Library Exchange Format (LEF)):
   - Cell information – dimensions of the cell and locations of the pins.
   - Interconnect information – dimensions, pitch, capacitance, and resistances for each metal layer.
   - Via information – dimensions, resistance, and layers that are connected.

2. Timing library information (usually expressed in Liberty Format):
   - Library characterization information – temperature, voltage and process.
   - Parameters used in the library – slew thresholds, input thresholds, output thresholds, and measurement units.
   - Cell information – delays, area, logic function, short-circuit power, switching power, leakage power. For flip-flops the hold and setup time requirements are also given.
   - Cell pin information – capacitances, maximum loads.

The geometry information in the LEF file is used for placement and routing. This information tells the program how to create standard cell rows for floorplanning, and the dimensions of each cell for placement.
Next, the pin locations, interconnect geometries, and via dimensions are used for routing, and once routing is complete, the capacitance and resistance information is used to extract information about the wire parasitics.

The timing and power information from the Liberty file is used for the timing and power analysis of the design. The timing information is used in conjunction with the wire parasitic information to create delay estimates and power estimates. This will be covered in more detail in Section 2.4.

### 1.4 The gate sizing problem

There are many variations on the gate sizing problem that are tailored to different goals. In this work, the following metrics are considered:

- Leakage power
- Dynamic power
- Clock period

The most common way to use these metrics is to minimize a combination of the leakage power and the dynamic power, with a constraint on the clock period. When timing closure is important, the objective might be to minimize the clock period, and in post-layout situations, a common optimization is to minimize noise and crosstalk violations.

The variables in the optimization process are the cells used to implement the gates. These cells can be swapped to decrease the delay or power, or modify the output signal waveform. The cells may be different sizes, and have different pin locations.

Equivalent cells that can be interchanged may be identified by the designer, or by the library by having the same footprint. For example, the inverter-type cells will have an “inverter” footprint, which identifies the family of cells that can be used to replace the gate. Each of these cells performs the same logic function, thus changing the cells does not affect the functionality of the design.

Formally, the problem may be written as a optimization problem. For example, with the vector of cell options $\vec{\omega}$, the delay-constrained
power optimization problem is:

\[
\begin{align*}
\text{minimize} & \quad \text{Power}(\vec{\omega}) \\
\text{subject to} & \quad \text{Delay}(\vec{\omega}) \leq \text{Delay}_{\text{max}}
\end{align*}
\] (1.1)

This form helps to summarize the objectives and constraints in the optimization process.

The delay is estimated using a Static Timing Analysis method (see Chapter ??) and timing constraints are usually expressed in a Synopsys Design Constraint (SDC) format. These constraints can be very complex, with constraints on the false paths, multicycle paths, multiple clock and power domain. In additions, parasitic information in a Standard Parasitic Exchange (SPEF) format is used to approximate the delay in the interconnect.

### 1.5 Notations and Acronyms

While new notation is avoided when possible, notation for certain key concepts are unavoidable. The notation for key symbols used in this paper are summarized in Table 1.1.
Gate sizing methods that optimize timing or use timing constraints rely on Static Timing Analysis (STA) based timers. Static timing analysis computes the delays and arrival times in the graph without requiring data inputs or simulations. This is in contrast to dynamic timing methods which require simulation vectors to find critical paths. In this chapter, the a broad overview of STA will be covered. Please refer to [125] for a comprehensive treatment of the subject, and [16] for a practical introduction.

The principle behind STA is to propagate the best- and worst-case delays through the circuit. The arrival times at each node are computed by traversing the circuit in topological order, starting with the primary inputs, and propagating the delays through to the primary outputs. The slacks are computed in reverse-topological order, starting with the required arrival times at each of the outputs and propagating these times through to the inputs. In sequential circuits, the inputs of the sequential elements are also considered to be primary outputs and the outputs are considered primary inputs.

When setup or long-path violations are of interest, the STA propagates the worst-case times through the graph to find the critical, or
max path. This is to find whether the signal will arrive at its destination by the required arrival time.

Similarly, when hold or short-path violations are of interest, the STA propagates the best-case times through the graph to find the shortest, or min path. This is used to find whether the signal arrives too fast at the input of a flip-flop or latch, causing the input state of the flip-flop to be set improperly, causing metastability at the output.

2.1 Concepts and terminology

Describing the properties of the circuit and its signal and delay characteristics requires some terminology. The first set of terms are used to describe the topology of the circuit. These terms are used to identify the inputs and the outputs of the circuit, the neighboring gates, and an ordering of the gates that can be used for performing the timing analysis.

- **Primary inputs** (PI)– input ports in the design that are driven by external sources.
- **Primary outputs** (PO)– output ports in the design that drive external loads.
- **Fanins of a gate** (∫(g))– the set of gates that drive the inputs of the given gate.
- **Fanouts of a gate** (∫o(g))– the set of gates that are driven by the given gate.
- **Fanin cone of a gate** – the set of gates whose outputs have a path to the input of the given gate. This includes the fanins of the gate, the fanins of the fanins, and so on.
- **Fanout cone of a gate** – the set of gates whose inputs have a path from the output of the given gate. This includes the fanouts of the gate, the fanouts of the fanouts, and so on.
- **Topological order** – a way of ordering, or numbering the gates where the fanins of a gate have a number smaller than the fanouts of a gate. When sequential elements are present, the sequential elements are placed at the top of the list. This ordering is generally not unique.
For example, in Figure 2.2 the primary inputs are \{IN[0], IN[1] \}. The primary outputs are \{OUT[0]\}. The fanins of gate G2 are \{G1\} and the fanouts of G3 are \{G2\}. In this example, a topological ordering is \{G1,G3,G2\}.

The second set of terms are used to describe the output waveform and delay characteristics. These terms are used to describe the time the signal arrives at a given point, and the characteristics of the waveform that arrives.

- **Cell rise time or rise delay**\(^1\) – time from when the input crosses 50% voltage to when the rising output crosses 50% voltage.

- **Cell fall times or fall delay** – time from when the input crosses 50% voltage to when the falling output crosses 50% voltage.

- **Cell rise transition or slew**\(^2\) – the time elapsed from when the output signal crosses the 30% voltage to when it crosses 70%.

- **Cell fall transition or slew** – the time elapsed from when the output signal crosses the 70% voltage to when it crosses 30%.

- **Arrival time** \((t_a)\) – the time that the signal crosses the 50% voltage threshold at a given point in the circuit. The time associated with a rising and falling signal are called the rise arrival time and the fall arrival time, respectively.

- **Required arrival time** \((t_r)\) – the time that the signal is required to cross the 50% voltage threshold for the given point in the circuit to be timing feasible. The required arrival time associated with a rising and falling signal are called the rise and fall required arrival time, respectively.

- **Slack** \((s)\) – the difference of the required arrival time and the arrival time, represents the amount of timing slack available at that point in the circuit.

\(^1\)The term rise time is generally used to refer to a specific rising delay or a falling delay.

\(^2\)The terms transition time and slew will be used interchangeably in this work.
An example of these concepts is shown in Fig. 2.1. This figure shows a falling transition for an inverter in a 45nm process. The input is assumed to be an ideal ramp, however, the output transition is not ideal. The input slew is 0.1 ns, the fall time is 0.2 ns, and the output slew is 0.12 ns.

Note that the 50% threshold for the delays, and the 30% to 70% thresholds for the slews are parameters that are set by the given timing library. The 50% threshold generally standard across libraries, but libraries may differ in the slew thresholds. In libraries today, a 20% to 80% or a 30% to 70% threshold are the most common.

At a given point in the circuit, the *arrival time* \( (t_a) \) is the time that a signal crosses the delay threshold. This is the time that the signal is said to arrive at the given point. Arrival times are a widely used to convert the arriving waveform into a single time. There are two types of arrival times – the minimum arrival time, which is the fastest time that a signal can arrive, and the maximum arrival time, which is the slowest time that a signal can arrive. The minimum arrival times are used to check for hold-time violations, and the maximum arrival times are used to check for setup-time and output arrival time requirements.
At a given point in the circuit, the required arrival time \( (t_r) \) is the time that the signal is required to arrive to meet the setup time and the output arrival time requirements. This is computed by subtracting the clock period with all the delays downstream from the gate. Thus, if the signal arrives by the required arrival time, the delay conditions at the primary output or flip-flop inputs should be met.

The main use of the required arrival time is in computing the slack \( (s) \). The slacks is defined as the difference between the required arrival time and the arrival times:

\[
s = t_r - t_a
\]  

Thus, it measures the amount of timing “slack” available at the point in the circuit and how much the timing can be increased at or must be decreased at that point in the design. Timing is infeasible where the slack is negative \( (s < 0) \), and timing is feasible where the slack is positive \( (s > 0) \).

The last set of terms are used to relate the signal waveforms and the arrival times between different adjacent parts of the design.

(1) **Timing arc** – a concept used to relate the delay between two adjacent points in the circuit.

(2) **Positive unate** – when a rising input to a gate causes a rising output, or a falling input causes a falling output.

(3) **Negative unate** – when a rising input to a gate causes a falling output, or a falling input causes a rising output.

The **timing arcs** are useful abstractions to connect the delays of adjacent points in the circuit. For example, the delay from an input A to output ZN might be a timing arc, or the delay between an output of a gate to an input of another gate might be another timing arc. This helps to conceptualize the circuit timing in terms of a graph with the delays, represented by timing arcs, along the edges.

The terms **positive** and **negative** unate are used to describe timing arcs in a cell. They connect the appropriate rising or falling arrival times with the corresponding rising or falling arrival time. For example, the timing arcs in an inverter cell are **negative unate** – a rising input will
always cause a falling output. In contrast, the timing buffer cell has only positive unate timing arcs— a rising input will always lead to a rising output. In the exclusive-OR gate, there are both positive unate arcs and negative, as a rising input can lead to a rising output (if both inputs are initially 0), or a falling output (if the other input is 1).

2.2 Computing arrival times and slacks

Signals are propagated through the design using a series of timing arcs or timing paths. These timing arcs connect:

1. Primary inputs to gate input pins
2. Gate input pins to gate output pins
3. Gate output pins to other gate inputs or primary outputs using interconnects (nets).
4. Clock signals to input hold and setup conditions
5. Clock signals to output “clock-to-Q” (C2Q) delays.

STA works through the following method.

1. Compute the delays and arrival times of the primary inputs (PI) and the flip-flop outputs.
2. In topological order, compute the delays and arrival times for the remainder of the gates.
3. Compute the arrival times at the primary outputs (PO) and the flip-flop inputs.
4. In reverse-topological order, compute the required arrival times and the slacks.

The arrival times are found by adding the delays of the different timing arcs.

However, there is a complication when the multiple inputs combine to form a single output. This happens in the case of multiple input gates, such as a 3 input AND, a multiplexer, or 2 input exclusive-OR. In this case, the maximum of the arrival times at the output are propagated for a setup time or late-mode analysis, and the minimum of the arrival times at the output are propagated for a hold-time or early-mode analysis.
2.2. Computing arrival times and slacks

Fig. 2.2 Static timing analysis example– finding the maximum path. The notation $t_a : [\text{rise}] | [\text{fall}]$ is used to denote the rise arrival times and the fall arrival times, and $d : [\text{rise}] | [\text{fall}]$ is used to denote the rise and fall delays.

Figure 2.2 gives a simplified example of the max path STA process. The STA computes the maximum path delay by computing the arrival times at:

1. the output of G3 by using the C2Q delay.
2. the output of G1 by using the C2Q delay.
3. G2 from G3 by adding the output delay with the interconnect delay of net E.
4. G2 from G1 by adding the output delay with the interconnect delay of net B.
5. G2 from IN[1] by adding the arrival time at IN[1] to the interconnect delay of net C.
6. the output of G2 by taking the maximum of the sum of the input arrival times and the gate delay.
7. the input of G1 by adding the arrival time at IN[0] to the interconnect delay of net A.
8. the input of G3 by adding the output arrival time at G2 with the interconnect delay of net D.
9. the primary output OUT[0] by adding the output arrival time
Fig. 2.3 Static timing analysis example – find the minimum path. The notation \( t_a : \) [rise] | [fall] is used to denote the rise arrival times and the fall arrival times, and \( d : \) [rise][fall] is used to denote the rise and fall delays.

In this example, the worst-case delay is a falling delay of 8 at flip flop G3. There are two equally critical paths, or the worst-delay paths with this delay. One is from G3 to G2 to G3, and another is from G1 to G2 to G3.

Finding the shortest path is similar, except that the minimum arrival times are propagated. Figure 2.3 modifies Figure 2.2 to compute the shortest path. The difference is in computing the arrival time at the output of G2. In this case, the minimum path, from the primary input \( \text{IN}[1] \) is propagated through. In this example, the hold requirement is met – the minimum delay is 2, while the hold requirement is also 2.

The slack is computed using an extra backwards (reverse topological) pass through the circuit. In the backwards pass, the required arrival times \( (t_r) \) are backwards propagated through the circuit. It starts with the primary outputs and the inputs to the flip-flops, setting the required arrival time to the latest arrival time that the signal can arrive – this is generally equal to the clock period, or the clock period minus the setup time. The required arrival times are propagated backwards by

at G2 with the interconnect delay of net D.
2.2. Computing arrival times and slacks

The required arrival times \( t_r \) are propagated from the outputs to the inputs, and the slack \( s \) is computed as \( s = t_r - t_a \). The notation \( t_r : [\text{rise}] | [\text{fall}] \) is used to denote the rise required arrival times and the fall required arrival times.

Subtracting the delays at each arc, until it reaches the primary inputs to the flip-flops and latches. Once the required arrival times are set, the slacks are computed as the difference between the required arrival times, and the actual arrival times:

\[
s = t_r - t_a
\]

Figure 2.4 shows an example of this process. It proceeds by computing the required arrival time at:

1. the input of G3 by subtracting the clock period with the setup time.
2. the primary output \( \text{OUT}[0] \) by setting it equal to the clock period;
3. the input of G1 by subtracting the clock period with the setup time.
4. the output of G4 by subtracting the wire delay from the minimum of required arrival time from (1) and (2).
5. the input of G2 from G3 by subtracting the required arrival time from (4) with the corresponding gate delay.
(6) the input of G2 from G1 by subtracting the required arrival
time from (4) with the corresponding gate delay.

(7) the input of G2 from \text{IN}[1] by subtracting the required ar-
rival time from (4) with the corresponding gate delay.

(8) the output of G1 by subtracting the required arrival time from
(6) with the wire delay.

(9) the output of G3 by subtracting the required arrival time
from (5) with the wire delay.

(10) the primary input \text{IN}[1] by subtracting the required arrival
time from (7) with the wire delay.

The examples above shows how to compute the delay, required ar-
rival times and the slacks as a function of the gate delays and intercon-
nect delays. The process of computing the gate delays and interconnects
delays will be explained below.

2.3 Interconnect delay

As scaling continues, interconnect delay is becoming an increasingly
large percentage of the total delay. This has increased the importance
of computing accurate wire delays, and has motivated fast and accurate
interconnect delay metrics \footnote{For a detailed analysis of the subject, please see [24] or [125].}

Interconnects are generally extracted from layouts in the form of
parasitic networks. These parasitic networks model the capacitance and
resistance properties of the interconnects and these networks are used
to analyze the propagation of the voltages down the wires.

The simplest and most common of these networks is the RC tree. This is defined as a tree with:

- one voltage source
- capacitances are connected to ground
- resistances to other nodes (but never to ground).

Figure 2.5 shows an example RC tree. There is a voltage source $V_S$ that
drives the tree, and there are multiple loads ($C_1$ to $C_7$) that need to
be charged. In the context of gates and interconnect, $V_S$ represents the
gate output waveform and $C_1$, $C_2$, $C_3$ and $C_6$ represent the parasitic capacitances from the interconnect. $R_1$ - $R_7$ are the parasitic resistances from the interconnect and $C_4$, $C_5$ and $C_7$ represent the loads from gate inputs or primary outputs.

There are more complex variations on the RC tree [125]. If the restriction to a tree is dropped and loops are allowed, then the resulting network is called an RC mesh. When inductors are allowed, the resulting network is called an RLC tree or RLC mesh. Lastly, a pair of RC trees that are connected together using capacitors is called a coupled RC tree. There are many variations on this, that emerge by mixing the different types of network topology (line, tree, mesh), parasitic elements (resistors, capacitors, inductors), and the coupling (coupled or not coupled).

The challenge in computing interconnect delay is to find the propagation delay and the transition time at the outputs. The most accurate method is to solve the differential equations related to the currents, charge and voltages in the parasitic network. However, this is computationally prohibitive in large designs, and especially in the context of gate sizing where fast delay estimates are needed to guide optimization.

The simplest method is the Elmore delay [56]. This method approximates the delay as the expected value of the unit step response applied

---

Fig. 2.5 Example RC tree. All capacitors are connected to ground but no resistors are connected to ground.
to the network. This is also equal to the first-moment of the impulse response, hence the Elmore Delay is also called the dominant-time constant, or first-order approximation of the delay.

The widespread use of the Elmore delay comes from the ease by which it is computed for RC trees. The Elmore delay for a path on a tree is computed by summing the product of each resistance and the total capacitances downstream. This gives the expression:

$$\text{Elmore Delay} = \sum_{\text{node } i \in \text{Path}} (R_i \cdot \sum_{j \in \text{downstream}(i)} C_j)$$  \hspace{1cm} (2.3)

For the example in Figure 2.5, the Elmore delay at $C_4$ is:

$$R_1 \sum_{i=1}^{7} C_i + R_2 \sum_{i=2}^{7} C_i + R_3 \sum_{i=3}^{5} C_i + R_4 C_4$$  \hspace{1cm} (2.4)

and the delay at $C_7$ is:

$$R_1 \sum_{i=1}^{7} C_i + R_2 \sum_{i=2}^{7} C_i + R_6 \sum_{i=6}^{7} C_i + R_7 C_7.$$  \hspace{1cm} (2.5)

The Elmore delay is usually scaled by $\ln(2)$ to account for the 50% threshold for the delay. This correlates well with the actual delay \[18\] (also see \[4\]) and can be used for rough delay estimates or for short interconnect. In \[67\], the Elmore delay is shown to be an upper bound on the actual delay, which justifies its use as a conservative estimate.

When accuracy is important, modern timers may employ more accurate methods that utilize higher order moments to improve the accuracy. Examples of this are the Asymptotic Waveform Evaluation Method \[116\] and the Krylov subspace methods \[59\] \[132\]. We refer the reader to \[24\] \[125\] for more details on these methods.

### 2.4 Gate delay models

The most accurate gate models are in the form of spice level netlists. These netlists contain transistor level information that can be used by transistor-level timing methods for static timing analysis\[4\]. These

\[4\] For example, the Synopsys NanoTime \[113\].
methods use fast-spice like simulations without the use of input vectors to create timing estimates that are comparable to spice simulations.

However, the only a small percentage of designs are able to utilize transistor-level timing as most designs are too large for transistor level simulation. Furthermore, these methods are too slow to be used in the process of optimizing large designs.

The library modeling standard that is used is the Synopsys Liberty modeling format. Currently, Liberty uses two different methods, the Nonlinear Delay Models (NLDM) and the current-source models (the Synopsys Composite Current Source (CCS) and the Cadence Effective Current Source Model (ECSM)). The NLDM format is a legacy format introduced in 1992, and the CCS/ECSM methods are newer methods from 2004, that were introduced to improve accuracy.

2.4.1 The Nonlinear Delay Model (NLDM)

The NLDM has long been the de facto standard for static timing analysis methods, and is slowly being replaced by the newer current-source methods. It utilizes tables that store rise and fall times, and output rise and fall transitions (slews), as a function of the input transition and the output capacitance load. The model also contains the capacitances for all of the input pins in the design.

The parameters that define the cell rise, fall, and transition times are given at the top of the Liberty file. The most common numbers in modern libraries are a 50% rise/fall threshold and a 20% to 80% or 30% to 70% slew rate.

A table for rise delay, fall delay, rise transition, and fall transition is provided for each cell in the library. Each of these tables is indexed by the input transition time, and the output capacitance. Table 2.1 provides an example of the cell rise times for a nominal sized inverter gate. The delays are increasing functions of the input transition, and the output load and note that both the input transition and output capacitance have a large effect on the output delay.

There are also tables for the clock hold time and setup time of
### Static Timing Analysis

<table>
<thead>
<tr>
<th>output capacitance (fF)</th>
<th>0.4</th>
<th>0.8</th>
<th>1.6</th>
<th>3.2</th>
<th>6.4</th>
<th>12.8</th>
<th>25.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>output rise delay</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0075</td>
<td>0.018</td>
<td>0.022</td>
<td>0.032</td>
<td>0.051</td>
<td>0.089</td>
<td>0.164</td>
<td>0.315</td>
</tr>
<tr>
<td>0.0188</td>
<td>0.024</td>
<td>0.028</td>
<td>0.038</td>
<td>0.056</td>
<td>0.094</td>
<td>0.170</td>
<td>0.321</td>
</tr>
<tr>
<td>0.0375</td>
<td>0.032</td>
<td>0.038</td>
<td>0.048</td>
<td>0.066</td>
<td>0.104</td>
<td>0.179</td>
<td>0.330</td>
</tr>
<tr>
<td>0.0750</td>
<td>0.042</td>
<td>0.051</td>
<td>0.065</td>
<td>0.087</td>
<td>0.124</td>
<td>0.199</td>
<td>0.349</td>
</tr>
<tr>
<td>0.1500</td>
<td>0.059</td>
<td>0.069</td>
<td>0.087</td>
<td>0.117</td>
<td>0.164</td>
<td>0.239</td>
<td>0.388</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.084</td>
<td>0.098</td>
<td>0.121</td>
<td>0.160</td>
<td>0.223</td>
<td>0.319</td>
<td>0.469</td>
</tr>
<tr>
<td>0.6000</td>
<td>0.129</td>
<td>0.145</td>
<td>0.174</td>
<td>0.224</td>
<td>0.305</td>
<td>0.433</td>
<td>0.628</td>
</tr>
</tbody>
</table>

Table 2.1 Rise times (delays) and transitions (slews) in ns for a nominal inverter gate as a function of the input transition time and the output capacitance.

<table>
<thead>
<tr>
<th>output rise transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0075</td>
</tr>
<tr>
<td>0.0188</td>
</tr>
<tr>
<td>0.0375</td>
</tr>
<tr>
<td>0.0750</td>
</tr>
<tr>
<td>0.1500</td>
</tr>
<tr>
<td>0.3000</td>
</tr>
<tr>
<td>0.6000</td>
</tr>
</tbody>
</table>

### clock transition (ns)

<table>
<thead>
<tr>
<th>clock transition (ns)</th>
<th>0.005</th>
<th>0.0141</th>
<th>0.0281</th>
<th>0.056</th>
<th>0.113</th>
<th>0.225</th>
<th>0.450</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0075</td>
<td>0.038</td>
<td>0.033</td>
<td>0.028</td>
<td>0.022</td>
<td>0.017</td>
<td>0.017</td>
<td>0.028</td>
</tr>
<tr>
<td>0.0188</td>
<td>0.043</td>
<td>0.038</td>
<td>0.034</td>
<td>0.027</td>
<td>0.022</td>
<td>0.023</td>
<td>0.033</td>
</tr>
<tr>
<td>0.0375</td>
<td>0.050</td>
<td>0.046</td>
<td>0.041</td>
<td>0.035</td>
<td>0.030</td>
<td>0.031</td>
<td>0.041</td>
</tr>
<tr>
<td>0.0750</td>
<td>0.064</td>
<td>0.060</td>
<td>0.055</td>
<td>0.049</td>
<td>0.044</td>
<td>0.044</td>
<td>0.053</td>
</tr>
<tr>
<td>0.1500</td>
<td>0.087</td>
<td>0.083</td>
<td>0.078</td>
<td>0.071</td>
<td>0.067</td>
<td>0.067</td>
<td>0.075</td>
</tr>
<tr>
<td>0.3000</td>
<td>0.124</td>
<td>0.119</td>
<td>0.114</td>
<td>0.107</td>
<td>0.101</td>
<td>0.100</td>
<td>0.107</td>
</tr>
<tr>
<td>0.6000</td>
<td>0.181</td>
<td>0.176</td>
<td>0.170</td>
<td>0.162</td>
<td>0.155</td>
<td>0.153</td>
<td>0.159</td>
</tr>
</tbody>
</table>

Table 2.2 Setup times for a rising input to a nominal D-Flip Flop as a function of the input transition and the clock transition.
2.4. Gate delay models

Fig. 2.6 Example of effective capacitance: an inverter driving an RC tree. The effective capacitance is less than the total capacitance of the tree: $C_{\text{eff}} < \sum_{i=1}^{4} C_i$. Adapted from [125].

sequential elements. The hold and setup times are given for rise and fall transitions, as a function of the input transition (slew), and the clock slew. Table 2.2 shows an example of the setup time for a nominal sized D-flip flop.

One of the limitations of this model is that the load capacitance is not well-defined at the output of a gate. Gates usually drive other gates through a series of interconnect that have their own intrinsic resistances. However, this has the effect of “resistive shielding” as the resistance hides some of the capacitance from the gate. Essentially, this resistance will cause the capacitances closer to the gate to be charged faster than the ones downstream from the gate. Thus, the waveform at the output of the gate will be faster than having the total capacitance present at the output of the gate.

To account for this, the timer must compute the effective capacitance of the gate (see [117, 98, 16, 57]) to predict accurate delays. As the amount of interconnect grows, and the resistances increase, this becomes a larger and larger problem. Figure 2.6 shows an example of this resistive shielding effect. Due to the resistors $R_1-R_4$, the effective capacitance seen at the output of the inverter is strictly less than the
total capacitance of the tree. As the resistances increase, the effective capacitance decreases.

Another limitation is the one-parameter model of the input and output waveforms [57]. The input and output waveforms are characterized only by their transition time. This limits the amount of flexibility and introduces modeling errors. However, many tools will re-model the NLDM library as a current-source model that will be discussed in Section 2.4.2 (see, for example, [79]). The NLDM tables are used to estimate the input-output current and voltage relationships. This helps to mitigate the limitations of the NLDM.

In the context of gate sizing, the NLDM has been the standard timing information for gate sizing algorithms for over a decade, and has influenced many gate sizing algorithms. Most algorithms will consider the gate delays to be functions of the input slew and output capacitance.

2.4.2 Current source models

Current source models were introduced to increase the accuracy of the gate models [145, 57, 82]. They use enhanced driver models, which model the output of the gates as function of time, and receiver models, which model both the loading and the Miller coupling effects. These libraries are generally much larger than the NLDM libraries, as they need to store voltage or current information over multiple time points, for each input load and slew value.

The driver models provide an output waveform for each input slew and output capacitance combination. In the CCS models, the current is given as a function of time, and in the ECSM model, the voltage is given as a function of time. This improves the accuracy of the timing simulation, up to 2% for a single stage, when compared to HSPICE [145].

The receiver models are used to capture the nonlinear characteristics of the receiver [145, 82]. These models use a two tables that give the receiver capacitance as a function of the input transition and the output load. The first table gives the input pin capacitance for the first half of the input waveform, and the second table gives the capacitance for the second half of the waveform. The capacitances vary greatly; for
2.5. Slew propagation

An important part of static timing analysis is the propagation of the input transitions, or slews. Not only is the delay important, but the output waveforms are equally important. For example, Table 2.1 shows that the input transition affects the delay by 2 to 7X! Thus, ignoring slew effects will cause major inaccuracies in the delay. Furthermore, ignoring slew in gate sizing obscures the possibility of increasing gate sizes to improve the downstream slew, and the corresponding downstream delays.

Computing the output slew is a difficult process when there are

<table>
<thead>
<tr>
<th>when</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>!A1 &amp; !A2</td>
<td>1017.82</td>
</tr>
<tr>
<td>!A1 &amp; A2</td>
<td>7340.61</td>
</tr>
<tr>
<td>A1 &amp; !A2</td>
<td>1204.57</td>
</tr>
<tr>
<td>A1 &amp; A2</td>
<td>10286.09</td>
</tr>
</tbody>
</table>

Table 2.3 Leakage powers for the NAND2_X1 gate. A1 and A2 are the two input pins.

example the nominal sized inverter in the Nangate library has capacitances that range from .43fF to .78fF!

2.4.3 Liberty power models

All Liberty models provide leakage power information – the maximum or average leakage power for the cell, and may additionally provide the leakage power for the different input combinations. For example, Table 2.3 shows the leakage information for the NAND2_X1 gate from the 45nm Nangate Library.

The Nonlinear Power Model (NLPM) is the power analog to the NLDM. This model provides a series of tables for the rise and fall internal power of the cell, as a function of the input transition and the output load. This internal power model gives the internal energy used at each transition, but excludes the power that is needed to charge the output load.

2.5 Slew propagation

An important part of static timing analysis is the propagation of the input transitions, or slews. Not only is the delay important, but the output waveforms are equally important. For example, Table 2.1 shows that the input transition affects the delay by 2 to 7X! Thus, ignoring slew effects will cause major inaccuracies in the delay. Furthermore, ignoring slew in gate sizing obscures the possibility of increasing gate sizes to improve the downstream slew, and the corresponding downstream delays.

Computing the output slew is a difficult process when there are
multiple inputs. This is because there may be a case where the input signals overlap. For example, Figure 2.7 shows the case of a two-input gate. The input A arrives at the gate later (in terms of the 50% delay threshold), but it finishes transitioning faster. The input B arrives at the input earlier, but has a much slower transition.

When computing the output slews for max path analysis, the slower slew is usually propagated. This is because the timing analysis is intended for design validation, and it is important to return worst-case numbers. Thus, in Figure 2.7 the arrival time of input A will be used to compute the output arrival time, and the slew of input B is used to compute the output slew. However, most timers can also propagate the slew associated with the greatest arrival time if specified.

Similarly, for min path analysis, the fastest slew is propagated, as this is the worst case. Thus, in Figure 2.7 the arrival time of input B will be used to compute the output arrival time, and the slew of input A is used to compute the output slew.

2.6 Power and clock domain considerations

Most modern designs will also use multiple clock domains, and possibly multiple power domains. Multiple clock domains arise naturally, as one clock might be used to synchronize the input of the design, another clock to drive the transfer within the design and a third clock that synchronizes the output of the design. Multiple power domains may show up due to performance considerations. A low-voltage power domain may be used to handle data paths that are not-critical, while
2.7 Additional considerations

A higher-voltage might be used for parts of the domain that need a higher voltage to process in time.

Multiple power domains require the computation of the appropriate delays for the proper supply voltage, and the delays through the voltage-level shifters. This would affect the delays and slews, and the method to compute the slews. For example, the delay would need to be computed with a higher voltage, and also the arrival time is 50% of the new voltage.

Multiple clock domains are a trickier consideration. While timing the design is straightforward within a clock domain, complexities emerge when crossing clock domains. Generally, the data signals use synchronization or handshaking methods between the different clock domains. However, these paths require special attention to make sure that these paths are handled correctly, or designers mark them to be ignored.

2.7 Additional considerations

There are many additional concepts that are used in Static Timing Analysis that will not be covered here:

- **On-Chip Variations** – modeling the variations in operating conditions (temperature and voltage) and manufacturing process (transistor dimension and threshold variations, interconnect dimension variations, etc.).
- **Crosstalk** – the effect of neighboring interconnects due to coupling capacitances.
- **Timing borrowing** – utilizing transparent latches to improve performance.

2.8 Block-based delay formulation

In the STA method, the delays are calculated by propagating the worst-case arrival times and slews at every gate output. This is called a block-based method, and is in contrast to the path-based method. In the path-based method, the delays and arrival times are computed independently. For the example in Figure 2.2, the path from G1→G2→OUT[0]
is computed separately from the path $G_1 \rightarrow G_2 \rightarrow G_3$, even though they overlap significantly. This method is more accurate, as less pessimism is needed when the actual path is known, but has the drawback of being computationally expensive – the number of paths is exponential in the number of gates.

The block-based delay can be formulated as a series of inequalities. The idea is that the arrival time at the output of a gate is the maximum of the input arrival times, plus the delay through the gate. Thus, at the output of any gate $g$:

$$t_{a(g')} + d_{g'} \leq t_{a(g)}, \forall g \in \text{fi}(g'), \quad (2.6)$$

or equivalently:

$$t_{a(g)} + d_g \leq t_{a(g')}, \forall g' \in \text{fo}(g). \quad (2.7)$$

For primary input gates, we have:

$$d_g \leq t_{a(g)}, \forall g \in \text{PI}, \quad (2.8)$$

and for primary outputs, we can write

$$t_{a(g)} \leq T_{\text{max}}, \forall g \in \text{PO}, \quad (2.9)$$

where $T_{\text{max}}$ is a constant representing the maximum delay constraint, for timing constrained problems, or as

$$t_{a(g)} \leq t_{a(\text{max})}, \forall g \in \text{PO}, \quad (2.10)$$

where $t_{a(\text{max})}$ is a variable that can be used to minimize the maximum delay. While these inequalities can be replaced by using the max functions, e.g.

$$t_{a(g)} = \max_{\forall g' \in \text{fi}(g')} \{ t_{a(g')} + d_{g'} \}, \quad (2.11)$$

the expression in terms of inequalities is better suited for use in optimization routines, as the max operator is generally non-differentiable.

The model can be improved to account for rise and fall times, and the different timing arcs in each gate. For example, the model in (2.6) can be improved to

$$t_{a(g'(\text{rise}))} + d_{g'(\text{rise})} \rightarrow g(\text{fall}) \leq t_{a(g(\text{fall}))}, \forall g \in \text{fi}(g'). \quad (2.12)$$
In addition, the delays can be written as functions of the gate type, \( \omega \), and the input slew:

\[
t_a(g'(\text{rise})) + d_{g'(\text{rise})} - t_{g'(\text{fall})}(\omega, \tau_{g'(\text{rise})}) \leq t_a(g'(\text{fall})), \forall g \in \text{fin}(g'). \quad (2.13)
\]

As this makes the notation difficult to follow, it will be omitted in the remainder of the text. However, it should be understood that the block-based delay formulations can also account for these subtleties.

### 2.9 Difference between STA timers

While the basic methodologies that the timers use are all similar, there is still a difference between the outputs of different timers. Timers used in place and route tools will generally use approximations for variation modeling, crosstalk and interconnect delay calculations. On the other hand, sign-off quality timers will use methods that are more accurate, while they may not be as fast.

The main benefit to using better timing methods is to reduce the pessimism. STA is designed to be conservative, as the resulting design must meet the timing. However, a large amount of pessimism may cause over-design, while a timing method that is not conservative enough may result in catastrophic results.

The difference between the timers used in place and route, and the timers used in signoff will be discussed in more detail in Section 5.2.1.

### 2.10 Statistical Static Timing Analysis (SSTA)

Statistical Static Timing Analysis algorithms update the STA framework to compute statistical delays and arrival times. There is an extensive literature on the subject \(^6\). In this section, we briefly describe the basic categories and describe the two main categories of SSTA methods – block-based methods and path-based methods.

#### 2.10.1 Block-based methods

Block-based methods for SSTA \([84, 146, 27, 150]\) are a natural extension of the STA framework. In STA, there are two main mathematical op-

\(^6\) For a good overview see \([90, 17, 61]\).
erations, the summation, when gate delays are added to arrival times, and the maximum operation, when the latest arrival time is propagate down the circuit (see Section 2.2). SSTA extends these operations to work with distributions, thereby making it possible to propagate statistical arrival times.

Gate delays and arrival times can be described using the canonical delay model [150]:

\[
a_0 + \sum_{i=1}^{n} a_i \Delta X_i + a_{n+1} \Delta R_a
\] (2.14)

where \( \Delta R_a \) and the \( \Delta X_i \) are zero mean, unit variance random variables. Each of the \( \Delta X_i \) describe the \( n \) different global sources of variation, with the \( a_i \) representing the sensitivity of the delay to the variation source. \( \Delta R_a \) describes the independent local variation, and the \( a_{n+1} \) describes the sensitivity to the local variation.

When the timing variations are Gaussian, the sum is easy to compute. As the sum of two Gaussian random variables is also Gaussian, the sum of \( d_a \) and \( d_b \) given by

\[
d_a = a_0 + \sum_{i=1}^{n} a_i \Delta X_i + a_{n+1} \Delta R_a
\] (2.15)

\[
d_b = b_0 + \sum_{i=1}^{n} b_i \Delta X_i + b_{n+1} \Delta R_b
\] (2.16)

can be summed directly to form the Gaussian random variable

\[
d_{a+b} = (a_0 + b_0) + \sum_{i=1}^{n} (a_i + b_i) \Delta X_i + \sqrt{a_{n+1}^2 + b_{n+1}^2} \Delta R_{a+b}
\] (2.17)

The maximum operation, however, is problematic, and is the main difficulty for block-based methods. The maximum of two Gaussian random variables is not, in general, Gaussian, and the resulting distribution cannot be computed in closed form. However, the mean and the variance of the resulting distribution can be computed using the formulae in [41], which [27] uses to reconstruct a Gaussian distribution that

\(^7\) Sensitivity can be computed as the partial derivative of the delay with respect to that variation source.
approximates the maximum. Applied to the canonical delay model, the maximum of $d_a$ and $d_b$ is approximated as $\hat{d}_{\text{max}}\{a,b\}$ as

$$\hat{d}_{\text{max}}\{a,b\} = (T_A a_0 + (1 - T_A)b_0) + \sum_{i=1}^{n} (T_A a_i + (1 - T_A)b_i) \Delta X_i \quad (2.18)$$

$$+ c_{n+1} \Delta R_{\text{max}}\{a,b\}$$

where $T_A$ is the probability that $d_a > d_b$, and $c_{n+1}$ is computed so that the variance of the approximation $\hat{d}_{\text{max}}\{a,b\}$ matches the actual variance.

Extension to this approximation to handle non-Gaussian variations and delay expressions are studied in [26, 155, 156, 83, 135, 38, 157, 32]. The non-Gaussian methods improve the accuracy of block-based SSTA by incorporating higher-order effects, or accounting the skewed delay distributions.

\subsection*{2.10.2 Path-based methods}

Path-based methods are studied in [21, 35, 62, 95, 112, 6, 8, 93, 118]. These methods select the most critical paths in the design, and then compute the statistical delay pdf using these critical paths, either by Monte-Carlo, or by using analytical method. They have the advantage of being more accurate because they do not need to approximate the statistical maximum of two random variables [118]. On the other hand, these methods are much more computationally intensive than the block-based methods, as the number of paths is generally believed to be exponential in the number of cells. However, [118] argues that the number of paths is sub-quadratic for a commonly used set of benchmarks, and that path-based SSTA can be used in conjunction with block based methods to improve accuracy.
Gate sizing, along with gate-length biasing and $V_t$ assignment, works by matching the drive-strengths of the transistors to the capacitive loads in the design. Larger gates are used to drive larger capacitive loads due to long interconnect wires, I/O pads, or large fanouts. However, improvements in the delay must weighed against increases in the power, and this trade-off motivates gate sizing.

### 3.1 Delay trade-offs

The relationship between the gate size, length and $V_t$ as a function of the delay can be approximated using an $\alpha$-power law model \cite{124}:

$$\text{Delay} \propto \frac{1}{W/L(V_{gs} - V_{th})^\alpha}.$$  \hspace{1cm} (3.1)

where $W$ is the gate width, $L$ is the gate length, $V_{gs}$ is the gate-to-source voltage, and $V_{th}$ is the threshold voltage. This means that the gate delay roughly linear in the gate length, and inversely proportional to gate widths.

For the 45nm commercial library, a sample of the effects of the
3.1. Delay trade-offs

Fig. 3.1 Normalized inverter gate delays for a commercial 45nm library.

delays is given in Figure 3.1.

- $V_{dd} = 1.1V$

$\alpha$ for this library is approximately 1.4.

Notice that the gate widths in this library are geometrically spaced – there are 10 gates sized 2.5 and under, and there are 10 gates between 3 and 16. This is because the inverse relationship between the gate delay and the gate width: reducing the delay by a factor of 2 requires the gate width to double in size.

The gates are in the library are spaced to cover a wide range of delays. Increasing the gate size will lead to a 10 to 15% reduction in the output delay, and switching the gate from minimum size to maximum size will reduce the output by approximately 12X. Increasing the threshold voltage from $v_{t0}$ to $v_{t1}$ or $v_{t2}$ to $v_{t3}$ increases the delay by about 10%. However, increasing the threshold voltage from $v_{t2}$ to $v_{t3}$ increases the delay by 35%.

However, increasing the gate widths will increase the capacitances of

---

1In this table, the rise and fall transition thresholds are 20% and 80%, and the delay threshold is 50%. For example, the rise transition time is the time measured from when the signal passes 20% of the supply voltage, to when it reaches 80% of the voltage. The pull-up delay is the time from when the signal reaches the input of the gate (at 50%) to the time that the output pin voltage reaches 50%.
The mechanism behind gate sizing

Fig. 3.2 Normalized inverter input pin capacitances in a commercial 45nm library.

its input pins, which are proportional to \( W \cdot L \). For example, Figure 3.2 shows that this capacitance is roughly linear in the size of the gate. This may cause the delay of the fanin gates to increase, and may have a cumulative negative effect on the delay.

Changing the threshold voltages has a milder effect on the input capacitances. Figure 3.2 shows that the capacitance is a mild function of the threshold voltage. Moving from \( v_{t0} \) to \( v_{t1} \) gives a 3% decrease in the capacitance, while moving from \( v_{t1} \) to \( v_{t2} \) gives a 4.5% decrease in the capacitance.

Due to this mild effect, increasing the threshold voltage is seen to have a negative effect on the delay. Though the input capacitance changes, the total effect on the delay is very likely to be negative. In contrast, increasing the gate size may not decrease the overall delay. The increase in the input capacitance may increase the delay more than the decrease in the output delay. This is why gate sizing requires a balancing of the loads in the design, rather than increasing the sizes to meet the delay constraints.

3.1.1 Delay and balancing loads

From a purely delay standpoint, the goal of gate sizing is to balance the capacitive loads with higher gate sizes. To illustrate this concept,
3.2 Power trade-offs

Power is an important concern in gate sizing. The amount of power dissipation has increased super-linearly with the shrinking of the transistor dimensions, and the power consumption is increasing faster than the ability to remove the heat from the devices. Furthermore, with the increasing focus on low power and mobile applications, low power is a major factor in the design process.

Power consumption is divided into static power and dynamic power. The static power is composed of the leakage powers, while the dynamic power is composed of the power used to switch the voltages within the gates. The focus on power has been shifting from dynamic power to static power. At the 45nm technology node, static power is comparable to the dynamic power consumption, and the static power consumption will continue to increase at a much faster rate than the dynamic power.

3.2.1 Dynamic power

The dynamic power is related to the energy used to switch logic states. This requires charging and discharging the interconnect and also includes the internal energy that is lost in the process of switching.

The dynamic power due to charging and discharging gates \( p_{\text{dynamic(cap)}} \) is directly related to the capacitance that is charged. Thus, if the total capacitance of a gate is \( C_g \), and the gate switches at a frequency of \( f_{\text{switch}} \), then the power dissipation is approximately:

\[
p_{\text{dynamic(cap)}} = f_{\text{switch}} \cdot \left( \frac{1}{2} C_g V_{\text{dd}}^2 \right)
\]

This expression is just the energy stored on the gate \( \frac{1}{2} C_g V_{\text{dd}}^2 \) times the number of times the energy will be dissipated per second \( (f_{\text{switch}}) \).

The capacitance of a gate is linearly related to the area of the gate, source and the drain:

\[
p_{\text{dynamic(cap)}} \propto \frac{W}{L}.
\]

Thus, increasing the gate widths will have a linear effect on the dynamic power due to capacitance charging.
The mechanism behind gate sizing

The other component of dynamic power is the short-circuit power, $p_{\text{dynamic(sc)}}$. This is the power that is lost when both the p-MOS and n-MOS are temporarily on while the gate is in transition. When the input transition is fast, then the short-circuit power is minimized. However, a slow input transition can cause a 5X to 10X increase in the short-circuit power, and using minimum sized devices may hurt overall power \[126\]. From \[124\], a short-circuit current expression derived using the $\alpha$-power law is:

$$p_{\text{dynamic(sc)}} \propto \tau \cdot V_{DD} \cdot \frac{W}{L} \cdot \frac{(1 - 2(\frac{V_{TH}}{V_{DD}}))^{\alpha+1}}{(1 - (\frac{V_{TH}}{V_{DD}})^\alpha)}. \tag{3.4}$$

where $\tau$ is the input transition time, and $\alpha$ is a fitting coefficient for the technology library\[2\]. This indicates that the short-circuit current is linear in the input transition time and the gate length, and is inversely proportional to the gate lengths.

Figure 3.3 gives the relative internal power dissipation for the inverter gates. The internal power refers to the power that is dissipated inside the boundaries of the cell itself, and does not include the power dissipated by charging the output load. The figure shows that the switching power is indeed a linear function of the size. When comparing threshold voltages, the $v_{t0}$ gate has the highest internal power

\[2\] For the 45nm Nangate Library, $\alpha \approx 1.4$. 

Fig. 3.3 Normalized inverter gate internal powers a commercial 45nm library.
3.2 Power trade-offs

3.2.2 Leakage power

The leakage power is the current that is dissipated when there is no change in the inputs. It is also referred to as the static power to contrast it with the dynamic power. Leakage became a significant source of power dissipation from the 130nm technology node and at 45nm it is comparable to the dynamic power dissipation. There are several sources of leakage power (see Figure 3.4). There is gate leakage from the gate to the substrate due to tunneling effects, leakage from the source to the drain due to subthreshold conduction, and leakage from the wells to the substrate.

The main source of leakage is due to subthreshold conduction from the source to the drain. This happens because the transistor is not fully “OFF”, and has a current proportional to:

$$\frac{W}{L} \cdot e^{-\frac{V_{gs} - V_{th}}{v_T}} \left(1 - e^{-\frac{V_{ds}}{v_T}}\right)$$

where \(v_T = 26mV\) is the thermal voltage and \(V_{gs}\) is the gate-to-source voltage. This expression shows that the subthreshold is a linear function of gate-width but is an exponential function of the threshold voltage \(V_{th}\).

The leakage power is a strong function of the gate inputs for multi-input gates. This is because the inputs affect the internal voltages of

Fig. 3.4 Sources of leakage power.
The mechanism behind gate sizing

<table>
<thead>
<tr>
<th>state</th>
<th>leakage (nW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 &amp; A2 &amp; A3</td>
<td>19.78</td>
</tr>
<tr>
<td>A1 &amp; A2 &amp; !A3</td>
<td>14.72</td>
</tr>
<tr>
<td>A1 &amp; !A2 &amp; A3</td>
<td>7.41</td>
</tr>
<tr>
<td>A1 &amp; !A2 &amp; !A3</td>
<td>0.48</td>
</tr>
<tr>
<td>!A1 &amp; A2 &amp; A3</td>
<td>15.59</td>
</tr>
<tr>
<td>!A1 &amp; A2 &amp; !A3</td>
<td>1.63</td>
</tr>
<tr>
<td>!A1 &amp; !A2 &amp; A3</td>
<td>7.15</td>
</tr>
<tr>
<td>!A1 &amp; !A2 &amp; !A3</td>
<td>1.27</td>
</tr>
</tbody>
</table>

Table 3.1 Leakage power for a NAND3 cell as a function of its inputs.

![Normalized inverter gate leakage powers in a commercial 45nm library.](image)

Fig. 3.5 Normalized inverter gate leakage powers in a commercial 45nm library.

stacked transistors. For example, Table 3.1 shows the leakages for the nominal sized NAND3 cell as a function of the input. The leakage values range from .48 nW to 19.78 nW—a range of over 40X! This range causes significant modeling error, as it is very difficult to predict the probabilities for each of the input states.

The exponential relationship between the leakage and the threshold voltage is the motivator for multiple-$V_t$ cells. Figure 3.5 shows that relative change in the leakage power as a function of the size. While the relationship between gate width and gate size are linear, the leakage
Gate length biasing \[66\] is another tool for optimization. The idea is to increase the lengths of gates with positive slack. The increase in gate length will have linear increases on the delay but a super-linear effect on the leakage power. Figure 3.6 shows how the delay and leakage power change as a function of the gate-length bias. Increasing the gate-length by 5% gives approximately 5% reduction in the leakage power, while creating an 10% improvement in the delay. This provides a finer resolution than the \(V_t\)-assignment, as this method can provide modest reductions in the leakage for modest trade-off in power.

### 3.3 Gate sizing examples

In this section, we provide examples that illustrate the trade-offs involved in gate sizing. These examples are adapted from the sizing benchmarks from \[65\].
3.3.1 Inverter chain example

Inverter chain driving a fanout of 32 minimum-sized inverters. Figure 3.7 shows the gate sizes, dynamic power, and leakage power as a function of the minimum delay. The power vs size shows an intuitive trend – the gate sizes increase as the power constraint decreases. This is because there is a large load at the end of the inverter chain, and larger gates are useful for driving this load.

Notice that the gate sizes increase starting from the gates closest to the load at the end. This points out an important fact about gate sizing: larger gates do not always help. For example, with all other gates at minimum size, increasing the size of gate g2 to 32X will *increase* the delay by 50%! It is important to remember that the goal of gate sizing is the balance the capacitive loads in the design. Thus, the size of gate g4 is increased first. Next, g3 is increased to account for the increased size of g4, and thereafter, g2 and g1 are increased.

The switching power (Figure 3.7(c)) and leakage power (Figure 3.7(d)) also follow this trend. The switching power increases because the cumulative sizes of the gates and hence, the total capacitance increases. The trend for the switching power follows (3.2), and the leakage power trend follows (3.5). Thus the trend in these plots is very similar to the trend in Figure 3.7(b).

The internal power, however, is not a monotonic function of the cumulative gate sizes. As the internal power is a function of the internal node capacitances and the short-circuit current, it is affected by the input transition times. Initially, as the gate sizes increase, the input transition times decrease, causing the total internal power to increase. However, as the gate sizes continue to increase, the internal capacitances dominate the internal power, and the power follows a similar trend as the switching power.

Notice that the scales on the dynamic power (Figure 3.7(c)) and the scale on the leakage power (Figure 3.7(d)) are different. This might suggest that the dynamic power consumption will be 1000X larger than the leakage power. However, this is misleading in this example. The switching power is a function of the switching frequency (see (3.2)), and in large designs, the likelihood that a node will switch is less than 1. This
3.3. Gate sizing examples

Fig. 3.7 Inverter chain driving a fanout of 32 (a capacitance equal to 32 minimum sized inverters). The schematic is shown in (a). Plots (b) to (d) show the gate sizes, dynamic power, and leakage power as a function of the minimum delay.
The mechanism behind gate sizing

Fig. 3.8 Mesh example from [65]. The optimum sizes for minimum delay are shown.

has the effect of making the dynamic and static powers comparable.

3.3.2 Mesh circuit example

The mesh circuit (see [65]) is another interesting example. In this example, there are a few gates with larger loads (g2, g3 and g5), and the remainder only have a fanout of one gate. To create a minimum delay configuration, the gate with the most fanout load, g5, reaches its maxi-
The delay in this example ranges from .077ns for the fastest configuration, and .131ns for the configuration with all gates at minimum size. The spread in the total powers is larger, ranging from .214mW at the minimum, and .614mW at the maximum. Note that there is no dip in the internal power (compared to the inverter chain example in Section 3.3.1), as the input transitions do not dominate the internal power.

3.3.3 Star circuit example

The star circuit in Figure 3.9 is an example where there is one critical node at the center. In this design, the g7, g8 and g9 gates have an increased load – each has 2 fanouts. However, the size of g7 is limited to its maximum size of 4X, which limits the sizes of the other gates.

The minimum delay to maximum delay range is .112ns to .073ns, with a corresponding total power range of .247mW to 1.025mW. For the minimum delay sizing, gate g7 is set to its maximum and the remainder of the gates are sized to balance the loads. Notice that the input gates
are at maximum size – this is to take advantage of being a *primary input*. In the design model the arrival time is the same, irrespective of the size of the primary inputs. Thus, they can be maximized to improve the transition times in the circuit.
Discrete gate sizing methods are designed to choose cells from standard cell libraries for each gate in the design. As the range of gate options is discrete, the problem is inherently, and has been shown to be NP-hard [89].

There are many methods that approach the problem of discrete sizing using a continuous sizing. The methods that use rounding generally have at least 8 gate sizes per cell [122]. Other methods use the continuous solution as a starting point, and then employ heuristics to map the gate to discrete sizes [40, 129, 76].

Other methods attack the discrete problem directly. For example, [46] uses an optimization-derived approach that uses gradient-like functions to minimize the delay or power. [96] uses a dynamic programming and Lagrangian relaxation based heuristics, and [109] uses slack allocation via linear programming to improve power. This chapter will survey the field of discrete gate sizing, however the coverage is not exhaustive. Methods that are not covered here include the randomized exhaustive search based methods [153], and graph based methods [48, 29, 28].

1 See, for example [40, 129, 76, 122]
4.1 Finding candidate moves and cell options

Gate sizing methods try to size the best candidate gates to achieve optimal trade-offs. This requires a method to find candidate gates, and evaluate them. The “Rank-based algorithms” in Section 4.3 and the slack budgeting methods in Section 4.4 methods is the idea of a sensitivity between the power and the delay:

\[ \sigma_d = \frac{\Delta p(g, \omega)}{\Delta \text{Delay}(g, \omega)}. \tag{4.1} \]

This sensitivity is used as a first-order type-analysis to determine how efficiently the delay can be improved for a change in power, and captures the delay information in the gate’s fanin cone. Thus, the algorithms can prefer gates with better trade-offs to meet a timing constraint with a lower power design.

A variant of this method is to use a power vs. slack sensitivity:

\[ \sigma_s = \frac{\Delta \text{Power}(g, \omega)}{\Delta \text{Slack}(g, \omega)}. \tag{4.2} \]

This does a better job in capturing the effects of the delay, because both downstream (fanout) and upstream (fanin) effects due to slew will also be accounted for. For example, downsizing a gate may decrease the arrival time of the input gates, but the decrease in slew may increase the rise and fall delays in its fanout cone. Thus, this is an improvement on the power vs. delay sensitivity above.

Computing this sensitivity requires computing both the power and delay for each of the cell-options \( \omega \). This requires multiple calls to the timer to evaluate how the delay changes over the different cell options. When slacks are involved, this requires recomputing the required arrival times as well.

The need to find and evaluate candidate gate moves is the most time consuming step in gate sizing. The challenge is to search the design and find gates and alternative cell options that: (1) will not violate timing and (2) will improve the objective. Many algorithms and most commercial tools use incremental methods to speedup this process.
4.2 Avoiding significantly suboptimal solutions

In discrete gate sizing and library cell assignment, it is difficult to gauge the progress of the optimization. In other words, it is difficult for the algorithm to measure how far it is from optimal and due to this, algorithms terminate whenever the sizing method is unable to optimize any further. This is a major concern for the quality of the result, as it becomes imperative that when the algorithm does get stuck, it is stuck at a solution that is relatively close to optimal. Thus, many algorithms will take a circuit-wide or global approach to gate sizing.

One of the reasons that the algorithm terminates in a significantly suboptimal solution is that because the evaluation metrics at the core of these methods are local (see 4.1). Candidate gates and cell options are evaluated by how they individually affect local delay, slacks, or power. However, the effects of large groups of gates is what is needed, and this discrepancy may cause sub-optimality.

Many methods make an effort to incorporate a circuit-wide perspective. For example, the “global sizing” method from [47] in Section 4.3 uses a gradual method, by making a series of smaller steps to perform the optimization gradually. The justification for this rests upon the assumption that local metrics are good guides for small changes but they are not a good guides for large changes. Thus, a series of smaller changes can help to improve the outcome.

The slack, slew and delay budgeting methods in Sections 4.4 and 4.8 incorporate a circuit-wide view by utilizing a pre-processing step to allocate the slacks and slews. These allocations may take into account the topology of the design, or the slacks, or the relationship between slacks of neighboring gates. These allocations are then used to guide optimization at a local level.

Lagrange multiplier weighting methods in Section 4.7 are also use circuit-wide metrics to guide local optimization. The Lagrange multipliers are updated according the the slack needs across the design. They are iteratively refined to balance the needs across different parts of the design.

Another reason for significantly suboptimal solutions is the large space of solutions that must be searched. For example, a 100,000 gate
design with 5 cell options per gate yields a search space of $5^{100,000}$. While this large search space can be reduced by pruning methods or dynamic programming, the number of options does not reduce down to a manageable size. However, the Dynamic Programming based-heuristics in Section 4.6 use a dynamic programming type search to find the best solution. Essentially, these methods perform a smart enumeration to quantify the interactions between different gates in the design, and prune the options that are suboptimal.

When evaluating discrete sizing methods, it is helpful to keep these considerations in mind. Most methods were motivated as an improvement upon the greedy heuristic in [60], and they incorporate specific methods to avoid the pitfalls of greedy optimization.

### 4.3 Score and Rank Algorithms

Score and Rank algorithms (SR) for gate sizing were developed as fast heuristics for gate sizing. These methods include TILOS [60], other greedy heuristics [92, 152, 72, 66], and its variants [46, 53, 137, 136, 138]. The defining characteristic of these methods is a scoring method for each gate that measures its ability to provide a power per timing tradeoff, and a ranking step to select the best gates for optimization. The gates with the highest ranking are evaluated and changed, and this process is iterated until no improvement is possible.

In brief, the steps in SR methods are summarized in Algorithms 1 and 2. The Algorithm 1 starts with a timing feasible design, and trades the extra slack for power savings. The cells which provide the best power vs. delay are chosen. On the other hand, Algorithm 2 starts with a timing infeasible design, and gates are changed to reduce the negative slack. Gates that

While all algorithms in this category share these fundamental steps, they are distinguished by their choice in scoring functions:

---

2 Note that the term “Score and Rank” is ours. We believe that it's a useful category that covers a large and important class of methods.
4.3. Score and Rank Algorithms

Input: Design with gates optimized for minimum delay
while Slack(\(G\)) > 0 do
    /* Step 1: Score the gates and cell options
        according to a given score function \(\sigma\). Choose
        the best option for each gate */
    /* Step 2: Choose candidate gate moves by ranking
        the gates according to the score */
    /* Step 3: Perform optimizations in decreasing
        order of ranking */
end

Algorithm 1: The general Feasible-Start Score and Rank Algorithm

Input: Design with all gates at minimum power
while Slack(\(G\)) < 0 do
    /* Step 1: Score the gates and cell options
        according to a given score function \(\sigma\). Choose
        the best option for each gate */
    /* Step 2: Choose candidate gate moves by ranking
        the gates according to the score */
    /* Step 3: Perform optimizations in decreasing
        order of ranking */
end

Algorithm 2: The general Infeasible-Start Score and Rank Algorithm

- TILOS [60] and [152] use the power vs delay metric
  \[ \sigma_d(g, \omega; \omega_0) = \Delta t_a(g, \omega; \omega_0)/\Delta p(g, \omega; \omega_0), \] (4.3)
  where \(\Delta p(g, \omega; \omega_0)\) is the change in the power \((p(g, \omega) - p(g, \omega_0))\), and \(\Delta t_a(g, \omega; \omega_0)\) is the change in the arrival times at the output of the gate \((t_a(g, \omega) - t_a(g, \omega_0))\). Note that this formulation accounts for changes in the fanin gate delays due to the changing input capacitances. \(??\) Check signs

\(\text{In their paper, the delay vs size sensitivity is used; however, as we are mainly interested in power, we present it as the delay vs power sensitivity.}\)
Methods for discrete gate sizing

- For gate-length biasing, [66] uses the score:
  \[ \sigma_s(g, \omega; \omega_0) = \frac{\Delta s(g, \omega; \omega_0)}{\Delta p(g, \omega; \omega_0)}, \tag{4.4} \]
  where \( \Delta p(g, \omega; \omega_0) \) is the change in the power, and \( \Delta s(g, \omega; \omega_0) \) is the change in the slack. The change in the slack accounts for timing changes in the fanout cone due to slew effects. This provides a more complete picture of the timing effects than (4.3).

- The Duet method [137, 136], along with [139], adapts the scoring function from [53]:
  \[ \sigma_{duet}(g, \omega; \omega_0) = \phi\left(\frac{\Delta d_\alpha(g, \omega; \omega_0)}{|s(g, \omega_0)| + \epsilon}\right) \tag{4.6} \]
  where \( \alpha \in G \) are the timing arcs in the design (see Section 2.1), \( \Delta d_\alpha(g, \omega; \omega_0) \) is the change in the delay of \( s_{min} \) is the minimum slack in the entire design, and \( \epsilon \) is a small number used for the case \( s_\alpha = s_{min} \). Note that in practice, the \( \Delta d_\alpha(g, \omega; \omega_0) \) values are computed using analytical models and Elmore delay approximations to improve runtime.

- Coudert [47] uses a relax function to score the gates:
  \[ \sigma_{Relax}(g, \omega; \omega_0) = (\alpha(\Delta p(g, \omega; \omega_0)) - \epsilon) \cdot \phi\left(\frac{\Delta s(g, \omega; \omega_0)}{|s(g, \omega_0)| + \epsilon}\right) \tag{4.7} \]
  where
  \[ \phi(x) = \begin{cases} 1 + x & \text{if } x \geq 0 \\ \frac{1}{1-x} & \text{otherwise} \end{cases} \tag{4.8} \]

These methods are summarized in Table 4.1. Note that in this table, the applications vary from transistor sizing to gate length biasing and \( V_t \) assignment. Also note that the starting point for each of these methods is different. Methods [60, 152, 66, 137, 136] start with a design with timing violations, with the gates at their minimum area and/or high-\( V_t \) configurations. These methods improve the timing until the design is timing feasible, trading power increases for delay reductions. In
4.3. Score and Rank Algorithms

<table>
<thead>
<tr>
<th>method</th>
<th>input design</th>
<th>score</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>[60]</td>
<td>min area</td>
<td>(4.3)</td>
<td>transistor sizing</td>
</tr>
<tr>
<td>[47]</td>
<td>min delay</td>
<td>(4.7)</td>
<td>gate sizing</td>
</tr>
<tr>
<td>[152]</td>
<td>min area</td>
<td>(4.3)</td>
<td>$V_t$, gate sizing</td>
</tr>
<tr>
<td>[66]</td>
<td>min delay</td>
<td>(4.4)</td>
<td>gate length biasing</td>
</tr>
<tr>
<td>[137, 136]</td>
<td>all gates at high-$V_t$</td>
<td>(4.5)</td>
<td>$V_t$</td>
</tr>
</tbody>
</table>

Table 4.1 Summary of Score and Rank methods for gate sizing.

contrast, [47, 66] start with a timing minimized design that meets the timing constraints. In these cases, positive slacks are traded for power reductions.

These methods will be discussed in greater detail below.

4.3.1 Greedy methods

Although Greedy methods were first applied to gate sizing over two decades [60] ago, and they are still widely in use [33]. These methods have the benefit of being scalable, easy to implement, and they perform reasonably well (see Chapter 5). They use a locally optimal choice at each step in the algorithm, which gives these methods the title greedy.

The well known TILOS method [60] was motivated as a fast heuristic to solve the continuous gate-sizing method. Their paper notes that the problem has a convex formulation and can therefore be solved optimally; however, due to runtime reasons, they provide a heuristic for approximate sizing.

The TILOS method for timing constrained area minimization starts with a design at minimum size (all gates sized down to their lowest power sizes), and then scores each gate on the critical paths using the score function in Equation (4.3). The gate with the least score is chosen for optimization and is set to the cell option with minimum delay. This continues until the timing constraint is met. As this algorithm starts with a timing infeasible design, we denote this as timing infeasible greedy algorithm.

This method can be used for delay minimization. In this case, the method terminates when the delay cannot be further decreased. This
Methods for discrete gate sizing

is will be used in Chapter 5 to perform greedy timing optimization.

This method has been adapted to other contexts. [152] applies this method to the simultaneous gate sizing and threshold voltage assignment problem, by adding multiple $V_t$ options into the optimization process. The method is identical to the TILOS algorithm, except that the cell options include $V_t$ variants along with the different gate width choices.

[66] applies the greedy heuristic to gate-length biasing. This method starts with a timing feasible design at minimum delay and all gates with nominal gate-lengths. The gate widths are set as fixed, and the method scores each of the gates and the cell options with the different gate length variants. The top ranked gates and options have their gate lengths increased, and this continues until no slack remains. As a variation on the original delay vs power metric however, this method uses a slack vs power metric to capture delay changes in the fanout cone due to slew effects.

[137, 136, 138] apply a modified score that accounts for the criticality of each gate – gates with less slack are given greater weights that gates with larger slack (see Equation (4.5)). Each timing arc is weighted as:

$$\frac{1}{s_\alpha - \min(slacks) + \epsilon}.$$  

(4.9)

where $\epsilon$ is a small positive constant. For the most critical timing arcs, the weight is $1/\epsilon$, and it decreases as $(s_\alpha - \min(slacks))$ increases. Thus, delays in timing critical paths can be improved even when other delays in non-critical paths are increased.

### 4.3.2 Global sizing

[47] attempts to improve the short-sighted deficiency of the greedy method. A score function “Relax” (4.7) is proposed to gradually trade-off the delay for the power. The rationale for this approach is given in [?] and is the following.

“Optimizing the delay gives plenty of alternatives for power optimization, i.e., going far away from the infe-
sible region makes power optimization less likely to be trapped in a local minimum.”

“The power optimization is done within the feasible region by relaxing the delay constraints using a penalty/benefit function, as opposed for instance to a greedy method that resizes as many non-critical nodes as possible to their minimal power. Such a greedy method can give low quality results for the following reason: resizing a few nodes to their local minimal power too “quickly” creates critical paths that can prevent most of the other nodes from being resized and saving more power.”

```
/* GLOBAL SIZING: Relax the timing gradually */
Input: Design with gates optimized for minimum delay
while Power improves and Slack(G) > 0 do
    /* Step 1: Score the gates and cell options according to σrelax (Equation 4.7) */
    /* Step 2: Compute a set of moves: find a maximal ordered subset that minimizes the Power (rank the gates according to score, and choose the top k gates) */
    /* Step 3: Update the scores for changed gates and gates in neighborhood of changed gates */
end
```

**Algorithm 3:** The Global Sizing method

In effect, the relax function is designed to prevent the algorithm from converging too quickly to a poor solution, as may be the case with greedy methods. This method will gradually trade the delay for power reductions, until it settles on a good solution.

In the Global Sizing Algorithm (Algorithm 3) there are two main differences from the TILOS algorithm. Firstly, the method requires the computation of a “maximal ordered subset of moves that minimizes the
Methods for discrete gate sizing

They suggest that this be done using a gradient-type approach, or conjugate gradient method. However, this step is also tantamount to the gate sizing problem itself. For the purposes of this survey, this is assumed to be equivalent to taking a subset of the best options with the best relax values, and applying them until the timing budget is used.

The second difference is the incremental updates to the sensitivities. Unlike the TILOS algorithm, which only scores the critical gates, the global sizing requires that all gates are scored. Thus, an incremental method is proposed where after the initial iteration, the neighborhood around each changed gate, which is defined as the gates a given number of levels away from it, are added to the update list, and are updated.

4.4 Slack / Delay budgeting methods

Delay budgeting methods were developed as fast, scalable methods for gate sizing \[31\] These methods are an improvement over the sensitivity and gradient methods in Section 4.3 because they can manage the interaction between gates. For example, it understands that the slack used on one gate in the design reduces the amount of slack left for other gates. Furthermore, these methods account for the topology of the design – it can account for low sensitivity gates that are bottlenecks and as they are on many critical paths.

In this paper, we will focus on the recent approach by \[109\]. This method works in two phases:

- **Phase I**: Allocate a slack budget to each gate in the design;
- **Phase II**: Use the slack budget for the gate to choose the appropriate size.

There is an important consideration for Phase I. When the design is not timing feasible, then the algorithm will need to distribute negative slacks, along with positive slacks. When a gate is allocated a negative slack, the gate must improve its timing. To circumvent this difficulty, a minimum delay design is used as a starting point, where the slacks

\[4\] \[49\] also provides a method for transistor sizing, applied to a single logic gate.
have been maximized.

\[ P_{\text{best}} \leftarrow \infty \]

\[ \text{while } \text{Power}(G) < P_{\text{best}} \text{ do} \]

\[ P_{\text{best}} \leftarrow \text{Power}(G) \]

\[ /* \text{Setup: Compute Power vs. Delay sensitivities} */ \]

\[ \text{foreach } g \in G \text{ do} \]

\[ \text{Compute } \sigma_g = \max_{\omega \in \{(g)\}} \{\sigma_s(g, \omega; \omega_0)\} \]

\[ \text{end} \]

\[ /* \text{Phase I: Allocate slacks to gates using LP} */ \]

Solve:

\[ \text{minimize } \sum_{g \in G} \sigma_g s_g \]

subject to \[ t_{a(g)} + s_g \leq t_{a(g')} \quad \forall g' \in \text{fanout}(g) \]

\[ 0 \leq t_{a(g)} \leq T_{\text{max}} \]

\[ 0 \leq s_g \leq D_{g,\text{max}} \]

(4.10)

with variables \( s_g, t_{a(g)} \) to get slack allocations \( s_g \).

\[ /* \text{Phase II: Use the slacks to reduce the power} */ \]

\[ \text{foreach } g \in G \text{ do} \]

\[ \text{Set library cell of gate } g \text{ to} \]

\[ \text{argmin}_{\omega \in \{(g)\}} \{s(g, \omega) - s(g, \omega_0) \leq s_g\} \{\text{Power}(\omega)\} \]

\[ \text{end} \]

Algorithm 4: Slack budgeting method in [109]

In Phase I, the slacks are allocated to the gates. This is done by first computing the power-delay sensitivities using [4.1]. These sensitivities are then used in the objective of a linear programming method (Algorithm 4). These sensitivities weight the slacks allocated to the gates, and the solution to the linear program creates a “maximally-weighted independent set” of gates, where the gates are weighted by their sensitivities.

In Phase II, the slacks are converted into reductions in power (see
Phase II in Algorithm 4). This step is done at a local level – each gate uses their slack allocation to reduce their power. For example, at a gate $g$, the options with a smaller power are evaluated to see if the change in delay is within the allocated slack budget. After the options are evaluated, the gate is changed to be the one with the least power that meets the budget.

The biggest limitation to this method that changes in cells will affect the slacks of neighboring gates. The sensitivities for each gate may change if any of their fanin or fanout gates change. An increase in the size of an input gate may prove a greater improvement in slack, while a decrease may cause the opposite effect. Similarly, an increase in the size of an output gate will improve the slack improvement of the current gate, and a decrease will reduce the slack improvement. These interactions are not considered by this method and are handled by iterating the method.

Moreover, due to slew effects, the delay sensitivities may also change if any gates up-stream had changed. For example, if a change upstream from a gate improves the slew dramatically, then this may dampen the improvement seen at the current gate. Similarly, a decrease in the slew may increase the improvement that comes from increasing a gate’s size.

Thus, this algorithm is sensitive to the order in which the slacks are distributed to the gates. Furthermore, due to the interactions, slack may remain after the first iteration. Thus, the method is iterated until all of the budget is used.

The work in [31] has a couple differences. Slacks do not need to be maximized in this case, and the algorithms works by iterating over the paths in the design (in order of most critical to least critical), and allocating their slack budget as:

$$ s_g = T_{\text{max}} \cdot \frac{t_i(1 - \eta_i)}{T_{\text{path}}} . $$

(4.11)

When a gate is assigned multiple times, the gate is given the first assigned timing budget.
4.5 Continuous sizing based methods

Continuous sizing based methods are well studied, and have been applied as heuristics for discrete gate sizing. These methods model the delay and power as continuous functions of the design parameters, and then use the models to formulate an optimization problem.

4.5.1 Linear Programming methods

The simplest models, and the fastest to optimize, are the linear models [13, 143]. Power models are roughly linear (see Section 3.2) and can be approximated as:

$$\text{Power} = \sum_{g \in G} p_{g, \text{nom}} \cdot w_g$$

where $w_g$ is the width of the gate or the size of the standard library cell and $p_{g, \text{nom}}$ is the power of a minimum sized gate or library cell.

The gate delay is not linear in the gate size (see Section 3.1), thus linear models for the gate delay require fitting. In [13], the gate delay is linearized by:

$$d_g = c_1 + c_2 w_g - c_3 \sum_{g' \in \text{fo}(g)} C_{g'} w_{g'}$$

where $c_1$, $c_2$, and $c_3$ are modeling coefficients, $w_g$ is the width of gate $g$ and $C_g$ is the input capacitance of $g$. These models can be improved to be any convex piecewise-linear function by using a series of inequalities:

$$d_g \geq c_4 + c_5 w_g - c_6 \sum_{g' \in \text{fo}(g)} C_{g'} w_{g'}$$

$$d_g \geq c_7 + c_8 w_g - c_9 \sum_{g' \in \text{fo}(g)} C_{g'} w_{g'}$$

$$...$$

For further reference on modeling, please see [31, 19, 121].
The resulting linear program, using the block-based delay formulation in Section 2.8, is:

\[
\begin{align*}
\text{minimize} & \quad \sum_{g \in G} p_g w_g \\
\text{subject to} & \quad t_{a(g)} + d_g \leq t_{a(g')}, \quad \forall g' \in \text{fanout}(g) \\
& \quad c_1 + c_2 w_g - c_3 \sum_{g' \in \text{fo}(g)} C_{g'w_{g'}} \leq d_g \\
& \quad 0 \leq t_{a(g)} \leq T_{\text{max}} \\
& \quad w_{\text{min}} \leq w_g \leq w_{\text{max}}
\end{align*}
\] (4.16)

4.5.2 Geometric Programming methods

An improvement on the linear model is to use posynomial models \[60\]. Posynomials are equations of the form:

\[
\sum_k \prod_j x_j^{\alpha_{jk}}
\] (4.17)

\(x_j\) is restricted to be positive but \(\alpha_{jk} \in \mathbb{R}\) can be positive or negative. This form can handle the standard transistor models for delay:

\[
\text{Delay}_g = R_g \frac{\sum_{g' \in \text{fo}(g)} C_{g'w_{g'}}}{w_g}
\] (4.18)

\[
= R_g w_g^{-1} \sum_{g' \in \text{fo}(g)} C_{g'w_{g'}}
\] (4.19)

This results in the geometric programming problem:

\[
\begin{align*}
\text{minimize} & \quad \sum_{g \in G} p_g w_g \\
\text{subject to} & \quad t_{a(g)} + R_g w_g^{-1} \sum_{g' \in \text{fo}(g)} C_{g'w_{g'}} \leq t_{a(g')}, \quad \forall g' \in \text{fanout}(g) \\
& \quad 0 \leq t_{a(g)} \leq T_{\text{max}}, \quad \forall g \in \text{PO} \\
& \quad w_{\text{min}} \leq w_g \leq w_{\text{max}}
\end{align*}
\] (4.20)

While this is not convex as written, it is convex when the change of variables \(w_g = e^{y_g}\) is made. Extensions to these models can be made for \(V_t\) and gate length assignment. Also, methods to improve the accuracy of these models can be made by including rise and fall times, wire delays, and estimating slew effects. However, convex models are preferred, as they have the property that there exists a unique optimum that can be found in polynomial time \[20\ 120\].

\[6\] see \[81\ 19\] for a discussion on posynomial models for gate sizing.
4.5.3 “Snapping” continuous sizes back to library cells

After the continuous model is optimized, the continuous sizes need to be mapped, or snapped, back to the available library cells. Intuitively, this is challenging when the number of library cells is sparse⁷, however [75] reports that even in dense libraries, with cell sizes of \{1, 2, 3, 4, 6, 8, 12, 16, 24, 32\}, snapping to the nearest library cell results in slack violations and infeasible designs.

Moving from a continuous solution to the discrete sizes is difficult, and not guaranteed to be optimal. For example, suppose that the continuous solution is used to limit each gate to two cell choices – the library cell that is larger, and the library cell that is smaller than the continuous size. This still leaves $2^{|G|}$ choices, and furthermore the optimal solution may not be one of these choices [40].

Heuristics for snapping continuous sizes are studied in [40, 129, 100, 159, 122]. Simple snapping to the closest library cell is used in [100, 122], and in [129], the gate sizing is re-performed after the $V_t$ is snapped, followed by the snapping of the gate sizes. In [40] the continuous solution via linear programming is used to limit the size options. Gates that are set at the minimum size by the continuous sizer are fixed to be at the minimum; the options for the remainder of the gates are two sizes that are smaller than, and greater to the continuous size. The algorithm then creates a state-space tree that is used to enumerate the different possibilities. They state that this process is tractable as the number of gates that are not at minimum size is “relatively small”.

[75] uses a dynamic-programming like enumeration approach. This approach traverses the graph breadth-first from primary input nodes to output nodes, enumerating all the possible solution possibilities. When the size of the enumerated space becomes too large, it is pruned using a locality-sensitive hash that diversifies the set of solutions.

4.5.4 Modeling errors

Modeling errors are a large limitation for continuous-sizing based approaches. Even with posynomial models, errors of up to 10% were re-

---

⁷ See [69] [11] for a discussion on selecting library cells
ported in [81] for a 0.25µm technology. Furthermore, these errors were reported for custom transistors; the transistors within standard cells have additional errors due to packing of the transistor within the standard cell format using transistor folding, diffusion sharing, and metal pins.

4.6 Dynamic Programming Based Algorithms

Dynamic programming [12] (DP) is an optimization method for problems with stages. The method breaks down complex problems over the stages by making a decision in the present, with the assumption that all future decisions will be optimal. For problems where the optimal future decisions can be modeled efficiently, this presents an attractive method for optimization.

In the context of gate sizing, dynamic programming methods provide a structure to break apart the problem using decision stages and cost-to-go functions. At each stage, cost-to-go functions recursively define the cost for the objective for all prior stages of the design. After all the stages have been traversed, the cost-to-go function is used to find the minimum cost solution.

Dynamic programming for gate sizing has been applied in both forward topological order [25, 75, 91], and reverse topological order [96]. In the forward topological order version of the method, the power and the arrival times are propagated in the form of cost-to-go functions:

\[
J_g(t_a) = \min_{\omega \in (g)} \left\{ p(g, \omega) + \sum_{g' \in \ell(g_k)} J_{g'}(t_a - d(g, \omega)) \right\}. \tag{4.21}
\]

The cost function returns the minimum power, as a function of the arrival time at the gate’s output. It is defined recursively as a function of the cost functions for its fanin gates, and the term \(J_{g'}(t_a - d(g, \omega))\) is the cost needed for the signal to arrive at the input by the time

\[\text{see [15] for an introduction to the subject}\]

\[\text{Note that [25] does not explicitly refer to itself as Dynamic Programming, but has the characteristics of DP.}\]
4.6. Dynamic Programming Based Algorithms

$J_{g_1}(t_a) = \begin{cases} 2 & \text{if } t_a = 1 \\ 1 & \text{if } t_a = 2 \end{cases}$

$J_{g_2}(t_a) = \begin{cases} 4 & \text{if } t_a = 2 \\ 3 & \text{if } t_a = 3 \\ 2 & \text{if } t_a = 4 \\ 1 & \text{if } t_a = 5 \end{cases}$

$J_{g_3}(t_a) = \begin{cases} 6 & \text{if } t_a = 3 \\ 5 & \text{if } t_a = 4 \\ 4 & \text{if } t_a = 5 \\ 3 & \text{if } t_a = 6 \end{cases}$

Fig. 4.1 Forward topological order dynamic programming example. A simple delay vs power model is used for the gates, where either Delay=1 and Power=2, or Delay=2 and Power=1.

$t_a - d(g, \omega)$, which is the given arrival time minus the delay at the current gate. To simplify notation, we can assume that $J_g(t_a) = \infty$ when $t_a < 0$, thus any infeasible arrival time will have a cost $\infty$.

An example of the forward version is shown in Figure [4.1]. In this example, a simple delay model is used with two cell options for each gate: either Delay=1 and Power=2, or Delay=2 and Power=1. At $g_1$, this means that the arrival time of the output ($t_a$) can be 1 with a power cost of 2 ($J_{g_1}(1) = 2$), or the arrival time can be 2 with a power cost of 1. At $g_2$ the computation becomes more complex – $J_{g_2}$ is computed as a function of its fanin, $J_{g_1}$. $J_{g_2}(2)$ can be achieved only using the delay 1 cell option, for $g_2$, and using $J_{g_2}(1)$, for a cumulative power cost of 4. Similar computations can be made for $J_{g_2}(3)$, however in this case the minimizer is not unique – either of the two cells, $g_1$ or $g_2$ can be at the minimum delay option. At the primary output, the cost-function is evaluated with the delay constraint to yield the optimal power, and the circuit can be traversed backwards to find the cell options that produce the minimum power design.

In reverse-topological order, the procedure works from the primary outputs towards the primary inputs. The power and the required arrival
Methods for discrete gate sizing

Fig. 4.2 reverse topological order dynamic programming example. A simple delay vs power model is used for the gates, where either Delay=1 and Power=2, or Delay=2 and Power=1. The required arrival time at the primary output is assumed to be 4.

\[ J_g(t_r) = \begin{cases} 6 & \text{if } t_r = 1 \\ 5 & \text{if } t_r = 0 \end{cases} \]

\[ J_{g'}(t_r) = \begin{cases} 4 & \text{if } t_r = 2 \\ 3 & \text{if } t_r = 1 \\ 2 & \text{if } t_r = 0 \end{cases} \]

\[ J_{g_i}(t_r) = \begin{cases} 2 & \text{if } t_r = 3 \\ 1 & \text{if } t_r = 2 \end{cases} \]

\[ t_r = 4 \]

An example of the reverse topological version is shown in Figure 4.2.

In this case, the cost functions are simpler than the forward topological case, as the options for negative required arrival times \( t_r < 0 \) can be eliminated. However, this can also be done in the forward topological case. Once the cost-to-go functions are propagated to the primary input, the cost-to-go function is evaluated with \( t_r = 0 \) to find the optimal configuration.

Note that in both cost-to-go functions, (4.21) and (4.23), the costs are computed by enumerating the different options at each gate; this information is usually stored in the form of a table. The size of this table can be reduced by removing options that are not Pareto-optimal, and this is implicitly done in the cost-to-go functions. For example, in
4.6. Dynamic Programming Based Algorithms

Fig. 4.3 Design with reconvergent fanout. This structure is problematic in Dynamic Programming formulations.

the forward topological case, \( t_a = 4 \) and \( p = 5 \) is certainly superior to \( t_a = 5 \) and \( p = 5 \) and the latter entry would be removed. This helps to reduce the complexity of the cost-to-go functions.

Dynamic programming based algorithms for sizing suffer from two limitations. The first limitation is the size of the solution space that must be propagated. Even after the non Pareto-optimal elements are pruned, the number of elements in the table may be very large. To combat this, [76] uses locality-sensitive hash functions to reduce the number of entries to the table. In [25, 96], the slews are ignored, reducing the number of entries to a manageable size. Furthermore, while [96] uses the dynamic programming methods for slack minimization, they rely on Lagrangian methods to perform the timing constrained power optimization (see Section 4.7). Secondly, they have difficulty with non-tree structures, e.g. when a gate has multiple paths that lead to it. This is present in the forward topological case in Figure 4.3. First, the cost-to-go function is incorrect. As written in (4.21), \( J_{g_4}(t_a = 4) = 8 \) because the cost of \( g_1 \) is double counted – it is present in \( J_{g_2} \) and \( J_{g_3} \). However, the actual minimum cost is \( J_{g_4}(t_a = 4) = 6 \).

There is a further inconsistency difficulty in the cell options that
Methods for discrete gate sizing are assigned by the DP. For a delay constraint of 4, the cell options are set by traversing the graph in reverse topological order. $g_4$ is set to the cell option that minimizes $J_{g_4}(t_a = 4)$, which is Delay=1 / Power=2 option. Next, suppose that $g_2$ is also set to Delay=2 / Power=1, but $g_3$ is set to Delay=1 / Power=2 – this is completely valid in the DP algorithm. In the perspective of $g_3$, it does not understand that there is a parallel reconvergent path through $g_2$ that it should coordinate with. This forces, $g_1$ to be Delay=1 / Power=2, and results in a total power of 7, which is greater than the optimal power 6.

[96] uses a heuristic to fix this problem by considering the problem in two steps. In the first step, when the cell options are being assigned, when multiple cell options are possible (when the minimizer in (4.21) is not unique), all of the cell options set as viable candidates for that cell. After this step is over, each gate may have multiple candidate cell options that it can be assigned to. This is reconciled in the next step, when the algorithm traverses the graph in topological order, and assigns the gate to the cell option that is locally optimal.

4.7 Lagrangian Relaxation

Lagrangian relaxation are widely used in gate sizing to perform continuous gate sizing (see, for example, [30, 128, 39, 151, 96, 77]) as an algorithm to solve the posynomial models in (4.20), and they have also been applied directly to discrete gate sizing [96]. They provide scalable methods for large-scale gate sizing, and they provide a useful way to convert constrained optimization problems into unconstrained optimization problems.

Lagrangian relaxation methods convert a constrained optimization problem into an unconstrained problem by using by using Lagrange multipliers[10], also known as dual variables, to add them to the objec-

---

For the optimization problem:

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 0, \quad i = 1, \ldots, m \\
& \quad g_i(x) = 0, \quad i = 1, \ldots, p \\
& \quad x_{\min} \leq x \leq x_{\max}
\end{align*}
\]  

(4.25)

The associated Lagrangian is:

\[
L(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i g_i(x).
\]  

(4.26)

Note that the constraints \(x_{\min} \leq x \leq x_{\max}\) do not appear in the Lagrangian (4.26). In gate sizing problems, it is more efficient to handle these constraints implicitly by restricting the domain of the function. Thus, the domain of \(x\) in (4.26) is restricted to \(x_{\min} \leq x \leq x_{\max}\).

The Lagrange dual problem associated with the Lagrangian is:

\[
\max_{\lambda \geq 0, \nu \in \mathbb{R}} \left\{ \min_{x_{\min} \leq x \leq x_{\max}} \{L(x, \lambda, \nu)\} \right\}.
\]  

(4.27)

This is the result of minimizing the Lagrangian over the primal variables, \(x\), and maximizing over the dual variables, \(\lambda\) and \(\nu\). This formulation is interesting when the minimization \(\min_{x_{\min} \leq x \leq x_{\max}} \{L(x, \lambda, \nu)\}\) can be performed faster than solving (4.25) directly.

The main motivation for solving the Lagrange dual problem (4.27) comes from two theoretical results:

- The solution to the dual problem (4.27) always less than the primal (4.25)
- The solution to the dual problem (4.27) is equal to the solution of the primal (4.25) for a wide class of convex continuous problems.\(^\text{11}\)

\(^{11}\)A sufficient, but not necessary, condition for equality comes from Slater’s condition:

1. \(x\) is continuous, and the domain of \(x\) is convex, connected set.
2. \(f_i\) is convex \(\forall i\).
3. \(g_j\) is linear \(\forall j\).
4. There is an \(x\) such that \(f_i < 0 \forall i\), and \(x_{\min} < x < x_{\max}\).
Using the Lagrange dual problem is said to be using Lagrangian relaxation\textsuperscript{12}, and it is a useful method to find lower bounds or optimize difficult problems. Generally speaking, these methods perform well when there is an efficient way to minimize $L(x, \lambda, \nu)$ over $x$ for fixed $\lambda$ and $\nu$.

Gate sizing is one of these problems that benefit from Lagrangian relaxation. In this case, the associated Lagrangian for (4.20) is:

$$L(x, t_a, \lambda) = \sum_{g \in G} p_g w_g +$$

$$\sum_{g} \sum_{g' \in \text{lo}(g)} \lambda_{g,g'} (t_{a(g)} + d(g, w) - t_{a(g')}) +$$

$$\sum_{g \in \text{PO}} \lambda_g (t_{a(g)} - T_{\text{max}})$$  \hspace{1cm} (4.29)

The associated Lagrangian dual problem is:

$$\max_{\lambda \geq 0} \left\{ \min_{t_a, w_{\text{min}} \leq w \leq w_{\text{max}}} \left\{ L(x, t_a, \lambda) \right\} \right\}$$  \hspace{1cm} (4.31)

In this expression, the minimization over $t_a$ is equal to $-\infty$ for certain values of $\lambda$. This can be seen by rewriting (4.34) as:

$$L(x, t_a, \lambda) = \sum_{g \in G} \left( \sum_{g' \in \text{lo}(g)} \lambda_{g,g} - \sum_{g' \in \text{hi}(g)} \lambda_{g',g} \right) \cdot t_{a(g)} + ...$$  \hspace{1cm} (4.32)

where the terms that do not involve $t_a$ are omitted. If any of the coefficients of $t_a$ are non-zero, then minimizing $L(x, t, \lambda)$ would yield $-\infty$ as the $t_a$ are free variables and are unrestricted. Positive coefficients would have $t \to -\infty$ and negative coefficients would have $t \to \infty$. Thus, the $\lambda$ is restricted to the dual feasible set that satisfies:

$$\sum_{g' \in \text{lo}(g)} \lambda_{g,g'} - \sum_{g' \in \text{hi}(g)} \lambda_{g',g} = 0$$  \hspace{1cm} (4.33)

For dual feasible $\lambda$, the Lagrangian reduces into a weighted delay-

\textsuperscript{12}See [SS] for a good reference on Lagrangian relaxation
power minimization:

\[
L(x, t_a, \lambda) = \sum_{g \in G} p_g w_g^+ + \sum_{g} \left( \sum_{g' \in \text{lo}(g)} \lambda_{g,g'} \right) d(g, w)^+ + \left( \sum_{g \in \text{PO}} \lambda_g \right) \cdot T_{\text{max}}.
\]

(4.34) \hspace{1cm} (4.35) \hspace{1cm} (4.36)

For continuous gate sizes and \(V_t\), there are very good methods to solve this problem (see \[120\]) simply by cycling through each gate and choosing the gate width that minimizes the power and weighted delay combination while keeping all the other gate sizes fixed.

This was also applied to discrete problems in \[96, 77\]. In this case, the formulation is identical to the continuous sizing case. However, the power and weighted-delay minimization is performed by enumerating the different cell options and choosing the best one. \[96\] uses Lagrangian Relaxation in conjunction with dynamic programming, while \[77\] uses only the Lagrangian Relaxation.

### 4.8 Slew targeting methods

Gate sizing can also be performed with slew targets \[73\]. This works on the idea that the delay is a monotonic function of the slew; thus improving the slew will also improve the delay. This is an interesting perspective on the timing closure problem to use the slews targeting as a proxy for timing closure.

This method is also one of the few discrete sizing methods that deal with slew directly. The most other methods ignore the slew, because it creates long range interactions between gates that is difficult to model (see Chapter 2). However, slew has a large effect on the delays and ignoring slews may introduce further sub-optimality into the design.

This method works by alternating between assigning slew targets, and optimizing the gates for the given slew targets, as shown in Algorithm 5. The slew targets are assigned as a function of the slacks.
Gates with negative slack will have their slack targets increased, while gates with positive slacks will have their slack targets decreased.

Central to the slack adjustment is the concept of \textit{local criticality}, lc(\(g\)). This is defined as the difference between the slacks of the input pins, and the minimum of the slacks of its fanins:

\[
\text{lc}(g) = \max_{g' \in \text{in}(g)} \{ s(g) - s(g'), 0 \}. \tag{4.37}
\]

A value lc(\(g\)) = 0 means that the gate is at least as critical as any of its fanins. However, a value of lc(\(g\)) > 0 implies that there is a fanin gate that is more critical and thus the current gate does not contribute to the worst case slack.

Following this rationale, gates with negative slack and lc(\(g\)) = 0 have their slew targets decreased, or tightened, to reduce timing violations. All other gates have their slew targets increased, to help recover power. These increases or decreases are changed as a function of the local criticality lc(\(g\)), slack s(\(g\)), damping factor \(\Theta_k\), and a constant \(\gamma\), which is related to \(\frac{\partial r(g)}{\partial s(g)}\).

In Step 3 of the algorithm, the slew targets are refined. This is to reduce slack targets that are unnecessarily high, when “cells cannot be enlarged further, or to locally non-critical cells that cannot be downsized sufficiently because of too large successors” \[73\]. To fix this, the slew target is reduced as by a factor of \(\lambda\) times the difference between the estimated slew of the most critical input pin:

\[
\tau_g^\text{target} = \tau_g^\text{target} + (\lambda) \cdot (s^p - \tau_g^\text{target}) \tag{4.38}
\]

The author reports that this method is very fast – 5.8 million cells are sized within 2.5 hours on a 3.0 GHz Xeon Server. Furthermore, they report good results for timing closure – an average worst negative slack of 6\% is reduced to 2\% after applying their algorithm.

\section{4.9 Linear programming based assignment methods}

Linear programming can also be used as an assignment procedure to assign gates to specific cell options \[34, 87, 5\]. In contrast to the continuous sizing methods in Section 4.5.1 where the variables represent the continuous gate sizes, each variable in this context is a binary variable,
/* Θ_k = step size & approximation parameter */
/* lc = local criticality */

foreach k ∈ {1, ..., k_{max}} do
  α_k = \frac{1}{\log(k+1)}

foreach g ∈ G do
  lc(g) = \max_{g' ∈ fi(g)} \{s(g) − s(g'), 0\}
  if (s(g) < 0) \&\& (lc(g) == 0) then
    \Delta_{\text{target}} = -\min \\{ (\Theta_k \cdot \gamma \cdot |s_{g'}|, \Delta_{\text{target}}) \}
  else
    sm = \max\{s(g), lc(g)\}
    \Delta_{\text{target}} = \min \\{ (\Theta_k \cdot \gamma \cdot |sm|, \Delta_{\text{target}}) \}
  end
  \tau_{\text{target}} = \tau_{\text{target}} + \Delta_{\text{target}}

end /* Project the slew target to feasible range */
\tau_{\text{target}} = \max\{\tau_{\text{target}}, \Delta_{\text{target}}^{\min}\}
\tau_{\text{target}} = \min\{\tau_{\text{target}}, \Delta_{\text{target}}^{\max}\}

end /* Step 2: Apply the slack targets */

foreach g ∈ G in reverse topological order do
  foreach ∀g' ∈ fi(g) do
    /* Create input slew estimates */
    \hat{\tau}(g') = \Theta_k \cdot \tau_{\text{target}}^{g'} - (1 - \Theta_k) \cdot s_{g'}
  end

Using the estimated input slews \hat{\tau}(g') set \omega_g as:
the minimum power cell option \omega that satisfies
\tau(g, \omega) > \tau_{\text{target}}
end /* Step 3: Refine slew targets */

foreach g ∈ G in topological order do
  \hat{\tau}(g') = \Theta_k \cdot \tau_{\text{target}}^{g'} - (1 - \Theta_k) \cdot s_{g'}
  s^p = \max\{g' | s_{g'} = s_{g}\} \{\hat{\tau}(g')\}
  \tau_{\text{target}} = \tau_{\text{target}} + (\lambda) \cdot (s^p - \tau_{\text{target}})
end

end

Algorithm 5: Slew target method in [73]
Methods for discrete gate sizing

\[ x_{g \leftarrow \omega} = \begin{cases} 1 & \text{if gate } g \text{ is assigned to cell } \omega \\ 0 & \text{otherwise} \end{cases} \]  

(4.39)

A value of \( x_{g \leftarrow \omega} = 1 \) means that gate \( g \) is implemented by the library cell \( \omega \). These variables are referred to as \textit{assignment variables} as they assign gates to library cells. In this method, the power objective is written as:

\[ \sum_{\forall g \in G} \sum_{\omega \in \Omega_g} \Delta p_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \]  

(4.40)

where

\[ \Delta p_g(\omega; \omega_0) = p_g(\omega) - p_g(\omega_0) \]  

(4.41)

The power consumption of gate \( g \) when implemented by the library cell \( \omega \) (\( \omega_0 \) is the current implementation of the gate). This power can be the leakage power or the dynamic power, or a weighted combination of the two. As the short circuit portion of the dynamic power is dependent on the input slew and output load, the current slew and load values are used to evaluate the \( \Delta \) terms. Although this may introduce modeling errors if the input gates change, this is still a good proxy for the actual power.

The delay constraints are written using into the linear program block-based formulation\(^\text{13}\):

\[ t_{a(g)} + \sum_{\omega \in \Omega_g} \Delta d_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \leq t_{a(g')}, \quad \forall g' \in \text{fanout}(g) \] 

\[ 0 \leq t_{a(g)}, \quad \forall g \]  

(4.42)

\[ t_{a(g)} \leq T_{\max}, \quad \forall g \in \text{PO} \]  

(4.43)

The \( t_{r(g)} \) are variables that are used to model the arrival times at the input of gate \( g \) and the \( \Delta d_g(\omega; \omega_0) \) term is the change in delay when the gate is changed from cell option \( \omega_0 \) to \( \omega \). This can be computed by trying the cell option and computing the change in the arrival times.

\(^{13}\)For simplicity, the difference between the rise and fall times are omitted, as well as the difference in delay for different input-output paths in a gate. However, they can easily be incorporated by adding separate rise and fall arrival-time variables and additional delay models for different the input-output paths in a gate.
As an improvement, the change in slack may also be used to model downstream effects on the delay [87].

Combining the power and delay terms results in the linear programming problem:

\[
\begin{align*}
\text{minimize} & \quad \sum_{g \in G} \sum_{\omega \in \text{CellOptions}(g)} \Delta \text{Power}_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \\
\text{subject to} & \quad t_a(g) + \sum_{\omega \in \text{CellOptions}(g)} \Delta \text{Delay}_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \leq t_a(g') \\
& \quad 0 \leq t_a(g), \forall g \\
& \quad t_a(g) \leq T_{\text{max}}, \forall g \in \text{PO} \\
& \quad 0 \leq x_{g \leftarrow \omega} \leq 1, \forall g, \omega \in \text{CellOptions}(g) \\
& \quad \sum_{\omega \in \text{CellOptions}(g)} x_{g \leftarrow \omega} \leq 1, \forall g 
\end{align*}
\]

(4.44)

Note that the assignment variable was originally intended to be binary – the variable should only be 0 or 1. However, to improve the runtime of this problem, it is relaxed so that the assignment variable is continuous between 0 and 1.

Once (4.44) is solved, then the next step is to map the \(x_{g \leftarrow \omega}\) into gate cell changes. In [34], the authors apply any \(x_{g \leftarrow \omega} > 0.99\). However, due to the delay interactions, this may cause timing violations. They correct these timing violations by running a linear programming assignment for minimizing timing:

\[
\begin{align*}
\text{minimize} & \quad \max\{t_a(\text{output}), T_{\text{max}}\} + \ldots \\
& \quad \epsilon \sum_{g \in G} \sum_{\omega \in \Omega_g} \Delta \text{Power}_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \\
\text{subject to} & \quad t_a(g) + \sum_{\omega \in \Omega_g} \Delta \text{Delay}_g(\omega; \omega_0) \cdot x_{g \leftarrow \omega} \cdots \leq t_a(g') \\
& \quad 0 \leq t_a(g), \forall g \\
& \quad t_a(g) \leq t_a(\text{output}), \forall g \in \text{PO} \\
& \quad 0 \leq x_{g \leftarrow \omega} \leq 1, \forall g, \omega \in \Omega_g \\
& \quad \sum_{\omega \in \Omega_g} x_{g \leftarrow \omega} \leq 1, \forall g 
\end{align*}
\]

(4.45)

where \(\epsilon\) is a small constant used to give more weight to the timing objective. After this is solved, and values of \(x_{g \leftarrow \omega} > 0.01\) are applied.

The main limitations with this method are with the delay model. This model does not account for the interactions in the delays between
gates in two ways. To understand the first limitation, suppose that an input gate to gate $g$ has decreased in size, diminishing its ability to drive its output gates. If the size of gate $g$ is then increased, then the delay up through $g$ will be worse than if the input had not changed. This is because the input gate is less able to handle the increase in capacitance from a increase in the gate size of $g$. Due to this limitation, [87] adds a further constraint on neighboring gates. That is, out of every pair of gates, only one gate should change at a time. This reduces the problems due to the changing capacitances of neighboring gates.

Another is the transition time (slew) interactions. Decreasing the slew of an input gate will improve the delays downstream while increasing the slew will have the opposite effect on the delay. Thus, any gate change may affect the $\Delta d_g(\omega; \omega_0)$ downstream from it. Furthermore, slow slews may improve the $\Delta d_g$ when gates are upsized, and faster slews may decrease the benefits of upsizing a gate. While this is partially captured by using the change in slack as a proxy for $\Delta d_g(\omega; \omega_0)$, it does not fully capture these effects.

Due to these limitations, [87] uses this approach for incremental gate sizing. The model has a better accuracy when the number of changes is not too large, thus it is well suited for finding a small number of gate changes that can be used to implement an Engineering Change Order (ECO).

### 4.10 Summary

The discrete gate sizing methods presented are summarized in Table 4.2. This chart indicates whether the method is well-suited (✓), applicable (⋆), or not applicable (−) in a given context.

Pre-layout means that the method is well-suited for optimization before the layout is fixed. Generally, this means that that method can make large changes to the design. In contrast, post-layout indicates that the method can be used incrementally when large changes to the design are to be avoided. Post-layout methods must be able to modify the current design, rather than start from scratch.

Power optimization indicates that the method can be used to minimize power given a timing constraint. The timing closure optimiza-
Table 4.2 Summary of discrete gate sizing methods

<table>
<thead>
<tr>
<th>Method</th>
<th>pre layout</th>
<th>post layout</th>
<th>power optimization</th>
<th>timing closure optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Score and Rank</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Slack and Delay Budgeting</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Continuous sizing based</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dynamic Programming</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Lagrangian relaxation</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Slew targeting</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
</tr>
<tr>
<td>LP-based assignment</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

The notation column indicates that the method can be used to improve timing, whether to find a minimum delay design, or to meet a timing constraint.
Comparing discrete gate sizing methods is a tricky task because there are many different contexts in which gate sizing is used (see Section 1.2 for more information):

- **Post-synthesis**: after the design is mapped to logic blocks
- **Post-placement**: after the locations of the cells are known, along with approximate information on the lengths of connecting wires.
- **Post-layout**: after routing and placement are completed. At this step, the layout, along with information on wire parasitics, is known.

Each of these contexts has differing flexibilities, and accuracy requirements. For example, post-synthesis gate sizing has the most flexibility, as the gates are not yet placed (given locations), and also has the loosest accuracy requirement. In contrast, post-layout gate sizing has the least flexibility, as each change in the cell may require moving and/or rerouting other cells. Furthermore, there is a high accuracy requirement – after this stage, the design must be timing feasible.
Along with these different contexts is the experimental setup. The library, buffering, timing model, benchmark type, timing constraints, and power measurements all affect the resulting sizing. For example, a small library granularity tests a method’s ability to handle only a few cells that are available of each type. Benchmarks without sequential elements (combinatorial benchmarks) generally have different distributions of critical paths than sequential benchmarks. Tighter timing constraints may test a method’s ability to create timing feasible designs.

These topics will be considered in this section.

5.1 Setting up gate sizing experiments

In gate sizing, there is a significant number of variables that affect the success of a gate sizing method. In this section we will discuss:

(1) Standard Cell Library – what cell options are available for each gate?
(2) Benchmarks and synthesis options – what are the characteristics of the test circuit?
(3) Timing constraints – how much timing slack is available?

5.1.1 Standard Cell Library

The most important factor in gate sizing experiments is the standard cell library. This is an even larger factor than the benchmark itself – the performance of a sizing algorithm will vary on the same benchmark under different standard cell libraries.

The two major qualities of a library that impact the gate sizing experiment are the range of the cells and the number of the cells. The range of the cells determines the range of the gate sizing method. Gates that have a small range, such as those with cells of 1X to 2X the minimum gate width, can provide only small impacts, as the sizing method can increase gate sizes to just 2X for large fanout loads. Most cell libraries will provide large ranges for popular gates and those used for buffering large fanouts (such as inverters and buffers). However, less popular gates (such as the AND-OR-INVERT) may have a small
Comparing discrete gate sizing methods

range, and the flexibility at these cells is very limited.

The granularity of a library impacts different algorithms differently. A library that is dense, with many gate sizes for each gate footprint, will generally work better on methods that are derived from continuous sizing methods. This is because the increased granularity reduces the penalty for rounding and snapping - rounding from 5.2X to 5X is much better than rounding down to 4X.

In contrast, other methods may fare poorer on dense libraries. Dynamic programming based algorithms may suffer from significant slowdowns due to the increase number of cell options. Greedy methods, which are heuristics, may fare poorer than their continuous sizing based counterparts. The granularity question – is the library sparse or dense – may cause one method to perform better than another.

However, there is no typical library density or range. One 65nm commercial library provides 11 versions of the buffer and inverter cells, and 4 versions of other cells. The Nangate Library [3], which is used for testing and research, provides 5 versions of the buffer and inverter cells, and 3 versions of other cells. In contrast, another 45nm library provides 20 versions of the buffer and inverter cells (ranging from .5X to 16X), and between 5-11 versions of other cells (ranging from .5X to 6X). In addition, there are 3 threshold voltage options for each cell. Therefore, it is important to understand the library that will be targeted before developing a gate sizing method.

5.1.2 Benchmarks and synthesis options

The type of benchmark and the synthesis options are the second most important factor in comparing gate sizing methods. The benchmarks are usually designs that have interesting sizes and topologies. For example, the ISCAS ‘85 benchmarks were combinatorial designs representing controllers, arithmetic logic units, and error correcting circuits [70]. The ISCAS ‘89 benchmarks are sequential designs[1] and are sections of real chips, traffic light controllers, and digital fractional multipliers [22].

At a high level, these benchmarks differ by the number of gates, the depth of their logic, the delay distributions of their paths, and

[1] These designs include flip-flops.
5.1. Setting up gate sizing experiments 79

The average number of fanouts and fanins for each of the gates. These metrics, however, say little of what kind of sizing method will work, or how difficult the design will be to size.

A complicating factor is the effect of the synthesis tool on the design. Table 5.1 shows the effect of different timing constraints on the resulting netlist. The leftmost columns represent synthesis for minimum delay, and the rightmost columns represent synthesis for minimum power. The delay spread between fastest and slowest synthesis outputs vary from 0.19ns to 6.9ns, depending on the benchmark. The size of the resulting netlists vary from 25 gates to 1309 gates, which means that the number of gates can double, depending on the design!

The difference between the synthesized outputs is the amount of buffering, and the type of gates that are used. When the timing constraint is generous, more complex gates are used, such as full adders, half-adders, or-and-inverts, and-or-inverts and so on. These gates are more power efficient that their lower level representations, but they do not offer the kind of flexibility that can be used to improve the delay. Conversely, tighter timing constraints will result in designs with many more inverters and buffers, and also more basic building blocks, such as not-ands.

An odd consequence of this effect is that synthesizing designs for

| c432  | 0.431 | 118  | 0.494 | 137  | 0.619 | 93   | 0.188 | 25   |
| c499  | 0.477 | 586  | 0.612 | 322  | 0.881 | 189  | 0.404 | 397  |
| c880  | 0.408 | 346  | 0.562 | 237  | 0.871 | 212  | 0.463 | 134  |
| c1355 | 0.473 | 598  | 0.599 | 356  | 0.850 | 222  | 0.377 | 376  |
| c1908 | 0.802 | 676  | 0.863 | 571  | 0.990 | 533  | 0.188 | 143  |
| c2670 | 0.610 | 1041 | 0.787 | 791  | 1.140 | 748  | 0.530 | 293  |
| c3540 | 1.096 | 1615 | 1.289 | 1359 | 1.675 | 1107 | 0.579 | 508  |
| c5315 | 1.090 | 2019 | 1.226 | 1867 | 1.498 | 1766 | 0.408 | 253  |
| c6288 | 1.845 | 3089 | 4.127 | 1656 | 8.692 | 1780 | 6.847 | 1309 |
| c7552 | 0.866 | 2894 | 0.977 | 2640 | 1.200 | 2481 | 0.334 | 413  |

Table 5.1 Effect of synthesis on the ISCAS ’85 benchmarks (adapted from [87])
Comparing discrete gate sizing methods

5.1.3 Timing constraints

The last factor that affect the gate sizing experiment is the timing constraint. ???

minimum delay provides the best examples for gate sizing. These designs not only have the most number of gates, but the most gates that have multiple cell options. For example, while there may only be 1 full-adder size, there are several inverter and nand cell sizes to play with. However, the effect of synthesis on the end design after place and route and additional optimizations is not clear. Table 5.2 shows the effect of different timing constraints on synthesis given in [78]. A larger slack at the

<table>
<thead>
<tr>
<th>Timing Constraint used @ synthesis</th>
<th>With a 2.0ns Clock</th>
<th>@ signoff</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>After Place and Route Optimizations</td>
<td></td>
</tr>
<tr>
<td>$T_{\text{max}}$</td>
<td>slack</td>
<td>$T_{\text{max}}$</td>
</tr>
<tr>
<td>1.60</td>
<td>1.829</td>
<td>0.171</td>
</tr>
<tr>
<td>1.80</td>
<td>1.912</td>
<td>0.088</td>
</tr>
<tr>
<td>1.90</td>
<td>1.888</td>
<td>0.112</td>
</tr>
<tr>
<td>1.95</td>
<td>1.926</td>
<td>0.074</td>
</tr>
<tr>
<td>2.00</td>
<td>1.912</td>
<td>0.088</td>
</tr>
<tr>
<td>2.10</td>
<td>1.912</td>
<td>0.088</td>
</tr>
<tr>
<td>2.20</td>
<td>1.88</td>
<td>0.120</td>
</tr>
<tr>
<td>2.40</td>
<td>1.838</td>
<td>0.162</td>
</tr>
</tbody>
</table>

Table 5.2 Effect of timing constraint in the synthesis on the final layout (adapted from [78]) for an AES encryption circuit from [1]. After synthesis, a timing constraint of 2.0ns is used.

5.2 Post-layout gate sizing considerations

Post-layout discrete gate sizing is done after cell placement, interconnect routing, and clock tree synthesis (see [87]). It is performed to eliminate any violations and modify the design to meet specifications. For
5.2. Post-layout gate sizing considerations

example, it may be performed to fix timing violations, signal integrity issues, maximum capacitance violations, or hold time violations [72].

5.2.1 Post-layout timing considerations

Because this is done after the placement and routing steps, the delay estimates utilize (see Chapter 2):

1. information about the interconnect, and interconnect parasitics;
2. the clock distribution tree with its associated skews;
3. the metal fills, effects of lithography on the gate length, and stress parameters.

These models generally do not have convexity properties, nor do they have simple analytical expressions. Thus, commercial timers use detailed approximations to ensure that the resulting delays are accurate.

Due to tradeoffs between accuracy and speed, timing engines used at the place and route stage use a faster (albeit less accurate) timing engine than that used at signoff. Furthermore, the parasitic extraction used at place and route generally is not as detailed as the signoff parasitic extraction [78]. This mismatch may result in iterating between design modification, and signoff.

An example of this mismatch is shown in Figure 5.1. Though the timing estimates correlate well, the values differ by a substantial margin. The error is as large as .2ns, and in many cases the place and route timer underestimates the delay, which may lead to multiple cycles of validation. This deviation increases the difficulty of the timing closure problem, as the tool used for sizing, placing and routing does not match with the tool that is used for verification. Thus, iterations of design changes in the form of Engineering Change Order (ECO), and signoff may need to be performed to create a timing feasible design.

Another problem that arises from this mismatch is the potential for suboptimality. While the signoff and verification may be successful, the design tool may be overly pessimistic if the timing estimates of the place and route timer are significantly less than the signoff timer. There may be pessimism in the timing for each path, as well as the
Comparing discrete gate sizing methods

Fig. 5.1 Mismatch between the timing estimates at place and route and at signoff. The gray line indicates when the two estimates match. Data adapted from, and courtesy of [78].

total worst-case delay, and minimum cycle time. As the discrepancy grows, the potential for this suboptimality also grows.

5.2.2 Incremental placement and routing considerations

Another larger consideration is the need for incremental placement and routing. This may be needed if the cells increase in size or the connection pins change location. While this is not important in the \( V_t \) assignment or gate-length biasing contexts, this is important in the gate sizing context where the cell size and the pin locations may change.

Incremental placement and routing is usually done after a series of gate sizing changes\(^2\) and may affect other aspects of the design. The change in cell via gate sizing may change the locations of the pins and require rerouting. This may also require neighboring gates to be moved, which may also require rerouting. After the incremental placement and routing, the design may again become timing-infeasible, and require an

\(^2\)While it can be done after each gate is changed, it is more efficient to handle these changes at one time, performing an incremental placement, followed by an incremental routing after a series of changes are made.
5.2. Post-layout gate sizing considerations

5.2.3 Measuring an Engineering Change Order (ECO)

In the post-layout context, there is an additional consideration of minimal design disturbances – while it is important for the design to have a minimal power, it is also important to consider the implementation cost. This is especially important when the verification process has already begun. Larger ECOs may require more time to implement and verify, and may cause unwanted changes that will lead to additional ECOs.

Examples of ECOs are shown in Figure 5.3. The changes are marked in black. The changes in Figure 5.3(a) are small compared to the changes in Figure 5.3(c). [87] proposes that this changed area be measured as an ECO area cost.

Another consideration proposed in [87] is the use of a ECO timing cost. This is the number of positive-slack output pins in a design that had their timing signal changed. These pins did not need any changes in their signal as they were positive-slack, thus, changes in their signal
Comparing discrete gate sizing methods

Fig. 5.3 Visual example of ECOs. The changed cells, wires and vias are marked in black.

Comparisons

5.3.1 Comparisons reported in the literature

The comparisons reported in the gate sizing literature are difficult to navigate. Most $V_t$ assignment papers ([137, 152, 136, 109, 138]) report improvements over no $V_t$ assignment. Similarly, [66] shows improvements over no gate-length biasing, and the improvements available from adding additional threshold voltages and gate-length variants. The re-

(a) $c_{area} = 27\mu m^2$, benchmark s38417  (b) $c_{area} = 277\mu m^2$, benchmark mult

(c) $c_{area} = 1159\mu m^2$, benchmark s35932
5.3. Comparisons

<table>
<thead>
<tr>
<th>method</th>
<th>comparison</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP+LP [97]</td>
<td>Budgeting [109]</td>
<td>min 1%</td>
</tr>
<tr>
<td>LSH [75]</td>
<td>Greedy [47]</td>
<td>9%</td>
</tr>
<tr>
<td>Continuous based [129]</td>
<td>Greedy [136]</td>
<td>6%</td>
</tr>
<tr>
<td>Global Sizing [47]</td>
<td>Greedy [131]</td>
<td>5.3%</td>
</tr>
<tr>
<td>Continuous based [40]</td>
<td>Greedy [94]</td>
<td>0%</td>
</tr>
<tr>
<td>Continuous based [122]</td>
<td>Com.¹</td>
<td>1%</td>
</tr>
<tr>
<td>LP assignment [31]</td>
<td>Com.¹</td>
<td>-3.6%</td>
</tr>
</tbody>
</table>

¹ These methods were compared against the results of a commercial synthesis tool.

Table 5.3 Improvements reported in literature

- The experiments are run on the Nangate Open Cell Library [3] and a commercial 65nm library. The static timing analysis tool used in this work is the UCLA Timer [3], which is based on Open Access [4].

5.3.2 Comparisons using UCLA Timer

Comparisons of gate sizing methods can be found in [105]. In this comparison, an LP assignment [109] and the LR+DP method [96] are compared against a feasible-start greedy algorithm (see Algorithm 1) that uses a $\Delta$Power/$\Delta$Slack sensitivity function to convert the slack in a minimum delay design into power savings. The experiments are run on the Nangate Open Cell Library [3] and a commercial 65nm library. The static timing analysis tool used in this work is the UCLA Timer [3], which is based on Open Access [4].

The results are shown in Table 5.3. In this table, positive values denote improvement over a timing-feasible greedy method. The improvements are generally small – on the order of 1% – with a maxi-

³Available at http://www.nanocad.ee.ucla.edu/Main/DownloadForm.
⁴See http://www.si2.org.
Comparing discrete gate sizing methods

<table>
<thead>
<tr>
<th></th>
<th>Nangate 45nm</th>
<th>Commercial 65nm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w$</td>
<td>$w$</td>
</tr>
<tr>
<td></td>
<td>LP</td>
<td>LR</td>
</tr>
<tr>
<td>c1355</td>
<td>0.20%</td>
<td>-0.56%</td>
</tr>
<tr>
<td>c1908</td>
<td>0.47%</td>
<td>1.41%</td>
</tr>
<tr>
<td>c3540</td>
<td>-0.20%</td>
<td>0.40%</td>
</tr>
<tr>
<td>c432</td>
<td>-0.20%</td>
<td>0.07%</td>
</tr>
<tr>
<td>c5315</td>
<td>0.11%</td>
<td>1.08%</td>
</tr>
<tr>
<td>c7552</td>
<td>0.37%</td>
<td>0.27%</td>
</tr>
<tr>
<td>b15</td>
<td>8.55%</td>
<td>8.54%</td>
</tr>
<tr>
<td>b17</td>
<td>6.50%</td>
<td>6.50%</td>
</tr>
<tr>
<td>b18</td>
<td>10.46%</td>
<td>10.46%</td>
</tr>
<tr>
<td>b20</td>
<td>0.30%</td>
<td>0.23%</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>2.66%</td>
<td>2.84%</td>
</tr>
</tbody>
</table>

Table 5.4 Comparisons using UCLA timer. Percentages denote improvements over a feasible-start greedy algorithm (see Section 4.3.1).

The minimum improvement of 10.46% on the b18 benchmark using the Nangate Library. The small sizes of the improvements suggests that the sizing method
In nanometer designs, the effects of variability are too large to ignore. This is best seen by examining the ITRS Roadmap 2009, which is a report sponsored by semiconductor companies that is designed to identify future challenges. This report predicts that in 2011:

- 11% - Parametric variability effects on sign-off delay
- 20% - \(V_t\) variability
- 60% - Performance variability
- 88% - Total power variability
- 255% - Leakage power variability.

Furthermore, this variability is projected to increase. Figure ?? shows the variability data through the year 2016. The increase is the greatest for the leakage power, which is projected to increase at a rate of 24% per year!

This has motivated statistical gate sizing methods to mitigate the “soaring leakage variability”. These methods use statistical models for delay and power to create designs that are robust against varia-

---

6

Statistical gate sizing
Fig. 6.1 Projected variability in key design parameters (adapted from [2]).

6.1 Motivating examples

6.2 The slack wall

An excellent example for statistical design is the “slack wall” [9]. In regular deterministic design, there is an incentive to make the delay for each path equal to the maximum allowed delay.

1. Paths slower than the maximum delay violate the timing and are therefore not allowed.
2. Paths faster than the maximum delay are not optimal – they can be slowed down to save power.

This has the effect of creating a “slack wall” – a large number of paths having a small positive slack.

While these slack walls are optimal from a deterministic point of view, they are terrible from a deterministic point of view. The minimum

\footnote{For an in-depth discussion of statistical variations and modeling, see [90].}
6.2. The slack wall

Fig. 6.2 Effects of a “slack wall” on statistical slack. The pdf for the worst-case slack is shown for 1, 10, 50 and 100 paths. The delay of each path is Gaussian with variance 100ps.

slack over all the paths will be worse than the slack of each one of its paths. This effect is illustrated in Example 6.1.

Example 6.1. Suppose that a 300ps guard-band is used to ensure a high yield. Also suppose that the delay variation of each path has a standard deviation of 100ps. Thus, the yield would be expected to be the 3-σ value 99.98%.

This is true when the paths are completely dependent, but it underestimates when the paths are not completely dependent. For example, Figure 6.2 plots the statistical slack probability density distribution (PDF) as a function of the number of independent paths. As the number of independent paths increases, the slack distribution shifts towards the negative slack direction. When there are 100 independent paths, the yield drops to 92% – a huge decrease from 99.98%.

This example shows that deterministic optimization may be overly optimistic and result in under-design. However, there are also cases where deterministic optimization may result in over-design [149].
Statistical gate sizing

<table>
<thead>
<tr>
<th>corner</th>
<th>temperature</th>
<th>supply voltage</th>
<th>process</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast</td>
<td>0C</td>
<td>1.25V</td>
<td>FastFast</td>
</tr>
<tr>
<td>typical</td>
<td>25C</td>
<td>1.1V</td>
<td>Typical</td>
</tr>
<tr>
<td>slow</td>
<td>125C</td>
<td>0.95V</td>
<td>Slow</td>
</tr>
</tbody>
</table>

Table 6.1 Corners in the Nangate 45nm library

6.3 Worst-case corners and overdesign

Most earlier design methods relied on corner based methods for design [108, 147, ?]. The corners were process and operating condition parameters that were used for validation. For example, a fast corner might be at a low temperature, have a high supply voltage, and use a fast process. Conversely, a slow corner might operate at a high temperature, have a low supply voltage and use a slow process. An example of the corners in the Nangate Library are shown in Table 6.1

When the variations become large, as in Figure 6.1, the parametric variability of the transistors cause variations in the timing and power that are on par with the fluctuations in the operating conditions. Due to this, the fast and slow process corners become inadequate for characterizing the design. One example of this inadequacy is the notion of a worst-case corner. Traditionally, this corner is taken as a $3\sigma$ value of the process variations. However, when there are $n$ independent sources of variation, each with standard variations $\{\sigma_1, ..., \sigma_n\}$, using the $3\sigma$ value for each of these parameters will actually result in a $\sqrt{n} \cdot 3\sigma$ value [68], which is must more conservative than is needed.

Even if a correct $3 - \sigma$ value for the delay can be found, it may still be conservative due to intra-die random variations. Inter-die variations have the potential to reduce the size of the variation. For example, consider an inverter chain with $N$ gates, whose delays are random variables, $D_1, ..., D_N$ that can be generated from $N+1$ IID $\mathcal{N}(0, 1)$ random variables $X_1, ..., X_N$ as:

$$
\begin{bmatrix}
D_1 \\
\vdots \\
D_N
\end{bmatrix} = 1 + \begin{bmatrix}
\alpha & 0 & \ldots & 0 & (1 - \alpha) \\
0 & \alpha & \ldots & 0 & (1 - \alpha) \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \ldots & \alpha & (1 - \alpha)
\end{bmatrix} \begin{bmatrix}
X_1 \\
\vdots \\
X_{N+1}
\end{bmatrix}
$$
The total delay of the inverter chain is the random variable defined by $D = \sum_{i=1}^{N} D_i$. When $\alpha = 1$, $D$ is mean $N$ and standard deviation $\sqrt{N}$. On the other hand, when $\alpha = 0$, $D$ is still mean $N$ but the standard deviation is $N$. For general $\alpha$, the mean is always equal to $N$, but the standard deviation varies as $\sqrt{N^2(1-\alpha)^2 + N\alpha^2}$. The main idea here is that the correlation between gates is important, as it impacts the variance. However, the impact of the correlation is also dependent on the number of stages, $N$ – the difference between the correlated and uncorrelated cases is $N - \sqrt{N}$ and grows as the stages grow. However, this is difficult to incorporate into corners, as the number of stages is design dependent.

The last motivation for statistical methods is due to the differences in the sensitivities of gates to the variation sources. Different library cells may respond differently to variations in gate lengths, oxide thicknesses, doping, line-edge roughness, etc. This difference may make the worst-case for one cell be a different set of parameters than another. In other words, it may be impossible for two cells to be at the worst-case scenario at the same time! Thus, characterizing them at their 3-σ points may be overly pessimistic.

These criticisms are not uncontested. In [106] the author argues that the drawbacks of corner based analysis may be exaggerated. He argues that corners can account for the correlations between gates by adjusting the corner to be less conservative. Furthermore, he argues that corner based analysis may not be overly conservative for the reasons shown in Figure 6.2. A larger number of critical paths will cause the yield to decrease, and thus a conservative approach may be needed.

### 6.4 Slack-wall methods

In [9], the authors present a method to avoid the effect of the “slack wall” in Section 6.2. This slack wall was the result of a deterministic timing constraint where each of the paths is required to be less than or equal to the clock period, but there is no incentive to be any faster. Thus, many of the paths have delays that are near the clock period, and when statistical delays are added, many of the paths violate timing.

To counter this, the deterministic timing constraint is replaced with
a modified timing constraint that gives paths an incentive to be faster than needed. This is done using a penalty – at each primary output $i$, the penalty is:

$$\sum_{i \in PO} e^{-t_{ai}/\sigma}. \quad (6.2)$$

where $\sigma$ is a constant that is used to tailor the profile of the slack wall. This is then added to the power objective and the resulting problem is solved as a multi-objective problem.

6.5 SSTA based methods

One branch of statistical gate sizing methods utilized the developments in Statistical Static Timing Analysis, which adapts the traditional Static Timing Analysis methodology to account for statistical variations (see Section 2.10). [7] implements the greedy sizing heuristic from Section 4.3.1 using the modified score function:

$$\sigma_{ta,\beta} = \frac{(t_{a,\beta}(g, \omega) - t_{a,\beta}(g, \omega_0))}{\Delta p(g, \omega; \omega_0)} \quad (6.3)$$

[68] uses the statistical slack version:

$$\sigma_{s,\beta} = \frac{(s_{\beta}(g, \omega) - s_{\beta}(g, \omega_0))}{\Delta p(g, \omega; \omega_0)} \quad (6.4)$$

where $s_{\beta}$ is the $\beta$-percentile of the slack. [139] utilizes this score as well, but uses the statistical slack version of (4.5) from Section 4.3 for timing closure. This is a straightforward adaptation of prior SSTA work – by changing the timing to statistical timing, methods become upgraded to statistical sizing methods.

6.6 Gate delay approximation heuristics

Several works use a continuous sizing framework that uses approximations for the statistical gate delay [102, 38, 100, 134, 113]. The idea here is to approximate the statistical delay of the design by using the statistical library cell delays in place of the deterministic gate delay. Specifically, a percentile of the cell delay over the variation sources, $\xi$, is used:

$$d_{\beta}(g) = \text{percentile}_{\beta}(d(g, \xi)). \quad (6.5)$$
When the underlying delays are Gaussian, this is equivalent to:

\[ d_{\beta}(g) = d(g, \xi) + \kappa \sigma_d(g, \xi) \]  

where \( \sigma_d \) is the standard deviation of the delay. This is then incorporated a block-based delay formulation in Section 4.5.2.

At first glance, this looks equivalent to the corner based formulation – each gate is set to be at a “worst-case” type delay corner. However, there are two differences. The first difference is that the standard deviation of the delay is often modeled as a function of the gate width and length using Pelgrom’s model \cite{114} for the variation in \( V_t \):

\[ \sigma_{V_t} \propto \frac{1}{\sqrt{WL}}. \]  

Thus, these heuristics have the advantage of using larger gates to reduce the variation in \( V_t \), and consequently, in the delay.

The next difference is in the choice in the \( \kappa \). Worst-case corner methods apply the same value of \( \kappa \) for all gates in the design (see \cite{127}). However, this precludes the ability to adjust the \( \kappa \) values to match the actual criticalities. Some gates may influence the statistical delay heavily if it is the bottleneck of many critical paths, while other gates may effect the statistical delay less if they contribute to few paths. In \cite{101,133}, heuristic methods to tune the \( \kappa \) are presented to improve the accuracy, and to reduce the pessimism described in Section 6.3.

6.7 Convex functions of statistical delay

Another approach to sizing with a yield constraint is discussed in \cite{51} and \cite{43}. The idea here is to use convex functions of statistical delay, as they have the attractive property that there are no local minima that are not global minima.

\cite{51} minimizes the Bin-Yield Loss function:

\[ \text{BYL} = - \int_{-\infty}^{0} s \cdot \text{Prob}(\text{slack} = s)ds. \]  

where \( \text{Prob}(\text{slack} = s) \) is the probability density function of the slack. This expression is the negative of the expected value of the negative slack. It is zero when there is no delay violations, in which case the
probability of a negative slack is zero. This function is convex when the delay of the circuit is convex, and thus it can be used as a convex proxy to maximize the yield.

[43] presents a convex approximation to the yield function. Their mean-excess delay function is given as:

\[
\text{MED}_\beta = \min_z \left\{ z + \frac{1}{1 - \beta} \int_z^\infty (t - z) \cdot \text{Prob}(\text{delay} = t) \, dt \right\}, \quad (6.9)
\]

where \( \text{Prob}(\text{delay} = z) \) is the probability density function for the delay. The utility of this function is in approximating percentiles:

\[
\text{MED}_\beta \geq \text{percentile}_\beta \quad (6.10)
\]

Thus, a constraint that the \( \beta \) percentile is less than \( T_{\max} \) can be approximated as:

\[
\text{MED}_\beta \leq T_{\max}. \quad (6.11)
\]

This is preferred over the percentile constraint, as the percentile is generally not a convex function of the delay.

These two functions, the BYL and the MED are similar – when \( t \) is equal to the clock period, this function is identical to the Bin-Yield Loss function.

### 6.8 Statistical Power Considerations

All of the prior discussion in statistical gate sizing has dealt with statistical delay. However, a natural question is whether the statistical power should also be considered – is it beneficial to perform gate sizing with a statistical power objective?

The statistical power can be represented as the sum of the statistical leakage power and the statistical dynamic power random variables:

\[
P = P_l + P_d. \quad (6.12)
\]

As Figure 6.1 indicates, the variations in the leakage power dominates the total variation in power, this will be the focus of this section. The leakage power random variable is generally modeled as lognormal, as in [119], and can be expressed as a function of the gate level leakage
6.8. Statistical Power Considerations

as:

\[
P_l = \sum_{g \in G} p(g, \omega)e^{-\gamma_g \Delta V_{tg}}e^{-\eta_g \Delta L_g}
\] (6.13)

where

- \(p(g, \omega)\) is the nominal power.
- \(\Delta L_g\) is a zero-mean random variable that describes the gate length variations for gate \(g\).
- \(\Delta V_{tg}\) is a zero-mean random variable that describes the threshold voltage variations for gate \(g\).
- \(\gamma_g\) and \(\eta_g\) are fitting coefficients.

When the \(\Delta L_g\) and \(\Delta V_{tg}\) are Gaussian, e.g. normal, the expression (6.13) is log-normal. Generally, \(\Delta V_{tg}\) and \(\Delta L_g\) contain two parts:

- **Within-die, or intra-die variations** – each device in a die will see different values of \(\Delta V_{twid,g}\) and \(\Delta L_{wid,g}\).
- **Die-to-die, or inter-die variations** – each device on a die has the same value of \(\Delta V_{tdtd}\) and \(\Delta L_{tdtd}\), but this value varies between different die. Note that there is no \(g\) subscript here – all gates on the same die see the same variation.

The random variable (6.13) is difficult to work with directly. Generally, the random variable must be interpreted. For example, the mean leakage power of (6.13) can be expressed as:

\[
E[P_l] = \sum_{g \in G} p(g, \omega)e^{\gamma_g^2 \sigma_{\Delta V_{tg}}^2 / 2}e^{\eta_g^2 \sigma_{\Delta L_g}^2 / 2}
\] (6.14)

The percentile, or quantile, of (6.13) is generally difficult to express in closed form. As an approximation, the mean + 3\(\sigma\) measure is often used:

\[
E[P_l] + 3(E[P_l^2] - E[P_l]^2)
\] (6.15)

with

\[
E[P_l^2] = \sum_{g \in G} \sum_{g' \in G} p(g, \omega)p(g', \omega)E[e^{-(\gamma_g \Delta V_{tg} + \gamma_{g'} \Delta V_{tg'})}e^{-(\eta_g \Delta L_g + \eta_{g'} \Delta L_{g'})}].
\] (6.16)
This function is non-linear, and requires the computation of the covariance terms.

In [42] the benefits of using a statistical power objective are evaluated. When the mean power is used to measure statistical power, the improvements were small – on the order of 1%. For the mean + 3 σ measure, the improvements can be significant (> 5%). However, the following linear proxy measure can be used in place of the more complex non-linear mean + 3 σ measure:

\[
[\text{proxy}] = \sum_{g \in G} p(g, \omega) e^{\gamma_g \sigma \Delta V_{1, tdt} + \eta_g \sigma \Delta L_{tdt} e^{\sqrt{2(\gamma_g \sigma \Delta V_{t, g, wid} + \eta_g \sigma \Delta L_{g, wid})}/2}}.
\]

(6.17)

Using this measure as an objective can be used to optimize to within 5% of the mean + 3 σ optimum.

6.9 Statistical Delay Considerations

There are many papers that show the benefits of using statistical delay. For example, [23] cites that a 20% to 30% power improvement is gained from using statistical delay in [36, 100, 139]. Unlike the case of statistical power objectives, a statistical delay objective or constraint provides a significant improvement.

It is important to recognize the comparison point for these improvements. In [36] a 19% improvement is found over worst-case design with a 6-σ guardband; [100] provides a 31% static power improvement over a 100%-yield worst-case scenario. These comparisons with overly conservative worst-case scenarios may exaggerate the benefits of statistical optimization, but they show the benefits of preventing overdesign.

[139, 23, 148] provide comparisons between full statistical optimization, and a design method called global guardbanding (see [23]). In global-guardbanding, the optimization is performed using nominal design methods, but the timing constraint is checked using a SSTA. The SSTA is used to adjust the timing constraint and to create a design that meets the statistical delay constraint exactly. This reduces the additional costs associated with overdesigning, and the infeasibility associated with underdesigning.

Comparisons between full statistical optimization and global guard-
banding are mixed. In [139], an improvement between 15% to 35% is shown over a sensitivity based method that employs an SSTA to verify the timing constraint. In this paper, the 15% to 35% improvement must then come from their use of statistical sensitivities. However, in similar experiments in [23, 148], a 6% improvement is shown. This indicates that a majority of the improvement comes from reducing the pessimism; however, there is still a small but significant improvement that can be gained from a full statistical optimization.
Acknowledgements

We would like to acknowledge.
References

References


References


References


References


References


References


References


References


108 References


References


