Power - Performance Optimization for Digital Circuits

Radu Zlatanovici

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-164
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-164.html

December 10, 2006

Power - Performance Optimization for Digital Circuits

by
Radu Zlatanovici

B.S. (Politehnica University, Bucharest, Romania) 1999
M.S. (University of California, Berkeley) 2002

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY of CALIFORNIA, BERKELEY

Committee in charge:

Professor Borivoje Nikolic, Chair
Professor Jan M. Rabaey
Professor Daniel Tataru

Fall 2006

Power - Performance Optimization for Digital Circuits

Abstract

Power - Performance Optimization for Digital Circuits

by

Radu Zlatanovici

Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolic, Chair

In recent years, power has become the most important limiting factor for electronic circuits. This has prompted a shift in the design paradigm for digital circuits: the traditional approach of achieving the highest performance by increasing the clock frequency has been replaced by a joint optimization for both power and performance.

This thesis puts into practice the new design paradigm for the power - constrained era: design as a power - performance optimization problem. The new circuit optimization framework provides a systematic methodology for the power - performance optimization of custom digital circuits at circuit and microarchitecture levels and is demonstrated on several examples.

The circuit optimization framework formulates the design as a mathematical optimization problem that is solved numerically. The user can select a wide array of models for the underlying technology, optimization variables, design goals and design constraints, with varying impacts on the convergence speed and the accuracy of the solutions. The formulation exploits the mathematical properties of the resulting optimization problems and in some cases can guarantee the global optimality of the solutions or verify their quality against a near-optimality boundary.

Four examples are used throughout the thesis to demonstrate the circuit optimization framework:

- a 64-bit Kogge-Stone static CMOS carry tree, to demonstrate the impact of adjusting different sets of design variables (like gate sizes, supply voltage and threshold voltage) in the power - performance optimization problem;
- a detailed study of 64-bit carry lookahead adders in 90nm CMOS, to illustrate how the circuit optimization framework can be used to build intuition into the subtle tradeoffs of a particular family of circuits;
- an IEEE-compliant single-precision fused multiply-add Floating Point Unit (FPU) in 90nm CMOS, to compare the circuit optimization framework with a commercial logic synthesis tool;
- a 64-bit Kogge-Stone static CMOS carry tree, to demonstrate how process variations can be included in a yield-constrained power - performance optimization.

One circuit from the 64-bit adder family in the second example has been built in silicon in order to confirm the correct operation of the framework through measurement. The 90nm chip performs single-cycle 64-bit additions in 240ps and consumes 260mW at the nominal supply voltage of 1V.

Professor Borivoje Nikolic, Chair

To my parents,

Dan and Rodica Zlatanovici

Table of Contents

List of Figures

List of Tables

Acknowledgments

Many people have helped me throughout the six years of graduate school and I wish to thank them all.

First and foremost I express my sincerest gratitude to my advisor, prof. Borivoje (Bora) Nikolic. Choosing a research advisor is the single most important decision a graduate student must make. I started to work with Bora in my first semester at Berkeley and looking back, there could not have been a better choice. This work would have not been possible without his support, guidance and vision. Thanks for everything, Bora - and I mean it!

Thanks to prof. Jan Rabaey and prof. Daniel Tataru for taking the time to read my dissertation, a long and probably boring manuscript. Thanks to my mentors Phil Strenski (IBM T.J. Watson Research Center) and Andrei Vladimirescu for broadening my field of understanding. Thanks to Ram Khrishnamurthy, Sanu Matthew and Stefan Rusu of Intel Corp. for the technical discussions and advice.

Thanks to my colleagues in 403 Cory and the BWRC for creating an enjoyable and productive environment: Engling Yeo, Socrates Vamvakos, Stephanie Augsburger, Ben Wild, Dejan Markovic, Yasuhisa Shimazaki, Josh Garrett, Bill Tsang, Liang-Teck Pang, Fujio Ishihara, Sean Kao, Farhana Sheikh, Zhengya Zhang, Zheng Guo, Renaldi Winoto, Sebastian Hoyos and Seng Oon Toh.

Thanks to the system administrators at BWRC Brian Richards, Kevin Zimmerman and Brad Krebs for taking any worries out of computing. Thanks to the BWRC staff Brenda

Vanoni, Jessica Budgin and Tom Boot for being always helpful, precise and prompt. Thanks to Ruth Gjerde for being such a bright spot in the Grad Office.

Outside the school environment I want to thank Andrei Roth for founding the Berkeley Organization of Romanian Students (BORS) and keeping it alive, even after his graduation. BORS made me feel as if a piece of my country moved every Friday evening to the IHouse Cafe.

Thanks to my ski pals and friends Jan Vevera, Stefan Safko, Akiko Hasuo, Jiri Vanicek and Sergiu Nedevschi for making all those trips to Tahoe very enjoyable. Thanks to Matei and Andrei Odobescu (commonly known as the "Dynamite Brothers") for the biking trips and awesome barbecues. Thanks to Sergio Bronstein for being my favorite neighbor and such a warm friend.

Last, but not least, I want to thank my parents, Dan and Rodica Zlatanovici for their unconditional love and support. Much of the credit for my accomplishments and for who I am, in general, belongs to them. It is interesting to note that they are both accomplished electrical engineers and researchers and it is no wonder that I chose the same career path. I like to think of them not just as parents, but as friends, mentors, and role models.

Berkeley, CA

December 7, 2006

1 Introduction

"*If the automobile industry advanced as rapidly as the semiconductor industry, a Rolls Royce would get half a million miles per gallon and it would be cheaper to throw it away than to park it.*" (Gordon Moore, 1998).

1.1 Digital circuit design in the power-constrained era

Under the auspices of Moore's law, the development of the semiconductor industry in the last decades has been nothing short of phenomenal. Such tremendous progress has been possible due to the advances in the underlying technology. For more than three decades, the performance of digital circuits has increased at an astonishing rate using a very consistent design strategy: achieve the highest possible performance by maximizing the clock frequency.

In recent years, the emergence of power as the main limiting factor has prompted a shift away from this traditional approach. The recent stop in frequency increase for the leading microprocessors coincides with the beginning of the power - constrained era for the design of electronic circuits. Contemporary designs are power-limited yet it is still performance that ultimately sells products. Therefore, a design strategy adequate for the power - constrained era is to achieve the highest possible performance under maximum power constraints. In some situations it is more convenient to reformulate it by using power as the design objective and performance as the design constraint: achieve the minimum possible power under minimum performance constraints. In either of the two formulations, the design becomes a *power - performance optimization*.

This thesis implements a systematic power - performance optimization methodology for custom digital circuits at circuit and microarchitecture levels. A new CAD framework is developed for this purpose and demonstrated on several examples.

1.2 Background and current state of the art

Achieving the optimal performance under power limits is a challenging task because it involves a hierarchical optimization over a number of discrete and continuous variables, with a combination of discrete and continuous constraints. It is commonly accomplished through architecture and logic design, adjustments in the transistor / gate sizing, supply voltages or selection of transistor thresholds.

Various optimization techniques have been employed traditionally in digital circuit design, which range from simple heuristics to fully automated CAD tools.

At circuit level, the method of logical effort [Sutherland99] provides an analytical solution for sizing relatively simple custom circuits for minimum delay. For complex circuits with multiple paths and interconnect parasitics there is no analytical solution and manual sizing becomes impractical. TILOS [Fishburn85] was the first tool that realized that the delay of logic gates expressed using Elmore's formula presents a convex optimization problem that can be efficiently minimized using geometric programming [Boyd03]. A broad class of combinational circuits can be optimized by the tool due to the efficiency of the employed numerical methods. While the convex delay models used by TILOS are rather inaccurate because of their simplicity, the result is guaranteed to be *globally optimal*.

Commercial logic synthesis tools use a different approach. The technology mapping step chooses gates of different sizes from a library of standard cells in order to meet a delay target. In a subsequent step, a power - driven remapping can be used to bring the power of the circuit within a specified budget without exceeding the delay target. The resulting ASICs are *feasible* (i.e. they meet all design constraints) but not *optimal* because no figure of merit is minimized in the process.

Delay optimization under constraints has been automated in the past for custom circuits as well. IBM's EinsTuner [Conn99] uses a static timing formulation and tunes transistor sizes for minimal delay under total transistor width constraints. The delay models are obtained through simulation for better accuracy; however this guarantees only *local optimality*.

The conventional delay minimization techniques can be extended to account for energy as well. For example, a combination of both energy and delay, such as the energy-delay product (EDP) has been used as an objective function for minimization. A circuit designed to have the minimum EDP, however, may not be achieving the desired performance or could be exceeding the given energy budget. As a consequence, a number of alternate optimization metrics have been used that generally attempt to minimize an $E^m D^n$ product [Penzes02]. By choosing parameters n and m a desired tradeoff between energy and delay can be achieved, but the result is difficult to propagate to higher layers of design abstraction. In the area of circuit design, this approach has been traditionally restricted to the evaluation of several different block topologies, rather than using it to drive the optimization.

In contrast, a systematic solution to this problem is to minimize the delay for a given energy constraint [Stojanovic02]. Note that a dual problem to this one, minimization of the energy subject to a delay constraint yields the same solution. Two extreme solutions to this problem for sizing at circuit level are well known. The minimum energy of the fixed logic topology block corresponds to all devices being minimum sized. Similarly, the minimum delay point is well defined: at that point further upsizing of transistors yields no delay improvement. [Zyuban02] proposes *hardware intensity* as a unified metric for both energy and delay that can be used to position the circuits in-between these two extremes.

In sequential circuits, the position of storage elements (also called the *cutset*) offers another opportunity for optimization. The process of moving the storage elements with the purpose to minimize or maximize a certain figure of merit is called *retiming* and the basic algorithms have been introduced in [Leiserson91] and subsequently refined in [Shenoy94]. Basic retiming is independent of any other design optimizations and its use has been hampered by the difficulty in propagating it through the various layers of abstraction. In a higher level analytical approach, [Zyuban03] derives the optimal balance of hardware intensities across the various stages in a pipeline in order to achieve minimum energy for a fixed throughput. In a subsequent work by the same authors, the hardware intensity concepts are used to perform a power - performance analysis of microprocessor pipelines at the architecture level [Zyuban04].

Custom datapaths are an example of power-constrained circuits with several loops in the design process. Designers traditionally iterate between schematics and layouts through successive resizings and retimings in order to achieve timing closure. The initial design is sized and pipelined using wireload estimates and is iterated through the layout phase until a set delay goal is achieved. The sizing and cutset are refined manually using the updated wireload estimates. Finally, after minimizing the delay of critical paths, the non-critical paths are balanced to attempt to save some power, or in the case of domino logic to adjust the timing of fast paths. This is a tedious and often lengthy process that relies on the designer's experience and has no proof of achieving optimality. Furthermore, the optimal sizing and cutset depend on the chosen supply and transistor thresholds.

An optimal design strategy would be able to minimize the delay under power constraints (or the other way around) by choosing supply and threshold voltages, gate sizes or individual transistor sizes, logic style (static, domino, pass-gate, etc.), block topology, degree of parallelism, pipeline depth, cutset, layout style, wire widths, etc.

1.3 Thesis overview

This thesis builds on the ideas of convex [Fishburn85] and gradient-based [Conn99] optimization techniques under constraints. The ideas presented here are integrated in a modular *circuit optimization framework* for custom digital circuits in the power - performance space. The optimization is performed at circuit and microarchitecture levels.

The circuit optimization framework is the main research contribution of this thesis. The goals of the framework are:

- offer maximum flexibility in the choice of design variables, logic family and models for the underlying technology;
- use the theory of mathematical optimization to transform the design problem in a form that can be reliably solved by existing numerical optimizers;
- whenever possible, offer a guarantee for the global optimality of the results;
- provide a way to evaluate the quality of the results when such an optimality guarantee does not exist;
- take process variations into account in order to optimize yield (currently in a separate module, not fully integrated with the rest of the framework).

The circuit optimization framework is described in a bottom - up fashion in the thesis. The presentation starts with the basic concepts beyond power - performance optimization and the fundamentals of mathematical optimization. Combinational circuits are analyzed first, dealing with issues such as gate sizing, supply and threshold voltage optimization. Then the presentation shifts to sequential circuits, integrating retiming in the framework. In the end process variations are added in the picture to pave the road to yield optimization.

Examples are provided throughout the thesis to illustrate the concepts and capabilities of the framework. The examples have two equally important goals:

1. to verify the circuit optimization framework; this serves a sanity check and ensures that the circuits designed by the framework are functional and perform “as advertised”;
2. to build design intuition.

The second goal helps the designer get a better understanding of the architectural tradeoffs and make wise design choices. Ultimately, a CAD tool is only as good as the designers who are using it. The skills, intuition and experience of the design team is what eventually determines the final outcome of any project. The circuit optimization framework can help the designers understand the subtle tradeoffs of the particular circuit they are working on and this is illustrated on a few examples in the thesis. One of the examples goes beyond a particular implementation and uses the circuit optimization framework to build general knowledge about a family of circuits (64-bit adders). One circuit in the family has been built in silicon and its performance and power measurements make it interesting in its own right, regardless of the methodology used to design it.

1.4 Thesis organization

Chapter 2 introduces the main concepts of power - performance optimization. A general discussion on the behavior of digital circuit in the power - performance space is followed by a review of the mathematical background of circuit optimization. The general architecture of the circuit optimization framework developed for this thesis is presented in the last section. Chapter 3 provides an in-depth look at the optimization of combinational circuits - from simple classroom cases to complex real-life circuits through successive refinements. A simple theoretical example is presented at the end of the chapter. Chapter 4 presents a detailed case study of 64-bit adders and their design tradeoffs in the power - performance space using the most advanced capabilities of the combinational circuit optimization framework. A proof-of-concept implementation of a state-of-the-art adder built in 90nm CMOS and designed using the described optimization framework is presented in the latter sections.

Chapter 5 describes the optimization of sequential circuits by building on the concepts from Chapter 3 and also presents an example in the end. Chapter 6 introduces process variations in the power - performance optimization. A theoretical framework for stochastic yield optimization is presented and demonstrated on a simple example. The conclusions and future directions are presented in Chapter 7.

2 Design as a Power - Performance Optimization Problem

This chapter introduces the main concepts of power - performance optimization.

It starts with a general discussion of the behavior of digital circuits in the power-performance space in Section 2.1, leading to the construction of a first optimal power - performance tradeoff curve. Section 2.2 discusses how these power - performance tradeoff curves can be used to compare different candidate architectures for the implementation of the same function. The choice of optimization variables and their impact in the power - performance space are described in Section 2.3.

The ensuing sections set the mathematical background for the circuit optimization framework. Section 2.4 introduces numeric optimization problems. Section 2.5 discusses the challenges incurred by general optimization problems and introduces convex optimization as a sub-family of optimization problems that is easier to solve. Section 2.6 presents the types of optimization problems most commonly used for digital circuits: geometric programs and generalized geometric programs in convex form. Section 2.7 introduces the Lagrange multipliers as a tool to analyze the sensitivity of the optimization problem to its parameters and constraints.

The general architecture of the circuit optimization framework developed for this thesis is introduced in Section 2.8. Detailed descriptions of the optimization framework are presented in Chapter 3 (for combinational circuits) and Chapter 5 (for sequential circuits).

2.1 Constructing the first energy - delay curve

In order to analyze the behavior of digital circuits in the power - performance space, the first step is to select appropriate power and performance metrics.

The performance of a digital circuit can be evaluated in several ways, using metrics such as clock frequency, number of operations per second, cycle time or simply the delay of the circuit. While such metrics can have different meanings and offer different information to the system designers, the *delay* of the circuit is a common denominator for most of them. Therefore, the *delay* of a circuit is the primary performance metric used in this thesis.

The power of a circuit can also be evaluated in several ways. For desktop applications such as high-performance microprocessors, peak power drives packaging and cooling constraints. For portable applications where battery life is the primary concern, the average energy per operation is a more appropriate power metric. Average power and peak energy per operation can be used as well if needed. Conversion between these power metrics is easy by taking into account the cycle time, leakage power and activity factors throughout the circuit. For simplicity, the *average energy per operation* is used as the primary power metric in this thesis and referred to simply as the *energy* of the circuit.

Using these metrics, the generic "power - performance space" translates into a concrete "*energy - delay space*" where the designs can be accurately placed, as shown in Figure 2-1.

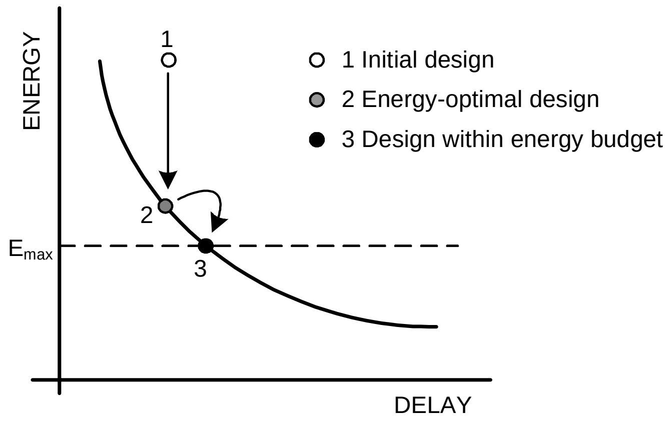

Figure 2-1. Designs in the energy - delay space.

Any design corresponds to a point in the energy - delay space. For instance, point 1 in Figure 2-1 can be an initial design. The first optimization to be performed is to remove any excess power without affecting performance (if possible). Point 2 corresponds to an energy-optimal design, where it is no longer possible to decrease the energy without decreasing the performance of the circuit. If point 2 still exceeds a pre-imposed energy budget E_{max} , then the delay of the circuit must be increased, to point 3. However, the delay of the circuit should be increased only as little as necessary to attain the desired energy budget E_{max} - i.e. it should attain the best performance in the limited power budget.

Points 2 and 3 in Figure 2-1 achieve some degree of optimality: they both represent circuits with the minimum delay for their energy *and* the minimum energy for their delay. They both sit on the *optimal energy - delay tradeoff curve*. Circuits corresponding to points above the curve (such as point 1 in Figure 2-1) are sub-optimal because there are other circuits that can perform the same function faster for the same energy or for less energy in the same time. From a power - performance perspective, circuits not sitting on the optimal power - performance tradeoff curve are wasteful and should be avoided.

For a circuit with a given topology, the optimal energy - delay tradeoff curve has two well defined end-points, as shown in Figure 2-2.

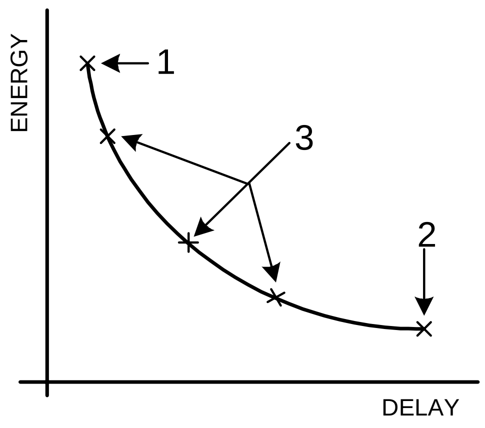

Figure 2-2. End-points of the optimal energy - delay tradeoff curve.

Point 1 represents the fastest circuit that can be designed using that topology. For a combinational circuit, this point can be obtained by solving the minimum delay problem using, for instance, the logical effort method from [Sutherland99]. Point 2 represents the lowest power circuit that can be obtained using that topology. In most cases this circuit can be obtained by making all transistors minimum size. Other design constraints such as minimum signal slopes imposed for signal integrity reasons may require larger devices.

In a different interpretation, point 1 corresponds to minimizing just the delay of the circuit D while point 2 corresponds to minimizing just the energy of the circuit, E. The points in-between the two extremes (marked “3” in Figure 2-2) correspond to minimizing various E^mD^n design goals - such as the well known energy-delay product, EDP. Therefore, the optimal energy - delay tradeoff curve is a *complete* representation of the behavior of a circuit in the energy delay space, containing all composite energy - delay design metrics.

The feasibility-based approach of commercial logic synthesis tools can be represented in the energy - delay space as shown in Figure 2-3. A synthesizer produces a design that meets the delay and energy targets D_{max} and E_{max} and therefore can lie anywhere in the “feasible region”. Different starting points can result in different paths through the energy - delay space and therefore different final circuits, because any design in the feasible region is acceptable. In most cases the synthesized design will not be on the optimal energy - delay tradeoff curve but in the interior of the feasibility region and hence suboptimal.

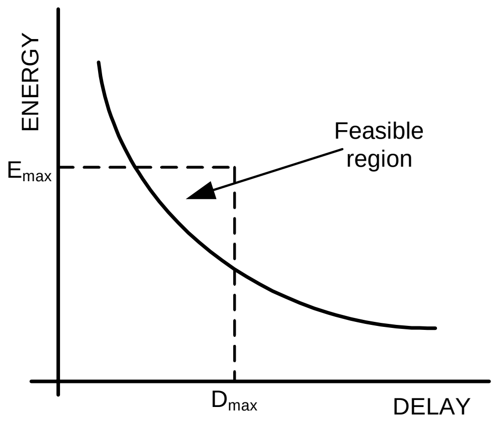

Figure 2-3. Feasibility vs. optimization-based design.

By contrast, an optimization-based approach will always produce a circuit on the optimal energy - delay tradeoff curve.

2.2 Using energy - delay curves for architecture selection

Optimal energy - delay tradeoff curves provide a natural comparison criterion for different circuit architectures implementing the same function. From a system level perspective, it is always desirable to implement the underlying circuit fabric such that it offers maximum performance for its power budget or the minimum power for its performance target. Figure 2-4 illustrates how to use optimal energy - delay tradeoff curves to choose between two candidate architectures implementing the same function.

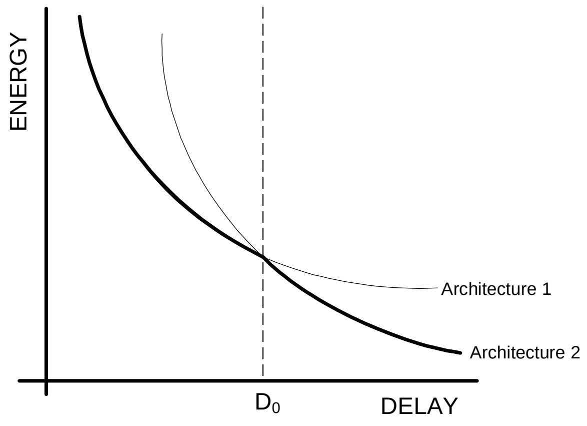

Figure 2-4. Architecture selection using energy - delay tradeoff curves.

For delay targets shorter than D_0, “Architecture 1” is better in the energy - delay sense (i.e. it offers better performance for the same power and lower power for the same performance). For delay targets longer than D_0, “Architecture 2” is the preferred choice. The envelope of the two curves can be considered the global energy - delay tradeoff curve for the block.

In general, the overall power - performance tradeoff curve of a logic block is the *envelope* of the individual tradeoff curves corresponding to different architectures. Such overall curves are made of *pieces* of architecture-specific tradeoff curves and therefore are not smooth like the ones shown in the previous section. The best architecture in a particular design point can be tracked back by identifying the piece in the overall curve that gives the envelope in that region of the energy - delay space.

2.3 Variables in the energy - delay optimization

Several design variables can be adjusted in the process of power - performance optimization. Some variables are commonly expected to be the subject of a design optimization process (such as transistor sizes and power supply voltage) while others may be fixed in advance due to various architectural and practical considerations (such as the cell height if using a standard cell library).

The choice of optimization variables greatly influences the final outcome of the optimization, as illustrated in Figure 2-5.

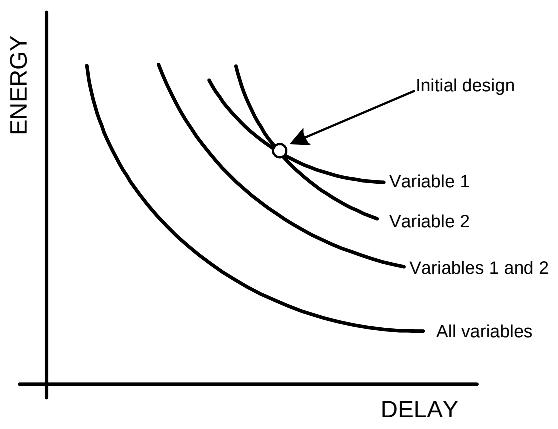

Figure 2-5. Impact of optimization variable choice in the energy - delay space.

Starting from an initial design, different energy - delay tradeoff curves can be obtained by tuning different variables individually. In Figure 2-5 “Variable 1” and “Variable 2” can be for example the supply voltage V_{DD} and the threshold voltage V_{TH}, adjusted through body bias. Adjusting each of these variables individually yields a certain energy - delay tradeoff. However, optimizing both variables *jointly* produces a circuits that are better in the energy - delay sense. If other variables (such as transistor sizes) are included in the optimization, the delay and energy of the resulting circuits can be further decreased.

In general, the more variables are included in the optimization, the better the result. Not including a variable in the optimization is equivalent to constraining it to its nominal value, hence reducing the space the optimizer can explore in its quest for the optimum.

Typical optimization variables can be (but are not limited to):

- gate / transistor sizes;
- power supply voltage;

- threshold voltage;
- bitslice height;
- bus wire size and spacing;
- other layout parameters concerning placement and routing strategies;
- latch / flip-flop positions;
- pipeline depth;
- degree of parallelism or multiplexing.

Some of these variables are continuous (e.g. power supply, gate sizes in custom designs) and some other are inherently discrete (e.g. flip-flop positions, degree of parallelism). While optimizing all possible variables at the same time is desirable due to the reasons discussed in the beginning of this section, this is impossible in most practical cases. A common approach is to do a layered design, in which different variables are tuned at different layers of abstraction. Parameters such as pipeline depth and degree of parallelism or multiplexing are usually adjusted in macro-architecture level optimizations, such as the ones performed in [Markovic04]. This thesis is mostly focused on the optimization of circuit and microarchitecture level parameters such as gate sizes, power supply voltage and flip-flop positions.

2.4 Optimization problems

“Optimization” has been a recurring word throughout this chapter and the goal of this work is to formulate the design as a power performance *optimization problem* and then solve that *optimization problem*. In general, a mathematical *optimization problem* can be expressed as [Boyd03]:

$$\min_{x \in \Re^n} f(x) \text{ such that } \begin{cases} g_i(x) \le 0 & i = 1 \ldots m \\ h_j(x) = 0 & j = 1 \ldots k \end{cases} \qquad \textbf{(2-1)}$$

The commonly used terminology is:

- x contains the *optimization variables* in an n-dimensional vector
- f is the *objective function*
- g_i are *inequality constraints*
- h_i are *equality constraints*

The general optimization problem (2-1) searches for the set of optimization variables x that minimizes the objective function f while satisfying the inequality constraints g and the equality constraints h. A point in the variable space that satisfies all (inequality and equality) constraints is called a *feasible point*. All feasible points define the *feasible set* of the optimization problem. The solution of the optimization problem (the x that minimizes f) must be within the feasible set. If the feasible set is void, the optimization problem has no solution and is said to be *infeasible*.

A general optimization problem such as (2-1) is very difficult to solve if no assumptions are made about the objective function and the constraints. Although there are some algorithms that are able to deal with general functions with a certain degree of success, they are usually slow and can get trapped in local minima. Figure 2-8 illustrates how an optimizer could converge to the incorrect solution. If the initial guess value is point 1, then the optimizer converges to the correct solution, point A. However, if the initial guess value is point 2, the optimizer will most likely stop at the incorrect solution point B which although a local minimum, is not the global minimum.

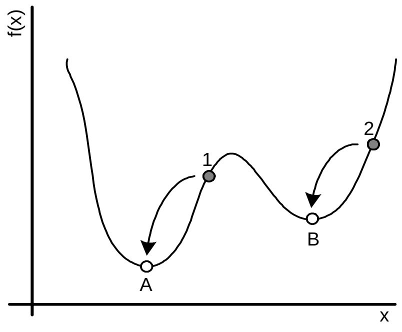

Figure 2-6. Local and global minima in general optimization problems.

The solution of general optimization problems is usually highly dependent on the initial guess point.

An alternative approach is to develop optimization algorithms for certain classes of problems and exploit their properties in a convenient way. This, however, poses another problem: real optimization problems may not fall within that specific class of problems the algorithm is capable to deal with in a very efficient way. The solution is to try to transform the actual optimization problem into one that satisfies all the assumptions of the algorithm by using various techniques such as changes of variables, adding slack variables, duality etc.

2.5 Convex optimization problems

One class of optimization problems for which very efficient algorithm exist is called *convex optimization* [Boyd03].

In a convex optimization problem the objective function and the feasible set must be *convex*. A convex function has the property that the chord joining two points lies above the function, as shown in Figure 2-7.

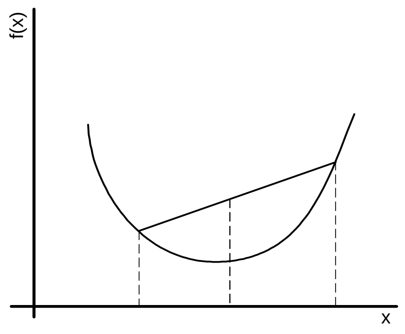

Figure 2-7. Convex function.

The key property of a convex function is that every *local* minimum is a *global* minimum. Hence the use of a convex function avoids any hill-climbing efforts. If a minimum is found, it is guaranteed to be the global minimum.

A *convex set* is a set for which the segment joining two arbitrary points of the set lies also within that set. The condition that the feasible set be convex translates into the following restrictions on the constraint functions *g* and *h*:

- inequality constraints g_i must be convex
- equality constraints h_i must be linear

The constraint on the *h* functions is very restrictive. However, circuit optimization problems do not usually have equality constraints. Therefore, the general form of a *convex optimization problem* as used for digital circuits is:

$$\min_{x \in \Re^n} f(x) \text{ such that } g_i(x) \le 0 \quad i = 1 \ldots m \quad \text{where } f \text{ and } g_i \text{ are convex functions} \qquad \textbf{(2-2)}$$

Some very efficient algorithms for solving convex optimization problems are commercially available. They generally use interior point iterative methods or some variations of the simplex algorithm [Boyd03]. Some of them (such as the Mosek toolbox [Mosek06] for Matlab) do not even require an initial guess value, completely eliminating its dependency.

The biggest problem when using convex optimizers is to put the real optimization problem in a convex form. Real life problems are usually not convex; for instance, the delay of a logic path is not a convex function, but can be made convex after a series of mathematical transformations. The next section introduces the most common forms of convex problems used in the optimization of digital circuits.

2.6 Geometric programs and generalized geometric programs

In this section we describe a family of optimization problems that are *not* convex in their natural form. These problems can, however, be transformed to convex optimization problems using a change of variables. The presentation in this section summarizes the essential characteristics of geometric programming without going into all the details. For a complete description, the reader is referred to [Boyd03] which contains an in-depth discussion about these problems and about convex optimization in general.

In order to define geometric programming we need to first introduce posynomial functions. A *posynomial* function is defined as:

$$p(x) = \sum_j \gamma_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \quad \alpha_{ij} \in \Re, \gamma_j \in \Re^+ \tag{2-3}$$

for $x_i > 0$. Each term of the sum is called a *monomial*. A posynomial is similar with a polynomial with the following 2 differences:

1. exponents of the variables can have arbitrary real values (including negative), not only integers
2. monomial coefficients must be positive real numbers (as opposed to polynomials where they can be negative as well).

A posynomial is not a convex function per se, but can be made convex using the following change of variables:

$$x_i = e^{z_i} \quad i = 1 \ldots n \tag{2-4}$$

This transformation is invertible for positive x_i's and therefore the original optimization variables can be easily retrieved when the algorithm finishes.

An optimization problem of the form

$$\min\ f_0(x) \text{ such that } \begin{cases} f_i(x) \le 1 & i = 1 \ldots m \\ h_j(x) = 1 & j = 1 \ldots k \end{cases} \qquad \textbf{(2-5)}$$

where $f_0, \ldots f_m$ are posynomials and $h_1, \ldots h_k$ are monomials is called a *geometric program* (GP). The domain of this problem is $D = \Re^n_{++}$ and the constraint $x_i > 0$ is implicit.

Geometric programs are not (in general) convex optimization problems, but they can be transformed to convex problems using the change of variables (2-4).

Generalized geometric programming (GGP) is an extension to GP that preserves the convexity of the re-mapped optimization problem for a wider family of functions than just posynomials. A function f is called a *generalized posynomial* if it can be formed from posynomials using the operations of addition, multiplication, positive (fractional) power and maximum. An optimization problem like (2-5) where $f_0, \ldots f_m$ are generalized posynomials and $h_1, \ldots h_k$ are monomials is called a *generalized geometric program* (GGP). As shown in [Boyd03], GGPs can be transformed into convex problems using the same change of variables (2-4) as for regular GPs. Moreover, while GGPs are much more general than GPs, they can be *mechanically transformed* into an equivalent regular GP by introducing a supplementary variable and an equality constraint for each individual posynomial component of each generalized posynomial.

GP and GGP are the most common forms of optimization used for digital circuits, as will be discussed in Chapter 3.

2.7 Sensitivity analysis of optimization problems: the Lagrange multipliers

Sometimes it is useful to know how changing a constraint or a parameter in the optimization problem will change its result. Such an analysis can provide insights into the bottlenecks of the problem and improve the robustness of the design.

Duality provides the theory behind optimization sensitivity considerations. This section presents a brief introduction of the main concepts of duality. The reader is referred to [Boyd03] for a detailed discussion on this topic.

For the optimization problem (2-1), the *Lagrangian function* is defined as:

$$L(x, \lambda, \nu) = f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{k} \nu_j h_j(x) \qquad \textbf{(2-6)}$$

λ_i is called the *Lagrange multiplier* associated with the ith inequality constraint $g_i(x) \le 0$; similarly, ν_j is the Lagrange multiplier associated with the jth equality constraint $h_j(x) = 0$. The λ and ν vectors are called the *dual variables* or *Lagrange multiplier vectors* associated with the problem (2-1).

The *Lagrange dual function G* is defined as the minimum value of the Lagrangian over *x*:

$$G(\lambda, \nu) = \inf_x L(x, \lambda, \nu) = \inf_x \left(f(x) + \sum_{i=1}^{m} \lambda_i g_i(x) + \sum_{j=1}^{k} \nu_j h_j(x) \right) \qquad \textbf{(2-7)}$$

When the Lagrangian is unbounded below in x, the dual function takes on the value $-\infty$. Since the dual function is the pointwise infimum of a family of affine functions of (λ,ν), it is concave, even if the optimization problem (2-1) is not convex.

The dual function yields lower bounds on the optimal value p^* of the problem (2-1). It is easily verified that for any $\lambda \geq 0$ and any ν:

$$G(\lambda, \nu) \leq p^* \tag{2-8}$$

Thus, G gives a lower bound on the solution that depends on some parameters λ,ν. The *best* lower bound that can be obtained from the Lagrange dual function provides can provide a stopping criterion for the original problem. This leads to the optimization problem:

$$max \quad G(\lambda, \nu) \text{ such that } \lambda \geq 0 \tag{2-9}$$

This problem is called the *Lagrange dual problem* associated with (2-1). In this context, the original problem (2-1) is called the *primal problem*. The doublet (λ^*,ν^*) is referred to as *dual optimal* or *optimal Lagrange multipliers*.

The Lagrange dual problem (2-9) is a convex optimization problem since the objective to be maximized is concave and the constraint is convex. This is true whether or not the primal problem (2-1) is convex or not.

The optimal value of the Lagrange dual problem, d^*, is by definition the best lower bound on p^* that can be obtained from the Lagrange dual function. In particular the simple but important inequality:

$$d^* \leq p^* \tag{2-10}$$

holds even if the original problem is not convex. This property is called *weak duality*. The difference *p*-d** is called the *optimal duality gap* of the original problem. It gives the gap between the optimal value of the primal and the best (i.e. greatest) lower bound on it that can be obtained from the Lagrange dual function. The optimal duality gap is always non-negative.

If the equality

$$d^* = p^* \tag{2-11}$$

holds, i.e. the duality gap is zero, it is said say that *strong duality* holds. This means that the best bound that can be obtained from the Lagrange dual function is tight.

Strong duality does not hold in general. But if the primal problem is convex, it holds under a set of straightforward conditions known as *Slater's constraint qualification conditions*, described in detail in Section 5.2.3 of [Boyd03].

All optimization problems involved in the optimization of digital circuits qualify under Slater's conditions for strong duality.

When strong duality holds, the optimal dual variables provide very useful information about the sensitivity of the optimal value with respect to perturbations of the constraints. Consider the following *perturbed* version of the original optimization problem (2-1):

$$\min_{x \in \Re^n} f(x) \text{ such that } \begin{cases} g_i(x) \le u_i & i = 1 \dots m \\ h_j(x) = v_i & j = 1 \dots k \end{cases} \tag{2-12}$$

The problem coincides with the original problem (2-1) when $u = 0$ and $v = 0$. When u_i is positive it means that we have relaxed the ith inequality constraint; when u_i is negative it means that we have tightened the constraint. The perturbed problem (2-12) results from the original problem (2-1) by tightening or relaxing each inequality constraint by u_i and by changing the right-hand side of the equality constraints by v_i. Let $p^*(u,v)$ be the optimal value of the perturbed problem (2-12). When the original problem is convex, the function p^* is convex in u and v [Boyd03].

If $p^*(u,v)$ is differentiable at (0,0) and strong duality holds, the optimal dual variables (λ^*, ν^*) are related to the gradient of p^* at $u = 0$ and $v = 0$:

$$\lambda_i = -\frac{\partial p^*(0, 0)}{\partial u_i} \qquad \textbf{(2-13)}$$

$$\nu_i = -\frac{\partial p^*(0, 0)}{\partial v_i} \qquad \textbf{(2-14)}$$

The optimal Lagrange multipliers are *exactly* the local sensitivities of the optimal value with respect to constraint perturbations.

Various interpretations of the optimal Lagrange variables follow directly from these equations. The sensitivity provides a measure of how active a constraint is at the optimum value x^*. If $g_i(x^*) < 0$ then the constraint is *inactive* and it follows that the constraint ca be tightened or loosened a small amount without affecting the optimal value and the associated optimal Lagrange multiplier is zero. The ith optimal Lagrange multiplier tells us how active the ith constraint is: if λ_i^* is small it means the constraint can be tightened or loosened a bit without much influence on the optimal value; if λ_i* is large, it means that the effect on the optimal value will be great.

All the optimizers used in this thesis solve the primal problem by solving the dual and computing the duality gap [Mathworks05], [Mosek06], [GGPLAB06] thus making the values of the optimal Lagrange multipliers available to the user. The dual solution of the optimization problem provides the data for the sensitivity analysis: for instance the values of the Lagrange multipliers can be used to identify the *critical paths* in a digital circuit.

2.8 Circuit optimization framework

The main contribution of this work is the development of a CAD toolbox to design digital circuits in the power - performance paradigm. This incurs formulating the design as a power - performance optimization problem (in most cases as a GP or GGP) and then solving it numerically. The CAD toolbox, further referred to as the “circuit optimization framework” is built around a versatile optimization core consisting of a static timer in the loop of a mathematical optimizer, as shown in Figure 2-8.

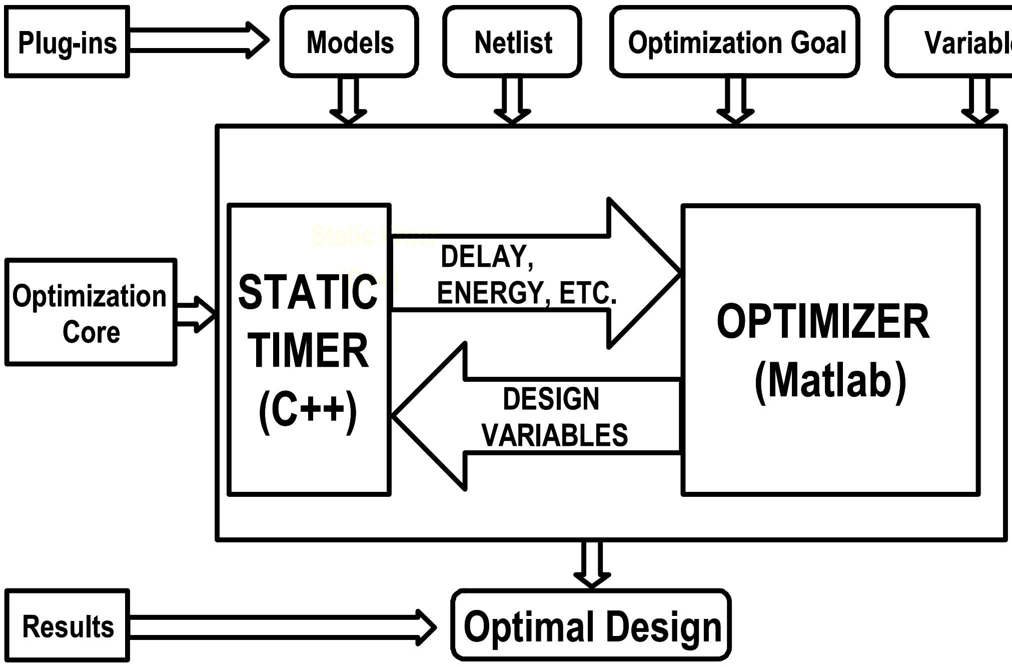

Figure 2-8. Circuit optimization framework.

The optimizer passes a set of specified design variables (as discussed in Section 2.3) to the timer and gets the resulting delay (as a measure of performance) and energy of the circuit, as well as other quantities of interest such as signal slopes, capacitive loads and, if needed, design variable gradients. The process is repeated until it converges to the optimal values of the design parameters, as specified by the desired optimization goal.

The inputs to the optimization core are:

- an optimization goal (e.g. "minimize delay subject to a maximum energy constraint");
- a set of design variables to be adjusted in the quest for the optimum (from the list in Section 2.3);

- a SPICE-like gate-level netlist of the circuit to be optimized (each gate is specified by its type as well as its input and output nodes). The netlist also includes an estimate of the *wire loads* from a preliminary layout floorplan sketch and the boundary conditions for the circuit (i.e. the maximum loads on the global inputs and outputs);
- user-specified models to be used by the static timer to compute delays, energy, power, signal slopes, leakage etc.

The circuit optimization framework is modular and can easily accommodate other optimization goals, design variables and models by simply swapping modules in and out.

Since the static timer is in the main speed-critical optimization loop, it is implemented in C++ to accelerate computation. It is based on the conventional longest path algorithm. The current timer is custom written and does not account for false paths or simultaneous arrivals, but it can be easily substituted with a more sophisticated one because of the modularity of the optimization framework. When optimizing sequential circuits, the static timer also has *retiming* capabilities, as discussed in detail in Chapter 5.

The framework can use several numerical optimizers. For GPs and GGPs, the solvers currently used are Mosek Optimization Toolbox [Mosek06] and GGPLAB [GGPLAB06]. For other problems (not necessarily convex) the *fmincon* function from Matlab Optimization Toolbox [Mathworks05] is used. Each solver has its own advantages and disadvantages: *fmincon* is slow but the most general and can deal with non-convex optimization problems. GGPLAB is written entirely in Matlab and works best for small and medium size GPs. Mosek has a rather large overhead for small problems, but can solve very large GPs with tens of thousands of variables, and can take advantage of modern multi-core and multi-processor computers to speed up the optimization.

Other solvers can be employed with minimal efforts due to the standard interface with the other modules of the framework.

3 Optimizing Combinational Circuits

This chapter provides an in-depth look at the optimization of combinational circuits. Starting from the simplest cases, the combinational circuit optimization problem is successively refined until it can handle real-life circuits with good accuracy.

The first step in setting up an optimization problem in the energy-delay space is to select models that will express the energy and delay of a circuit as a function of the optimization parameters. Section 3.1 presents the Elmore delay-based linear model for a gate and a logic path. The model is at the foundation of the logical effort sizing method [Sutherland99]. Using this model, the simplest optimization problem - unconstrained delay minimization by tuning gate sizes - is set up in Section 3.2. The rather inaccurate initial delay model is improved in Section 3.3 and Section 3.4 by including the effects of keepers for dynamic gates and signal slopes. Section 3.5 discusses how wire parasitics can be included in the optimization problem. Energy and area expressions are presented in Section 3.6 and the constrained optimization problem for combinational circuits is set up. Section 3.7 shows how to optimize combinational circuits with multiple paths using a static timer. At this point the models are similar to the level-1 models used in commercial logic synthesis tools [Synopsys04]. Arbitrarily accurate tabulated models are discussed in Section 3.8. Section 3.9 shows a way to evaluate the quality of a result obtained from an optimization problem whose optimality cannot be guaranteed. Section 3.10 extends the optimization problem to include the power supply voltage and the transistor threshold voltage as optimization parameters in addition to the gate sizes.

Section 3.11 shows a simple example that uses analytical models to illustrate the capabilities of the optimization framework. The full capabilities of the optimization framework are demonstrated in Chapter 4 through a detailed case study on 64-bit adders using tabulated models.

This chapter presents an extended version of our previously published work [Zlatanovici05].

3.1 Simplest delay expressions: logical effort

The model described in this section is based on the assumption that the delay of a logic gates is a linear function of the load capacitance. It is the foundation of a gate sizing method described in [Sutherland99] called *logical effort*.

3.1.1 Delay of a static gate

Using the above linear assumption, the delay of a logic gate is:

$$t_{D,abs} = 0.69 \cdot R_{drive} \cdot (C_{int} + C_L) \qquad \textbf{(3-1)}$$

where R_{drive} is the equivalent driving resistance, C_L is the load capacitance and C_{int} is the internal self-loading capacitance of the gate. If γ is the ratio of the internal self-loading capacitance and the input capacitance of the gate $\gamma = C_{int}/C_{in}$ the delay equation (3-1) can be rewritten as:

$$t_{D,abs} = 0.69 \cdot R_{drive} \cdot C_{int} \cdot \left(1 + \frac{C_L}{\gamma C_{in}}\right) = \tau_{gate} \cdot \left(\gamma + \frac{C_L}{C_{in}}\right) \qquad \textbf{(3-2)}$$

where τ_{gate} is an intrinsic time constant for the gate. Normalizing everything to the intrinsic time constant of an inverter τ_{inv}, the delay of a gate can be written as:

$$t_D = \frac{t_{D,abs}}{\tau_{inv}} = \frac{\tau_{gate}\gamma}{\tau_{inv}} + \frac{\tau_{gate}}{\tau_{inv}} \cdot \frac{C_L}{C_{in}} = p + g \cdot \frac{C_L}{C_{in}} \qquad \textbf{(3-3)}$$

The delay of a gate is therefore determined by three factors:

1. $p = \frac{\tau_{gate}\gamma}{\tau_{inv}}$ - *intrinsic delay,* a fixed part that depends on the internal self-loading of the gate, dominated by the source and drain capacitances of the transistors at the output node;
2. $g = \frac{\tau_{gate}}{\tau_{inv}}$ - *logical effort,* capturing the influence of a gate's topology on its ability to drive loads; it is independent of the sizes of the transistors in the circuit;
3. $h = \frac{C_L}{C_{in}}$ - *fanout* or *electrical effort,* capturing the influence of the electrical environment of the gate; it is equal to the ratio between the load and the input capacitance of the gate.

Thus, the delay formula for one gate is:

$$t_D = p + g \cdot h \qquad \textbf{(3-4)}$$

Eq. (3-4) shows that a gate is characterized by two parameters for each input and for each transition: the intrinsic delay *p* and the logical effort *g*. The $g \cdot h$ product is usually called the effort delay of the gate.

The delay of a logic gate, as represented by (3-4) is a linear relationship. Figure 3-1 shows this relationship graphically (delay as a function of the fanout) for an inverter and a 2-NAND gate.

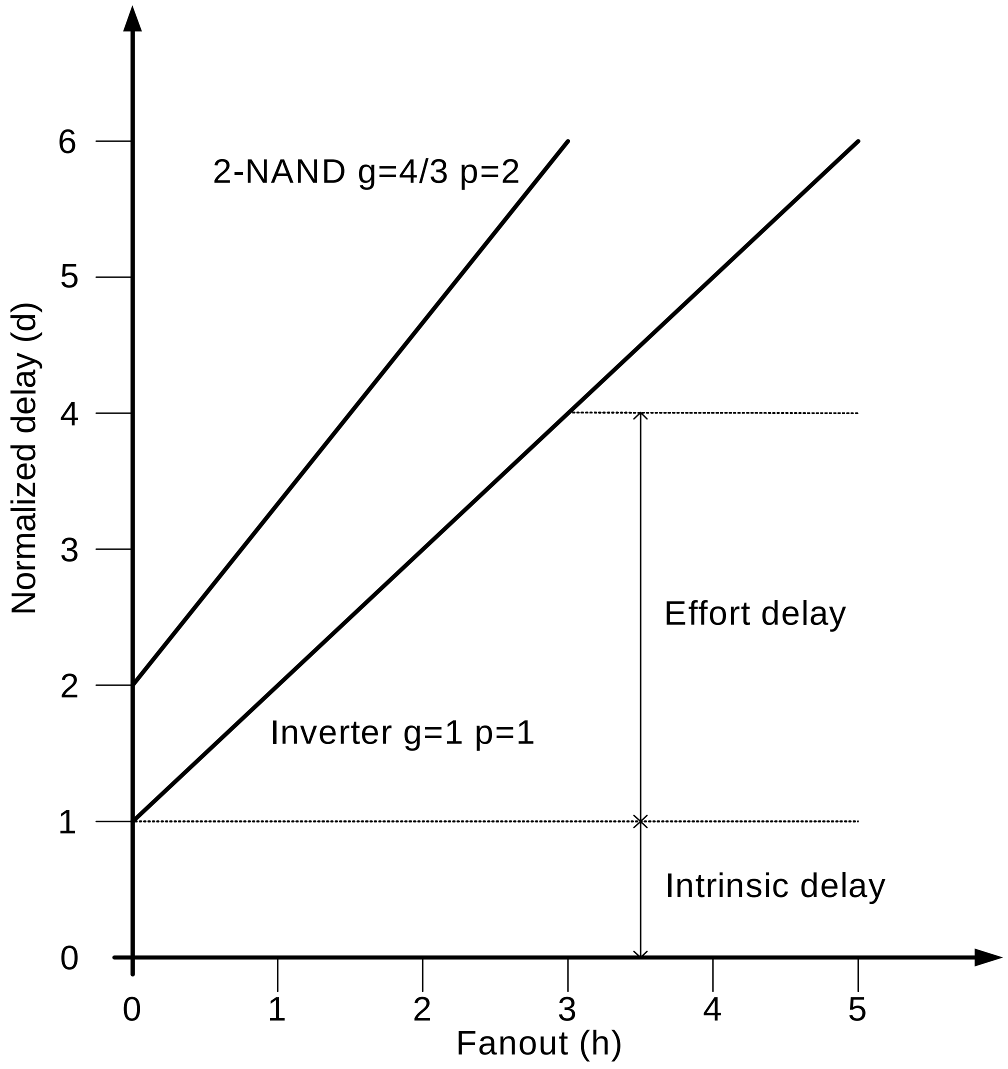

Figure 3-1. Delay of an inverter and a 2-NAND as a function of the fanout.

The logical effort of a gate (*g*) can have several equivalent definitions. It represents the number of times worse the gate is at delivering output current than would be an inverter with the same input capacitance [Sutherland99]. Therefore, by definition, an inverter has

g=1. The logical effort g is also the slope of the gate's delay vs. fanout curve divided by the slope of an inverter's delay vs. fanout curve. This definition is more useful for measurement and calibration purposes, and is illustrated in Figure 3-1.

The intrinsic delay of a gate (p) can also have several definitions. It represents the delay of the gate without any external load and it is determined mainly by the drain and source capacitance of the transistors. In Figure 3-1 the intercept of the delay vs. fanout curve with the Y axis is the intrinsic delay of the gate.

The main limitations of the model are:

- it is uncalibrated if the simplest definitions are used for the parameters; the model can be calibrated using the delay vs. fanout curves as shown in Figure 3-1;
- it does not account for slopes (nominal slopes are considered). The model has good accuracy when slopes are fast but becomes inaccurate when slopes are slow due to the limited gain of the gates and the decreasing accuracy of the linear approximation. The problem can be alleviated by imposing an upper bound on the fanout in order to keep the slopes reasonable;
- the model has difficulty handling gates with multiple paths from an input to the output. For instance a gate implementing $Z=(A+(A+B)C)'$ has two paths from input A to the output - one through each term. Depending on the internal transistor sizing and input combination, each of the two paths can be critical. The model cannot capture this situation unless a separate set of parameters is defined for each *path* in the gate.

3.1.2 Delay of a logic path

Let's consider first a logic path composed solely out of static gates, with negligible interconnect, and driving a load capacitance C_L, as shown in Figure 3-2.

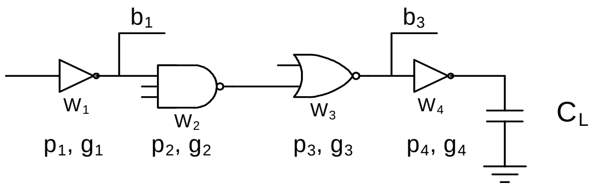

Figure 3-2. Logic path with branching and negligible interconnect.

At each node we can define the branching effort *b* as:

$$b = \frac{C_{on\ path} + C_{off\ path}}{C_{on\ path}} \qquad \textbf{(3-5)}$$

The input capacitance of gate *i* is:

$$C_{in,i} = g_i W_i \qquad \textbf{(3-6)}$$

where W_i is the relative size of gate *i*. Using (3-4), the path delay is the sum of the individual gate delays along that path:

$$t_D = \sum_{i=1}^{N} (p_i + g_i \cdot h_i) = \sum_{i=1}^{N} p_i + \sum_{i=1}^{N} g_i \cdot \frac{C_{load,i}}{C_{in,i}} = \sum_{i=1}^{N} p_i + \sum_{i=1}^{N} g_i \cdot \frac{b_i g_{i+1} W_{i+1}}{g_i W_i} \qquad \textbf{(3-7)}$$

and therefore:

$$t_D = \sum_{i=1}^{N} p_i + \sum_{i=1}^{N} \frac{b_i g_{i+1} W_{i+1}}{W_i} \qquad \textbf{(3-8)}$$

(3-8) is the basic formula for the delay of a logic path. It does not take into account any interconnect effects or keepers for dynamic gates. (3-8) is a *posynomial* in W_i and hence can be conveniently used in geometric programs.

3.2 The first optimization problem: unconstrained delay minimization

Using (3-8) the first optimization problem can be set up as follows:

$$\min_{W_i} \left(\sum_{i=1}^{N} p_i + \sum_{i=1}^{N} \frac{b_i g_{i+1} W_{i+1}}{W_i} \right) \text{ such that } \begin{cases} C_{in} \le C_{in,max} \\ W_i \ge 1 \end{cases} \qquad \textbf{(3-9)}$$

where C_{in} is the input capacitance of the path, computed using (3-6). C_{in} is constrained to a maximum value $C_{in,max}$ that determines the overall electrical effort of the path. The $W_i \ge 1$ constraint, although obvious for a circuit designer, must be included explicitly in order to ensure that the optimizer produces a manufacturable circuit.

The optimization problem (3-9) can be solved analytically. The well known solution, described in detail in [Sutherland99], finds that for minimum delay all stages in a path should have the same stage effort:

$$f_i = g_i \cdot h_i = \sqrt[N]{\prod_{j=1}^{N} g_j b_j \cdot \frac{C_L}{C_{in,max}}} \text{ for all } i = 1 \ldots N \qquad \textbf{(3-10)}$$

The gate sizes W_i can be found by working from either end of the path using $C_{in,i} = C_{out,i} \cdot g_i / f_i$ and (3-6).

3.3 Including keepers for dynamic gates

If a dynamic gate has no keeper, like the one in Figure 3-3, its logical effort is computed the same way as for a static gate. In this case *g*=1/3 and it is independent of the relative size of the gate, *W*.

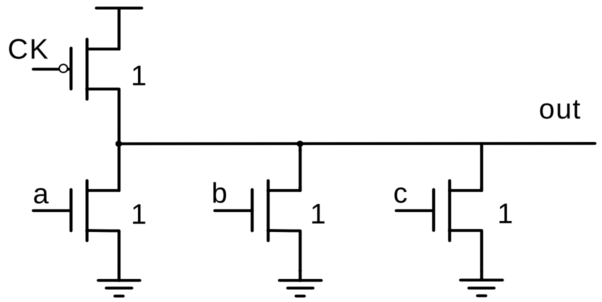

Figure 3-3. Dynamic 3-NOR gate with no keeper.

When a keeper is added, the logical effort of dynamic gates becomes dependent on the size of the gate (unlike static gates). The size of the PMOS keeper and the feedback inverter usually do not change when the gate scales (Figure 3-4). This means that the logical effort of a dynamic gate is no longer independent of its size.

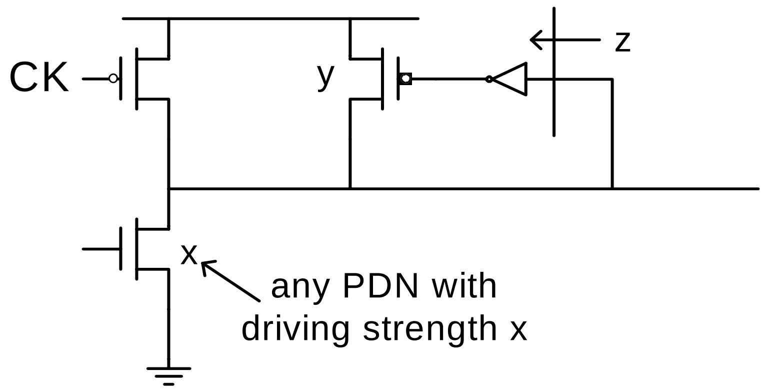

Figure 3-4. Typical dynamic gate with keeper.

With the notation from Figure 3-4, the driving strength for the pull-down transition of this gate is the same as the driving strength of an inverter with the NMOS size of *x*-*y*/2. If the logical effort of the gate without the keeper (i.e. $y = 0$) is *g*, then the logical effort of the whole gate is:

$$g' = g \cdot \left(1 + \frac{y}{2W}\right) \quad \textbf{(3-11)}$$

where *W* is the relative size of the pull-down network *only*. The advantage of using this representation is that the input capacitance of the gate is still $C_{in,i} = g_i W_i$ (using the original g_i).

Theoretically, the intrinsic delay of the gate should also increase when the PMOS keeper is added. However, because the gate becomes increasingly nonlinear, the linear logical effort model is more imprecise. Simulations have shown that delay as a function of load is still linear with the slope given by (3-11) but the intrinsic delay stays the same for reasonably small keepers (Figure 3-5). The graph shows the delay of a dynamic gate without a keeper and with a keeper sized such that its driving strength is 25% the driving strength of the PDN. Therefore, although logical effort theory would predict an increase in the intrinsic delay, we will consider that it stays the same when keepers are added.

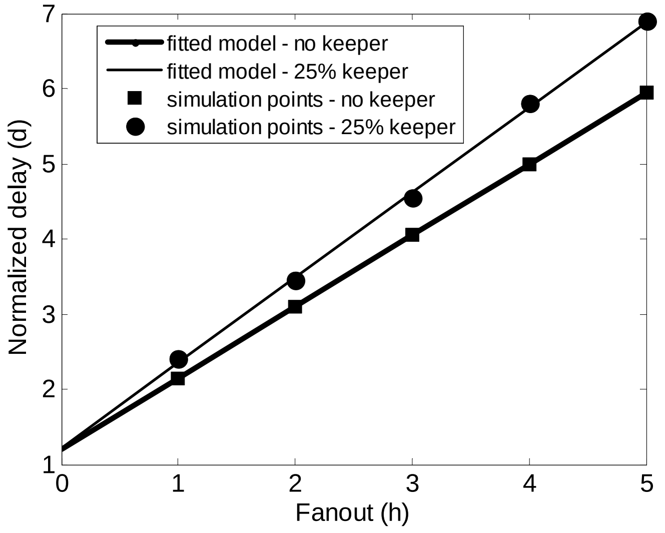

Figure 3-5. Delay of a dynamic gate with and without a keeper.

If the path contains dynamic gates, (3-8) becomes:

$$t_D = \sum_{i=1}^{N} p_i + \sum_{i=1}^{N} \left(1 + \frac{y_i}{2W_i}\right) \cdot \frac{b_i g_{i+1} W_{i+1}}{W_i} \tag{3-12}$$

where y_i contains the keeper sizes (set to zero if the gate is static). (3-12) is a posynomial in both W_i and y_i.

The inverter used by the keeper puts an additional load on the output of the gate. There are two possibilities to account for it:

1. include the effect of this load in the intrinsic delay term of the gate;

2. include it as an additional output load by lumping it into the wire capacitance at that node.

The first solution has the disadvantage that the intrinsic delay will also depend on the size of the gate (because the inverter does not scale with the gate). However, the wire capacitance at the output node of the gate does not scale with the gate either, and therefore the second solution is more convenient. In conclusion, the input capacitance of the inverter (z) will be lumped into the wire capacitance at the output node using the expressions in Section 3.5. If no significant (actual) wire capacitance is defined at the output node (which is common for dynamic gates), a virtual wire can be defined for the purpose of representing the keeper inverter.

3.4 Including signal slopes

The accuracy of the first order model described in the previous sections can be significantly improved by taking signal slopes into consideration. When driven with non-step input, the rise / fall time can be absorbed into the equivalent driving resistance of the gate, $R_{drive,slope}$ (different from the R_{drive} extracted from the I-V characteristics). The output delay is linearly dependent on the input rise / fall time and therefore eq. (3-4) can be modified to account for signal slopes by adding an extra term:

$$t_D = p + g \cdot \frac{C_L}{C_{in}} + \eta \cdot t_{slope,in} \tag{3-13}$$

where $t_{slope,in}$ is the 10%-90% signal slope at the input of the gate. Since (3-13) requires the values of the signal slopes, a new equation is needed to propagate them through the path:

$$t_{slope,out} = \lambda + \mu \cdot \frac{C_L}{C_{in}} + \nu \cdot t_{slope,in} \quad \textbf{(3-14)}$$

Each input of each gate is characterized for each transition by a set of six parameters: p, g, η for the delay, and λ, μ, ν for the slope. This delay model is similar to the level-1 models used in commercial logic synthesis tools [Synopsys04] and yields reasonable accuracy for a first order analysis.

Equations (3-13) and (3-14) can be written as posynomials in the gate sizes, W_i:

$$t_D = p + g \cdot \frac{\sum_{\text{driven gate inputs}} K_i W_i}{K W_{current}} + \eta \cdot t_{slope,in} \quad \textbf{(3-15)}$$

$$t_{slope,out} = \lambda + \mu \cdot \frac{\sum_{\text{driven gate inputs}} K_i W_i}{K W_{current}} + \nu \cdot t_{slope,in} \quad \textbf{(3-16)}$$

Since (3-6) is no longer accurate when using (3-13), a seventh parameter, K_i, is introduced to compute gate input capacitances; K_i is the input capacitance of the gate when $W_i = 1$.

If $t_{slope,in}$ is a posynomial, then t_D and $t_{slope,out}$ are also posynomials in W_i. By specifying fixed signal slopes at the primary inputs of the circuit, the resulting slopes and arrival times at all the nodes will also be posynomials in W_i, suitable for a geometric program.

In the case of dynamic gates, equations (3-13) and (3-14) can be extended to account for keepers in the same way as described in Section 3.3.

3.5 Accounting for wire parasitics

Interconnect loading and delay are modeled through wire capacitance and resistance. In early stages of the design, wire lengths are assumed to be constant and independent of the sizes of neighboring gates.

Wire capacitance can be readily included in equations (3-13) and (3-14) by simply changing the capacitances at the nodes with:

$$C_{node} = C_{wire} + \sum_{gate\ inputs\ at\ node} K_i W_i \tag{3-17}$$

and therefore:

$$t_D = p + g \cdot \frac{C_{wire} + \sum_{driven\ gate\ inputs} K_i W_i}{KW_{current}} + \eta \cdot t_{slope,in} \tag{3-18}$$

$$t_{slope,out} = \lambda + \mu \cdot \frac{C_{wire} + \sum_{driven\ gate\ inputs} K_i W_i}{KW_{current}} + \nu \cdot t_{slope,in} \tag{3-19}$$

Equations (3-18) and (3-19) are posynomials in W_i.

In deep sub-micron technologies the resistive effect of interconnect is playing an increasingly important role. A π model as shown in Figure 3-6 can be used with reasonable accuracy [Amrutur01].

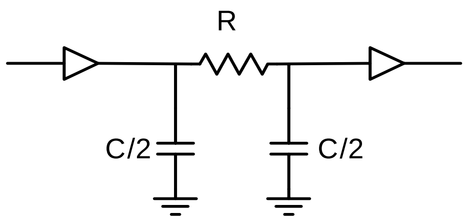

Figure 3-6. π wire model.

According to [Amrutur01] the delay of this structure is:

$$t_D = t_D(R = 0) + \frac{R_{wire} \cdot \left(\frac{C_{wire}}{2} + C_{after\ wire}\right)}{\alpha} \tag{3-20}$$

where α is a fitting parameter close to 1 and $t_D(R=0)$ is (3-18). Equation (3-20) introduces an extra term in (3-18) for each node that has a wire. Since the extra term is posynomial is W_i, the final equation maintains this property.

Each wire is described by two parameters: C_{wire} and R_{wire}. They depend on the particular geometry of that wire: length, width, distance to adjacent wires, metal layer and number of vias. A floorplan sketch is required in order to estimate the geometry of the wire early in the design.

3.6 Energy and area expressions

The energy per operation can be computed using:

$$E = \sum_{all\ nodes} \alpha_i C_i V_{DD}^2 + T_{cycle} \cdot \sum_{all\ gates} W_j P_{leak,j} \qquad \textbf{(3-21)}$$

where α_i is the activity at node i and $P_{leak,j}$ is the average leakage power of gate j. The equation separates switching energy and leakage, and does not account for crowbar current.

Eq. (3-21) is also a posynomial for a fixed cycle time T_{cycle}: the first term is just a linear combination of the gate sizes while the second term is another linear combination of the gate sizes multiplied by the cycle time. If the cycle time is computed at optimization time as the maximum of all path delays, (3-21) becomes a *generalized posynomial*, the optimization problem becomes a *generalized geometric program* (GGP) and it is still easily solvable [Boyd03].

If needed, the area of the design can be estimated using a simple linear combination of the gate sizes:

$$A = \sum_{i=1}^{N} u_i W_i \qquad \textbf{(3-22)}$$

where u_i is the area of the gate when $W_i = 1$. Eq. (3-22) is also a posynomial in W_i.

For datapaths, the area equation (3-22) can be reformulated to express the number of tracks occupied in each bitslice.

3.6.1 Constrained optimization problem

The energy-delay optimization problem can be formulated in two ways:

1. as an energy-constrained delay minimization:

$$\min_{W_i} \quad t_D \text{ such that } \begin{cases} E \le E_{max} \\ C_{in} \le C_{in,max} \\ W_i \ge 1 \\ t_{slope,j} \le t_{slope,max} \end{cases} \qquad \textbf{(3-23)}$$

2. as a delay-constrained energy minimization:

$$\min_{W_i} \quad E \text{ such that } \begin{cases} t_D \le t_{D,max} \\ C_{in} \le C_{in,max} \\ W_i \ge 1 \\ t_{slope,j} \le t_{slope,max} \end{cases} \qquad \textbf{(3-24)}$$

(3-23) and (3-24) are equivalent and they should yield the same result if the optimal values for delay / energy are used in the corresponding constraint in the other problem. Both are (generalized) geometric programs because all the objective and constrained functions are (generalized) posynomials.

3.7 Optimizing combinational circuits using a static timer

Real circuits have more than one logic path. The overall delay of a circuit *D* is the maximum of a set of posynomials, i.e. the delays of all the paths through the circuit. Even small circuits can have a very large number of paths in which case it is not practical to form and expression for *D* by listing all the paths. As an example, the circuit in Figure 3-7 (taken from [Boyd05]) has 7 paths from the primary inputs to the primary outputs.

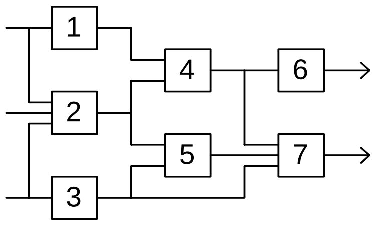

Figure 3-7. A combinational logic block.

If D_i is the delay of block $_i$, the worst case delay is given by:

$$D = \max(D_1 + D_4 + D_6, D_1 + D_4 + D_7, D_2 + D_4 + D_6, \\ D_2 + D_4 + D_7, D_2 + D_5 + D_7, D_3 + D_5 + D_6, D_3 + D_7) \quad \textbf{(3-25)}$$

A simple recursion [Boyd05] can be used to calculate D without enumerating all the paths. An intermediate variable T_i is defined for each gate representing the maximum delay over all paths that start at a primary input and end at gate i. T_i can be interpreted as the latest time at which the output of gate i can transition, assuming the primary input signals transition at $t = 0$. For this reason, T_i is sometimes called the latest signal arrival time at the output of gate i. T_i can be expressed using the following recursion:

$$T_i = \max_{j \in FI(i)} (T_j + D_i) \quad \textbf{(3-26)}$$

where $FI(i)$ is the fanin of gate i. This recursion states that the latest signal arrival time at the output of a gate is equal to the maximum signal arrival time of its fanin gates, plus its own delay. The starting condition is that $T_i = 0$ if i is a primary input. The recursion (3-26) shows that each T_i is a generalized posynomial of the gate sizes W_i since generalized posynomials are closed under addition and maximum. The delay D of the whole circuit is given by the maximum of all T_i's, which is the same as the maximum over all output gates (since the delay of a gate cannot be negative), hence also a generalized posynomial:

$$D = \max_i T_i = \max_i \{T_i | i \text{ an output gate}\} \tag{3-27}$$

For the circuit in Figure 3-7 the recursion is:

$$\begin{aligned}
T_i &= D_1, \quad i = 1, 2, 3 \\
T_4 &= \max(T_1, T_2) + D_4 \\
T_5 &= \max(T_2, T_3) + D_5 \\
T_6 &= T_4 + D_6 \\
T_7 &= \max(T_3, T_4, T_5) + D_7 \\
D &= \max(T_6, T_7)
\end{aligned} \tag{3-28}$$

For this small example, the recursion gives no significant savings over (3-25), but in larger circuits the savings can be dramatic.

If using (3-25) on a multi-path circuit, the constrained optimization problems (3-23) and (3-24) will be generalized geometric programs. With the recursion (3-26), the constrained optimization problems (3-23) and (3-24) can be reformulated as plain geometric programs. For instance (3-23) becomes:

$$\min_{W_i} \; t_D \text{ such that } \begin{cases} E \le E_{max} \\ C_{in} \le C_{in,max} \\ W_i \ge 1 \\ t_{slope,j} \le t_{slope,max} \\ T_i = 0 \quad \text{for primary inputs} \\ T_j \le D \quad \text{for all } j \\ T_j + D_i \le T_i \quad \text{for } j \in FI(i) \end{cases} \tag{3-29}$$

(3-24) can be reformulated is a similar fashion as a plain geometric program.

Although (3-29) introduces an extra variable for each node in the circuit, the constraints are sparse: each of the timing constraints involves only a few variables (assuming reasonable fanins and fanouts). The sparsity can be exploited by the GP solver to obtain great efficiency [Boyd03].

The recursion (3-26) describes a static timer. Therefore, for real circuits with multiple paths, a static timer is the tool of choice for performing all circuit-related computations. Each edge in the timing graph of the circuit translates into a posynomial constraint in the optimization problem.

The optimization problem (3-29) can be extended to account for the difference in $t_{H\text{-}L}$ and $t_{L\text{-}H}$ in a straightforward fashion: two intermediate variables $T_{i,H}$ and $T_{i,L}$ can be defined for each gate, representing the maximum delay over all paths that start at a primary input and end at gate i when computing a 1 or a 0, respectively. The recursion (3-26) becomes:

$$\begin{aligned} T_{i,H} &= \max_{j \in FI(i)} (T_{j,L} + D_{i,L-H}) \\ T_{i,L} &= \max_{j \in FI(i)} (T_{j,H} + D_{i,H-L}) \end{aligned} \quad \textbf{(3-30)}$$

where $D_{i,H\text{-}L}$ and $D_{iL\text{-}H}$ are the delays for the high-to-low and low-to-high transitions at the output of the gate, respectively.

Static timers can have various degrees of sophistication. The custom timer written for this thesis is rather straightforward and does not have the advanced features of the most advanced commercial timers (such as false path detection, simultaneous arrivals etc.). The timer does compute activity factors at the nodes through logic simulation with a large number of random input vectors. The activity factors are used in energy computations and their expressions differ for static and dynamic circuits [Rabaey03].

Due to the modularity of the framework, any available static timer with selectable delay models can be easily used in the optimization with the proper interface.

3.8 Using tabulated models

The linear delay models described in the previous sections lead to geometric programs that can be easily solved by commercial optimizers. Although reasonably accurate for the early stages of the design, they are inadequate when very accurate calculations are required.

For ultimate accuracy, all analytical formulas can be replaced with look-up tables. For instance, (3-13), (3-14) and their respective parameters can be replaced with a look-up table like Table 3-1for each input of each gate and for each transition:

Table 3-1. Example of a tabulated delay and slope model (NOR2 gate, input A, rising transition)

C_{load}/C_{in}	$t_{slope,in}$	t_D	$t_{slope,out}$
1	20 ps	19.3 ps	18.3 ps
...	...	...	...
10	200 ps	229.6 ps	339.8 ps

The table can have as many entries as needed for the desired accuracy and density of the characterization grid. Actual delays and slopes used in the optimization procedure are obtained through linear interpolation between the points in the table. The grid can be non-uniform, with more points in the mid-range fanouts and slopes, where most designs are likely to operate. Linear or more sophisticated interpolation can be used to compute the actual delays and slopes between the points of the characterization grid.

Additional columns can be added to the tables for different logic families for instance if a dynamic gate is characterized this way, the relative size of the keeper to the pull-down network needs can be included, as well.

The analytical models described in the previous sections can be obtained by fitting their respective expressions to the data in the tables. Figure 3-8 shows a comparison of the actual and predicted delay for the rising transition of a gate for a fixed input slope and variable fanout, thus evaluating the accuracy of the analytical models. Since the actual delay is slightly concave in the fanout, the linear model is pessimistic at low and high fanouts and optimistic in the mid-range.

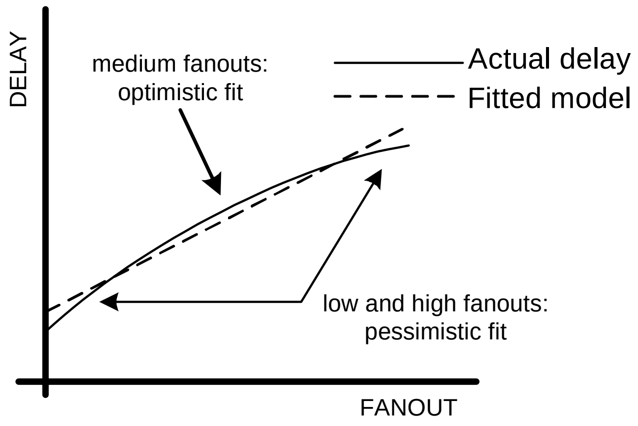

Figure 3-8. Accuracy of fitted models.

The choice of models in the static timer greatly influences the convergence speed and robustness of the optimizer. The circuit optimization framework can use both analytical and tabulated models, depending on the desired accuracy and speed targets. Table 3-2 shows a comparison between the two main choices of models. Closed form analytical models can usually be forced into a convex form as described in this chapter as well as using other mathematical operations such as changes of variables and the introduction of additional (slack) variables [Boyd03].

Table 3-2. Comparison between analytical and tabulated models.

ANALYTICAL MODELS	TABULATED MODELS
- limited accuracy	+ very accurate
+ fast parameter extraction	- slow to generate
+ provide circuit operation insight	- no insight in the operation of the circuit
+ can exploit mathematical properties to formulate a convex optimization problem	- can't guarantee convexity, optimization is "blind"

On the other hand, tabulated models provide excellent accuracy at the points of characterization, but sacrifice all mathematical properties, including convexity.

3.9 Near-optimality boundary

If using tabulated models, the resulting optimization problem, even with using the change of variables from (2-4), cannot be proved to be convex. However, although not absolutely accurate to the last picosecond, the analytical models that describe the behavior of the circuits closely approximate the tabulated models. Thus, the resulting optimization problem is nearly-convex and can still be solved with very good accuracy and reliability by the same optimizers as before [Mathworks05], [Mosek06]. While the global optimality of the result of such a nearly-convex problem cannot be guaranteed, its quality can be checked against a well-defined a near-optimality boundary.

The example in Figure 3-9 shows a comparison of the analytical and tabulated models and the corresponding near-optimality boundary.

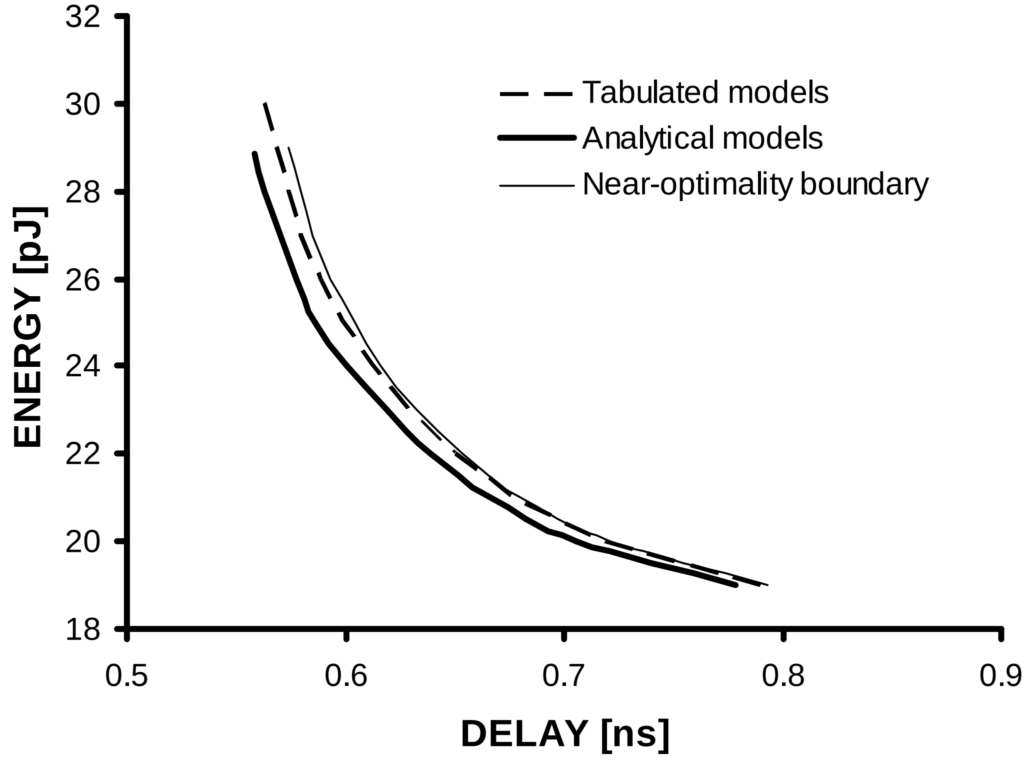

Figure 3-9. Analytical vs. tabulated models and near-optimality boundary.

The figure shows the energy-delay tradeoff curves for an example 64-bit Kogge-Stone carry tree in static CMOS using a 130nm process. The same circuit is optimized using each of the two model choices discussed in this section. Both models show that the fastest static 64-bit carry tree can achieve the delay of approximately 560ps, while the lowest achievable energy is 19pJ per transition. The analytical models are slightly optimistic because the optimal designs exhibit mid-range gate fanouts where the analytical models tend to underestimate the delays (Figure 3-8).

The near optimality boundary is obtained in two steps:

1. perform the optimization using analytical models that guarantee the global optimality of the result;

2. measure the delay and energy of the designs obtained in step 1 using the (accurate) tabulated models.

The near-optimality boundary curve represents a set of *designs optimized using analytical models, but evaluated with tabulated models*. Since those designs are guaranteed to be optimal for analytical models, the boundary is within those models error of the actual global optimum. However, if an optimization using the correct models (tabulated) converges to the correct solution, it will always yield a better result than a re-evaluation of the results of a different optimization using the same models. Therefore, if the optimization with tabulated models is to converge correctly the result must be within the near-optimality boundary i.e. will have a smaller delay for the same energy. If a solution obtained using tabulated models is within the near-optimality boundary it will deemed near-optimal and hence acceptable.

Ultimately, it is not the actual values of the delay and energy, but the values of the design parameters (e.g. the gate sizes W_i) that matter. A correctly converged optimization using tabulated should yield a set of design parameters that is not very far from the set of design parameters resulted from the convex optimization. In most practical examples, the two sets of gate sizes are so close that grid snapping eventually cancels all differences. Only very large transistors (e.g. for buffers driving heavy loads) are usually snapped differently to the grid, but due to their size the relative error remains small.

In a more general interpretation, optimizing using tabulated models is equivalent to using a trusted timing sign-off tool whose main feature is very good accuracy (i.e. a very sophisticated static timer) in the main loop of the optimizer. The result of such an optimization is not guaranteed to be globally optimal. The near-optimality boundary is obtained by running the timing sign-off tool on a design obtained from an optimization that can guarantee the global optimality of the solution. The comparison is fair because the power and performance figures on both curves are evaluated using the same (trusted and accurate) timing sign-off tool.

3.10 Supply and threshold as optimization parameters

In order to tune supply and threshold voltages, the models must include their dependencies. A gate equivalent resistance can be computed from analytical saturation current models (a reduced form of the BSIM3v3 [Toh98], [Garret04]):

$$R_{EQ} = \frac{2}{V_{DD}} \int_{V_{DD}/2}^{V_{DD}} \frac{V_{DS}\mathrm{d}V_{DS}}{I_{DSAT}} = \frac{3}{4}\frac{V_{DD}(\beta_1 V_{DD} + \beta_0 + V_{DD} - V_{TH})}{WK(V_{DD} - V_{TH})}\left(1 - \frac{7}{9}\frac{V_{DD}}{V_A}\right) \quad \textbf{(3-31)}$$

Using (3-31), supply and threshold dependencies can be included in the delay model. For instance, (3-13) becomes (3-32) with (3-14) having a very similar expression:

$$t_D = c_2 R_{EQ} + c_1 R_{EQ}\frac{C_L}{C_{in}} + (\eta_0 + \eta_1 V_{DD})t_{slope,in} \quad \textbf{(3-32)}$$

[Garret04] presents a way to translate (3-32) into a posynomial using a change of variables. The model is accurate within 8% of the actual (simulated) delays and slopes around nominal supply and threshold, over a reasonable yet limited range of fanouts (2.5 to 6). For a +/- 30% range in supply and threshold voltages the accuracy is 15%.

3.11 Example: optimizing a simple circuit using analytical models

This example demonstrates the effect of using different sets of optimization variables on a 64-bit Kogge-Stone carry tree [Kogge73] implemented in static CMOS using a general purpose 130nm process. The most complete analytical models described in this chapter are used for energy and delay.

Figure 3-10 shows the optimal energy-delay curves for the circuit under test in two situations:

1. Only gate sizes are optimized for various fixed supplies (labeled "Fixed V_{DD}" on the figure);
2. Gate sizes and supply are optimized jointly (labeled "Optimal V_{DD}" on the figure).

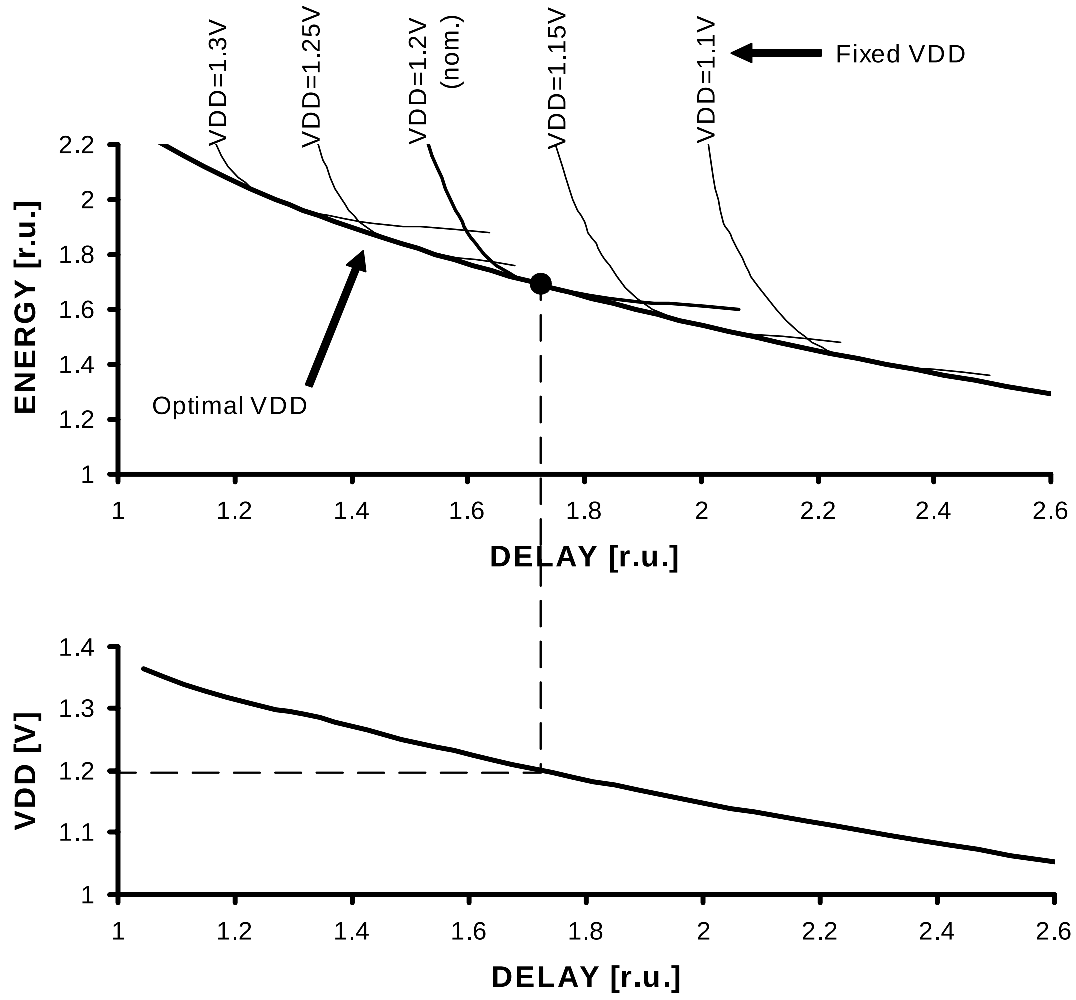

Figure 3-10. Energy - delay tradeoff curves for sizing-only optimization, joint sizing - V_{DD} optimization, and the corresponding optimal V_{DD}.

Figure 3-10 also shows the corresponding optimal supply voltage for case 2. An obvious observation is that by allowing the optimizer to choose the power supply instead of constraining it to a fixed value, the resulting circuit is better in the energy-delay sense (it consumes less at the same delay or is faster at the same energy). A few and interesting conclusions can be drawn from the figure:

- The nominal supply voltage is optimal in exactly one point, where the V_{DD} = 1.2V curve is tangent to the optimal V_{DD} curve. In that point, the sensitivities of the design to both supply and sizing are equal [Markovic04]. The same statement is true for any other value of the supply voltage;
- Power can be reduced in different ways, depending on the position of the design on the fixed V_{DD} curve. At the slow end of the curve (long delays, right side), decreasing V_{DD} and up-sizing the gates to obtain the same performance yields a circuit with lower power. At the fast end of the curve (short delays, left side), the most effective way to reduce the power is to downsize the gates and increase the V_{DD} to compensate for the performance loss. Using an analytical approach, [Markovic04] reached the same conclusion: the relative sensitivities of the design to sizing and supply dictate the most efficient way to reduce the power consumption, even if sometimes it can be counter-intuitive;
- Achieving the last few picoseconds of delay reduction is very expensive in energy because of the large sizing sensitivity (curves are very steep at low delays). This justifies why in the fast region of the curves it is more expensive to speed up the circuit by up-sizing rather than just increasing V_{DD}.

The graphs in Figure 3-10 are obtained using the nominal threshold voltage V_{TH} of the process. If the transistor thresholds can be adjusted (for instance through body biasing), they can be included in the optimization as well. Figure 3-11 adds the optimal energy-delay curve when sizing, V_{DD} and V_{TH} are optimized jointly to the graph from Figure 3-10. Since another variable is set free, the resulting circuit will be again better in the energy-delay sense.

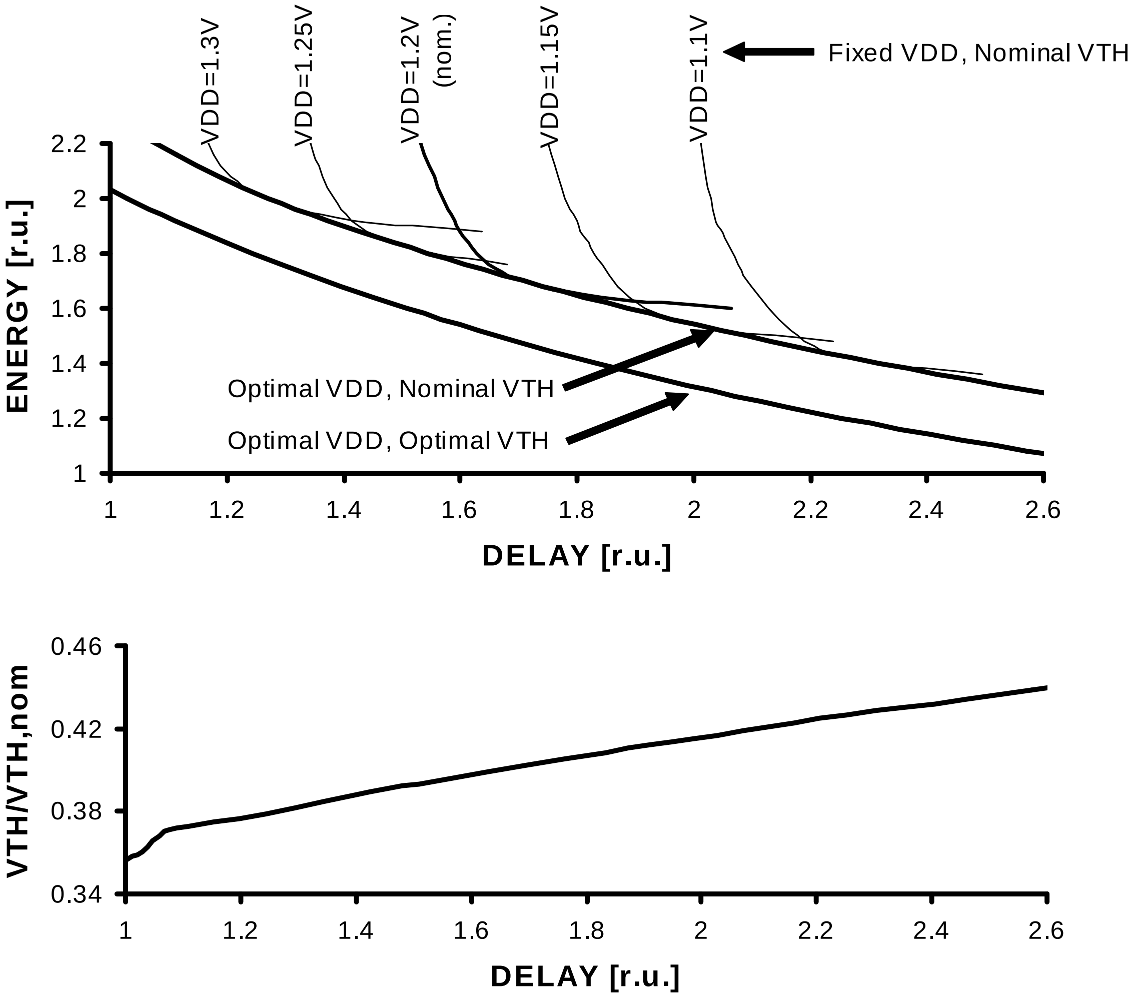

Figure 3-11. Energy - delay tradeoff curves for joint sizing - V_{DD}- V_{TH} optimization, and the corresponding optimal V_{TH}.

The corresponding optimal threshold voltage (one for all the transistors in the circuit) is shown in the bottom graph in Figure 3-11 and it is normalized to the nominal V_{TH} of the process (the fixed value used in Figure 3-10). For such a high activity circuit (a carry tree), the optimal threshold is well below the nominal threshold. The increased leakage due to lower threshold is recuperated by the downsizing afforded by the faster transistors with lower threshold because the circuit is more sensitive to sizing than to thresholds. The conclusion is reversed for low activity circuits, such as memories. The analytical approach in [Markovic04] leads to the same conclusion as well.

4 Case Study: Optimizing 64-bit Adders Using Tabulated Models

The extended example in this chapter demonstrates the use of the most accurate tabulated models on a real-life application: 64-bit carry-lookahead adders. Beyond simply demonstrating the functionality of the optimizer, experimenting with 64-bit adders produces results that are interesting in their own right.

The example is complex and examines the various design choices for 64-bit adders and their impact in the energy - delay space. Several 64-bit adder topologies are optimized in a typical multi-issue high-performance microprocessor environment in order to choose the optimal structure. The optimal structure is then build in a 90nm general purpose CMOS process and the measured results are compared to the predictions of the optimizer.

Section 4.1 motivates the choice of the 64-bit adder example. Section 4.2 describes the optimization setup. Section 4.3 shows a first comparison between a carry-lookahead (CLA) and a straightforward ripple-carry adder (RCA) in order to set a proper scale for the analysis. Six categories of design choices for CLA adders are defined in Section 4.4, establishing a common set of notations. The analysis of their impact in the power - performance space is presented in Section 4.5, highlighting the resulting design rules in the end. The result of this analysis is to pick the optimal adder topology for the chosen optimization setup and environment. Section 4.6 discusses how the technology parameters and adder environment influence this choice. Section 4.7 shows a runtime analysis for the optimizer. The test chip implementing the optimal adder in our 90nm general purpose bulk CMOS process is described in Section 4.8 and the corresponding measurement results are presented in Section 4.9.

4.1 Motivation

Fast and energy efficient single-cycle 64 bit addition is essential for today's high-performance microprocessor execution cores. Wide adders are part of the highest power-density locations on the processor, creating thermal hotspots and sharp temperature gradients [Mathew05]. The presence of multiple ALUs in modern superscalar processors [Fetzer02], [Naffziger06] and of multiple execution cores on the same chip [Naffziger06], [Rusu06], [Golden06] further aggravates the problem, severely impacting circuit reliability and increasing cooling costs. At the same time, wide adders are also critical performance - limiting blocks inside the ALUs, AGUs and FPUs of microprocessor datapaths. Ideally, a datapath adder would achieve the highest performance in a small power budget and have a small layout footprint in order to minimize interconnect delays in the core [Mathew05]. These seemingly contradictory requirements pose a difficult problem when choosing the optimal adder architecture. The designer has several degrees of freedom to optimize the adder for performance and power. There is a choice of tree radices, lateral fanouts, with full or sparse implementation of the conventional or Ling's carry-lookahead equations, as well as the circuit design style.

Although adder design is a well-researched area and the number of published research papers on the subject is very large, fundamental understanding of the impact of various design choices in the power - performance space is still incomplete. Traditionally Kogge-Stone parallel prefix trees [Kogge73] have been the most commonly used in high-performance contexts. Their main features are minimum logic depth, regular structure and uniform fanout. Their main disadvantages are the large number of gates and wires, leading to high power consumption. An implementation of a 64-bit adder using a Kogge-Stone tree is reported in [Park00]. The number of nodes and connections in the tree can be reduced by trading it off for increased logic depth, such as the sparse Han-Carlson tree [Han87]. Many sparse tree implementations have been reported in recent years, with sparseness of 2 [Mathew01], [Kao06], sparseness of 4 [Naffziger96], [Shimazaki03], or variable [Mathew05].

An alternate logic optimization investigates different implementations of the carry equations. Ling's equations can lead to simplifications of parallel prefix nodes [Ling81], [Doran88] and reduced transistor stack height.

Existing adder comparisons can be classified in two categories by their use of optimization techniques:

- without optimization:
    - simulation-based study of the impact of wires on the adder delay with fixed gate sizes [Huang00];
    - logic manipulation-based study of the impact of carry tree topology on logic depth [Beaumont-Smith01];

- with optimization:

  •optimal transistor sizing for minimum delay using logical effort [Dao01];

  •optimal transistor sizing in the energy - delay space using a combination of logical effort and net load assignment for static adders [Dao03].

This chapter presents an thorough analysis of the design tradeoffs for 64-bit carry-lookahead adders in a typical high-performance microprocessor environment and the actual design of an optimal adder in a general purpose 90nm CMOS process. The impact in the power - performance space is analyzed for design choices in six categories:

1. set of equations
2. logic style
3. carry tree radix
4. carry tree lateral fanout
5. carry tree sparseness factor
6. sizing strategy

The design tradeoff analysis extends the conclusions from our publication [Zlatanovici03]. The implementation example is a more detailed description of our design from [Kao06].

4.2 Optimization setup

In order to make a fair comparison between various implementations, a generic 64-bit adder structure is constructed, as shown in Figure 4-1. It is a conventional architecture, featuring a carry tree, a sum-precompute block and a sum-select multiplexer.

The carry tree is composed of two sub-trees: a generate sub-tree, implementing the AND-OR equations of the generate terms (G) and a propagate sub-tree implementing the AND equations of the propagate terms (P). The sum-precompute block precomputes the sums at each bit assuming incoming carries of 0 and 1 for S0 and S1 respectively. The final multiplexer selects the appropriate sums based on the carry signals computed by the carry block. In most cases, the carry tree and the sum-select multiplexer are the critical path, while the sum-precompute block is non-critical.

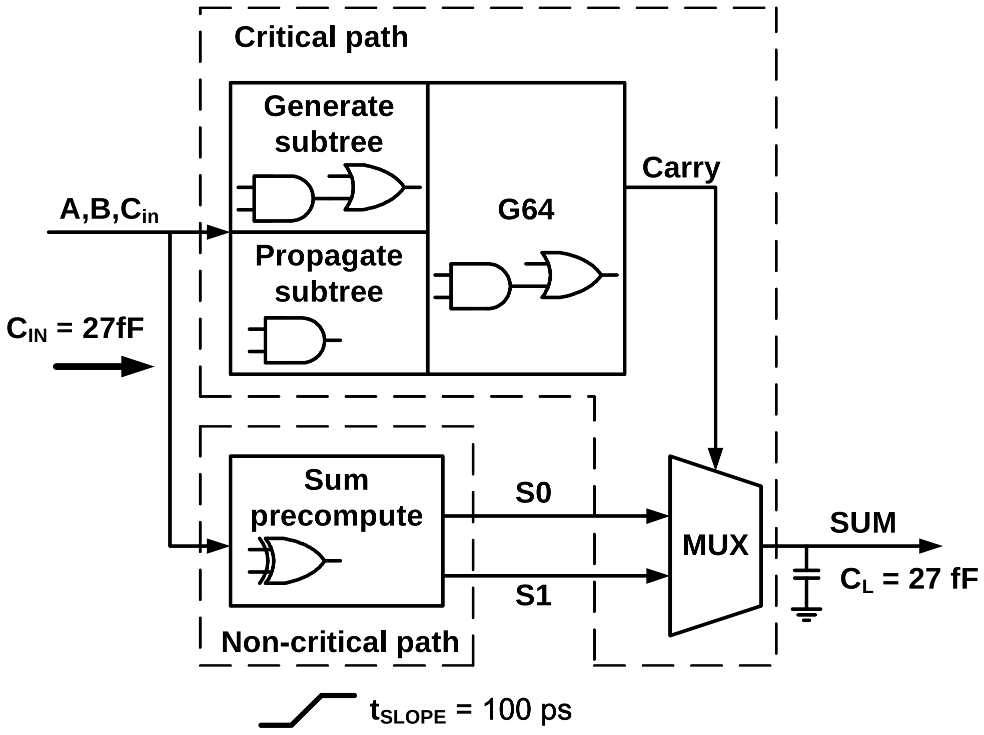

Figure 4-1. Generic 64-bit adder block diagram and optimization setup.

In the subsequent sections, this generic architecture is implemented in a general purpose 90nm CMOS process using various choices for the set of equations, logic family, carry tree architecture, layout strategy and technology parameters. The optimization problem (3-23) is solved at different energy constraints for each architecture and a corresponding optimal energy - delay tradeoff curve is plotted in order to analyze the impact of these choices in the energy-delay space.

The external environment is the same for all the adders optimized in the subsequent sections and it is also shown in Figure 4-1. In a high-performance integer execution unit, the microarchitecture sets the constraints for the adder design. The selected bitslice height of 24 metal tracks accommodates a wide-issue architecture. The bitslice height and adder topology determine the length of the long "prefix" wires in the carry tree. In this study, the input capacitance per bitslice is limited to 27fF and the output loading capacitance of the adder equals its input capacitance, assuming that a buffer would be used to drive the local loop-back bus. The output capacitance and bitslice height are changed only in the analysis in Section 4.6 to reflect different adder environments. The slopes in the circuit are constrained to 100ps, for signal integrity reasons.

4.3 Carry lookahead vs. ripple carry

Conventional wisdom says that ripple carry adders (RCA) have always the lowest power, due to their extreme simplicity. RCAs are usually considered inadequate for wide adders due to the linear dependency of the delay as a function of the bitwidth. Instead, carry lookahead adders (CLA) have a logarithmic dependency and are the preferred solution in all microprocessors. Since the RCA is a reference design in all cases, it is interesting to see what limits it sets for all other adder designs in the energy - delay space. Figure 4-2 shows a comparison of a RCA with a straightforward implementation of a CLA in static CMOS. The delay is normalized to fanout-of-4 (FO4) inverter delays and the energy is normalized such that the RCA has a relative energy of 1. The tradeoff curve corresponding to the CLA is extended to the right at the same energy level for comparison purposes only.

As shown in Figure 4-2, a 64 bit CLA can be easily made about 10 times faster than the corresponding RCA. The RCA consumes about half the power of the smallest CLA while being 6 times slower than it.

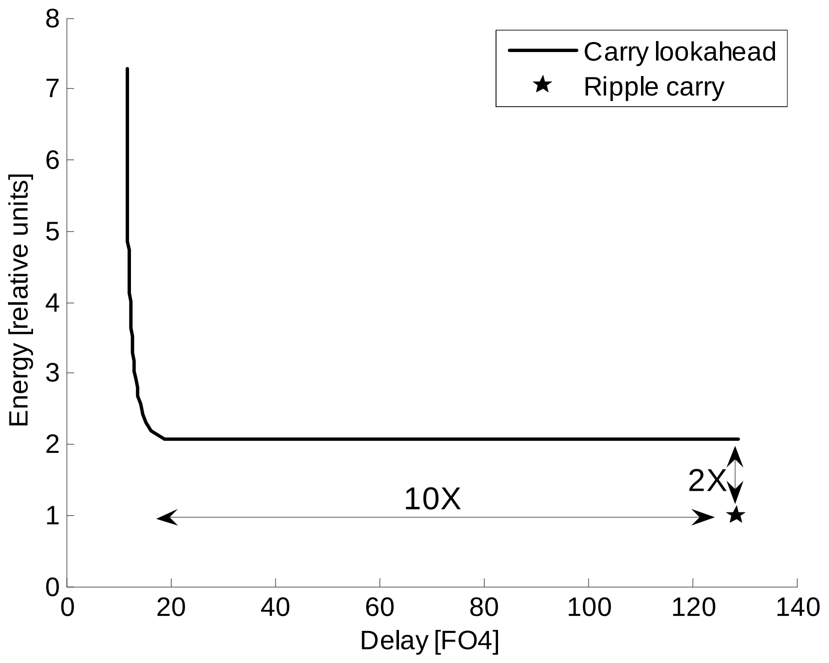

Figure 4-2. Comparison of 64 bit carry lookahead and ripple carry adders in the energy-delay space.

Due to the nature of the comparison, the graph in Figure 4-2 is inherently badly scaled. However, the graph helps set the scale for all subsequent comparisons, by showing where the ultimate lowest power limit is, and the cost in performance required to achieve that limit.

4.4 Design choices

4.4.1 Set of logic equations

The conventional implementation of the carry-lookahead adder uses the well known generate-propagate equations [Rabaey03]. If a_i and b_i are the input operands, the sum bits S_i can be computed using the following recursive equations:

$$\begin{aligned} g_i &= a_i b_i \\ p_i &= a_i \oplus b_i \\ G_i &= g_i + p_i G_{i-1} \\ S_i &= a_i \oplus b_i \oplus G_{i-1} \end{aligned} \qquad \textbf{(4-1)}$$

The quantity that is propagated to the next stage is the carry out of bit *i*. Ling's equations [Ling81] are an alternative to the classical CLA. These equations propagate Ling's pseudo-carry H_i instead:

$$\begin{aligned} g_i &= a_i b_i \\ t_i &= a_i + b_i \\ H_i &= g_i + t_{i-1} H_{i-1} \\ S_i &= t_i \oplus H_i + g_i t_{i-1} H_{i-1} \end{aligned} \qquad \textbf{(4-2)}$$

The potential advantage of using Ling's equations comes after unrolling the recursions [Doran88]. For instance, unrolling the recursions (4-1) and (4-2) for a group of 4 bits results in:

$$\begin{aligned} G_3 &= g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 \\ H_3 &= g_3 + g_2 + t_2 g_1 + t_2 t_1 g_0 \end{aligned} \qquad \textbf{(4-3)}$$

The terms in the H_3 equation have fewer factors. When implemented in CMOS, Ling's scheme requires fewer transistors in the stack of the first gate. However, the sum computation for Ling's pseudo-carry equations is more complicated. For a conventional CLA, the sum-precompute block from Figure 4-1 must implement:

$$\begin{aligned} S_i^0 &= a_i \oplus b_i \\ S_i^1 &= (a_i \oplus b_i)' \end{aligned} \qquad \textbf{(4-4)}$$

where S_i^0 is the precomputed value of the sum for an incoming carry (G_i) of 0 and S_i^1 for an incoming carry of 1. If a Ling CLA scheme is used, the sum-precompute equations are:

$$\begin{aligned} S_i^0 &= a_i \oplus b_i \\ S_i^1 &= a_i \oplus b_i \oplus (a_{i-1} + b_{i-1}) \end{aligned} \qquad \textbf{(4-5)}$$

and they require more hardware for their implementation. Ling's equations effectively move some complexity from the carry tree into the sum-precompute block.

4.4.2 Logic style

The usual choices for the logic style when designing a 64-bit adder are static CMOS, domino, compound domino and compound domino with stack height reduction.

In this context "domino logic" is a family in which a dynamic gate is always followed by a (skewed) static inverter. By contrast, in "compound domino logic" a dynamic gate can be followed by an arbitrary (skewed) static gate, such as an AND-OR-INV that can merge a number of carry signals.

A variant of compound domino takes advantage of the possibilities to reduce the transistor stack height in the gates by reformulating the logic equations [Park00]. By using the absorption property of the generate and propagate signals, the unrolled carry lookahead equations (4-3) can be rewritten as:

$$\begin{aligned} G_3 &= (g_3 + g_2 + g_1 + p_1 g_0)(g_3 + p_3 p_2) \\ H_3 &= (g_3 + g_2 + g_1 + g_0)(g_3 + t_2 t_1) \end{aligned} \quad \textbf{(4-6)}$$

These equations can be implemented by two dynamic gates (one for each parenthesis) followed by a (skewed) static NOR2 gate. Such an implementation style reduces the tallest transistor stack height by 2 in each of the equations (4-3) and is further referred to as "compound domino with stack height reduction".

4.4.3 Carry tree radix

The focus of this chapter is on 64-bit adders; however, all carry tree figures depicted in this and the next two subsections are for 16-bit trees, for simplicity. Carry-in and carry-out signals are omitted from the figures for the same reason, although they are included in the optimizations.

The common legend for all carry tree drawings is (similar to [Huang00]):

- white square: generate / propagate gate;
- black circle: carry merge gate;
- white rhomboid: sum select multiplexer.

The *radix* of a carry tree is defined as the number of carries merged in each step. A radix 2 carry tree is shown in Figure 4 and a radix 4 tree in Figure 5. The radix determines the number of stages needed in a tree in order to compute all the required carries. A 64-bit adder requires 3 radix 4 stages or 6 radix 2 stages. Mixed-radix trees can also be used; for instance a 64-bit carry tree can be implemented in 4 stages using a radix 4-3-3-2 scheme.

An alternate term for radix that is used sometimes in literature is *valency*. The full meaning of the figure captions will be explained in the section defining tree stage lateral fanout.

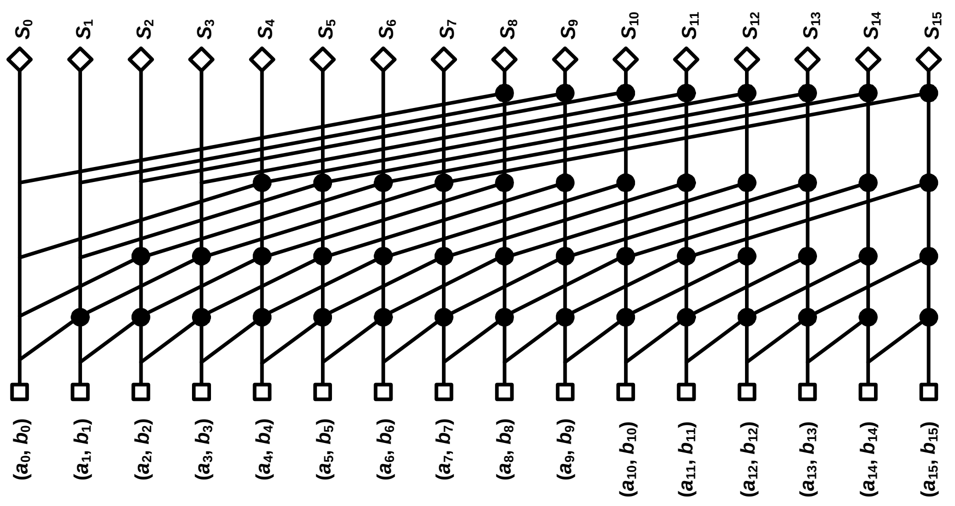

Figure 4-3. 16-bit radix 2 1-1-1-1 (Kogge-Stone) carry tree.

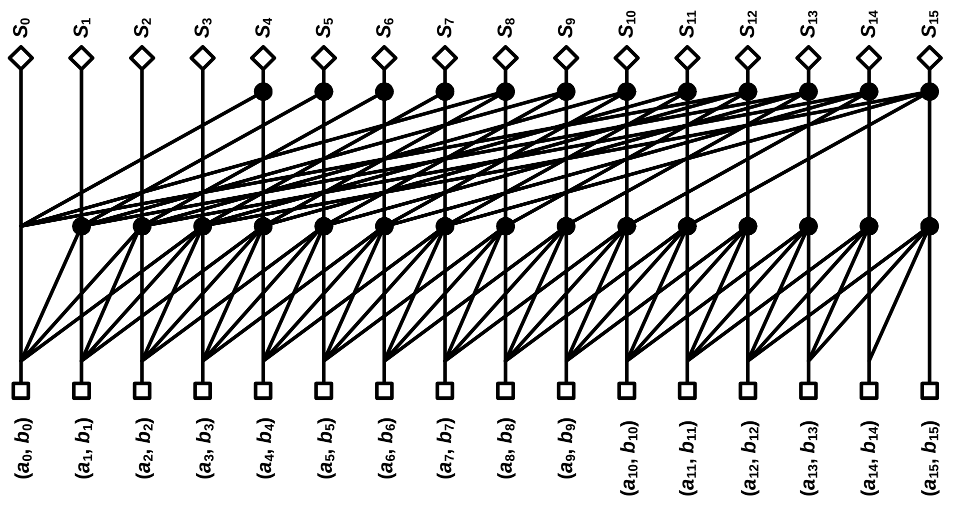

Figure 4-4. 16-bit radix 4 1-1 (Kogge-Stone) carry tree.

4.4.4 Tree stage lateral fanout

Ladner and Fischer introduced the minimum-depth prefix graph [Ladner80] based on earlier theoretical work [Ofman63]. The longest lateral fanning wires go from a node in the tree to N/2 other nodes. Capacitive fanout loads become particularly large for later levels in the graph, as increasing logical fanout combines with the increasing span of the wires. Kogge and Stone [Kogge73] addressed this fanout issue by introducing the "recursive doubling" algorithm. Using the idempotency property, the lateral fanout at each node is limited to one, at the cost of a dramatic increase in the number of lateral wires and logic gates at each level. Knowles introduced a new family of minimum-depth adders [Knowles01] spanning all the range from Ladner - Fischer to Kogge - Stone, with various lateral fanouts. Each member of the family is uniquely labeled by listing the lateral fanouts at each level, from the stage nearest to the output, back towards the inputs. Several terms are used for “lateral fanout” in the literature, among which are “branching” [Beaumont-Smith01] and simply “fanout” [Harris03]. In this thesis the term “lateral fanout” will be used with the same meaning as in [Knowles01], in order to avoid confusion with the electrical fanout of the gate.

A sample of the Knowles family of radix 2 minimum depth trees are shown in Figure 4-5 (8-4-2-1 tree, the original Ladner-Fischer tree), Figure 4-6 (4-4-2-1 tree) and Figure 4-7 (2-2-2-1 tree). The tree in Figure 4-3 is a 1-1-1-1 tree (Kogge-Stone). The Knowles labeling can be extended to higher radix trees as well - for instance the radix 4 tree from Figure 4-4 is a 1-1 tree, hence a Kogge-Stone tree.

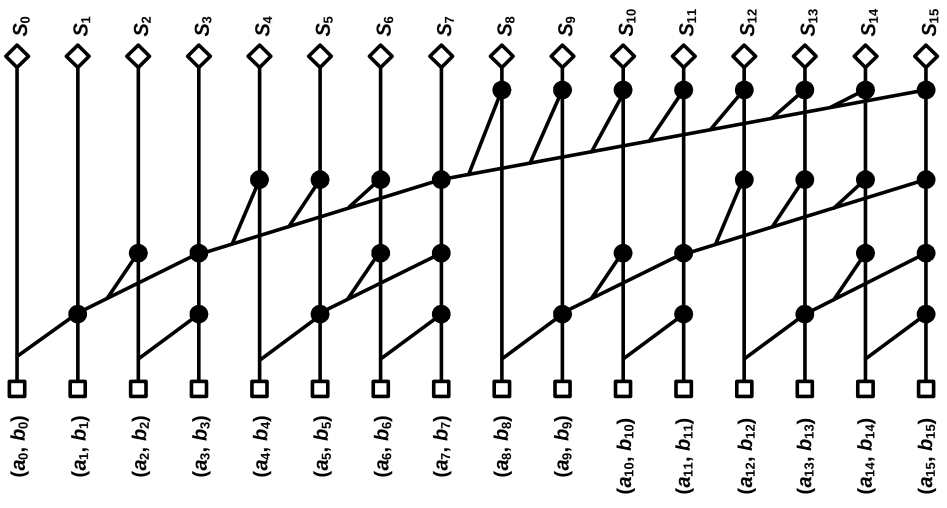
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15}
(a_0, b_0) (a_1, b_1) (a_2, b_2) (a_3, b_3) (a_4, b_4) (a_5, b_5) (a_6, b_6) (a_7, b_7) (a_8, b_8) (a_9, b_9) (a_{10}, b_{10}) (a_{11}, b_{11}) (a_{12}, b_{12}) (a_{13}, b_{13}) (a_{14}, b_{14}) (a_{15}, b_{15})

Figure 4-5. 16-bit radix 2 8-4-2-1 (Ladner - Fischer) carry tree.

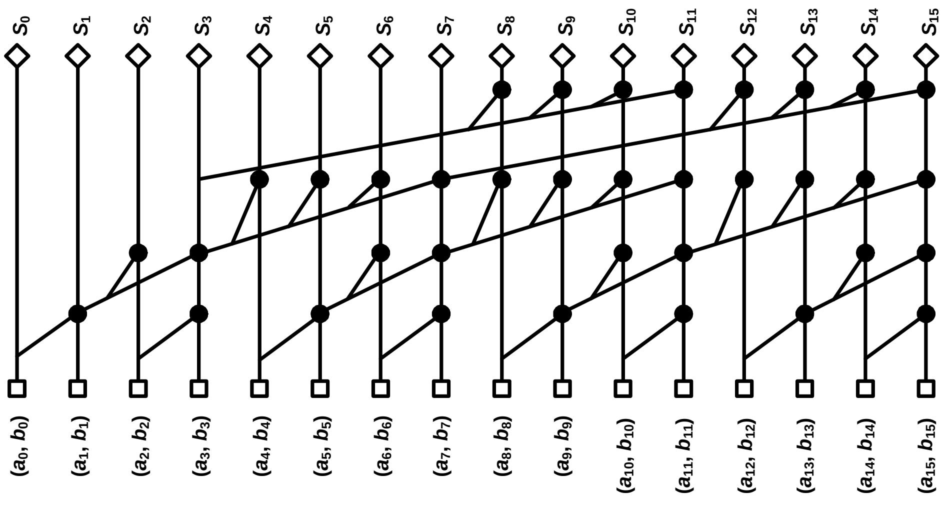
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15}
(a_0, b_0) (a_1, b_1) (a_2, b_2) (a_3, b_3) (a_4, b_4) (a_5, b_5) (a_6, b_6) (a_7, b_7) (a_8, b_8) (a_9, b_9) (a_{10}, b_{10}) (a_{11}, b_{11}) (a_{12}, b_{12}) (a_{13}, b_{13}) (a_{14}, b_{14}) (a_{15}, b_{15})

Figure 4-6. 16-bit radix 2 4-4-2-1 carry tree.

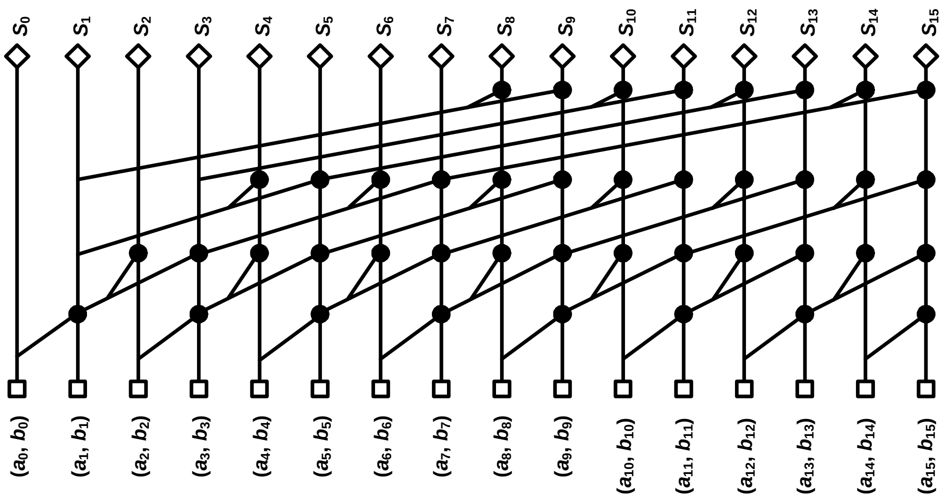

Figure 4-7. 16-bit radix 2 2-2-2-1 carry tree.

4.4.5 Tree sparseness

All the carry trees discussed so far are "full" trees because they compute the final carry at every bit. However, it is possible to compute only some of the carries and select the sum based only on the available carry bits. For instance, one can compute only the even carries (H_0, H_2, H_4,..., H_{62}) in the CLA block and use them to select the multiplexers in the sum-select stage. The gates and wires corresponding to the eliminated carries are pruned, dramatically reducing the complexity of the tree. The resulting tree is sparse, with a sparseness of 2, as exemplified in Figure 4-8.

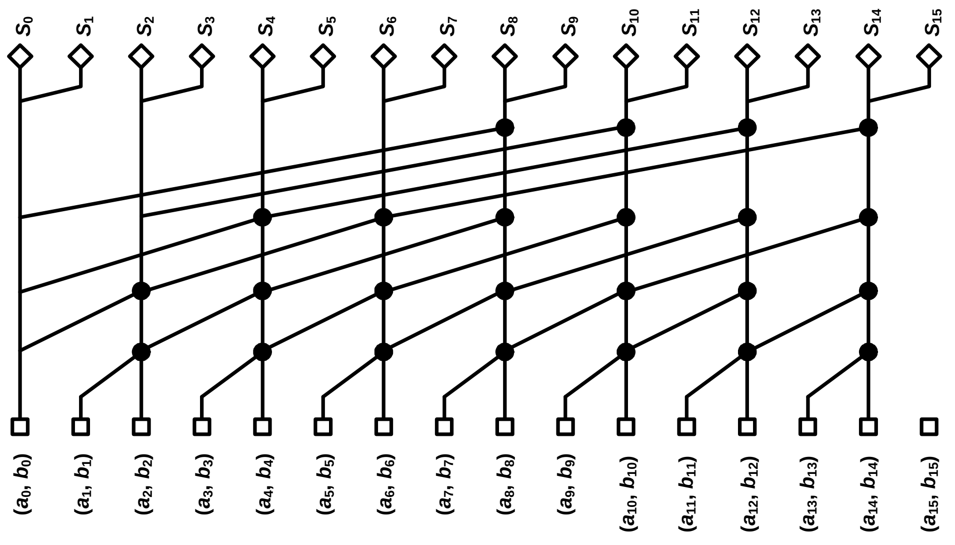

Figure 4-8. 16-bit radix 2 1-1-1-1 sparse-2 carry tree.

The most significant bits are not used in the carry computation because the last carry is not computed. However, those bits are used in the pre-computation of the sum.

The sum-precompute block is more complex in sparse trees, but still can be removed from the critical path. Even order pre-computed values for the sum are given by eq. (4-5) for Ling's carry scheme, but odd order sums must be pre-computed by unrolling the carry recursion (4-2) once:

$$\begin{aligned} S_i^0 &= a_i \oplus b_i \oplus (a_{i-1} b_{i-1}) \\ S_i^1 &= a_i \oplus b_i \oplus [a_{i-1} b_{i-1} + (a_{i-1} + b_{i-1})(a_{i-2} + b_{i-2})] \end{aligned} \tag{4-7}$$

Sparseness can be larger than two and can be applied to any carry tree. A sparse-4 version of the tree in Figure 4-3 is shown in Figure 4-9. In this case, the carry recursion must be unrolled once, twice and three times every four bitslices in the sum precompute block.

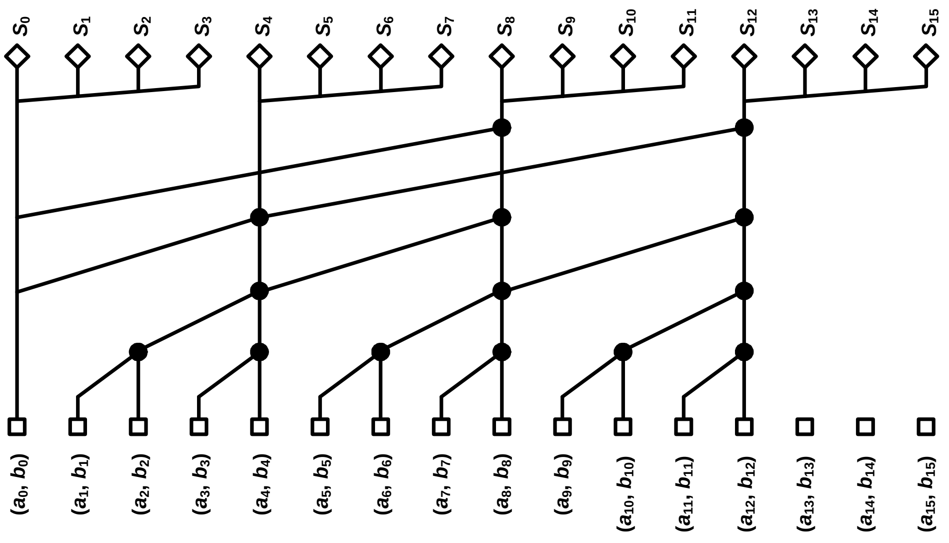

Figure 4-9. 16-bit radix 2 1-1-1-1 sparse-4 carry tree.

The sparse-2 and sparse-4 versions of the Ladner-Fisher tree are shown in Figure 4-10 and Figure 4-11. It should be noted that sparseness reduces the actual lateral fanout in the tree: the third stage has a fanout of 8 in the full tree, 4 in the sparse-2 tree and 2 in the sparse-4 tree. However, in order to uniquely identify the tree and keep sparseness as an independent variable, *the fanout notation corresponding to the full tree is used for all its sparse versions*. Therefore, although the tree from Figure 4-11 appears as a 2-2-1-1 at first sight, it is labeled as "8-4-2-1 sparse-4" because it is a sparse-4 version of the full 8-4-2-1 tree (from Figure 4-5). The design from [Mathew05] is an implementation of a 64-bit version of this tree (Figure 4-11).

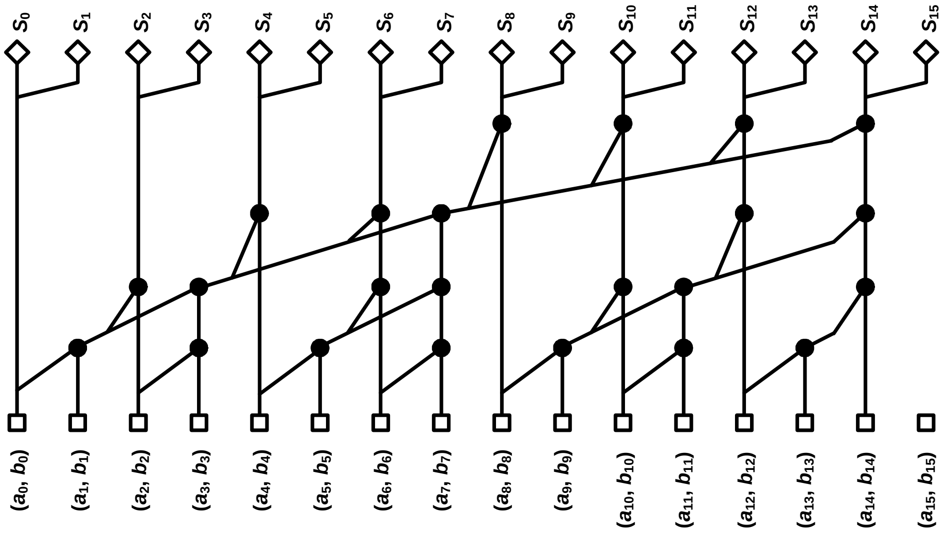

Figure 4-10. 16-bit radix 2 8-4-2-1 sparse-2 carry tree.

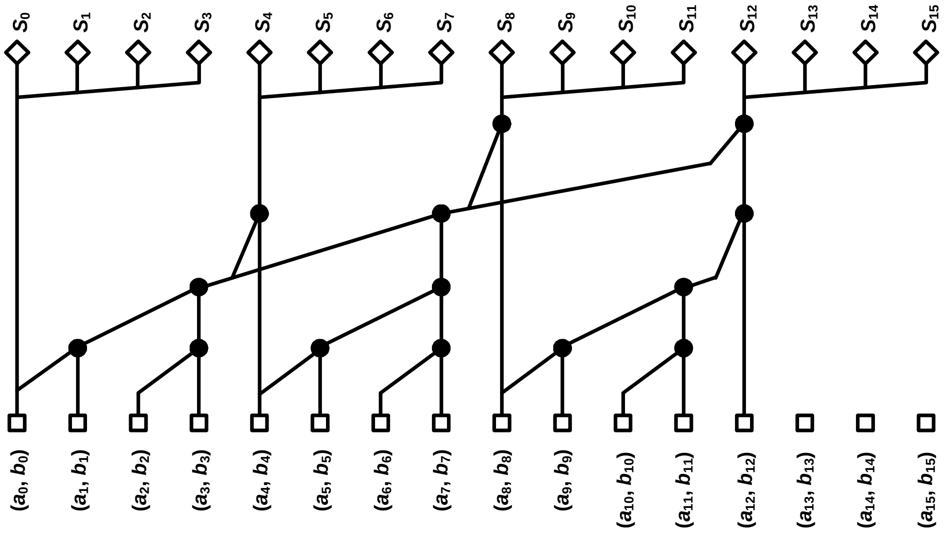

Figure 4-11. 16-bit radix 2 8-4-2-1 sparse-4 carry tree.

By using this notation, the space of minimum-depth carry trees can be represented along 3 independent axes: radix, lateral fanout and sparseness. Figure 13 shows this representation, highlighting the location of the well known Kogge-Stone and Ladner-Fischer tree and of a few recent implementations.

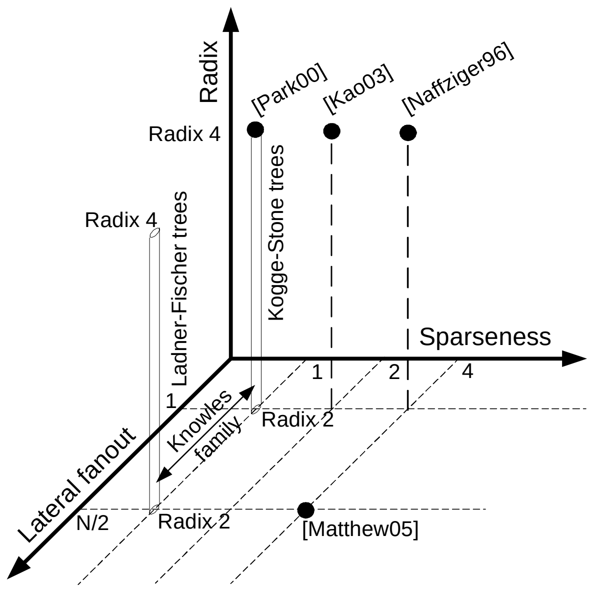

Figure 4-12. Three-dimensional representation of minimum-depth carry tree space.

4.4.6 Sizing strategy

An adder is a regular structure that is suited for bitsliced implementations. A common sizing strategy is to *group* all identical gates in a stage such that they have the same size. Figure 4-13 shows the common way to group gates in a 16-bit carry tree. All gates with identical function in a stage a grouped, meaning they have the same size. For instance, all G2 gates in the first stage are identical and have the same size. Similarly, all P2 gates in the first stage are identical and have the same size (although not the same size as the G2 gates).

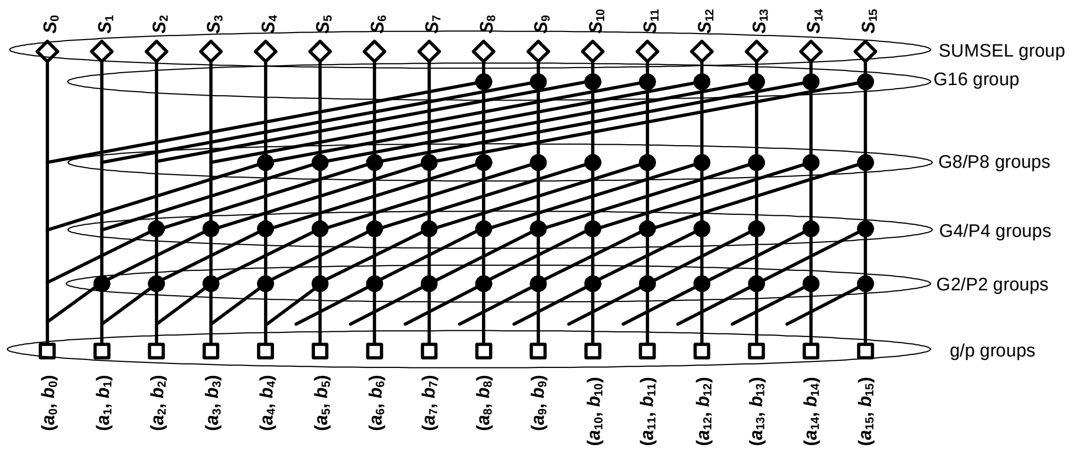

Figure 4-13. Gate grouping in a carry tree.

Gate grouping speeds up the design process by reducing the number of variables in the optimization and allowing layout reuse. The size and distribution of groups can be varied. For example, to make the timing window in footless domino implementation, some of the lower bits in higher stages can be downsized or footed. In the extreme case, each gate could be individually sized (*flat sizing*).

4.5 Exploration of design choices

4.5.1 Set of logic equations

Figure 4-14 shows the energy - delay tradeoff curves for radix 4 and radix 2 domino adders implemented in domino logic using classical CLA and Ling equations.

At high speeds, where the carry tree is in the critical path, the switch to Ling's equations is advantageous: by lowering the stack height in the first stage, Ling's equations allow the first gate in the carry tree to be larger for the same input capacitance. When driving the same load (next stage and corresponding interconnect), the speed increases. At very low speeds with most gates minimum sized, the sum-precompute block appears in the critical path and the simple classical CLA equations offer the lowest power.

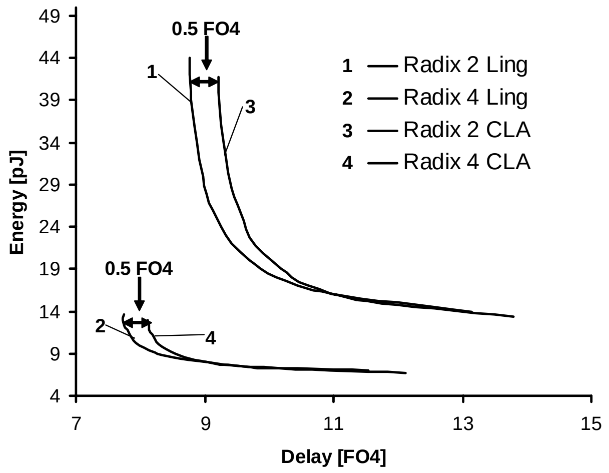

Figure 4-14. Energy - delay tradeoff curves for domino adders implemented using classical CLA and Ling equations.

4.5.2 Logic style

Figure 4-15 shows the optimal energy - delay tradeoff curves for 64-bit adders that implement the architecture from Figure 4-1 in the four logic families from Section 4.4.2. Two curves are plotted for domino logic - corresponding to a radix 2 and radix 4 architecture of the carry tree, respectively. The static and compound domino implementations use radix 2 architectures for the carry tree.

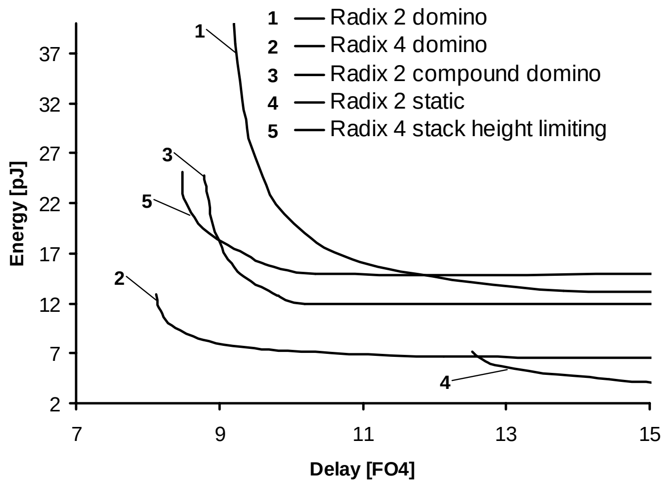

Figure 4-15. Energy - delay tradeoff curves for adders implemented in various logic families.

For comparison purposes, horizontal lines in the figure extend the energy level of "point 2" from Figure 2-2 and represent only additional available slack. Figure 4-15 shows that for long cycle times a static implementation is preferred due to its lower power consumption. Due to their high activity factors, dynamic circuits cannot translate the extra-slack into power savings for cycle times beyond 9 FO4. However, static adders can only achieve the minimum 12.5 FO4 delays when designed at "point 1". If the required delay is shorter than 12.5 FO4, a dynamic design is required.

The radix-4 domino design has the lowest energy - delay curve in Figure 4-15, at high speeds. Radix-2 compound domino design approaches the radix-4 domino design in speed, but has higher energy. Radix-2 compound domino adders can be implemented in the same number of stages as a radix-4 domino. However, they suffer from an increased impact of the wires: compound domino adders have dynamic wires that traverse multiple bitslices and prudent design practices require that such wires be shielded. As stated in the beginning of Section 4.2, all adders are optimized in the exact same conditions, including wires; however, the shields required by compound domino consume routing resources, increasing the actual length of the wires as well as their capacitance. This extra-capacitance is in the critical path, leading to an increase in delay and power consumption. This reduces the performance of compound domino adders and is taken into account in Figure 4-15.

The logic design of the adder that uses stack height limiting, implemented in compound domino, recuperates the speed loss due to extra wiring capacitance because all long wires are driven by static gates. However, these long wires must be driven by a stack of 2 PMOS transistors in the NOR2 gates characteristic for this logic implementation. While providing relatively short delays due to a small number of stages and reduced stack height, the large PMOS transistors in the NOR2 gates and the high count of transistors and clocked gates make this logic family less attractive because of increased power consumption.

4.5.3 Carry tree radix

Figure 4-16 shows the optimal energy - delay tradeoff curves for 64-bit domino adders implemented with carry trees of different radices.

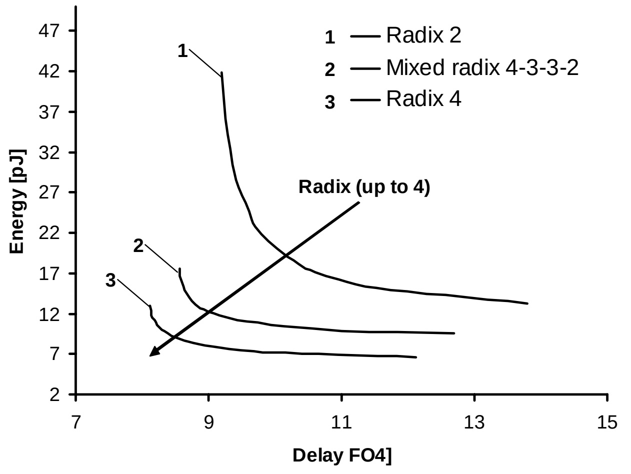

Figure 4-16. Energy - delay tradeoff curves for adders implemented with Kogge-Stone trees of various radices.

For the loading conditions described in Figure 4-1, radix 4 trees are closer to the optimal number of stages as described in [Sutherland99] and achieve the best performance and lowest power.

Using the logical effort formalism from [Sutherland99] it can be easily shown that if wire loads and stack effects are ignored, all carry trees have the same branching effort and the same logical effort. For the same external loading conditions (i.e. the same electrical effort), the overall effort of the carry tree is a constant regardless of the chosen architecture. For the light loading conditions from Figure 4-1, the number of stages should be kept at a minimum and hence the highest feasible radix should be used. As shown in Figure 4-16, radix 2 adders (6 stages) are the slowest, mixed radix 4-3-3-2 adders (4 stages) have significantly better performance and radix 4 adders (3 stages) achieve best performance.

The result holds for heavier loadings as well: if the optimal number of stages increases, it is always better to drive high loads with buffers (inverters) rather than with the last gates of the adder (complex AND-OR-INV gates or multiplexers).

The limiting factors in the above rule are the same as the simplifying assumptions in the logical effort analysis: wire loads and stack effect.

- Higher radix trees will have longer wires closer to the inputs: the first stage of a radix 4 tree needs to drive wires spanning 12 bitslices; the first stage of a mixed radix 4-3-3-2 tree needs to drive wires spanning 8 bitslices; the first stage of a radix 2 tree needs to drive wires spanning only 2 bitslices. Therefore, the advantage of a higher radix tree is eroded in processes with a high wire capacitance - such as modern deep submicron technologies. Section 4.6 elaborates on this aspect.
- The maximum transistor stack height in a carry tree is usually equal to the radix of the stage. However, the tallest stack that can be used is effectively limited by the poor V_{DD}/V_{TH} ratio of deep submicron technologies. A stack that is too high can cause a slow-down beyond the simple predictions of logical effort and significant signal slope degradation. In such a situation, stack height limiting compound domino can help maintain the speed of the adder (with a significant cost in power).

4.5.4 Tree stage lateral fanout

The optimal energy - delay tradeoff curves for radix 2 trees across the Knowles family are presented in Figure 4-17 (full trees), Figure 4-18 (sparse-2 trees) and Figure 4-19 (sparse-4 trees).

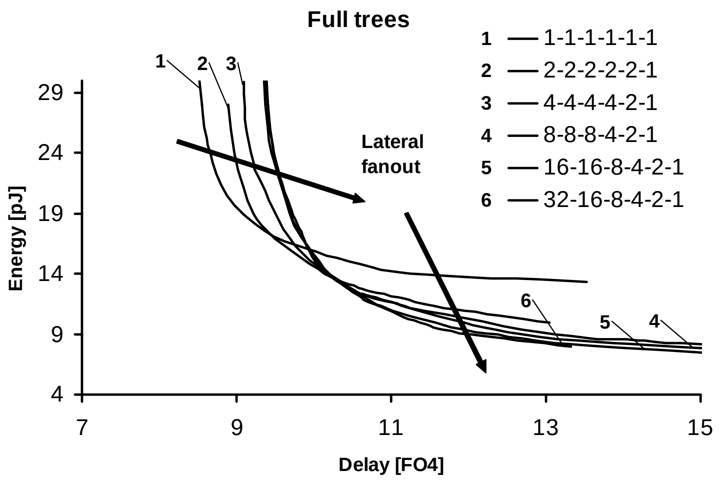

Figure 4-17. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 full trees with different lateral fanouts.

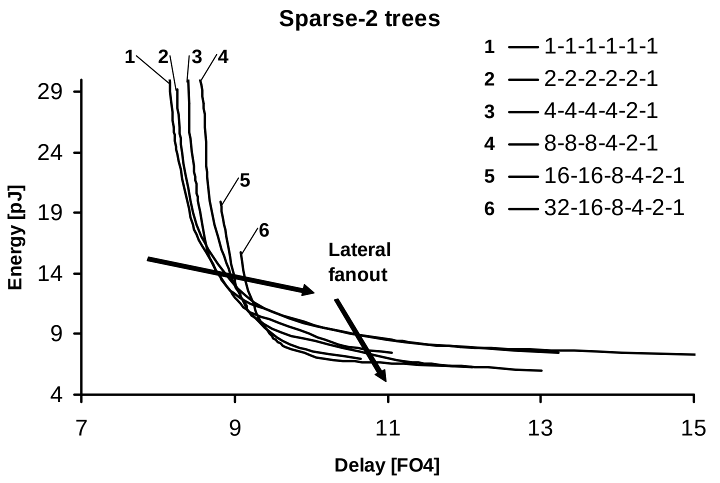

Figure 4-18. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 sparse-2 trees with different lateral fanouts.

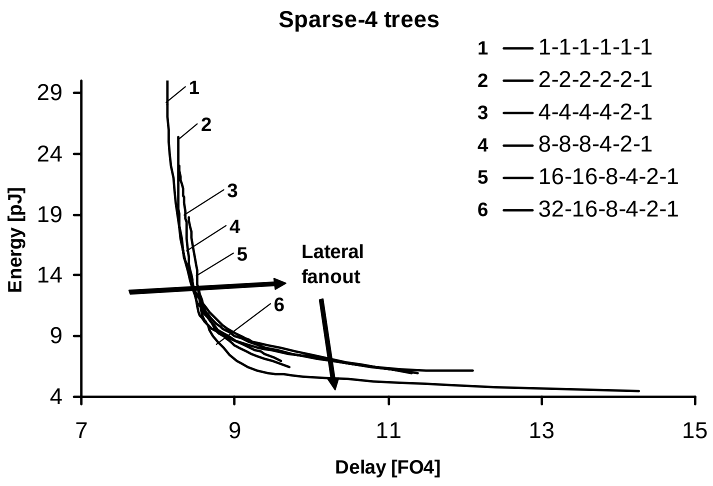

Figure 4-19. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 sparse-4 trees with different lateral fanouts.

Trees with high fanout have low degrees of redundancy, with the Lander-Fischer tree computing the minimum number of carries and having the highest loading along the critical path. At the other extreme, a Kogge-Stone tree computes all the carries and uses the redundancy to reduce the fanout, offering the lowest loading along the critical path. Consequently, the tradeoff along the fanout axis in the carry tree space is length of wires and gate fanout vs. number of gates.

Figure 4-17 shows that for full trees, the lower the fanout, the higher the maximum speed. At the other end of the curves, the higher the fanout, the lower the minimum achievable power.

Figure 4-18 and Figure 4-19 show the same effect for sparse-2 and sparse-4 trees, but with progressively smaller differences. Although the order is maintained throughout the whole spectrum, the differences in performance and power for sparse-4 trees are negligible across the fanout range. The reason for the reduced impact of lateral fanout on very sparse trees is because of the fact that sparseness reduces the effective fanout of the tree. As the trees are pruned to sparse-4, their critical paths become similar, with differences only in the branching factors at the input (lower) and output (higher), hence the similar delays. Although the number of gates is reduced, the increased output branching requires the remaining gates to be upsized by a roughly equal factor, resulting in similar energy consumption as well.

4.5.5 Tree sparseness

Figure 4-20, Figure 4-21, Figure 4-22 and Figure 4-23 show the impact of tree sparseness along the fanout dimension of the carry tree space (for fanouts of 1-1-1-1-1-1, 4-4-4-4-2-1, and 32-16-8-4-2-1) for domino and compound domino adders. Figure 4-24 shows the impact of sparseness along the radix dimension of the space for 1-1-1-1-1-1 radix 2 and radix 4 trees.

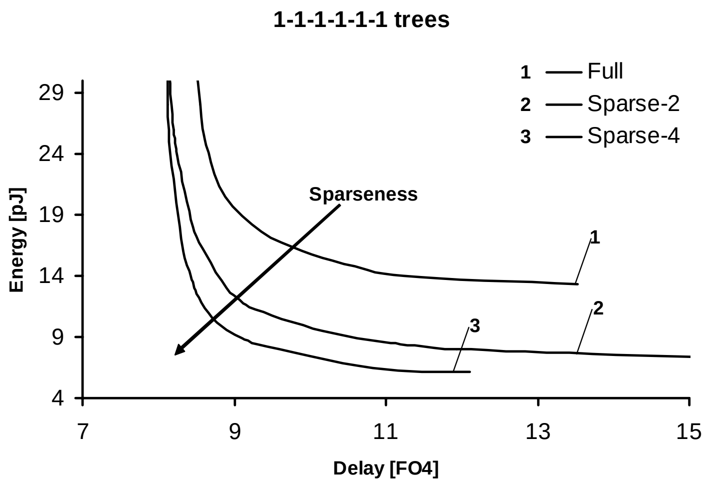

Figure 4-20. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 1-1-1-1-1-1 trees with different sparseness factors.

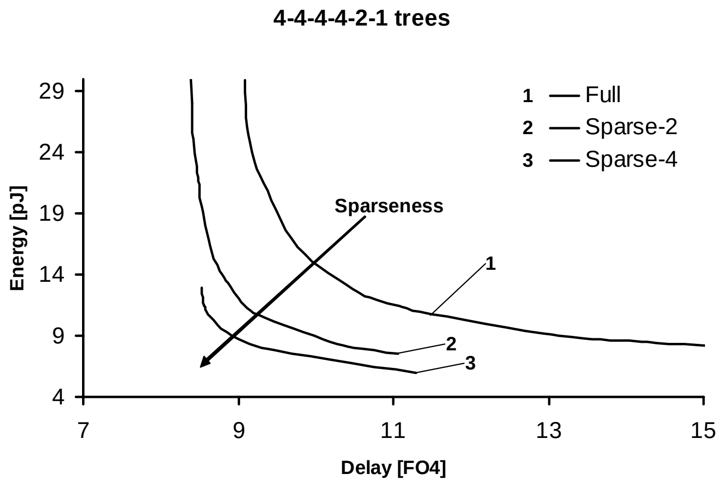

Figure 4-21. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 4-4-4-4-2-1 trees with different sparseness factors.

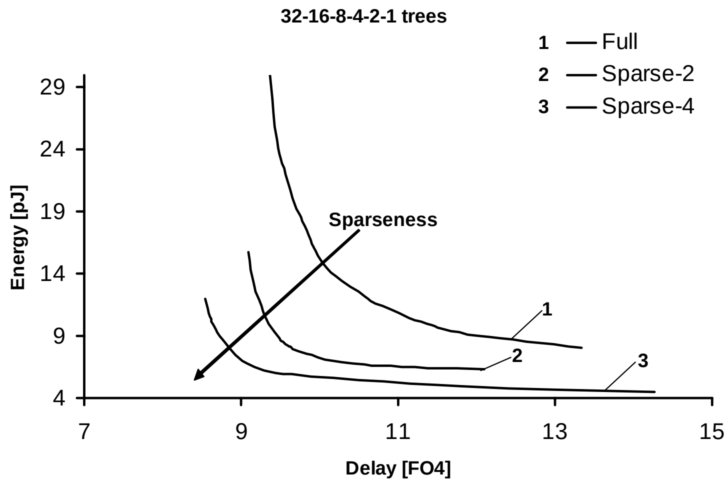

Figure 4-22. Energy - delay tradeoff curves for Ling domino adders implemented using radix 2 32-16-8-4-2-1 trees with different sparseness factors.

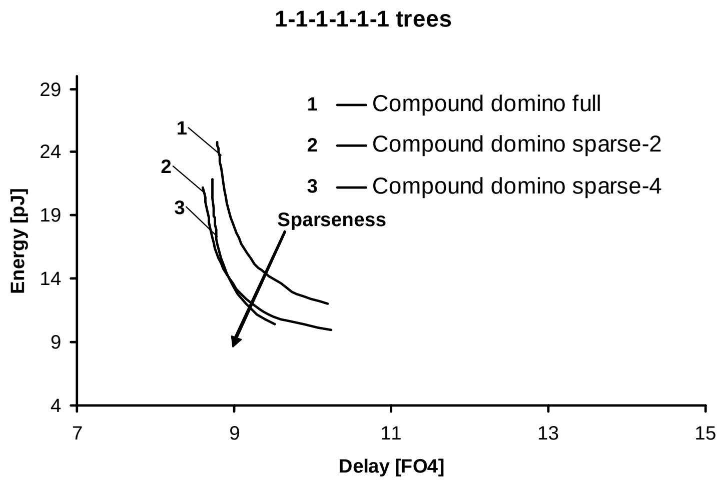

Figure 4-23. Energy - delay tradeoff curves for Ling compound domino adders implemented using radix 2 1-1-1-1-1-1 trees with different sparseness factors.

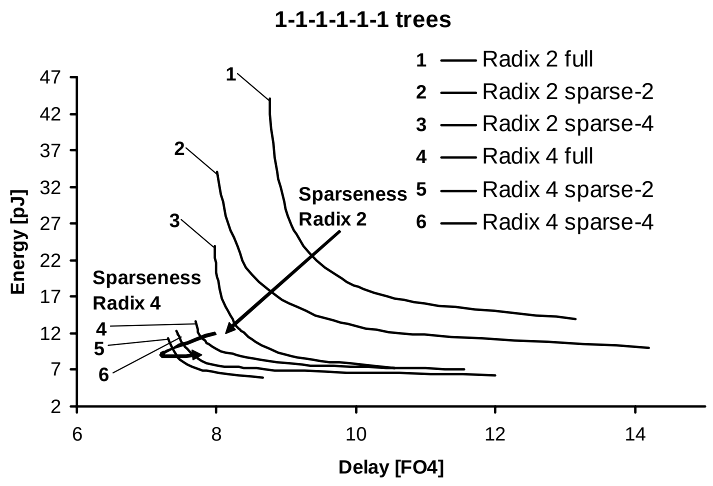

Figure 4-24. Energy - delay tradeoff curves for Ling domino adders implemented using 1-1-1-1-1-1 trees with different sparseness factors.

Adders with sparse trees differ from adders with full trees in three fundamental ways:

1. the gates and connections in the carry tree are pruned, reducing the size of the tree;
2. the sum-precompute block becomes more complex by unrolling the carry iteration at bit indexes where carries are not directly available;
3. the branching at the output of the carry tree increases.

These differences have several consequences in the power - performance space:

1. a smaller carry tree with less gates and less wires can be upsized for better performance in the same power budget;

2. reduced input loading for the carry block: a sparse tree has fewer gates in the first stage and therefore the load on the input is smaller; thus, for the same input capacitance, the input gates on the critical path can be made larger, resulting in a faster adder; also, as a result, larger gates will drive the internal wiring;

3. larger output load for the carry tree: one carry output must drive a number of gates equal to the sparseness factor of the tree, thus slowing down the circuit. Optimal delay is obtained through upsizing the critical path;

4. reduced internal branching due to gate pruning, speeding up the adder; the effect is more pronounced in high fanout trees and non-existent in Kogge-Stone derivatives (fanout of 1);

5. more complex sum-precompute blocks slow down the critical path through additional branching from the input and extra power consumption;

The overall result is a balance of all the above factors. The results of making a tree sparse depend on the configuration of the original tree. Increasing the sparseness factor benefits adders for which factors 1, 2 and 4 from the above list dominate factors 3 and 5.

Factor 1 is dominant for trees are very large, with many gates and many wires. Radix 2 trees with high redundancy (and reduced fanout) are excellent candidates for good power - savings and speed-ups due through sparseness due to this factor, as shown by Figure 4-24. On the other hand, radix 4 trees are smaller with fewer gates and fewer wires, reducing the effectiveness of factor 1. The same is true for compound domino adders - although radix 2, the number of gates in the tree is the same as in the radix 4 domino tree.

Factor 2 is generally small and benefits mostly higher radix trees, with long wires start to appear already after the first stage.

Factor 3 mostly affects higher radix trees, with less stages available to drive the extra load. For radix 2 trees, the impact of this factor is reduced because the large number of stages allows an adequate tapering for the increased load. A radix 2 sparse 2 tree is still amenable to extra pruning, although as shown in Figure 4-20 and Figure 4-24 the benefit of increasing the sparseness factor to 4 decreases significantly. On the other hand, the smaller number of stages in a radix-4 tree makes factor 3 more important when the output load increases. As shown in Figure 4-24, pruning a radix 4 tree to a sparseness factor of 2 results in an adder that is faster and consumes less. Pruning the tree further down to a sparseness factor of 4 trips the balance of the above factors, resulting in a slower and more wasteful adder.

Factor 4 is particularly dominant for high-fanout trees, where sparseness provides dramatic speed improvements and power reductions, as shown by Figure 4-21 and Figure 4-22.

Factor 5 has uniform influence throughout the tree spectrum and its impact is particularly pronounced for high sparseness factors. All the figures in this subsection reflect a decrease of the power/performance gain at the sparse-2 to sparse-4 transition when compared to the full to sparse-2 transition.

As with Ling's equations, pruning a carry tree in order to make it sparse effectively moves complexity from the carry tree in the sum-precompute block. As long as the carry tree remains the critical path, this move is advantageous. However, as the complexity of the carry tree decreases, the critical path may shift in the sum-precompute block, making the pruning ineffective.

It should be noted that the complexities in the carry tree and sum-precompute block scale differently with the sparseness factor: for a sparseness factor of *N*, the carry tree is generally *N* times smaller than the full tree. However, the sum-precompute block needs to repeatedly unroll the carry iteration at each bitslice: if G_i and G_{i+N} are available, the carry iteration needs to be unrolled once at bit index $i+1$, twice at $i+2$, and up to $N-1$ times at $i+N-1$. In *N* bitslices of a sparse-*N* tree, the carry iteration needs to be unrolled $1+2+...+N-1=N(N-1)/2$ times. Therefore, a $O(N)$ decrease in the complexity of the carry tree leads to a $O(N^2)$ increase in the complexity of the sum-precompute block.

4.5.6 Sizing strategy

All the curves presented so far in this chapter use the grouped sizing strategy. While convenient for design purposes, gate grouping can have a negative impact on the performance and power consumption of the adder. Grouping gates is equivalent to introducing equality constraints pertaining to the gate sizes in the optimization problem. Consequently, the feasible set of the problem is reduced and the value of the optimum is worsened. An optimization with a flat sizing strategy is always bound to yield a better solution in the energy - delay space (although the design costs might become prohibitive).

Of the three dimensions of the minimum depth carry tree space, the sizing strategy has the most impact along the fanout axis. Figure 4-25 shows the delay reductions that can be achieved for point 1 in Figure 2-2 when using a flat sizing strategy as opposed to a grouped sizing strategy.

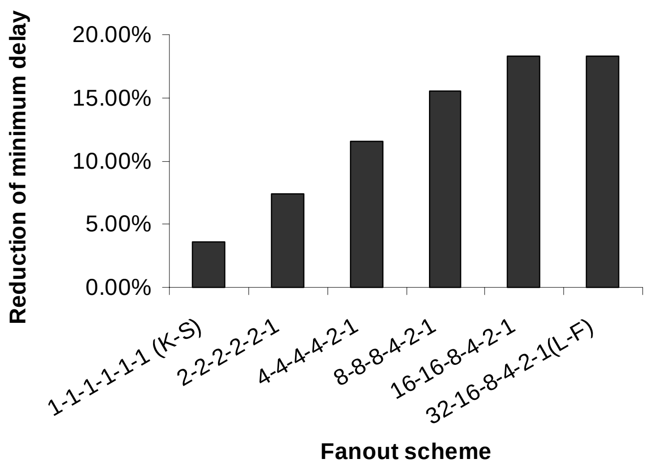

Figure 4-25. Minimum delay reduction for full trees when using a flat sizing strategy.

Structures with regular fanout (such as the Kogge-Stone tree) obtain very little benefit from a flat sizing strategy (3.6% delay improvement). On the other hand, structures with irregular fanout (such as the Ladner-Fischer tree) can be significantly sped up by ungrouping the gates. In such a situation, high fanout gates can be upsized without increasing the power consumption and input loading of lower fanout gates, resulting in a 18.5% speed increase.

Along the sparseness axis of the minimum-depth carry tree space, the speed increases due to ungrouping the gates follow the same percentages from Figure 4-25 for the actual fanout in the tree (not the same as the fanout label, as explained in Section 4.4.5).

4.5.7 Summary of adder design rules

The analysis of the impact of main design choices on adder behavior in the energy - delay space can be summarized in a set of 7 design rules, that can guide the designer when choosing the architecture of a 64-bit adder:

1. Use Ling's equations unless the lowest possible power is required;
2. For delay requirements longer than 12.5 FO4 use static CMOS. For less than 12.5 FO4 use domino;
3. Use the highest feasible radix for the carry tree;
4. Use the lowest lateral fanout for highest speed or highest lateral fanout for lowest power;
5. Sparseness reduces the impact of lateral fanout;
6. Sparseness is most beneficial for adders with large carry trees, high lateral fanout and relatively small sum-precompute blocks;
7. Flat sizing is most beneficial for adders with irregular structure and high fanout. Gate grouping has low impact for regular adders.

4.6 Fastest adder across different technologies

All the adders optimized so far in this chapter use a reference 90nm general purpose bulk CMOS process and the environment specified in the beginning of this section. While the design rules formulated in the previous subsections are general, the conclusions on which adder architecture is fastest in a given environment are dependent on the parameters of the particular process used for the analysis and the particular environment of the adder.

The circuit optimization framework can be used to investigate the influence of certain technology parameters on the behavior of digital circuits in the power - performance space. In the case of 64-bit adders, two technology parameters can significantly influence the design choices:

- C_{wire}/C_{gate} ratio ("wire capacitance ratio"), where C_{wire} is the lumped capacitance of 1μm of the interconnect used to route the carries across bitslices and C_{gate} is the capacitance of 1μm of minimum length transistor gate;
- C_{drain}/C_{gate} ratio ("drain capacitance ratio"), where C_{drain} is the drain capacitance of a single finger 1μm transistor.

The environment of the adder is usually reflected in the height of the bitslice. This in turn determines the wire capacitances as well as the output capacitance of the adder. A taller bitslice will increase the wire loads on the internal nodes of the adder but will decrease the output load because the layout will be narrower. The effect of the bitslice height can be transposed in the same coordinates as the technology (C_{wire}/C_{gate}, C_{drain}/C_{gate}) by simple scaling operations:

- a taller bitslice height is equivalent to a technology with a proportionally higher wire capacitance ratio;
- a smaller output load is equivalent to a technology with a smaller drain capacitance ratio that will yield the same delay when resized for the new load.

Figure 4-26 shows a partition of the (C_{wire}/C_{gate}, C_{drain}/C_{gate}) space, highlighting the architecture of the fastest adder in each region. In this figure, the delays corresponding to "point 1" in Figure 2-2 are compared across processes with different wire and drain capacitance ratios and for different bitslice heights using the above equivalency.

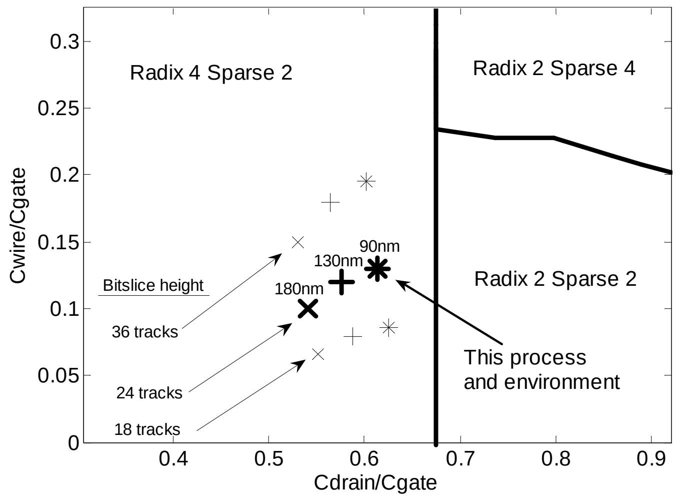

Figure 4-26. Fastest adder across different processes and environments.

All adders in Figure 4-26 use Ling's equations and are implemented in domino logic. For the 90nm process used in this analysis, the fastest architecture is radix 4 sparse 2, as concluded in Figure 4-24 in Section 4.5.5. This adder has been built in 90nm CMOS. Section 4.8 presents the design details and Section 4.9 the measurement results.

Figure 4-26 highlights the historical trend on the last three bulk CMOS processes from the same foundry. While the optimal architecture is the same in all three cases, the scaling trend points toward a different optimal architecture in the near future.

An increased drain capacitance ratio decreases the driving capability of a gate, degrades the slopes in the circuit and reduces the tallest acceptable transistor stack. Consequently, the highest feasible radix of the carry tree is also reduced. Even if a taller stack may be acceptable, the increased self-loading of the gates penalizes architectures with higher radix due to their longer wires and higher branching. The impact of this parameter on 64-bit adder performance is predicted by the design rule from Section 4.5.3: as shown in Figure 4-26, for processes with high drain capacitance ratios (exceeding 0.67), radix 2 architectures become faster than radix 4.

Interconnect parameters can also influence the architecture of the optimal 64-bit adder. If wire capacitance is significant, factor 1 in Section 4.5.5 becomes dominant and trips the balance towards sparse architectures. Taller datapaths (exemplified by the 36 track example in Figure 4-26) will tend to favor designs with high sparseness and many stages for appropriate tapering (radix 2). Figure 4-26 shows that radix 2 sparse 4 adders offer the best performance in very aggressive processes with high wire and drain capacitance ratios. At that point, adders with good energy efficiency can be obtained by using designs with high lateral fanout, as shown in Section 4.5.4) sparse trees with high fanout achieve speeds very close to the low fanout trees but with lower power consumption.

4.7 Runtime analysis and gate grouping

The runtime of the framework depends primarily on the size of the circuit, the sizing strategy and the type of models used in the optimization.

A typical 64-bit domino adder with about 1300 *grouped* gates is optimized with tabulated models on a 2.4GHz Opteron-based workstation with 2 GB of RAM running 64-bit Linux in 10 to 30 seconds if the constraints are rather lax. When the constraints are particularly tight and the optimizer struggles to keep the problem feasible, the runtime increases to about 90 seconds. A full power - performance tradeoff curve with 100 points can be obtained in about 45 minutes on such a machine. For grossly infeasible problems the optimizer provides a certificate of infeasibility in a matter of seconds. The optimization times can be reduced by 2-4x by using analytical models.

Changing the sizing strategy to *flat* increases the optimization time by 5-8x for analytical models and by 10-20x for tabulated models.

4.8 Fastest adder: test chip implementation

A test chip implementing the adder with the fastest architecture has been fabricated in a general purpose 90nm bulk CMOS process using standard V_{TH} transistors. The chip contains 8 64-bit adder cores and the corresponding testing circuitry (Figure 4-27) and is 1.6 x 1.7 mm in size.

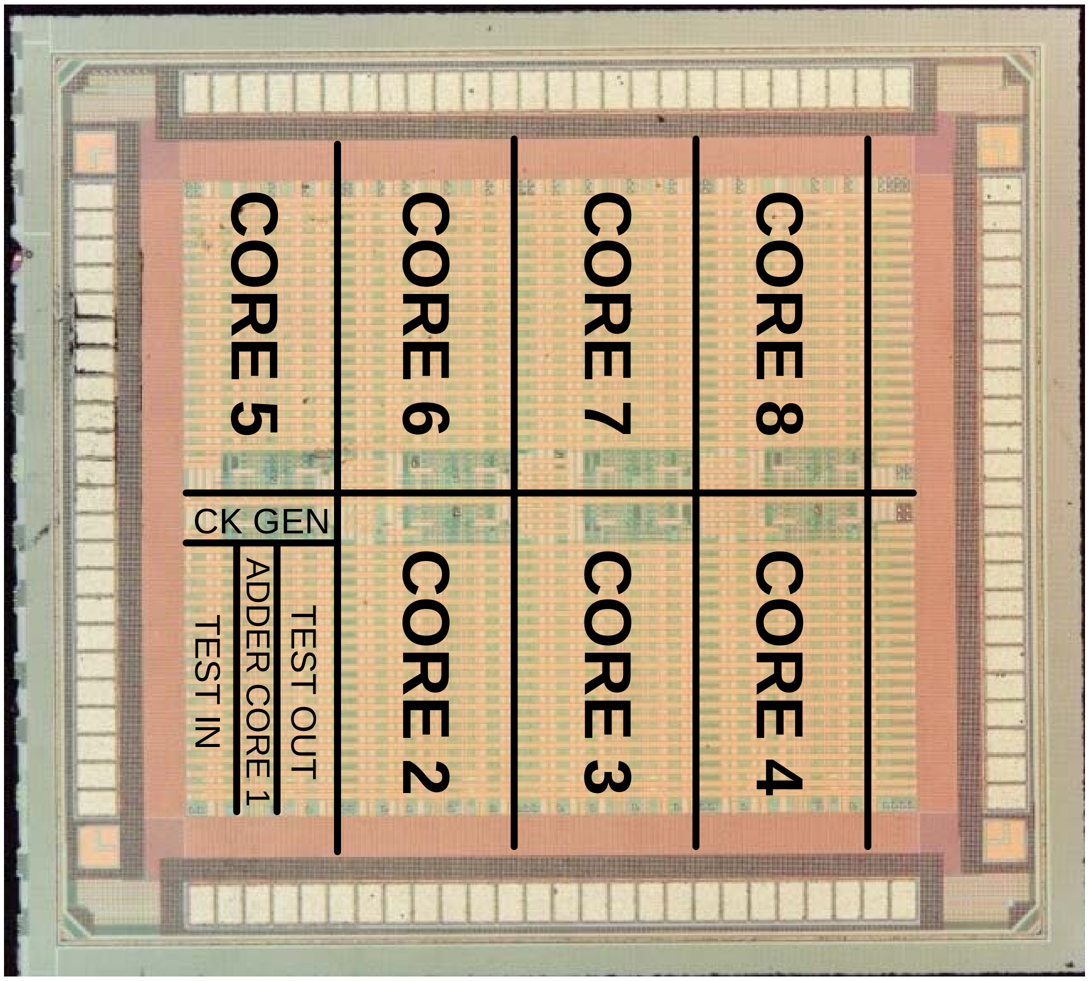

Figure 4-27. 90nm test chip micrograph.

The size of an actual adder core is 417 x 75 μm^2. The technology offers 7 metal layers and one poly layer, with a nominal supply voltage of 1V for the core. The chip is fully custom designed (the only standard cells used are the pin pads) and uses only standard threshold (SVT) devices.

As concluded in the, Section 4.5 for this process the fastest 64-bit adder architecture uses a radix 4 sparse 2 carry tree implementing Ling's equations in domino logic and the sum-precompute block in static CMOS. A sizing strategy with gate grouping has been used for this adder.

The complete diagram of the carry tree as implemented on the testchip is shown in Figure 4-28.

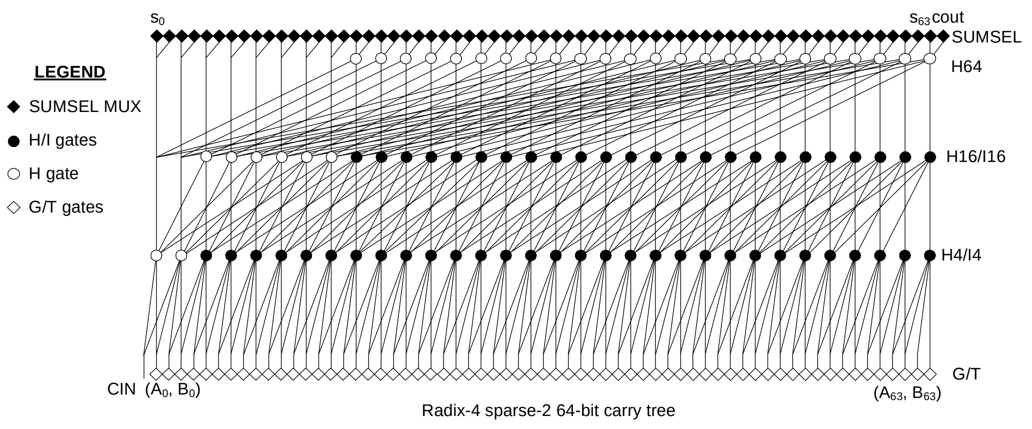

Figure 4-28. 64 bit radix sparse 2 carry tree, as implemented on 90nm testchip.

The *g* and *t* signals are computed by the G and T gates at each bitslice using (4-2); a 3-stage radix 4 sparse 2 carry tree computes even order carries. Each carry signal drives two sum select multiplexers, therefore selecting two sums.

Figure 4-29 shows the block diagram of the adder and on-chip testing circuitry, highlighting the logic families used in the core, and Figure 4-30 shows the corresponding timing diagram.

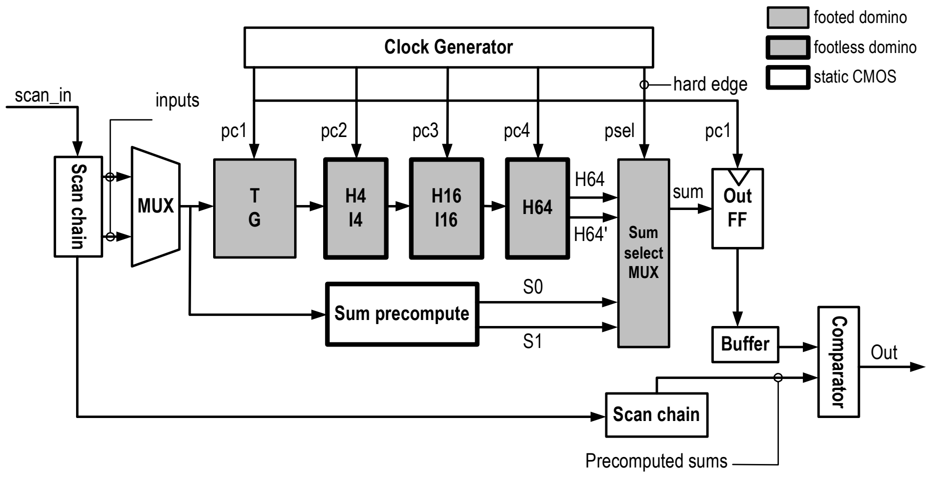

Figure 4-29. Adder and test circuitry block diagram.

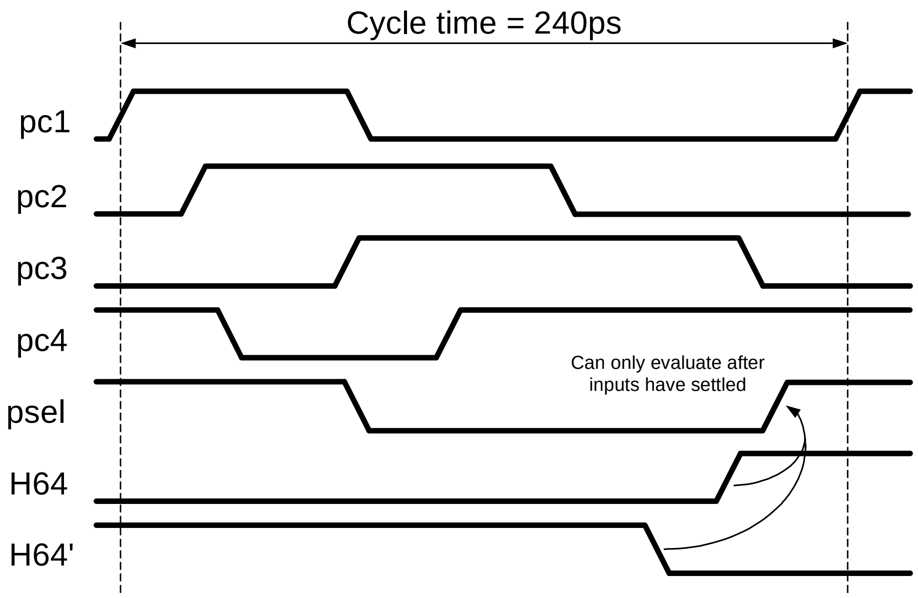

Figure 4-30. Adder timing diagram.

Delayed-precharged domino logic is used in the carry tree in order to hide the precharge phase from the overall cycle time. Most stages in the critical path use footless domino logic with stack node precharging when needed. Since the monotonicity of the global inputs *a*[63:0] and *b*[63:0] cannot be guaranteed, the first stage is implemented using footed domino logic. Figure 4-31 shows circuit details of such footed and footless domino gates.

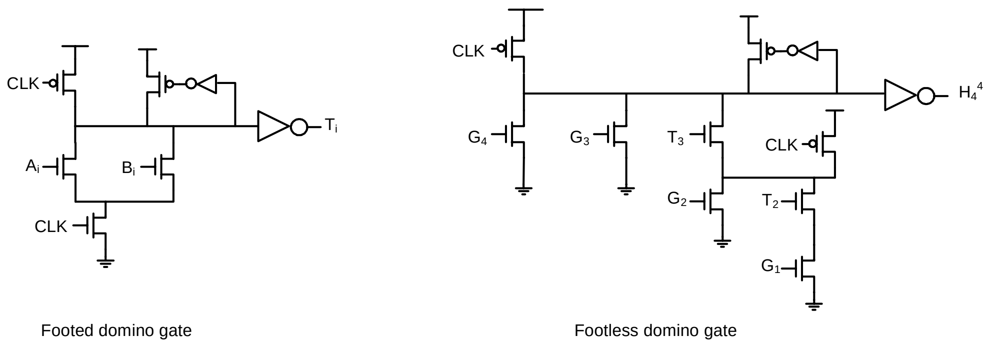

Figure 4-31. Example of footed and footless domino gates.

The inputs of the sum-select mux, S0[63:0], S1[63:0], are outputs of a static block and non-monotonic thus *psel* must be a hard clock edge (Figure 4-30). Critical timing edge arrivals can be fine tuned at runtime through the chip's scan chain in order to ensure correct functionality and best performance.

Using footless domino logic where possible increases the speed of the adder and reduces stack heights and transistors counts. These benefits do not come for free, and Figure 4-30 illustrates the cost of using footless domino. As opposed to a footed gate, a footless gate must be in evaluation when its earliest inputs arrive - otherwise crowbar current can occur (see Figure 4-31 for transistor level schematic). Any dynamic gate (footless or footed) must be in evaluation when its latest input arrives, in order to ensure correct operation of the circuit. When moving further away from inputs (or registered signals), the spread between the fastest and slowest paths in the circuit increases, therefore increasing the amount of time a footless domino gate must be in evaluation. After a few stages, this requirement becomes very stringent and leaves very little time for the precharge phase. Consequently, the precharge phase becomes critical and transistors must be upsized. At *pc4* the precharge phase is just as critical as the evaluation phase! Footless domino gates require bigger precharge transistors that slow down the evaluation path through their drain capacitance and increase the power consumption on the clock lines and in the clock distribution network. The tight constraints on the precharge transistors also reduce the design margins for timing signals, thus requiring runtime edge adjustment through the test scanchain.

The floorplan of the adder is shown in Figure 4-32.The bitslice height is selected to be 24 metal tracks, enough to accommodate the multiple loopback buses of modern multi-issue microprocessor architectures.[Fetzer02].

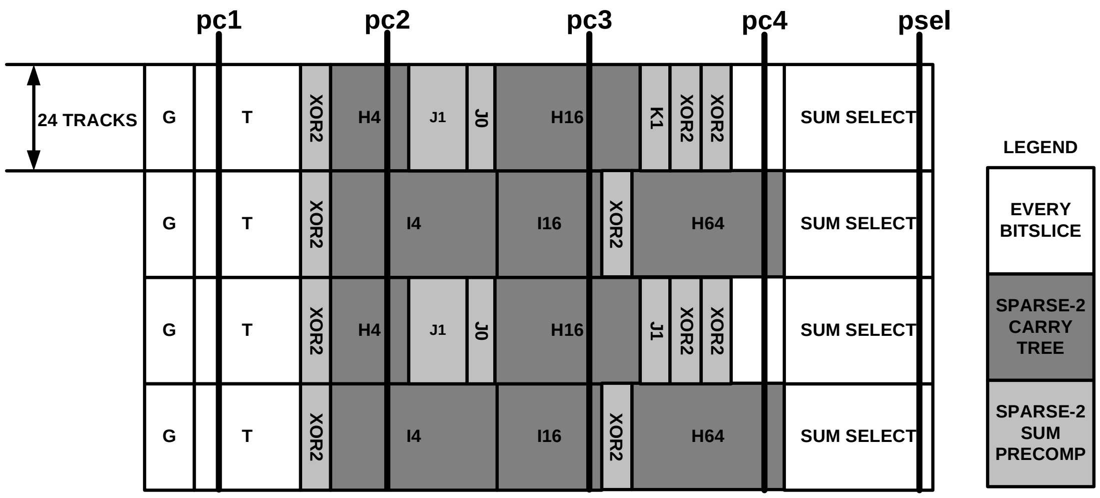

Figure 4-32. Sparse adder floorplan.

Due to the sparsity of the tree, the bitsliced floorplan has a period of 2 bitslices (Figure 4-32 shows 4 bistlices). Some blocks are repeated every bitslice (such as the G/T gates and the sum select multiplexer) while other are repeated every other bitslice (the actual H and I carry gates, the sum precompute gates). The floorplan is assembled such that the sum precompute gates occupy the space freed by the pruned carry gates in the sparse tree. The result is a very compact layout with very little unused space. Moreover - and very important from a functional perspective - the blocks are aligned such that the clock lines are always straight. The lack of jogs and branches on the clock lines helps meeting the very tight timing constraints imposed by the footless domino style.

4.9 Fastest adder: measured results

Figure 4-33 shows the average worst case delay (the delay for the worst case input combination, averaged across all the measured chips). At the nominal supply voltage of 1V the average delay of the adder is 240ps or 7.5 FO4, in good agreement with the 7.3 FO4 estimate from the circuit optimization framework.

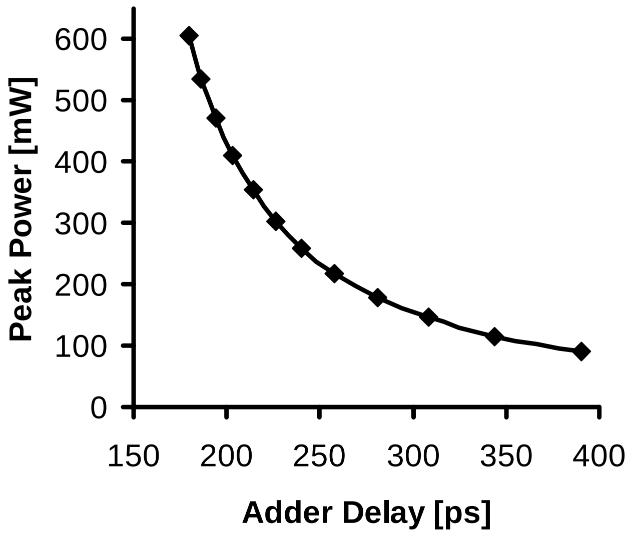

Figure 4-33. Measured worst case delay.

The fastest chip from the batch had a delay of 226 ps at 1V. Increasing the supply voltage to 1.3V reduces the delay of the adder to an average of 180ps. It should be noted that the design has been optimized at 1V; a re-optimization at 1.3V yields a slightly faster design, as shown by the example in Section 3.11.

The adder consumes 260mW at 1V supply voltage in the worst case, as shown in Figure 4-34. The worst case power is obtained (and measured) for a different input combination than the worst case delay. In Figure 4-34 the power includes the adder core and the clock generation and buffers and excludes the test circuitry. Increasing the power supply voltage to 1.3V and reducing the clock period to 180ps results in a worst case power dissipation of 606mW. As discussed in Section 3.11, if the design were to be re-optimized at 1.3V with a 180ps delay constraint, the power dissipation would be much less than 606mW.

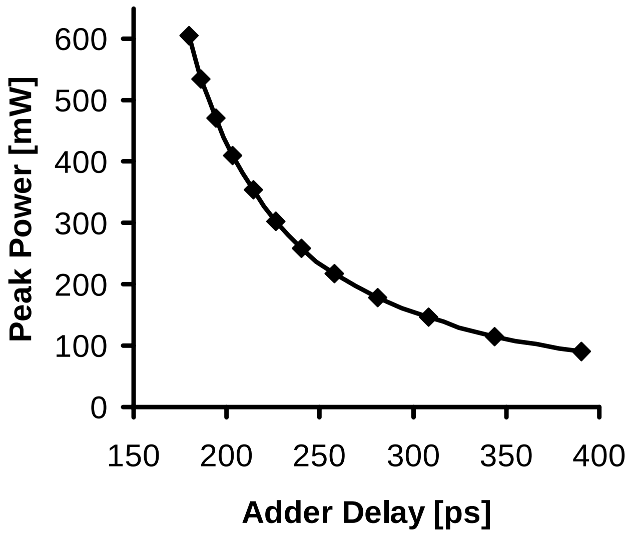

Figure 4-34. Measured worst case power.

Figure 4-35 shows the power distribution inside a core when running the worst case power input combination. For this high activity circuit using standard threshold (SVT) devices, leakage is small at 1%.

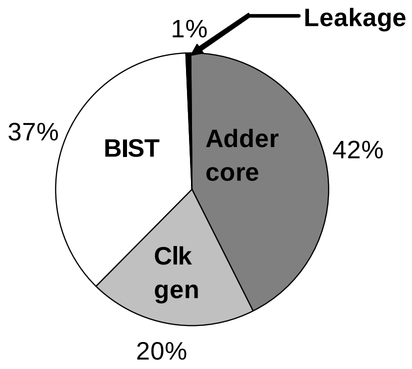

Figure 4-35. Power distribution in a core for worst case power input combination.

The clock generator and buffers consume almost half of what the actual adder core consumes *in the worst case*. The clock generator and buffers have the highest power density possible on a chip (inverters switching every cycle) and Figure 4-35 shows that their power consumption is very significant, thus requiring a very careful design of the power distribution network.

The measured power numbers cannot be readily compared with the estimates of the optimizer. First, the optimizer does not account for the power dissipated in the clock generator and buffers (but does include the power on the actual clock lines). Second, the optimizer can only compute *average* energy per operation (and hence *average* power), while the measurements show *peak* power. In the optimization, node activities are computed through logic simulation and they include an inherent averaging effect. In an experimental setup, it is impossible to measure the average power across 2^{129} input combinations and instead the peak power is more relevant.

The reason for doing the optimization and the comparison on *average* power is that such an optimization shows the power contributions of all blocks and paths and allows us to draw conclusions on where the power is spent in the adder. In an optimization and comparison on peak power only (i.e. for only one input combination), certain paths in the circuit would have never been sensitized and may have never switched thus may had been incorrectly sized (either very slow or very big). Working back from the measured peak power by accounting for node activities, the average energy per operation of the adder core is 10.33pJ, very close to the 10.4 pJ prediction of the optimizer at 1V supply voltage.

5 Optimizing Sequential Circuits

This chapter provides and in-depth look at the optimization of sequential circuits in the power - performance space. It builds on the concepts presented in Chapter 3 for combinational circuits by including the position of storage elements in the optimization.

Just like their combinational counterparts, sequential circuits can be optimized for minimum cycle time or minimum energy. Optimization problems similar to (3-23) and (3-24) can be formulated by simply replacing the “delay” of the combinational circuit with the “cycle time” of the sequential circuit and including storage element positions (the “cutset”) in the list of optimization variables. Although both problem versions make sense, it is more common to constrain the cycle time of a sequential circuit due to throughput requirements and minimize its energy consumption. Therefore, this chapter will focus on solving the following optimization problem:

$$\min_{cutset,\, W_i} \quad E \text{ such that } \begin{cases} T_{cycle} \le T_{cycle,max} \\ C_{in} \le C_{in,max} \\ W_i \ge 1 \\ t_{slope,j} \le t_{slope,max} \end{cases} \qquad \textbf{(5-1)}$$

The repositioning of storage elements while preserving the logic structure is called *retiming* and has been first presented in [Leiserson91] for circuits with edge-triggered flip-flops. Retiming can have several goals, such as minimizing the clock period, or minimizing the energy consumption subject to a maximum clock period. The first solutions to these problems have been presented in [Leiserson91].

The subsequent sections in this chapter present a way to include *retiming* in the power - performance optimization framework under certain assumptions. More precisely, the described methodology focuses on performing a *joint sizing and retiming optimization* on a sequential circuit.

Sizing and retiming cannot be performed simultaneously on a circuit: retiming by itself, as introduced in [Leiserson91], assumes that gate delays do not change; on the other hand, resizing gates implicitly changes their delay. Due to this fundamental conflict, sizing and retiming must be done *one at a time, iteratively,* as shown in Figure 5-1.

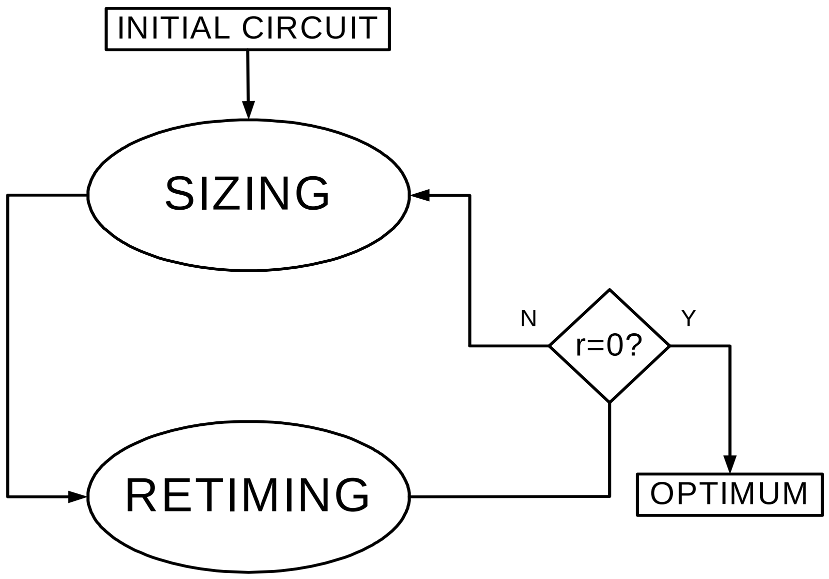

Figure 5-1. Iterative sizing and retiming of a sequential circuit

The circuit is iteratively resized and retimed for the same goal (e.g. minimum energy with maximum cycle time constraint) until the process converges. The convergence criteria is straightforward: iterations stop when no flip-flops are moved by the retiming process (i.e the total retiming *r* is zero). At this point, attempting to resize the circuit will not yield any improvement because retiming did not change anything from the previous sizing step.

In order to achieve optimality, the two steps must be coupled - i.e. at least one of the two steps must be modified to take into account the results of the other step (beyond just the current netlist). For instance, if the classical retiming algorithms [Leiserson91], [Shenoy97] were to be used, information about the previous sizing step would be lost. An optimally sized circuit will have many otherwise faster paths slowed down to the delay dictated by the cycle time constraint in order to save as much energy as possible. Without any other information, such paths will be regarded as critical by the retimer, thus fixing the corresponding flip-flops in their current positions due to the cycle time constraints and ending the iterations. In fact, such paths are not really critical, and the extra delay introduced in a cycle by a retimed flip-flop can be easily compensated by upsizing the logic in that cycle. For a correct optimization, the retimer should know and make use of a measure of the criticality of the paths ending at each flip-flop. Such criticality measures can be obtained from the preceding sizing step. Thus, the retiming steps have to be *sizing-aware retimings* and use modified algorithms that take such path criticality measures from the sizing optimizations.

As shown later in the chapter, in order to solve the optimization problem (5-1), the iteration from Figure 5-1 needs to be operated in two modes:

- minimum period mode;

- period-constrained minimum energy mode.

Each mode requires a sizing algorithm and a retiming algorithm. The methodology presented in this chapter uses the sizing step unchanged from Chapter 3 and couples it with new sizing-aware retiming algorithms. Therefore, two sizing-aware retiming algorithms are needed:

- a sizing-aware minimum period retiming algorithm;

- a sizing-aware period-constrained minimum energy retiming.

The resulting mixed rigorous / heuristic approach uses optimality-guaranteed algorithms for certain subproblems and near-optimality heuristics for the rest.

Section 5.1 introduces the definitions, notations and basic concepts of retiming from [Leiserson91] and [Shenoy97]. The two basic algorithms from the article are presented in Section 5.2 (minimum period retiming) and Section 5.3 (period-constrained minimum energy retiming). Section 5.4 describes how retiming can be coupled with sizing for the purpose of joint sizing and retiming power - performance optimization. Section 5.5 presents the sizing-aware version of the minimum period retiming and Section 5.6 the sizing-aware version of the period-constrained minimum energy retiming. An example on using the joint sizing - retiming power - performance optimization on a Floating Point Unit (FPU) is presented in Section 5.7. The example compares the results obtained by [Synopsys04] and the described methodology.

5.1 Definitions

This section introduces the definitions, notations and basic concepts of retiming as presented in [Leiserson91] and [Shenoy97].

A *sequential circuit* is an interconnection of logic gates and storage elements which communicates with its environment through primary inputs and primary outputs. A sequential circuit can be represented by a *directed graph $G(V,E)$*, where each vertex v corresponds to a gate v. Each directed edge e_{uv} represents a flow of signal from the output of gate u at its source to the input of gate v at its sink. Each edge has a weight $w(e_{uv})$ which indicates the number of registers that the signal at the output of gate u must propagate through before it is available at the input of gate v. If there is an edge from vertex u to vertex v, u is called a fan-in of v and v is called a fan-out of u. The sets of fan-outs (fan-ins) of u is denoted by $FO(u)$ ($FI(u)$). Each vertex v has a constant delay associated with the each of its inputs, $d_u(v)$, $u \in FI(v)$.

Two special vertices called the *global source* and the *global sink* are introduced in the graph to capture the interaction of the circuit with its environment. Edges directed from the global source represent the primary inputs and edges directed to the global sink represent primary outputs. Both special vertices have delays of 0. An edge from the global sink to the global source with the appropriate weight can be included to model an *external* loopback bus from the primary outputs to the primary inputs.

A *retiming* is a labeling of the vertices $r\colon V \rightarrow Z$ where Z is the set of integers. the weight of and edge e_{uv}, after retiming is denoted by $w_r(e_{uv})$ and is given by:

$$w_r(e_{uv}) = r(v) + w(e_{uv}) - r(u) \qquad \textbf{(5-2)}$$

A positive (negative) retiming label *r(v)* for a vertex *v* represents the number of registers moved from its outputs (inputs) towards it inputs (outputs). A retiming of zero implies no movement of registers. Figure 5-2 illustrates the retiming notations on a simple logic gate. The retiming of a circuit is an assignment of retimings to all the combinational gates in the circuit.

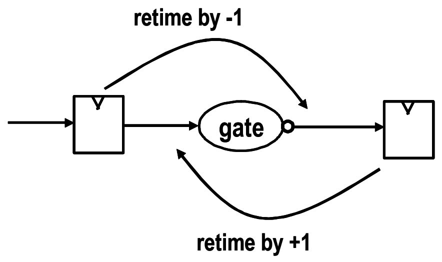

Figure 5-2. Retiming a gate

A path *p* is defined as a sequence of alternating vertices and edges such that each successive vertex is a fanout of the previous vertex and the intermediate edge is directed from the former to the latter. A path can start and end at vertices only. The existence of a path from vertex *u* to vertex *v* is represented as $u \to v$. The weight of the path *w(p)* is the sum of the edge weights for the path.

The delay of a path *d(p)* is the sum of the delays of vertices along the path. A zero-weight path is a path with *w(p)=0*. The *clock period* of the circuit is determined by the following equation:

$$c = \max\{d(p) | w(p) = 0\} \qquad \textbf{(5-3)}$$

that is the delay of the slowest zero-weight path.

Retiming makes the following assumptions:

1. the delay $d_u(v)$ of vertex v is non-negative for all $v \in V$;
2. the edge weight $w(e)$ of edge e is non-negative for all $e \in E$;
3. every directed cycle in G contains at least one register;
4. all registers are edge-triggered D flip-flops, clocked by the same signal with identical skew to all registers;
5. flip-flops are assumed to be ideal, with zero clock-to-Q delay, setup time and hold time;
6. the delay of a gate does not change when flip-flops are rearranged.

Among the assumptions summarized above, the first three are easily handled. Most synchronous registers can be modeled using D flip-flops, therefore the 4th assumptions only prevents the retiming of circuits with asynchronous set/reset registers. Since retiming involves a repositioning of registers, precise skew considerations are difficult to handle this early in the design process. A nominal tolerance of clock signals can be easily introduced to model clock skew. A nominal clock-to-Q delay and a nominal setup time can be incorporated in the algorithms by providing a margin around the clock period (as is done in the subsequent sections). Hold time violations can be easily corrected after retiming.

The most serious restriction is the last one, assuming that gate delays do not depend on flip-flop positions. Since registers are repositioned, loading at the gate outputs are difficult to predict in advance. The only way in which the delays of gate can be guaranteed to remain the same after retiming is to make sure that capacitive loadings at all nodes remain the same regardless of the positions of the flip-flops in the circuit. This can be accomplished if all D type flip-flops have $C_L/C_{in} = 1$. This choice presents two main advantages:

- inserting or removing a flip-flop on a wire does not change its loading, thus the delay of the gate driving that wire stays the same;
- all flip-flops in the circuit will have the same setup and hold times. The size of the flip-flop is determined by the value of the capacitive load on the wire it is inserted on (C_L); thus, all flip-flops operate in the same electrical environment and will have identical timing parameters.

The choice of having all D flip-flops with $C_L/C_{in} = 1$ is effectively an equality constraint in the sizing optimization. If flip-flops were allowed to have different gains (C_L/C_{in} ratios) a better optimization result could be obtained. All algorithms and results in the subsequent sections of this chapter assume that $C_L/C_{in} = 1$ for all D flip-flops.

5.2 Minimum period retiming

The objective of this retiming is to obtain a circuit with the minimum clock period without any consideration to the energy penalty due to the increase in the number of flip-flops. The minimum period retiming algorithm is based on the FEAS relaxation algorithm [Leiserson91] - “feasible clock period test”.

The FEAS algorithm determines if a retiming exists for a specified target clock period c. If such a retiming exists, FEAS computes it as well. If no “legal” retiming exists for the target clock period c, FEAS provides a certificate of infeasibility.

Let *Δ(v)* denote the largest delay seen along any combinational path that terminates at the output of *v*. It denotes the latest arrival time at *v*:

$$\Delta(v) = d(v) + max\{\Delta(u) | u \in FI(v), w(e_{uv}) = 0\} \quad \textbf{(5-4)}$$

It can be shown that the clock period can be given by the expression:

$$c = max\{\Delta(v) | v \in V\} \quad \textbf{(5-5)}$$

Algorithm FEAS: Given a synchronous circuit *G(V,E,d,w)* and a desired clock period *c*:

```
1. For each $v \in V$ set $r(v)=0$
2. Repeat $|V|-1$ times {
      2.1 Compute retimed edge weights using Eq. (5-2)
      2.2 Compute $\Delta(v)$ for all $v \in V$ using Eq. (5-4)
      2.3 For all $v \in V$ such that $\Delta(v)>c$, set $r(v)=r(v)+1$
      }
3. Compute retimed edge weights using Eq. (5-2)
4. Compute retimed clock period $cr$ using Eq. (5-5)
5. If $cr > c$ then there is no feasible retiming for target clock
period $c$
      else the current values of $r$ yield a legal retiming
```

The algorithm requires has a O(|*V*||*E*|) complexity. FEAS can be used as a decision algorithm in a bisection to find the minimum feasible clock period.

In sequential circuits with a constraint on throughput (i.e. with a clear target on the clock period) it is not usually necessary to compute the minimum clock period but instead it suffices to test if the desired clock period target is still feasible.

5.3 Period-constrained minimum energy retiming

In practice there are several solutions to the minimum period retiming problem with a large variation in the number of flip-flops amongst them. This is expected since the formulation does not impose any restriction on the number and size of flip-flops. Since the gate sizes and the capacitances in the circuit do not change during retiming, the energy consumption of the circuit changes only when flip-flops are inserted, deleted or repositioned.

The aim of minimum energy retiming is to minimize the energy of the flip-flops for a target clock period. Under the assumption that all flip-flops have the same energy consumption, the minimum energy retiming problem reduces to seeking a solution with the minimum number of flip-flops in the circuit.

The number of flip-flops in a circuit after a retiming *r* is given by:

$$R = \sum_{e \in E} w_r(e) = \sum_{e \in E} w(e) + \sum_{v \in V} r(v) \cdot (|FI(v)| - |FO(v)|) \qquad \textbf{(5-6)}$$

The first term in the summation is a constant representing the number of flip-flops in the original circuit, so it can be dropped from the optimization objective. The second term is a linear sum of the retiming labels.

Under the $C_L/C_{in} = 1$ assumption for all flip-flops, different flip-flops have different energy costs depending on the actual value of C_L. This is modeled by assigning each edge *e* a cost $\beta(e)$ proportional to the cost of adding a flip-flop along *e*.

The objective function to minimize is given by:

$$\sum_{v \in V} r(v)\left(\sum_{e \in FI(v)} \beta(e) - \sum_{e \in FO(v)} \beta(e)\right) = \sum_{v \in V} \alpha_v r(v) \qquad \textbf{(5-7)}$$

where α_v is a constant coefficient summarizing the cost (and benefit) of moving 1 flip-flop from all the outputs of vertex *v* to all its inputs.

The constraints on the retiming vector *r* translate into 2 sets of inequalities:

1. non-negativity of edge weights after retiming:

$$r(v) - r(u) \geq -w(e_{uv}) \quad \text{for any} \quad e_{uv} \in E \qquad \textbf{(5-8)}$$

2. correct clocking at clock period c requires that the delay of zero-weight paths after retiming be less than *c*. In fact, the correct way to formulate this constraint is to require that all paths with a delay more than *c* contain at least one register:

$$r(v) - r(u) \geq -w(u \to v) + 1 \quad \text{for any path } u \to v \quad \text{such that } d(u \to v) > c \qquad \textbf{(5-9)}$$

With these constraints, the period-constrained minimum energy retiming can be formulated as a *linear program* (LP) where *r(v)* are the optimization variables:

$$\begin{aligned} &min \sum_{v \in V} \alpha_v r(v) \quad \text{such that} \\ &r(v) - r(u) \geq -w(e_{uv}) \quad \text{for any} \quad e_{uv} \in E \\ &r(v) - r(u) \geq -w(u \to v) + 1 \quad \text{for any path } u \to v \quad \text{such that } d(u \to v) > c \end{aligned} \qquad \textbf{(5-10)}$$

Upon closer examination, (5-10) is an *integer linear program* (ILP). Since the edge weights *w* are integers, the first constraint specifies integer differences between all the elements of the retiming vector. The problem is defined up to an additive constant - it is easy to show that adding the same amount to all elements of the retiming vector does not change the final configuration of the circuit (Eq. (5-2)). Thus, even if the actual values of the elements of the retiming vector are not integers, their differences are always guaranteed to be integers.

[Leiserson91] and [Shenoy97] present a way to convert this LP to its dual, a network flow cost minimization problem that is easier to solve with the tools available at the time of writing of those articles. Modern LP solvers such as [Mosek06] generate and solve the dual problem automatically thus removing the need for the user to explicitly formulate it.

As a linear program, period-constrained minimum energy retiming is a convex optimization problem and its solution (if any) comes with an *optimality guarantee*. The solution indicates the positions of the flip-flops in the circuit that provide the achieve the minimum energy while still meeting the maximum cycle time constraint. Circuits optimized with this algorithm usually contain significantly less flip-flops than those optimized just for minimum period (Section 5.2) because the energy minimization is obtained by reducing the number and size of flip-flops within the problem constraints.

5.4 Coupling sizing and retiming

As described in the introduction of this chapter (Figure 5-1 and related explanations), sizing and retiming are coupled using two mechanisms:

1. successive iterations of sizing and retiming with the same objectives and constraints;
2. modifications of the retiming algorithms to make them sizing-aware.

The main property of this approach is that all circuits generated throughout the iterations, at the end of each resizing and retiming steps are *feasible* designs (i.e. they are functionally correct and meet all design constraints). The initial design for each step of the iteration is feasible and hence a potential solution. Consequently, the value of the objective function in the optimization is guaranteed not to increase at any step. If a change occurs in the circuit (a flip-flop is moved or a gate is resized), the next step can only decrease the value of the objective function (for instance resize the circuit such that the energy is less for the same clock period). This observation guarantees that the iterative process cannot continue forever, because the objective functions (clock period, energy) cannot be arbitrarily small. Also, the process cannot have an asymptotic behavior because retiming is a discrete transformation.

The approach is similar to interior point optimization methods [Boyd03] due to the key property that intermediate solutions are *feasible* at each intermediate step. One of the main problems with interior point methods is supplying an initial guess point that is feasible, in order to start the iterations. The iterative algorithm in Figure 5-1 requires only that the result of the *first* step be feasible. While this is a relaxation on the original interior point method limitation, it is not very useful because for most digital circuits the result of the first (sizing) step is not feasible. Therefore, it is desirable to always have a way to supply a feasible starting point to the sizing - retiming iteration.

[Boyd03] presents a rigorous way to find a feasible starting point for interior point methods by solving another (secondary) optimization problem with similar properties to the original one. The solution of the secondary optimization problem is the initial guess point for the original problem. The construction of the secondary optimization problem makes choosing its own starting point trivial. [GGPLAB06] follows the method from [Boyd03] closely and uses such a two-step approach for solving geometric programs without requiring an explicit initial guess point from the user.

A similar idea can be applied to the sizing - retiming iteration: supplying a feasible starting point by first iterating on a different problem. Figure 5-3 shows the two-step approach used to solve (5-1).

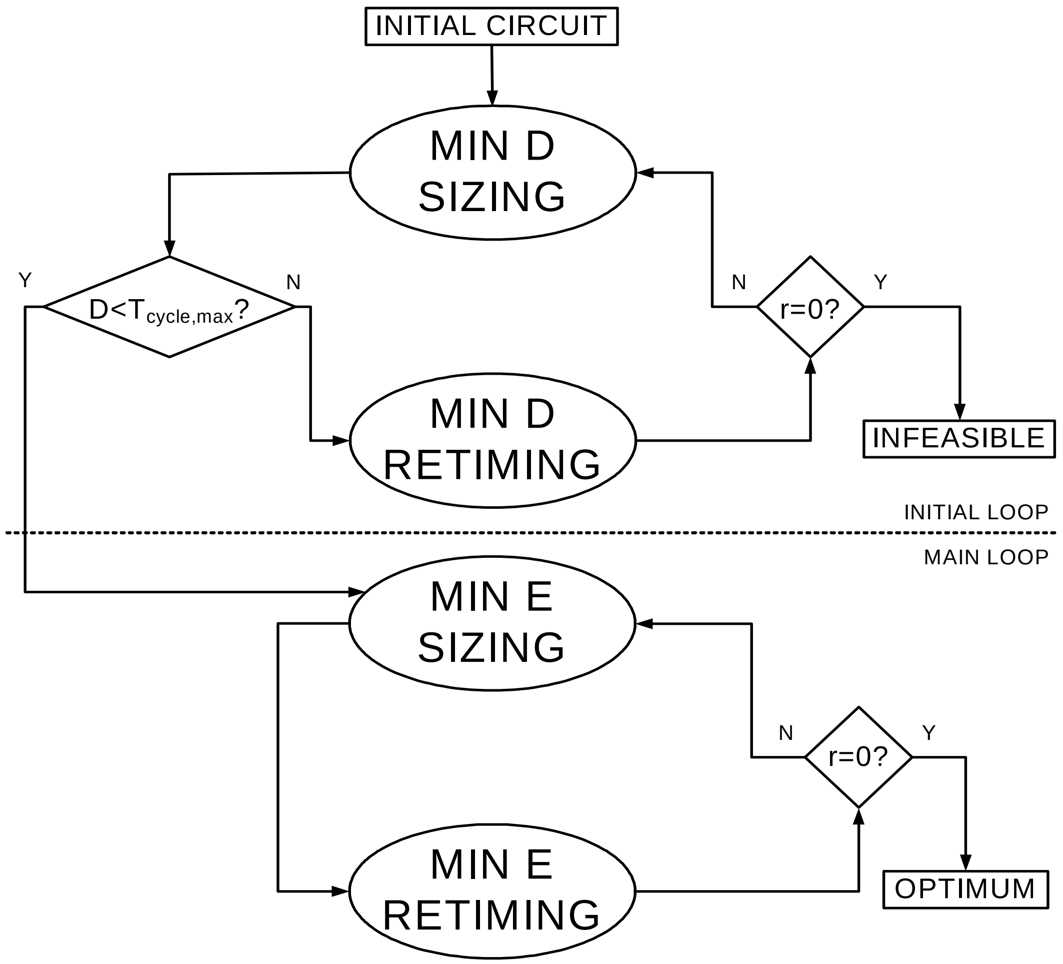

Figure 5-3. Coupling sizing and retiming for a two-step period-constrained energy minimization problem

The initial loop iterates in minimum period mode and supplies a feasible starting point for the main loop by performing an unconstrained cycle time minimization. In fact, it is not necessary to wait until the initial loop converges. Once the cycle time has gone below the throughput constraint, the current point is feasible for (5-1) and can be used a starting point in the main loop iterations in period-constrained minimum energy mode. If the initial loop converges and the cycle time is still too long, a certificate of infeasibility is issued for the constrained energy minimization problem (there is no circuit that meets the cycle time constraint). The initial loop can take any starting point and will produce a circuit that meets all constraints except cycle time after the first step, if such a circuit exists.

The sizing steps in Figure 5-3 are formulated and solved as described in Chapter 3 with additional margins on the delay constraints to account for flip-flop setup time and clock-to-Q delay, as explained in Section 5.1. As explained in the introduction of this chapter, the retiming steps in Figure 5-3 are actually *sizing-aware retimings* and use modified algorithms that take such path criticality measures from the sizing optimizations.

The next two sections describe these modified algorithms for sizing-aware minimum period and minimum energy retiming. Both algorithms make use of the *Lagrange multipliers* obtained from the corresponding sizing optimization.

5.5 Sizing-aware minimum period retiming

The sizing-aware minimum period retiming is coupled with the minimum period sizing optimization problem:

$$\min_{W_i} \quad T_{cycle} \text{ such that } \begin{cases} T_j \le T_{cycle} \quad \text{for all nodes with FFs} \\ C_{in} \le C_{in,max} \\ W_i \ge 1 \\ t_{slope,j} \le t_{slope,max} \\ T_i = 0 \quad \text{for primary inputs} \\ T_j \le D \quad \text{for all } j \\ T_j + D_i \le T_i \quad \text{for } j \in FI(i) \end{cases} \tag{5-11}$$

The period of the circuit is determined by the arrival times at the flip-flops, as reflected by the first set of constraints in (5-11). Thus, all paths in the circuit can be considered to end at the flip-flops and their degree of criticality is readily available from the sizing optimizer in the set of corresponding optimal Lagrange multipliers for those respective constraints.

The sizing aware retiming must identify the most critical paths - i.e. the ones with the highest optimal Lagrange multipliers for the period constraint at their terminal flip-flop - and favor a negative retiming for their terminal flip-flops. Using the retiming convention from Figure 5-2, a negative retiming ($r < 0$) means a flip-flop is moved backwards, thus removing one gate delay from its critical path and loosening its timing constraint.

The structure of the retiming algorithm remains similar to the one presented in Section 5.2 for minimum period retiming except for the routine used to compute the clock period. Instead of using the actual clock period c and the actual arrival times $\Delta(v)$, the FEAS algorithm is run with a set of different quantities, the *sensitivity-adjusted clock period* c_S and the *sensitivity-adjusted arrival times*, $\Delta_S(v)$.

For each vertex v in V, let λ_v be defined as follows:

- $\lambda_v = 0$ if the weight of all edges originating at v is zero;
- $\lambda_v = \sum_i \lambda_i^*$ for all edges i with non-zero weight originating at v.

The sensitivity-adjusted arrival times are computed using a modified version of (5-4) that takes the λ_v vector into account:

$$\Delta_s(v) = (d(v) + max\{\Delta(u)|u \in FI(v), w(e_{uv}) = 0\})(1 + \lambda_v) \qquad \textbf{(5-12)}$$

Because $\lambda_v = 0$ for nodes that have no flip-flops, the sensitivity-adjusted arrival times remain the same along the logic paths until they reach the flip-flops. At the very end of the path (at the flip-flop), a correction factor is applied to reflect its criticality. If the path is not critical ($\lambda_v = 0$), the sensitivity-adjusted arrival time remains unchanged. Very critical paths, with high λ_v will have a significantly increased sensitivity-adjusted delay, thus favoring negative retiming.

The sensitivity-adjusted clock period is defined in a similar way as the actual clock period (5-5), but using the sensitivity-adjusted arrival times instead:

$$c_s = max\{\Delta_s(v)|v \in V\} \qquad \textbf{(5-13)}$$

This sensitivity-adjusted clock period includes a measure of how critical is the "most critical" path in the circuit. By repeatedly applying the FEAS algorithm, the sensitivity-adjusted clock period c_s is minimized in a similar way as the real clock period in the stand-alone min-period retiming. The difference is that in the end, not only the critical path delay is minimized, but its delay sensitivity is minimized as well.

Since the sensitivities obtained through the Lagrange multipliers are defined locally, the algorithm is best behaved if the retiming vector is constrained to +/- 1, meaning that flip-flops can jump only one gate at a step. If flip-flops were allowed to jump two or more gates at a step, the criticality measure provided by the optimal Lagrange multipliers would be rather inaccurate due to their local scope. The consequence of this choice is a slight increase in the number of iterations in the initial loop in Figure 5-3.

A dramatic speed-up in execution time can be obtained by improving the FEAS algorithm using the following technique introduced in [Shenoy94]. The technique works for both stand-alone and sizing-aware minimum period retiming. It is most beneficial for the sizing-aware case due to repeated iterations in the initial loop in Figure 5-3.

It is empirically observed that if c (or c_s) is feasible, then the retiming labels converge rapidly before completing $|V|$ - 1 iterations in the FEAS algorithm (line 2). On the other hand, one cannot determine that a (sensitivity-adjusted) clock period is infeasible until all $|V|$ - 1 iterations have been exhausted and retiming labels have failed to converge. Thus, any hope of speeding up minimum period retiming must focus on detecting if a clock period is infeasible *before* completing the requisite $|V|$ - 1 iterations, if possible.

The detection principle is illustrated in Figure 5-4.

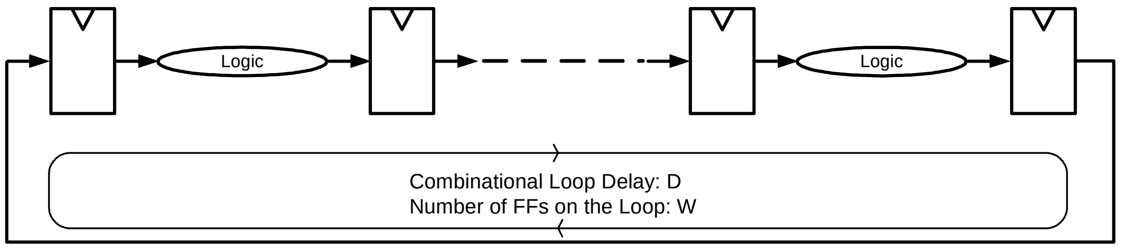

Figure 5-4. Critical cycle detection

A routine detects the loops in the graph and computes their corresponding combinational delay and weight (number of flip-flops along the loop). Because retiming cannot change the number of flip-flops in a loop of a circuit [Leiserson91], a (sensitivity-adjusted) clock period *c* can be feasible only if:

$$c \geq \frac{D}{W} \qquad \textbf{(5-14)}$$

A target clock period *c* means that a *W*-cycles computation must be completed in a time of no more than *cW* with ideal flip-flop positioning. If the combinational delay along a loop exceeds *cW,* that loop will prevent *any* feasible retiming at clock period *c*.

The implementation of this technique results in dramatic speed-up in execution time for both stand-alone [Shenoy94] and sizing-aware minimum period retiming.

5.6 Sizing-aware period-constrained minimum energy retiming

The sizing-aware period-constrained minimum energy retiming is coupled with the period-constrained minimum energy sizing optimization problem:

$$\min_{W_i} \quad E \text{ such that } \begin{cases} T_j \leq T_{cycle} \quad \text{for all nodes with FFs} \\ C_{in} \leq C_{in,max} \\ W_i \geq 1 \\ t_{slope,j} \leq t_{slope,max} \\ T_i = 0 \quad \text{for primary inputs} \\ T_j \leq D \quad \text{for all } j \\ T_j + D_i \leq T_i \quad \text{for } j \in FI(i) \end{cases} \qquad \textbf{(5-15)}$$

If the stand-alone minimum energy retiming algorithm from Section 5.3 is used, the iterations in the main loop from Figure 5-3 will stall. The main idea of the sizing-aware algorithm is to have the retimer *look ahead* at the next resizing step and try to estimate (approximate) what the objective function (energy) will be after resizing. The post-resizing estimated energy is used as the objective in the retiming LP instead of the actual energy. The optimal Lagrange multipliers from the previous sizing step are used for the estimation of the energy after retiming and the next resizing step.

It should be noted that because the objective function in the optimization problem (5-15) is energy, the meaning of the Lagrange multipliers for the flip-flop timing constraints is ratios of normalized energies to normalized delays $(\Delta E/E)/(\Delta T/T)$. In the previous section the meaning of Lagrange multipliers was a ratio of normalized delays.

Using the same λ_v definitions as in the previous section, Figure 5-5 depicts the situation when a flip-flop is to moved forward (sometimes referred to as "forward retiming") corresponding to *r* = -1.

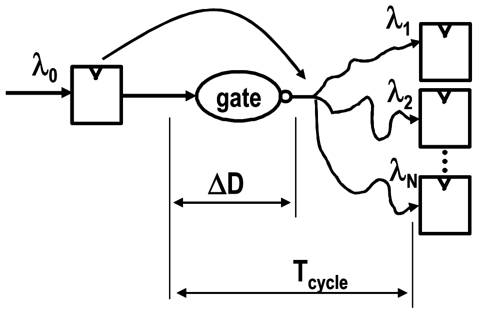

Figure 5-5. Predicting post-sizing energy for forward retiming

λ_0 represents the optimal Lagrange multiplier for the timing constraint at the flip-flop to be moved in the previous sizing step. $\lambda_1 \ldots \lambda_N$ represent the optimal Lagrange multipliers of the timing constraints for all the flip-flops at the end of paths originating at the output of the gate to be jumped. The irregular lines in the figure represent paths through combinational logic.

The energy of the circuit after retiming is predicted (exactly) by (5-7). In order to approximately predict the energy after the next resizing step, a correction factor is introduced for $\alpha(v)$ at the current node in (5-7):

$$\Delta E = E \cdot \frac{\Delta D}{T_{cycle}} \left(\lambda_0 - \sum_{i=1}^{N} \lambda_N \right) \qquad \textbf{(5-16)}$$

where ΔD is the delay of the gate jumped by the flip-flop, as shown in Figure 5-5.

Moving the flip-flop forward adds an extra logic gate in the previous cycle (the one driving the retimed flip-flop) with a delay ΔD. In order to keep the same cycle time, the previous cycle must be upsized and its energy consumption will increase. The first term in (5-16), $E \cdot (\Delta D / T_{cycle}) \lambda_0$, accounts for the energy increase in the previous cycle based on the corresponding sensitivity.

On the other hand, the current cycle looses one logic gate with delay ΔD and can therefore be downsized (and slowed down) until the same cycle time is achieved. The energy savings in the current cycle are captured by the second term in (5-16), $\Delta E = -E \cdot (\Delta D / T_{cycle}) \sum_{i=1}^{N} \lambda_N$.

Eq. (5-16) is a first order expansion of the actual post-resizing energy based on the local gradient. The approximation is most accurate for circuits with long pipeline stages where ΔD represents a small portion of the overall cycle time T_{cycle}. Since the energy estimations are based on the Lagrange multipliers which are local in scope, the approximations are accurate only for $r = 0$ or $r = -1$.

A very similar formula is used for backward retiming ($r = +1$). Instead of considering all paths *beginning* at the current flip-flop, the contributions of all the paths *ending* at it are included. Although the formula has the exact same expression as (5-16), the indices of the Lagrange multipliers refer to *different* flip-flops in the circuit for the same current node. Similarly, the formula is an accurate approximation only for $r = 0$ or $r = +1$.

Using different formulas for forward and backward retiming makes it impossible to formulate one single optimization problem for both forward and backward retiming: the objective function depends on the result. The optimizer has no way of knowing which formula to use in the objective function unless it already knows the solution (the r vector). Consequently, the sizing-aware retiming is split in two parts:

- forward retiming ($r \leq 0$);
- backward retiming ($r \geq 0$).

Of course, a resizing must performed between the two steps in order to obtain the optimal Lagrange multipliers if any flip-flop has been moved. Separating forward and backward retiming has the advantage of preventing the collapse of two-gate cycles, as shown in Figure 5-6.

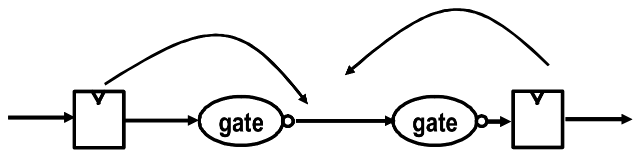

Figure 5-6. Two-gate cycle collapsed by retiming

The disappearing cycle causes a large error in the approximation (5-16) that cannot be recovered in the resizing step.

A very high Lagrange multiplier at a flip-flop signifies a very critical path and will push the flip-flop backward (thus allowing the cycle to be downsized). A small or zero Lagrange multiplier signifies a non-critical path and the flip-flop can move forward (thus increasing the delay) at little energy cost.

Because of the split between the forward and backward retiming, the main loop in Figure 5-3 must be restructured to accommodate the unidirectional retiming. Figure 5-7 shows the new main loop flow.

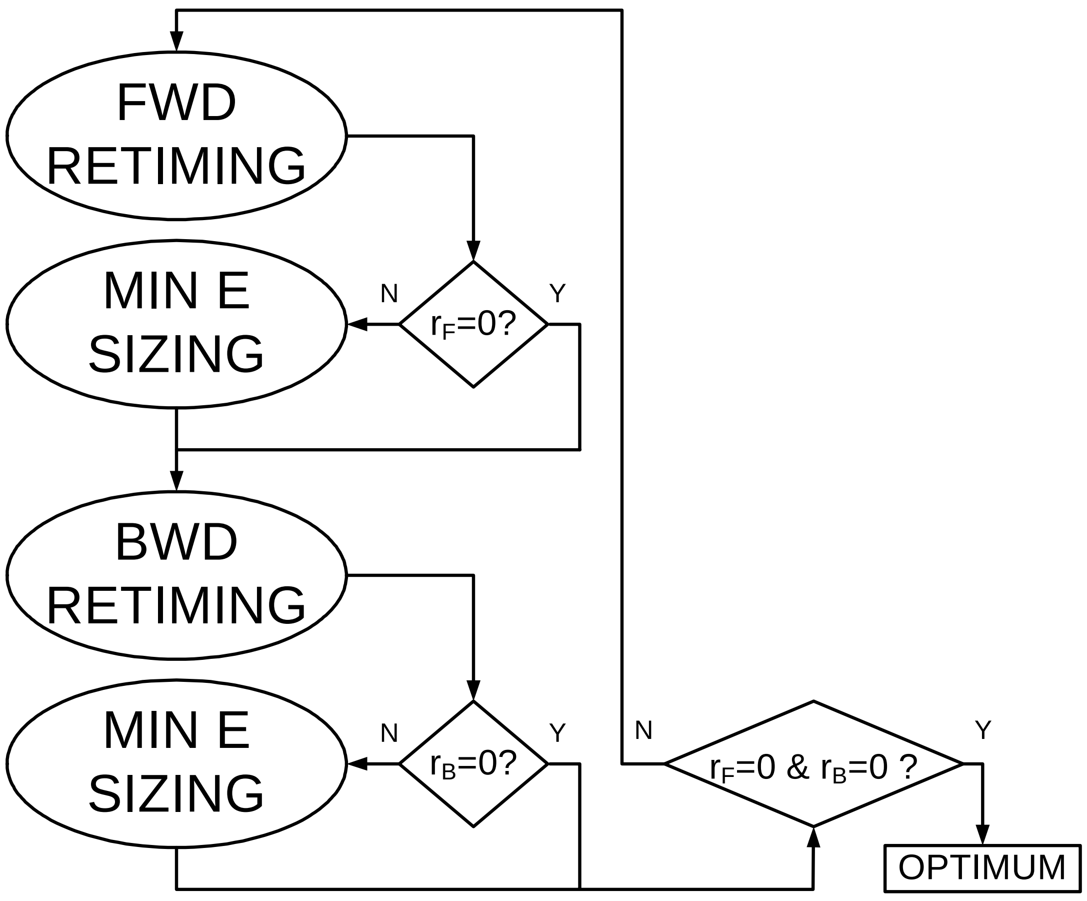

Figure 5-7. Main loop for sequential circuit optimization using unidirectional retiming

Forward and backward retiming are performed in an alternate fashion, each step followed by a resizing if any change has occurred. The iteration ends when neither of the retmings causes any change to the circuit.

The delay constraint in the sizing-aware unidirectional retimings is no longer equal to T_{cycle}, like in the stand-alone retiming. Using such a constraint will prevent some non-critical paths from being retimed. In order to minimize energy, the sizing step will downsize a non-critical path until either one of the following three conditions occurs:

1. all gates along the path are minimum size;

2. the slope constraints along the path become active;

3. the delay of the path reaches *c*.

In the last situation, the non-critical path will have a delay of *c* and will have a non-zero (albeit small) Lagrange multiplier. If the delay constraint remains unchanged, such a path cannot be retimed despite not being really critical. In order for any retiming to occur in such a situation, the delay constraint must be increased by one gate delay. Since different gates have different delays in the circuit, the delay constraint is increased by the delay of the slowest gate.

To summarize, sizing-aware period-constrained minimum energy retiming solves the same linear program (5-10) as its stand-alone counterpart with the following differences:

1. replace $\alpha(v)$ in (5-7) by $\alpha(v) + \Delta E$ with ΔE given by (5-16);

2. increase the delay constraint from c to $c + \max[d(v)]$;

3. separate forward and backward retiming as shown in Figure 5-7.

A dramatic speed-up in execution time can be obtained by using the following clock constraint pruning technique introduced in [Shenoy94]. The number of clock period related constraints in the LP (5-10) can be very large even for small circuits. However, many of them are redundant and can be eliminated.

In the original formulation (5-10), clock period constraints are required for all pairs of vertices (u, v) such that $d(u \to v) > c$. To see why a smaller set of clock period constraints is sufficient for the LP (5-10), note that if

$$r(v) - r(u) \geq -w(u \rightarrow v) + 1 \quad \textbf{(5-17)}$$

is true for a sub-path of a path, then it is true for the entire path. Hence a clock period constraint need only be added to vertex *v*, reachable from *w*, such that:

$$d(w \rightarrow v) > c \quad \text{and} \quad d(w \rightarrow u) \leq c \quad \text{for any} \quad u \in \{w \rightarrow FI(v)\} \quad \textbf{(5-18)}$$

Figure 5-8 illustrates the constraint pruning principle from Eq. (5-18).

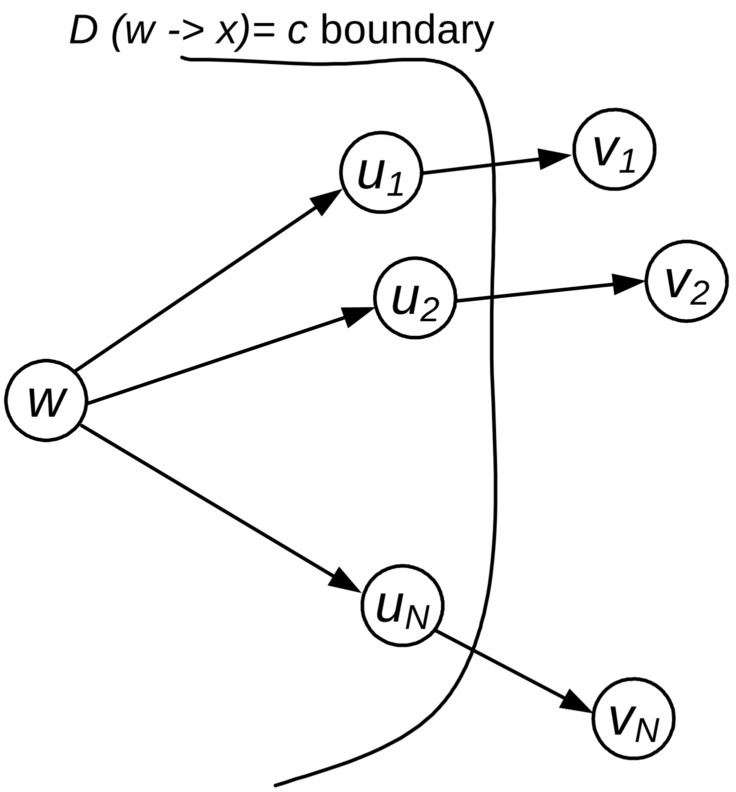

Figure 5-8. Clock period constraint pruning principle

The figure shows the boundary in the graph where the combinational delay from node *w* is *c*, the clock period constraint. Only the nodes immediately after that boundary (v_1, v_2, ... v_n) need a clock period constraint in the LP (5-10). A constraint for any node beyond v_1, v_2, ... v_n will be satisfied automatically because a similar constraint is satisfied for a node of the path between the w and that node.

5.7 Example: single-precision Floating Point Unit (FPU)

This example demonstrates the capabilities of the circuit optimization framework on a real-life sequential circuit: an IEEE-compliant single-precision Floating Point Unit (FPU). The goal of this example is to synthesize the FPU from a behavioral description and optimize it for minimum energy consumption at a cycle time of 2ns, with a latency of 4 cycles, using a general purpose 90nm CMOS technology. A commercial synthesis tool [Synopsys04] is used to generate a first design from a high level Verilog description. The design is then optimized separately by the same synthesis tool and by the circuit optimization framework, in order to compare the results.

The block diagram of the FPU is shown in Figure 5-9.

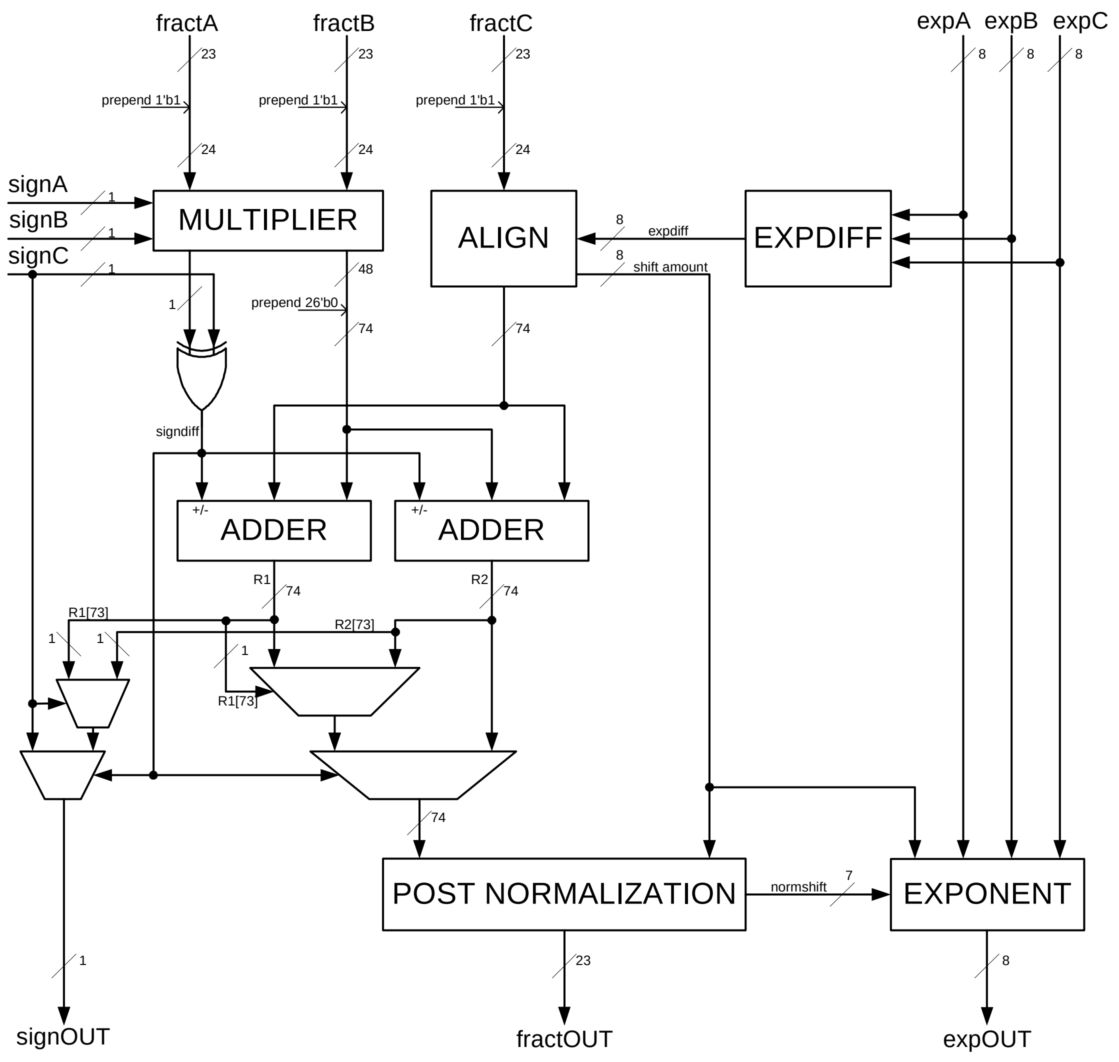

Figure 5-9. Block diagram of a 3-input fused multiply-add single-precision Floating Point Unit (FPU)

The FPU takes three 32-bit input operands, A, B and C and computes (+/-)(C+/-A*B). Each operand is separated in a fraction part (23 bits), an exponent part (8 bits) and a sign bit. The architecture of the FPU is fused multiply-add and is inspired by the OpenCores FPU implementation from [OpenCores06]. The mantissa path consists of a multiplier for computing A*B, the alignment shiftier for C, two wide adders and the post-normalization block. The exponent path has a block that computes the exponent differences required to align C and the final incrementers that adjust the exponent of the results based on the shifting amounts required in the post-normalization of the mantissa.

The design flow used for this example is shown in Figure 5-10. The flow has three starting points:

1. a behavioral Verilog description of the circuit to be tested. In this case, the behavioral Verilog description of the FPU follows closely the block diagram from Figure 5-9, with straightforward descriptions for each block and the same connections between blocks[1];
2. a library of standard cells;
3. a set of design constraints (cycle time, latency, slope constraints, load capacitances etc.).

From these three inputs, the fully automated flow produces three circuits with the same function. The power and performance metrics for these designs are compared in the end, in order to demonstrate the circuit optimization framework.

1. Behavioral Verilog description for the FPU provided by Seng Oon Toh

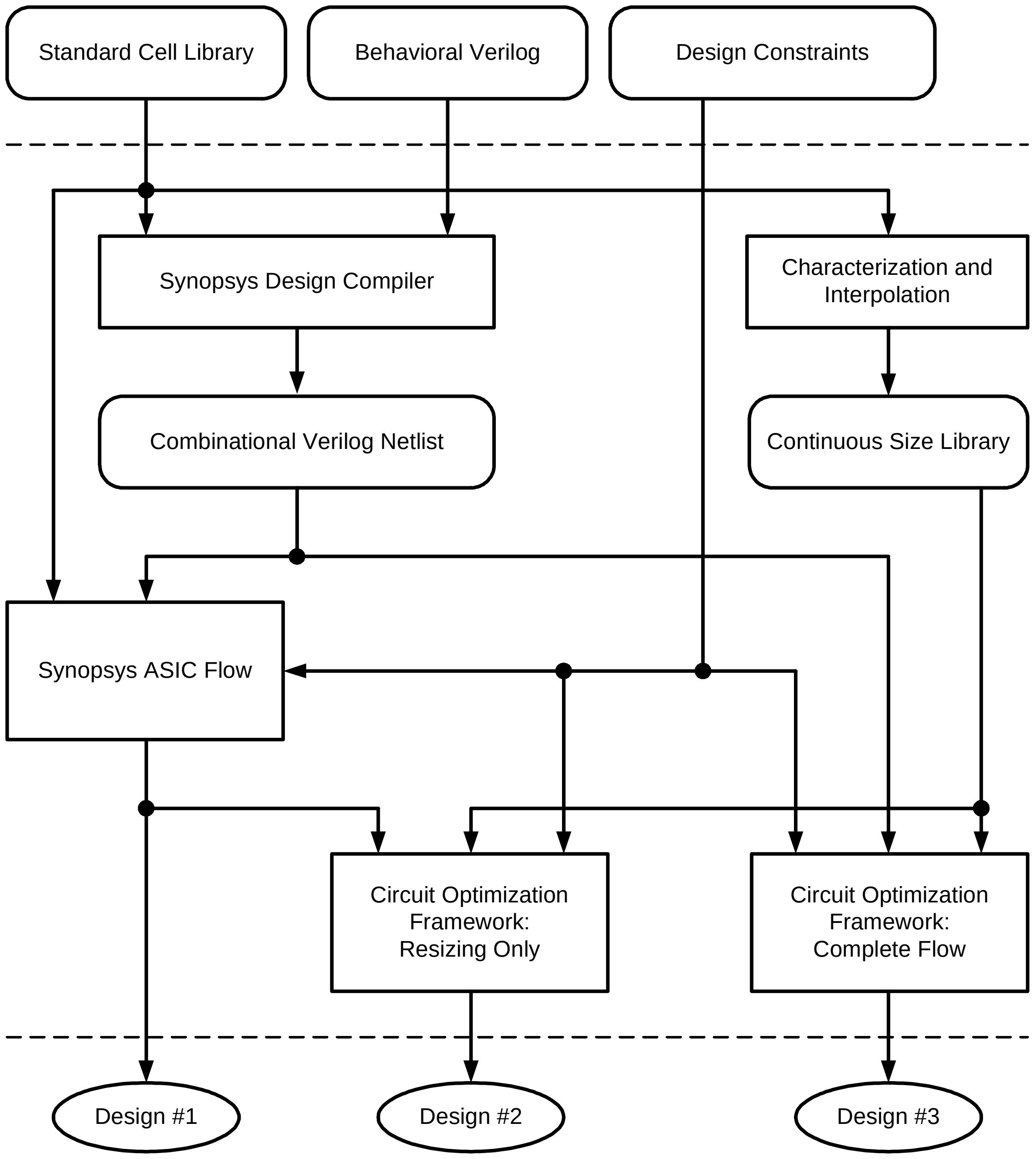
Standard Cell Library
Behavioral Verilog
Design Constraints
Synopsys Design Compiler
Characterization and Interpolation
Combinational Verilog Netlist
Continuous Size Library
Synopsys ASIC Flow
Circuit Optimization Framework: Resizing Only
Circuit Optimization Framework: Complete Flow
Design #1
Design #2
Design #3

Figure 5-10. Design flow used for the FPU

The circuit under test (FPU) is fist synthesized in a purely combinational fashion using Synopsys Design Compiler [Synopsys04]. The resulting combinational Verilog netlist is subsequently optimized on two separate paths:

- retiming and re-mapping using the standard-cell based Synopsys ASIC flow [Synopsys04] for a delay target of 2ns and 4 cycle latency (Design #1);
- continuous resizing and sizing-aware retiming by the circuit optimization framework for the same constraints (Design #3).

For comparison purposes, a third design is produced by applying a continuous resizing to Design #1 using the circuit optimization framework, without any retiming (Design #2). The purpose of this design is to compare only the retiming algorithms from the ASIC flow with those from the circuit optimization framework.

Since the only technology data provided to the flow is the discrete-sized standard cell library, a few extra processing steps are required in order to use the continuous sizing capabilities of the circuit optimization framework. The standard cell library needs to be translated into a continuous size library containing the gate models described in Chapter 3.

A logic gate usually has several entries in the standard cell library corresponding to the different available sizes - and each entry has a complete set of parameters. For continuous sizing, a logic gate must have only *one* entry in the continuous size library, characterized by only *one* set of parameters for *all* possible sizes. The translation is done in two steps:

1. characterization of each entry in the standard cell library; at the end of this step, each library entry has its own set of parameters;
2. removal of the discrete sizing dependency using interpolation; in this step the library entries with the same function are grouped together and one global set of parameters is computed for the group[1].

The flow from Figure 5-10 is fully automated using a set of custom written parsers and scripts. The parsers, written using the lex [Paxson95] and yacc [Donnelly95] utilities, allow the circuit optimization framework to read all Verilog netlists produced by the ASIC flow (by translating them into its internal SPICE-like format) and to use any standard cell library as a basis for building its own continuous size library.

Figure 5-11 shows the comparison of the three designs for the single precision FPU.

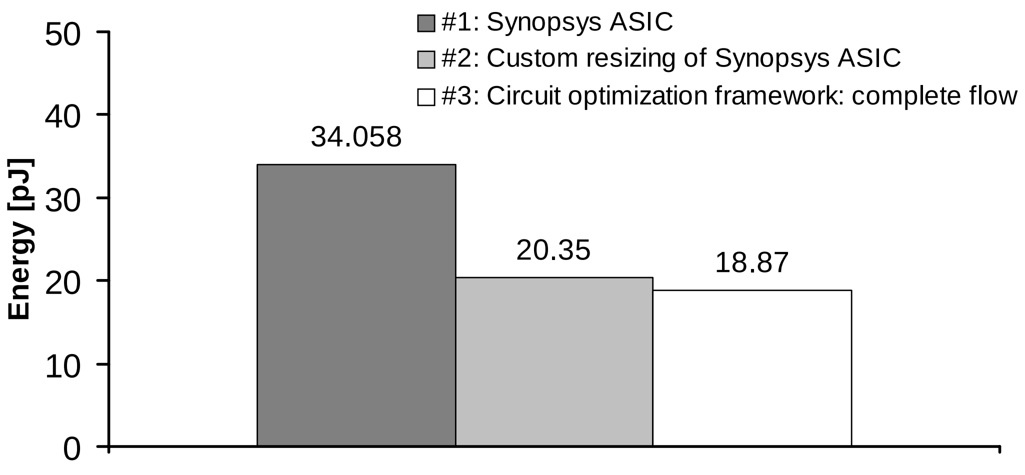

Figure 5-11. Minimum energy achieved by the FPU for 2ns cycle time and 4 cycle latency through various optimization methods

Design #1 (labeled “Synopsys ASIC”) represents the design as produced by Synopsys ASIC flow after retiming and re-mapping. For a 2ns cycle time, the average energy per operation is 34.058 pJ. This is a standard cell design, with discrete sizing levels as provided by the library.

1. interpolation scheme developed in cooperation with Seng Oon Toh

Design #2 (labeled “custom resizing of Synopsys ASIC”) represents a custom (continuous) resizing of the combinational logic of the Synopsys ASIC without any retiming. The positions of the registers in the circuit are kept unchanged, but all the logic gates are resized using the circuit optimization framework in order to minimize the energy subject to a 2ns maximum cycle time. The minimum size of the gates is constrained to be the same as the smallest gate of the same type in the standard cell library. The average energy per operation decreases to 20.35pJ.

Design #3 (labeled “circuit optimization framework: complete flow”) represents the best design obtained by the circuit optimization framework, after custom resizing and sizing - aware retiming. This design is obtained by coupling sizing and retiming as shown in Figure 5-3 and Figure 5-7 and using the algorithms presented in this chapter. The average energy per operation for this design is 18.87 pJ for the same 2ns cycle time and latency of 4 cycles.

Table 5-1 shows a breakdown of the energy savings in the 2ns 4-cycle FPU design.

Table 5-1. Breakdown of energy savings for the FPU

Source of energy savings	Amount of energy savings
Energy savings from to custom sizing	40.2%
Energy savings from sizing-aware retiming	7.2%
Total energy savings from ASIC design	44.5%

As can be seen from Table 5-1, the bulk of the energy savings come from resizing (about 40%). This is the gain obtained by using continuously-sized gates instead of discrete-sized standard cells. In Figure 5-11 these savings are represented by the energy difference between Design #1 and Design #2. Both designs have the same cutset (as produced by the ASIC flow) but use different sizing strategies.

The additional 7.2% of savings (from the custom sized design) are due to the sizing-aware retiming algorithms. In Figure 5-11 these savings are represented by the energy difference between Design #2 and Design #3. Both designs are continuously sized but use different retiming algorithms.

The total energy savings provided by the circuit optimization framework over the ASIC flow amount to 44.5% for this example. In Figure 5-11 these savings are represented by the energy difference between Design #1 and Design #3.

The computing resources and runtime required for optimizing this circuit are significant. The FPU has 8288 instances of combinational gates and the sizing optimizations are done flat. The final design has 1364 flip-flops in the retiming graph that can be merged into 715 bigger flip-flops (the graph representation from [Leiserson91] requires flip-flops with branching outputs to be split and later re-merged, if needed). The initial loop from Figure 5-3 yields a feasible design after the first iteration and the main loop from Figure 5-7 requires 5 iterations to converge to a final cutset. The total runtime for the optimization of the FPU is approximately 55 hours on a Sun V40Z machine with 4 2GHz Opteron processors and 16GB of memory. The retiming algorithms are single-threaded and cannot take advantage of multiple processors in the current implementation. [Mosek06] has a multi-threaded GP solver that provides a 2.5x speed-up for resizing on the 4-way machine when compared to a single-core machine. The memory requirements for the sizing-aware retiming are also substantial at approximately 6GB.

The percentages in Table 5-1 are representative for designs placed in the middle of the power - performance space (far away from the endpoints of the optimal energy - delay tradeoff curve). Experiments performed on parts of the FPU[1] show how the energy savings change along the power - performance spectrum:

- high performance circuits benefit most from the custom sizing and retiming capabilities of the circuit optimization framework. Circuits close to point 1 in Figure 2-2 are very sensitive to sizing (as shown in Section 3.11) and even a small change in a gate size can cause a significant speed increase or decrease. The coarse sizing granularity of the ASIC flow incurs large power and performance penalties for such circuits. On the other hand, continuous sizing and the retiming coupled with it are best suited for fine tuning such sensitive designs. The total amount of energy savings increases significantly as does the percentage attributable to continuous sizing.
- low power circuits benefit least from the custom sizing and retiming capabilities of the circuit optimization framework. Circuits close to point 2 in Figure 2-2 are generally minimum-sized and hence not very sensitive to sizing (as also shown in Section 3.11). Because of the low sizing sensitivity, sizing-aware retiming reverts to classical retiming and does not provide any additional energy savings. In the extreme case of point 2 with all the gates are minimum-sized, the circuit optimization framework does not provide any improvement over the ASIC flow.

1. Experiments performed by Seng Oon Toh and still in progress

6 Dealing with Process Variations: Robust Optimization

In aggressive nanometer-scale CMOS, process variations are critically affecting the design of digital integrated circuits. Scaling supply voltages, increasingly non-ideal device characteristics and increasingly hostile electrical environments are degrading design margins to the point where functionality becomes difficult to assure, and energy and delay become difficult to predict. Power - performance optimization is not immune to the perils of process parameter fluctuations. This chapter presents a theoretical framework to include process variations in the optimization process in order to design robust circuits with good manufacturing yield. While not fully developed and integrated with the rest of the circuit optimization framework, it provides insight on the effect of process variations on digital circuits and is a good starting point for future research.

It is not the purpose of this chapter to analyze the causes and sources of process variations in modern CMOS technologies. [Frank04] contains an excellent survey on these topics. Instead, the focus of this chapter is on how to deal with these variations in the optimization process and how to perform a robust optimization. Section 6.1 discusses the main approaches for dealing with parameter variations during design optimization. The selected method for digital power - performance optimization, stochastic geometric programming, is presented in detail in Section 6.2 and a simple but representative example follows in Section 6.3.

6.1 Types of robust optimization

There are three main types of robust optimization:

- optimization across process corners;
- optimization with uncertain parameters (in some mathematical literature it is called simply "robust optimization" [Ben-Tal98]);
- stochastic optimization.

Traditionally, robust design for electrical circuits meant analyzing *process corners*. The designer had to make sure that the circuit still operates correctly and meets the specs even in the worst situation. In optimization terms, this means adding constraints to the optimization problem corresponding to all process corners (such as TT, TF, FF, ... TS, SS):

$$\min_{x \in \Re^n} f(x) \text{ such that } \begin{cases} g_{i,TT}(x) \le 0 & i = 1\dots m \\ g_{i,TF}(x) \le 0 & i = 1\dots m \\ g_{i,FF}(x) \le 0 & i = 1\dots m \\ \dots & \\ g_{i,SS}(x) \le 0 & i = 1\dots m \end{cases} \qquad \textbf{(6-1)}$$

where $g_{i,XY}$ represents the set of design constraint at the XY process corner. If a certain value of the objective function must be achieved in order to ensure correct functionality (e.g. a minimum required throughput for a pipeline), appropriate constraints can be added across all corners. A more relaxed approach is *product binning* when the objective function is just a measure of the performance of the circuit and does not impair correct functionality. Microprocessors are the typical example for clock frequency binning.

Optimization across process corners has the advantage of being very simple. Moreover, the input data required in (6-1) is readily available since most foundries provide models for their transistors and interconnects at multiple process corners. Adding constraints of the same type does not change the mathematical properties of the optimization problem - so if the optimization problem in the typical case is convex, so is the one across all the corners.

The disadvantage of corner optimization is that it usually results in overly conservative and over-designed circuits. Figure 6-1 illustrates this conservatism for the simple case of two gaussian variables.

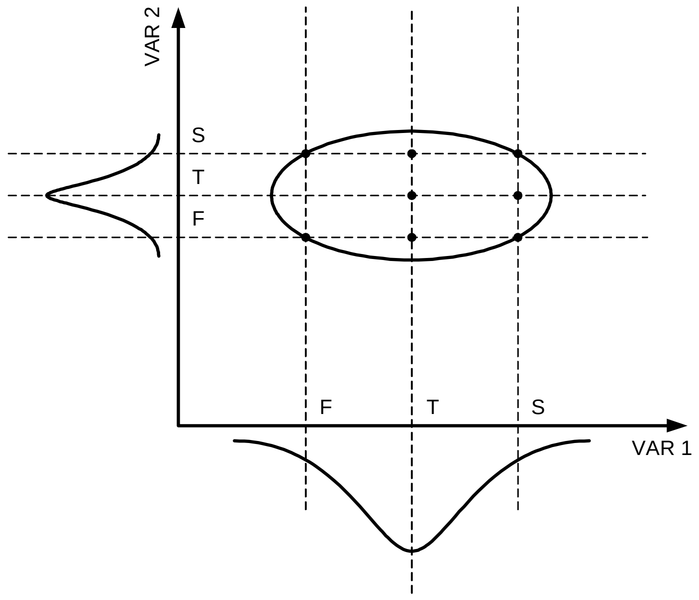

Figure 6-1. Conservatism of corner optimization

The typical (T) corner corresponds to the center of the distributions of the two variables. The fast (F) and slow (S) process corners correspond to yield-imposed boundaries on the distributions. A process corner - based design ensures that the circuit operates correctly in the corners of the rectangle defined by the F and S boundaries for each variable. However, the end result in this case is that the circuit operates correctly in an ellipsoid circumscribing the rectangle. It should be noted that there are many ellipsoids that can circumscribe the same rectangle. The actual ellipsoid is determined by the joint distribution of the two parameters and in particular by their correlation. If the two parameters are uncorrelated, the two diameters of the ellipsoid are determined only by the individual variances of the variables.

Since the ellipsoid covers more space than the rectangle, the circuit can actually sustain more variations than just those specified by the process corners - hence it is over-designed. The over-designing translates directly into performance losses and power costs in the optimization process.

The corner optimization approach has another theoretical limitation: the fact that the circuit operates correctly in the corners does not guarantee (at least theoretically) that it will operate correctly at any point inside the cube defined by those corners. Figure 6-2 illustrates an unlikely but theoretically possible situation in which corner optimization fails.

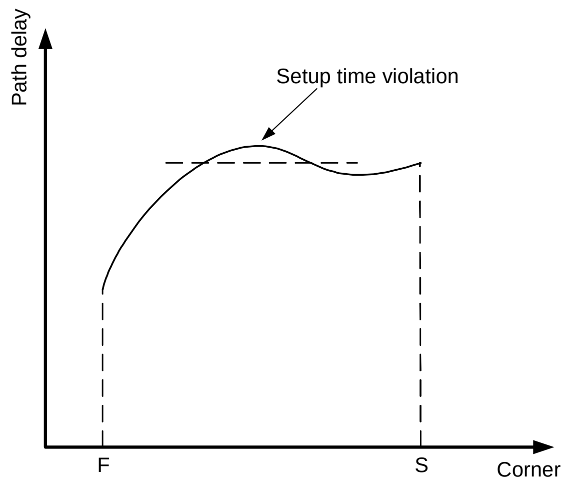

Figure 6-2. How designing for process corners can produce non-functional circuits

The figure presents a simplified situation in which the delay of a path depends on only one process parameter. While such a situation is not possible with the simpler delay models described in Chapter 3, the very non-linear characteristics of deeply scaled transistors can (at least theoretically) produces such a non-monotonicity in a performance metric. Since there is no constraint in the optimization problem to make sure that the circuit still works at the intermediate points between the process corners, a setup time violation (or other error) is possible in this situation.

A more thorough approach that extends corner exploration is the optimization with *uncertain parameters*:

$$\begin{aligned} &\min_{x \in \Re^n} f(x) \text{ such that} \\ &\quad g_i(x, \zeta) \le 0 \quad i = 1 \ldots m \\ &\text{for all} \quad \zeta \in K \end{aligned} \qquad \textbf{(6-2)}$$

In this case the constraint functions depend explicitly on the optimization variables *x* and on a set of parameters ζ that can fluctuate *anywhere* within an acceptable set, *K (*for instance the interior of the cube defined by the process corners). The optimization problem (6-2) requires that the constraints be satisfied at *all* the points of the *K* set, not just at its corners.

If the original problem is convex and the set *K* has certain properties, the resulting optimization problem with uncertain parameters can also be put in convex form. A comprehensive presentation about convex optimization with uncertain parameters can be found in [Ben-Tal98] (the authors use the name “robust convex optimization” for it).

Since geometric programming is the most common form of optimization for digital circuits, its uncertain counterpart is of particular interest. Section 3.5 of [Ben-Tal98] presents a way to transform a GP with uncertain parameters into a regular GP if the constraints g_i are affine in ζ and the acceptable parameter set *K* is an ellipsoid. The choice of an ellipsoid is not random: it can be exactly the ellipsoid from Figure 6-1.

While interesting from a theoretical standpoint, the GP with uncertain parameters in an ellipsoid is not practical due to the difficulties of specifying the ellipsoid. In terms of optimization results, it is even more conservative and pessimistic than optimization across process corners because it imposes correct functionality constraints on all the points of the acceptable parameter set.

Both optimization strategies presented so far in this section attempt to put *deterministic* bounds on *random* parameter variations and design the circuit within those bounds. A more natural approach is to let the process parameters be random - and therefore treat them like *random variables* - and include *probability distributions* in the optimization. The resulting *stochastic optimization* problem requires that the constraints be satisfied with a certain probability, η:

$$\min_{x \in \Re^n} f(x) \text{ such that } P[g_i(x, \zeta) \le 0] \ge \eta \qquad \textbf{(6-3)}$$

In (6-3) the uncertain parameters ζ are random and are characterized by a joint probability distribution function and η can be interpreted as the *yield* (sometimes this type of problem is also called *yield optimization*). It should be noted that the optimization variables x and the uncertain parameters ζ are disjoint sets. In stochastic optimization it is not possible for an optimization variable to be uncertain at the same time. This is a limitation since in reality typical optimization variables such as gate sizes are subject to the same random variations when the chip is manufactured.

Enforcing a minimum yield constraint is essentially equivalent to specifying the set where acceptable parameters lie - like the K set in (6-2). If the parameters ζ are jointly Gaussian, the probabilistic constraint from (6-3) has the equivalent of an ellipsoid in the parameter space (like in Figure 6-1). The ellipsoid can be directly computed from the statistical properties of the process parameters and the minimum yield, η. This is very important because it enables stochastic optimization for digital circuits.

The next section shows how to transform a stochastic GP (with direct application to digital circuits) into a regular (deterministic) GP. The ellipsoid and its associated constraint is just an intermediate step in the calculation with no other relevance.

6.2 Stochastic geometric programming with affine uncertainties

Consider the following deterministic GP:

$$\min c^T x \quad \text{s.t.} \quad \sum_{j=1}^{k} a_j \cdot e^{\beta_j^T x} \leq 1 \qquad \textbf{(6-4)}$$

The GP is already in convex form by using the exponential substitution (2-4) and has only inequality constraints, for simplicity. In this particular case the objective function is constrained to be a linear function. Any optimization problem can be converted to an equivalent problem with linear objective function using the epigraph form [Boyd03]:

$$\min f(x) \quad \text{s.t.} \quad g(x) \leq 0 \text{ is equivalent to } \min t \quad \text{s.t.} \quad g(x) \leq 0 \text{ and } f(x) \leq t \qquad \textbf{(6-5)}$$

We will make the assumption that only the a_j coefficients are uncertain whereas the exponents β_j and objective function coefficients *c* are certain. For the models used in digital circuits optimization these assumptions are not very restricting. Indeed, for the models presented in Chapter 3 the exponents are always +/-1 or +/-2 with very little uncertainty. Instead, the gate parameters such as *p* and *g* are subject to significant uncertainty and they appear as affine coefficients in the constraints. Also, in the epigraph form the objective function is just *t* as shown in (6-5), with the coefficient equal to 1, deterministic.

Since a_j are the only uncertain parameters and the constraint functions are affine in a_j, this type of problem is called *affinely parametrized* optimization problem.

Assuming a is a jointly Gaussian random vector with mean $\bar{a}$ and covariance matrix Σ (denoted as $a \sim N(\bar{a}, \Sigma)$) we can formulate the stochastic optimization problem derived from (6-4):

$$\min c^T x \quad \text{s.t.} \quad P\left[\sum_{j=1}^{k} a_j \cdot e^{\beta_j^T x} \le 1\right] \ge \eta \tag{6-6}$$

In (6-6) the deterministic constraint is replaced by a probabilistic constraint: we want to satisfy the constraint with a probability of at least η when the uncertain parameters a_j vary according to their distribution. η can be interpreted as the desired *yield*. All subsequent mathematical derivations assume that η is at least 50%.

There is no available optimizer that can handle explicit probabilistic constraints. In order to solve such a problem it is necessary to remove the probability operator $P[...]$ from the constraints and transform the stochastic optimization problem into an equivalent deterministic problem. The following mathematical derivation shows how to transform the stochastic GP into a deterministic GP.

Let u be a random variable defined as:

$$u = \sum_{j=1}^{k} a_j \cdot e^{\beta_j^T x} \tag{6-7}$$

Because u is a linear combination of the components of a jointly Gaussian random vector (a), u is also a Gaussian random variable itself:

$$u \sim N(\bar{u}, \sigma) \tag{6-8}$$

and the probabilistic constraint from (6-6) can be written as:

$$P(u \le 1) = P\left(\frac{u}{\sigma} \le \frac{1}{\sigma}\right) = P\left(\frac{u-\bar{u}}{\sigma} \le \frac{1-\bar{u}}{\sigma}\right) \ge \eta \qquad \textbf{(6-9)}$$

For obvious reasons $\frac{u-\bar{u}}{\sigma} \sim N(0, 1)$ and therefore:

$$P\left(\frac{u-\bar{u}}{\sigma} \le \frac{1-\bar{u}}{\sigma}\right) = \Phi\left(\frac{1-\bar{u}}{\sigma}\right) \qquad \textbf{(6-10)}$$

where:

$$\Phi(z) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{z} e^{-\frac{t^2}{2}} dt \qquad \textbf{(6-11)}$$

is the cumulative distribution of a Gaussian random variable with zero mean and unit variance. The Φ function and its inverse are readily available in Matlab and other mathematical packages.

The probabilistic constraint (6-9) can be rewritten as:

$$\frac{1-\bar{u}}{\sigma} \ge \Phi^{-1}(\eta) \qquad \textbf{(6-12)}$$

which comes down to:

$$\bar{u} + \sigma \cdot \Phi^{-1}(\eta) \le 1 \qquad \textbf{(6-13)}$$

In order to compute the mean and variance of the Gaussian random variable *u* we must note that the optimization variables *x*, although unknown to us are *not random variables*. To compute the mean and variance (just another kind of mean) we must average only across those variables that are random (a_j) and not across the certain (but unknown) variables. Using the composition rules for jointly Gaussian random vectors, the mean of u is:

$$\bar{u} = \sum_{j=1}^{k} \bar{a}_j \cdot e^{\beta_j^T x} \tag{6-14}$$

and the variance σ is:

$$\sigma = \sqrt{F^T \Sigma F} = \sqrt{\sum_{p=1}^{k} \sum_{q=1}^{k} (\Sigma)_{p,q} e^{(\beta_p + \beta_q)^T x}} \tag{6-15}$$

where $F = \left[e^{\beta_1^T x}, \ldots, e^{\beta_k^T x}\right]^T$.

Replacing (6-15) in (6-13) the probabilistic constraint becomes:

$$\sum_{j=1}^{k} \bar{a}_j \cdot e^{\beta_j^T x} + \Phi^{-1}(\eta) \cdot \sqrt{\sum_{p=1}^{k} \sum_{q=1}^{k} (\Sigma)_{p,q} e^{(\beta_p + \beta_q)^T x}} \leq 1 \tag{6-16}$$

(6-16) is a deterministic constraint that is equivalent to the original probabilistic constraint and represents the ellipsoid from [Ben-Tal98] in the e^x variables. Although (6-16) could be used "as is" in a blind optimization, it has no convexity properties nor is it a posynomial. However, through further mathematical processing (6-16) can be split into two posynomial constraints in order to obtain a GP equivalent with the original stochastic optimization problem.

Note that the following two optimization problems are equivalent:

$$\min x \quad \text{s.t.} \quad a + \sqrt{b} \le 1 \text{ is equivalent to}$$

$$\min x \quad \text{s.t.} \quad a + t \le 1 \text{ and } \sqrt{b} \le t \Leftrightarrow \frac{b}{t^2} \le 1 \qquad \textbf{(6-17)}$$

The newly introduced variable t is called a *slack variable* [Boyd03]. Using (6-17) the original probabilistic constraint breaks down into two deterministic constraints, both posynomials:

$$\begin{aligned} &\sum_{j=1}^{k} \bar{a}_j \cdot e^{\beta_j^T x} + e^t \le 1 \\ &[\Phi^{-1}(\eta)]^2 \cdot \sum_{p=1}^{k} \sum_{q=1}^{k} (\Sigma)_{p,q} e^{[(\beta_p + \beta_q)^T x - 2t]} \le 1 \end{aligned} \qquad \textbf{(6-18)}$$

Therefore, the stochastic version of the deterministic GP (6-4) when its constraints are affinely parametrized is also a GP:

$$\begin{aligned} &\min c^T x \quad \text{s.t.} \\ &\sum_{j=1}^{k} \bar{a}_j \cdot e^{\beta_j^T x} + e^t \le 1 \\ &[\Phi^{-1}(\eta)]^2 \cdot \sum_{p=1}^{k} \sum_{q=1}^{k} (\Sigma)_{p,q} e^{[(\beta_p + \beta_q)^T x - 2t]} \le 1 \end{aligned} \qquad \textbf{(6-19)}$$

(6-19) is called the *stochastic GP* and is significantly more complex than the original deterministic GP. It should be noted that the stochastic GP is a regular (deterministic) GP that can be solved my conventional optimizers such as [Mosek06], [GGPLAB06] and [Mathworks05].

Each constraint with k monomials in the deterministic GP produces a constraint with k monomials, another constraint with k^2 monomials, and an additional variable in the stochastic GP. Although the increase in complexity might seem drastic, it is still manageable because when using a static timer the constraints do not have too many monomials as demonstrated by (3-29).

The most difficult part in a stochastic GP is collecting the data for the probability distribution of the parameter vector, a. The mean $\bar{a}$ is easily identified with the nominal values of the parameters but the covariance matrix Σ is more difficult to fill. Correlation data is not readily available from foundries (like process corner data) and must be measured or computed separately.

Statistical static timing [Jess03] can be used to compute the required correlation coefficients from a small set of global process variables (such as temperature and threshold voltage). Statistical static timing tools are not fully developed and there is little data available to use them. A more direct approach is to measure within-die and die-to-die variations and correlations for a set of controlled structures on a test chip and then derive rules to infer what the correlation coefficients will be in the general case [Pang06].

6.3 Example: optimal orientation of the critical path in a 64-bit carry tree

The purpose of this example is to test the stochastic optimization flow and to analyze the impact of variations on the design of a simple circuit. The circuit under test is the critical path of a 64-bit radix 2 carry tree implemented in static CMOS. The example is technology independent and uses the logical effort delay model and normalizations [Sutherland99].

The critical path of the carry tree is represented in a manner similar to Figure 3-2. It consists of 6 gates with the appropriate branching efforts at the nodes. The path drives a load C_L=64 r.u. (relative units) and the maximum input capacitance is constrained to 1 r.u. Each gate is characterized by 2 parameters, *p* and *g*, that are subject to variations. Since the *p* terms can be lumped together, their sum is considered as a single parameter. Consequently, the path has seven parameters (6 *g*'s and one lumped *p*) and a 7x7 correlation matrix.

The mean of the parameters is set to their nominal values. A recent work by Pang et. al. [Pang06] offers a way to compute the correlation matrix for the path parameters based on its orientation on the chip. The authors of [Pang06] identified two different process variation mechanisms along the two dimensions of the chip:

In slit-and-scan photolitography, a narrow slit of light is shone through the mask and both the mask and wafer are moved such that the image of the reticle field is projected onto the wafer. In the direction of the slit, variations are due to lens aberrations and result in more correlated features. In the scan direction (orthogonal to the slit direction), variations are due to dosage and scan speed, which are better controlled. Hence, in the scan direction there are less systematic variations and features are less correlated. For the chip presented in [Pang06], the slit direction is horizontal and the scan direction is vertical. The correlation coefficient of the features decreases in an approximately linear fashion from one to zero over 500μm in the horizontal direction (more correlation) and over 300μm in the vertical direction (less correlation).

The gates in the path under test are spaced according to their position in a real 64-bit adder, spanning 500μm (64 bitslices) and the correlation coefficients are computed using the above rule for each orientation. A stochastic unconstrained delay minimization is solved for the path at various yield targets in three cases:

1. a “reference” case, where all the parameters are independent random variables, with variances inferred from the measurements in [Pang06];
2. a “horizontal” case, where the critical path is routed mostly along the slit direction of the chip;
3. a “vertical” case, where the critical path is routed mostly along the scan direction of the chip.

Figure 6-3 shows the minimum achievable delay for various yield targets in these three cases. The delay is normalized to the delay of the fastest carry tree obtained in the deterministic optimization. Figure 6-4 shows the corresponding area of the carry tree.

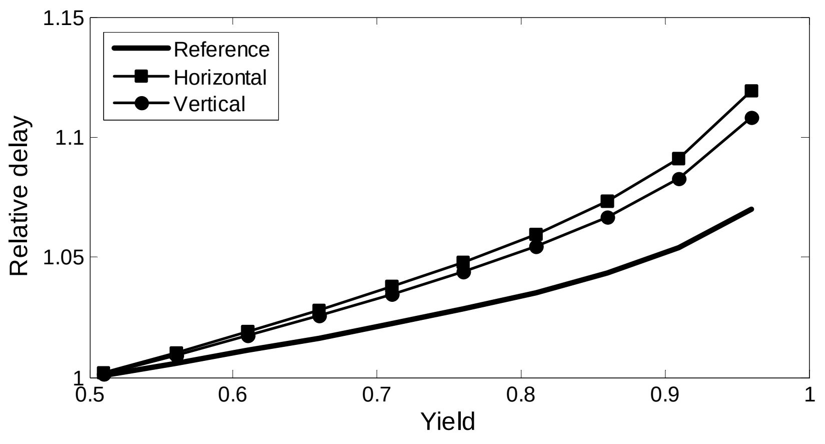

Figure 6-3. Unconstrained stochastic delay minimization for 64-bit carry tree critical path: normalized minimum delay.

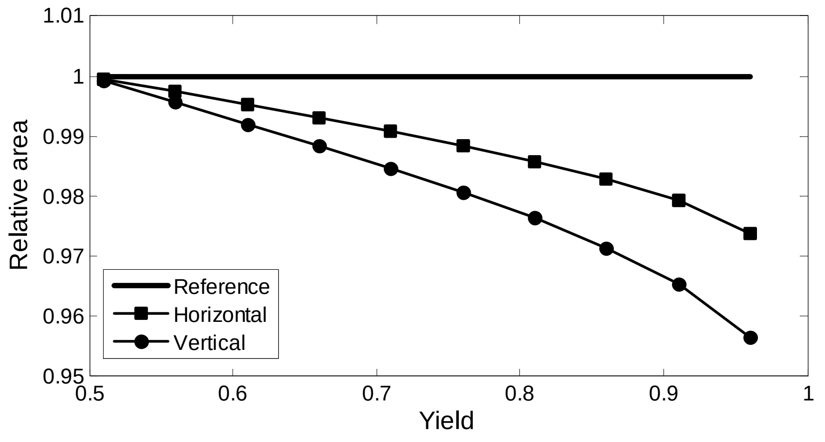

Figure 6-4. Unconstrained stochastic delay minimization for 64-bit carry tree critical path: normalized area.

The "vertical" layout, with less correlations between the gates, is both smaller and faster than the "horizontal" layout. In this case correlations slow down the circuit: if a gate is slower, there is high likelihood that neighboring gates are slower, too (in fact, in this example the correlation distance is equal to the length of the path for the horizontal case, therefore *all* gates in the path are affected), thus producing a slower circuit at the same yield target. If the gates are less correlated, the variations are more likely to average out along the path, thus reducing the delay at the same yield target.

The line for the reference case with independent parameters in Figure 6-4 is perfectly horizontal; in fact all the designs along that curve are identical to the deterministic design. Since the individual variances of the parameters are all the same, the deterministic design produces a well balanced delay distribution. The difference that exists in the delays from Figure 6-3 corresponds to different points on this delay distribution: the same design can have different yields at different delay targets.

While the differences are small (up to 1% for delay and 3% for area) for such a simple circuit, this example shows what are the possible gains of taking the statistical characteristics of the process variations into account at design time. Statistical characteristics do not include just the individual variances of the parameters, but their joint moments as well.

Accounting for correlations is not an easy task. It requires filling a correlation matrix that spans the whole circuit and that is usually not sparse. Moreover, correlation data is not readily available from the foundries. This emphasizes the importance of the research into measuring and calculating such correlations [Pang06] and into generating the correlation coefficients without the need for a huge non-sparse matrix [Jess03].

Dealing with process variations, in general, and robust optimization for digital circuits, in particular, are still areas open for research and significant work is still needed. This chapter presented a framework that can be used to tackle the problem of robust optimization for digital circuits, but leaves several unanswered questions and unsolved problems. First and foremost, the basic assumption is that all the parameters that vary are jointly Gaussian. While assuming variations to be Gaussian has been very common practice throughout many decades of literature, recent work such as [Qin04] concludes that some process parameters follow a log-normal distribution. For small variations the Gaussian approximation generally holds, but the trend is opposite, towards large variation amplitudes in deeply scaled technologies.

The example in this chapter demonstrates the importance of capturing the correlations between process parameters across the die and the cost of ignoring them in the power - performance space. The mathematical theory is helped by the jointly Gaussian assumption in which a correlation matrix is enough to characterize the joint distribution of all variables. However, even using this simplest way of characterizing correlation, the task of gathering the data for the covariance matrix of a real process has proved very difficult in practice.

7 Conclusions

This thesis addresses the topic of power - performance optimization for custom digital circuits at circuit and microarchitecture levels. It:

- formulates the design as a power - performance optimization problem;
- presents a custom-written optimization framework to solve this problem;
- demonstrates the framework on practical circuits.

The circuit optimization framework provides a systematic solution to the problem of maximizing the performance of the circuit in a limited power budget and to its dual - minimizing the power of the circuit with a minimum performance constraint. The main contributions of the framework are:

- modular design enabling great flexibility in the choice of models with various degrees of complexity and accuracy;
- static timing - based approach removing the dependency of input vector patterns and ensuring a conservative and robust design;
- formulation of the design as a mathematical optimization problem that is solved numerically;
- guarantee for the global optimality of the result for certain classes of analytical models leading to convex optimization for combinational circuits;

- verification of the quality of the results against a tight near-optimality boundary for combinational circuits; the boundary is computed using analytical models leading to convex optimization and can be use to check the quality of the results obtained with very accurate but non-convex models (e.g. tabulated models);
- for sequential circuits, a combination of optimality guarantee / near - optimality check for the logic with near - optimal heuristic - based retiming for the storage elements;
- development of a stochastic yield optimization methodology that maintains the same type of optimality guarantee as the original deterministic problem (not fully integrated with the rest of the framework);

The circuit optimization framework is demonstrated on four examples of practical circuits:

- a 64-bit Kogge-Stone carry tree implemented in static CMOS in a general purpose 130nm process. Using analytical models, this example demonstrates the impact in the energy - delay space of optimizing different sets of design variables (such as gate sizes, supply voltage and threshold voltage). The main conclusion of this example is that the optimal design has always equal sensitivities to all the designs variables. This may sometimes lead to counter-intuitive solutions, such as reducing power by increasing the supply voltage (but downsizing to retain the same performance);
- a detailed study of 64-bit carry lookahead adders in a general purpose 90nm bulk CMOS process. Using tabulated models, the study investigates the impact in the energy - delay space of several adder design choices: set of logic equations, logic family, carry tree radix, carry tree lateral fanout, carry tree sparseness, sizing strategy, adder environ-

ment and process parameters. A set of design guidelines is formulated for these choices in order to guide the selection of the best adder architecture in a specific environment. A proof of concept implementation of the fastest adder in 90m CMOS in a typical high-performance microprocessor environment is used to verify the optimizations. The optimal architecture in this case is a radix 4 sparse-2 clock-delayed domino carry tree implementing Ling's equations. The measurements show an average delay of 240 ps (7.5 FO4) across all the chips in the batch, at the nominal supply voltage of 1V and with a power consumption of 260mW for the worst case input combination.

- an IEEE-compliant single-precision fused multiply-add Floating Point Unit (FPU) implemented in static CMOS in a general purpose 90nm process. Using analytical models, this example compares the results of designing the same block using a commercial logic synthesis tool and the circuit optimization framework. The framework offers most benefits for circuits in the high-performance end of the spectrum due to the continuous sizing process coupled with a sizing-aware retiming algorithm. In the low-power end of the spectrum the benefits decrease significantly because most gates have minimum size. For a mid-range FPU design with 2ns cycle time and latency of 4 cycles, the circuit optimization framework saves 40.2% of the power through continuous gate sizing and an additional 7.2% through retiming over the standard-cell based ASIC design flow.
- the critical path of a static CMOS 64-bit adder in a generic normalized technology. Using analytical models, this example shows the impact of process parameter variations in the energy - delay space and demonstrates stochastic yield optimization. With

normalized variances of process parameters predicted to approach 0.5 for the 45nm technology node, designing circuits with good yield can incur a high cost in power and performance. Knowledge of within-die and die-to-die correlations helps alleviate the negative impact of variations and accounting for them in the yield optimization can improve performance and save power. This example shows that routing a critical path in the direction in which process parameters are least correlated produces the fastest and smallest circuit at the same yield target.

Future work on the circuit optimization framework is focused on four major directions:

- full integration of the stochastic yield optimization methodology with the rest of the framework; this will have to address the issue of collecting data about the correlations in the circuit;
- inclusion of clock generation and distribution in the optimization process; the current version takes into account the power on the clock lines but not what is required to generate and distribute those signals;
- integration of the framework in a general synthesis environment; this would allow the exploration of logic restructuring in the power - performance optimization process and a shift to standard cell-based ASIC design;
- development of a hierarchical abstraction of the underlying circuit fabric to characterize functional blocks in the power - performance space; this would allow an effective communication with the system architects and provide better integration of the whole design optimization process.

Bibliography

[Amrutur01] B. Amrutur, M. A. Horowitz: “Fast low-power decoders for RAMs”, IEEE Journal of Solid-State Circuits vol.36 10, p. 1506-1514, 2001

[Beaumont-Smith01] A. Beaumont-Smith, C.C. Lim: “Parallel prefix adder design”, 15th Symposium on Computer Arithmetic, p.218-225, 2001

[Ben-Tal98] A. Ben-Tal, A. Nemirovski: “Robust convex optimization”, Mathematics of Operations Research vol. 23 p. 769-805, 1998

[Borkar00] S. Borkar: presentation at UC Berkeley IC seminar, Fall 2000, unpublished

[Boyd03] S. Boyd, L. Vandenberghe: “Convex optimization”, Cambridge University Press, 2003

[Boyd05] S. Boyd, S.J. Kim, D.D. Patil, M. A. Horowitz: “Digital circuit optimization via geometric programming”, to appear in Mathematics of Operations Research

[Conn99] A. R. Conn, I. M. Elfadel, W. W. Molzen Jr., P. R. O'Brien, P. N. Strenski, C. Visweswariah, C. B. Whan: “Gradient - based optimization of custom circuits using a static timing formulation”, Proceedings of Design Automation Conference DAC, p. 452 - 459, 1999

[Dao01] H. Q. Dao, V. G. Oklobdzija: “Application of logical effort techniques for speed optimization and analysis of representative adders", 35th Asilomar Conference on Signals, Systems and Computers, vol. 2 p. 1666-1669, 2001

[Dao03] H. Q. Dao, B. R. Zeydel, V. G. Oklobdzija: "Energy minimization method for optimal energy - delay extraction" Proceedings of European Solid-State Circuits Conference, p. 177-180, 2003

[Donnelly95] C. Donnely, R. Stallman: "Bison, the YACC-compatible parser generator", documentation to bison, 1995

[Doran88] R. W. Doran: "Variants of an improved carry look-ahead adder", IEEE Transactions on Computers, vol. 37 p. 1110-1113, 1998

[Fetzer02] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, Z. Chengyu, E. Busta, B. Mohammad: "A fully bypassed six-issue integer datapath and register file on the Itanium-2 microprocessor", IEEE Journal of Solid-State Circuits vol. 37 no.11 p. 1433-1430, 2002

[Fishburn85] J.P. Fishburn, A. Dunlop, A. E.: "TILOS: A posynomial programming approach to transistor sizing", IEEE International Conference on Computer - Aided Design, p. 326-328, 1985

[Frank04] D. J. Frank: "Parameter Variations in sub-100nm CMOS technology", presentation at ISSCC2004 Microprocessor Design Forum

[Garret04] J. Garrett: Master thesis, UC Berkeley, 2004

[GGPLAB06] GGPLAB: A simple Matlab toolbox for geometric programming, http://www.stanford.edu/~boyd/ggplab/

[Golden06] M. Golden, S. Arekapudi, G. Dabney, M. Haertel, S. Hale, L. Herlinger, Y. Kim, K. McGrath, V. Palisetti, M. Singh: “A 2.6 GHz dual-core 64bx86 microprocessor with DDR2 memory support”, International Solid-State Circuit Conference, p. 104-105, 2006

[Han87] T. Han, D. A. Carlson: “Fast area efficient VLSI adders”, Symposium on Computer Arithmetic, p. 49-56, 1987

[Harris03] D. Harris: “A taxonomy of parallel prefix networks”, Asilomar Conference on Signals, Systems and Computers, p. 2213-2217, 2003

[Huang00] Z. Huang, M. D. Ercegovac: “Effect of wire delay on the design of prefix adders in deep-dubmicron technology”, Asilomar Conference on Signals, Systems and Computers, p. 1713-1717, 2000

[ITRS05] International Technology Roadmap for Semiconductors, 2005 Edition, http://www.itrs.net/Links/2005ITRS/Home2005.htm

[Jess03] J. A. G. Jess, K. Kalafala, S. R. Naidu, R. H. J. Oten, C. Visweswariah: “Statistical timing for parametric yield prediction of digital integrated circuits”, Design Automation Conference p. 932-937, 2003

[Kao06] S. Kao, R. Zlatanovici, B. Nikolic: “A 240ps 64b carry-lookahead adder in 90nm CMOS”, International Solid-State Circuits Conference, p. 438-439, 2006

[Knowles01] S. Knowles: “A family of adders”, Symposium on Computer Arithmetic, p. 277-281, 2001

[Kogge73] P. M. Kogge, H. S. Stone: “A parallel algorithm for efficient solution of a general class of recursive equations”, IEEE Transactions on Computers, 786-793, 08/1973

[Krishnamurthy02] R. Krishnamurthy, et al, "Dual supply voltage clocking for 5 GHz 130 nm integer execution core," Symposium on VLSI Circuits, p. 128-129, 2002

[Ladner80] R. E Ladner, M. J. Fischer: “Parallel prefix computation”, JACM 27(4) p.831-838, 1980

[Leiserson91] C. E. Leiserson, J. B. Saxe: “Retiming synchronous circuitry”, Algoritmica 6(1) p. 5-35

[Ling81] H. Ling: “High speed binary adder”, IBM Journal of Research and Development, vol.25, no.3, p. 156-166, 1981

[Markovic04] D. Markovic, V. Stojanovic, B. Nikolic, M. Horowitz, R. W. Brodersen : “Methods for true energy performance optimization”, IEEE Journal of Solid-State Circuits vol. 39 p. 1282-1293, 2004

[Mathew01] S. Mathew, R. Krishnamurthy, M. Anders, R.Rios, K. Mistry, K. Soumyanath: “Sub-500ps 64b ALUs in 0.18µ¦m SOI/bulk CMOS: design & scaling trends”, International Solid-State Circuits Conference, p. 318-319, 2001

[Mathew02] S. Mathew, M. Anders, R. Krishnamurty, S. Borkar: “A 4GHz 130nm address generation unit with 32-bit sparse-tree adder core”, Symposium on VLSI Circuits, p.126-127, 2002

[Mathew05] S. Mathew, M.A. Anders, B. Bloechel, T. Nguyen, R.K. Krishnamurthy, S. Borkar: “A 4GHz 300-mW 64-bit integer execution ALU with dual supply voltages in 90nm CMOS”, IEEE Journal of Solid-State Circuits vol. 40 no.1 pp. 44-51, 2005

[Mathworks05] Mathworks, Matlab optimization toolbox users guide Version 3, www.mathworks.com

[Moore65] G. E. Moore: “Cramming more components onto integrated circuits”, Electronics Magazine vol. 38, No. 8, 1965

[Moore75] G. E. Moore: “Progress in digital integrated electronics”, International Electron Devices Meeting, p. 11, 1975

[Moore03] G. E. Moore: “No exponential is forever. But “forever”can be delayed!” International Solid-State Circuits Conference, p.20-23, 2003

[Mosek06] Mosek Optimization Toolbox 4.0, online documentation at www.mosek.com

[Naffziger96] S. Naffziger: “A sub-nanosecond 0.5µm 64b adder design”, International Solid-State Circuits Conference, p. 210-211, 1996

[Naffziger06] S. Naffziger, B. Stackhouse, T. Grutkowski, D. josephson, J. Desai, E. Alon, M. Horowitz: “The implementation of a 2- core multi-threaded Itanium family processor”, IEEE Journal of Solid-State Circuits vol. 41 no. 1 pp. 197 - 209, 2006

[Nakagome93] Y. Nakagome, K. Itoh, M. Isoda, K. Takeuchi, M. Aoki, "Sub-1-V swing internal bus architecture for future low-power ULSIs," IEEE Journal of Solid-State Circuits, vol. 28 no. 4, pp. 414-419, 1993.

[Ofman63] Y. Ofman: On the algorithmic complexity of discrete functions, Soviet Physics - Doklady 7(7) pp.589 - 591, 1963

[OpenCores06] www.opencores.org

[Pang06] L.T. Pang, B. Nikolic: “Impact of layout on 90nm CMOS process parameter fluctuations”, Symposium on VLSI Circuits, p. 69-70, 2006

[Pangjun02] J. Pangjun, S. S. Sapatnekar, "Low-power clock distribution using multiple voltages and reduced swings," IEEE Transactions on VLSI Systems, vol. no. 3. p. 309-318, 2002.

[Park00] J. Park, H. C. Ngo, J. A. Silberman, S. H. Dhong: “470ps 64bit parallel binary adder”, Symposium on VLSI Circuits p. 192-193, 2000

[Paxson95] V. Paxson: “Flex v2.5, A fast scanner generator, documentation to flex”, 1995

[Penzes02] P. I. Penzes, A. J. Martin: “Energy - delay efficiency of VLSI computations”, Great Lakes Symposium on VLSI, p. 104-111, 2002

[Puri03] R. Puri, et al.: “Pushing ASIC performance in a power envelope”, Design Automation Conference, p. 788-793, 2003

[Qin04] H. Qin, Y. Cao, D. Markovic, A. Vladimirescu, J. Rabaey: “SRAM leakage suppression by minimizing standby supply voltage”, IEEE International Symposium on Quality Electronic Design, p. 49-52, 2004

[Rabaey03] J.M Rabaey, A. Chandrakasan, B. Nikolic: “Digital integrated circuits: a design perspective”, 2nd edition, Prentice-Hall 2003

[Rusu06] S. Rusu, S. Tam, H. Muljono, D. Ayers, J. Chang : “A dual-core multi-threaded Xeon processor with 16MB L3 Cache”, International Solid-State Circuits Conference, p. 102-103, 2006

[Shenoy94] N. Shenoy, R. Rudell: “Efficient implementation of retiming”, International Conference on Computer Aided Design , p. 226-233, 1994

[Shenoy97] N. Shenoy: “Retiming: theory and practice”, Integration, the VLSI journal, vol. 22 p. 1-21, 1997

[Shimazaki03] Y. Shimazaki, R. Zlatanovici, B. Nikolic: “A shared-well dual-supply-voltage 64-bit ALU”, International Solid-State Circuits Conference, p. 104-105 2003

[Stojanovic02] V. Stojanovic, D. Markovic, B. Nikolic, M. A. Horowitz, R. W. Brodersen: “Energy - delay tradeoffs in combinational logic using gate sizing and supply voltage optimization”, European Solid-State Circuit Conference p. 211-214, 2002

[Sutherland99] I. Sutherland, R. Sproul, D. Harris: “Logical effort”, Morgan-Kaufmann, 1999

[Synopsys04] Synopsys Design Compiler users manual Version 2004.12

[Toh98] K. Y. Toh, P. K. Ko, R. G. Meyer: “An engineering model for short-channel CMOS devices”, IEEE Journal of Solid-State Circuits vol. 23 p. 950-958, 1998

[Usami95] K. Usami and M. A.Horowitz, "Clustered voltage scaling for low-power design," International Symposium on Low Power Electronics and Design, p. 3-8, 1995

[Usami00] K. Usami, M. Igarashi, "Low-power design methodology and applications utilizing dual supply voltages," Asia and South Pacific Design Automation Conference, p. 123-128, 2000

[Zlatanovici03] R. Zlatanovici, B. Nikolic: “Power- performance optimal 64-bit carry-lookahead adders”, European Solid-State Circuit Conference , p. 321-324, 2003

[Zlatanovici05] R. Zlatanovici, B. Nikolic: Power - performance optimization for custom digital circuits”, PATMOS, p. 404-415, 2005

[Zyuban02] V.Zyuban, P.Strenski: “Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels”, International Symposium on Low Power Electronics and Design, p.166-171, 2002

[Zyuban03] V.Zyuban, P.Strenski: “Balancing hardware intensity in microprocessor pipelines”, IBM Journal of Research and Development, vol.47, no.5/6, p. 585-598, 2003

[Zyuban04] V.Zyuban, D.Brooks, V. Srinivasan, M. Gschwind, P. Bose, P. N. Strenski, P. G. Emma: “Integrated analysis of power and performance for pipelined microprocessors”, IEEE Transactions on Computers, vol. 53 p.1004-1016, 2004