SIMULTANEOUS DELAY, YIELD, AND TOTAL POWER OPTIMIZATION IN DEEP-SUBMICRON CMOS TECHNOLOGY USING GATE-SIZING, THRESHOLD VOLTAGE ASSIGNMENT, NODAL CONTROL, AND TECHNOLOGY MAPPING

by

Hsinwei Chou

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

December 2004

To my family, for all their support throughout my academic endeavor.

ACKNOWLEDGMENTS

I thank Professor Charlie Chung-Ping Chen for guiding me during my master’s study and helping me discover the wonderful field of EDA. This research would not be possible without his invaluable suggestions and support. I also thank the University of Wisconsin-Madison for giving me the opportunity to continue my education and advance my knowledge. Lastly, I thank my colleagues at the UW-Madison VLSI-EDA lab for their helpful discussions and their friendship over the past two years.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

DISCARD THIS PAGE

LIST OF TABLES

DISCARD THIS PAGE

LIST OF FIGURES

SIMULTANEOUS DELAY, YIELD, AND TOTAL POWER OPTIMIZATION IN DEEP-SUBMICRON CMOS TECHNOLOGY USING GATE-SIZING, THRESHOLD VOLTAGE ASSIGNMENT, NODAL CONTROL, AND TECHNOLOGY MAPPING

Hsinwei Chou

Under the supervision of Assistant Professor Charlie Chung-Ping Chen
At the University of Wisconsin-Madison

Charlie Chung-Ping Chen

ABSTRACT

The aggressive scaling of CMOS technology below 90nm brings forth many great new physical challenges that must be addressed in the forefront of the design cycle. In addition to meeting the usual timing and dynamic/switching power budgets, designers must also now carefully consider issues such as leakage current/power, crosstalk noise, electromigration, IR drop, and even manufacturing process variations. This thesis explores several novel techniques for mitigating the effects of leakage current and process variations. In the first part of this thesis, a circuit-tuning method for simultaneous optimization of critical delay, dynamic power, leakage power, and yield will be proposed. The method is based on Generalized Lagrangian Relaxation, and involves gate-sizing and multiple threshold voltage assignment. Experimental results show that this method can not only effectively tune a circuit with over 15,000 variables and 8,000 constraints in under 7 minutes, but it can also minimize the impact of process variations on timing. In the second part of this thesis, a ROBDD-based nodal control technique called Sectoral Partial Vector Control (SPVC) will be proposed to address the minimization of CMOS subthreshold leakage current/power. Together with leakage-aware technology mapping, this method was empirically shown to be capable of reducing the total static leakage power consumption by as much as 69%. Overall, the two proposed techniques in this thesis can be used either independently or together to provide a comprehensive and effective circuit optimization framework for today's complex designs.

Chapter 1

Introduction

Due to the aggressive scaling of CMOS technology, today's designs face a myriad of physical problems not seen in previous generations of technology. This chapter gives a brief introduction to two of the most critical challenges among these problems in leakage current/power and manufacturing process variations.

1.1 Leakage Current

In accordance with Moore's Law, CMOS technology scales from one generation to another approximately every 18 months. Specifically, Vdd scales down by 1/S every generation, where S denotes the scaling factor. The reason why Vdd has been scaling down over time is because of the fact that the dynamic or switching power consumption of a transistor is quadratically dependent on its supply voltage level, as it can be seen in the following equation:

$$P_{dynamic} = \alpha * f_{clk} * C_{load} * Vdd^2 \tag{1.1}$$

This scaling methodology of Vdd and other device parameters by a uniform factor of S is known as constant field scaling, and has been the industry standard for the past decade. However, since the reduction of Vdd alone would lead to a lower overdrive (V_{gs} - V_t) and thus a lower drive current, the device threshold voltage level has also been scaling down every generation in order to sustain performance. Unfortunately, this has negatively impacted the device subthreshold leakage current over time, as it can be seen from the following equation:

$$I_{leakage} = \frac{I_o}{W_o} * W * 10^{\frac{V_{gs}-V_t}{S}} \tag{1.2}$$

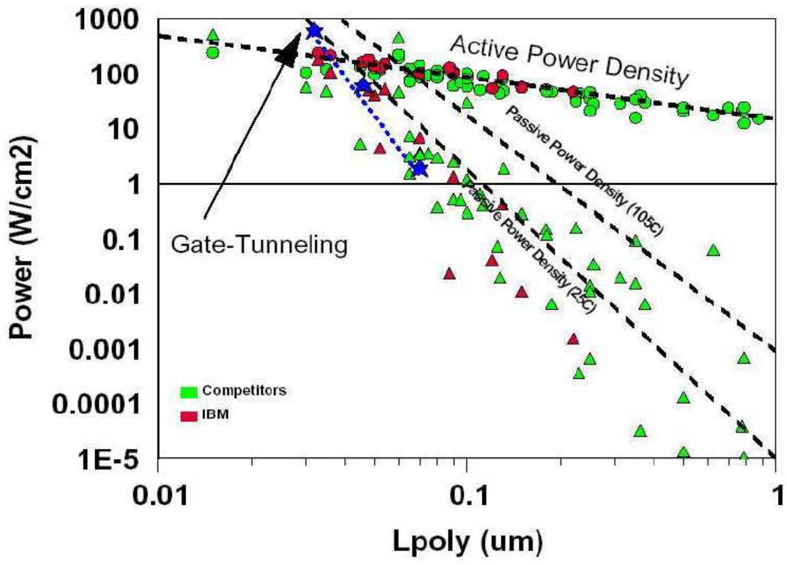


Figure 1.1 Power density as a function of the channel length, Lpoly

Equation 1.2 shows that the subthreshold leakage current of a device is inversely and exponentially proportional to its V_t level. Thus, the lowering of V_t through scaling has led to an exponential increase of leakage current/leakage power over the past decade. Consequently, in today's 90nm technology, leakage power already accounts for close to 50% of the total power consumption in Intel Pentium 4 microprocessors [1]. As it can be seen from Figure 1.1 [2], this will only continue to get worse as the leakage power is expected to overtake dynamic power sometime in the near-future as the largest contributor of power. Thus, for today's designers, minimization of both dynamic and leakage power is critical for achieving power closure.

1.1.1 Leakage Current Components

In general, the leakage current is comprised of several components. Referring to Figure 1.2, the eight different leakage mechanisms present within a transistor are as follows:

1. **P-N Junction Reverse-Bias Current** (I_1)
   Current which leaks from drain to bulk when the drain voltage is reverse-biased with respect to the bulk potential (thereby forming a natural, reverse-biased P-N junction).

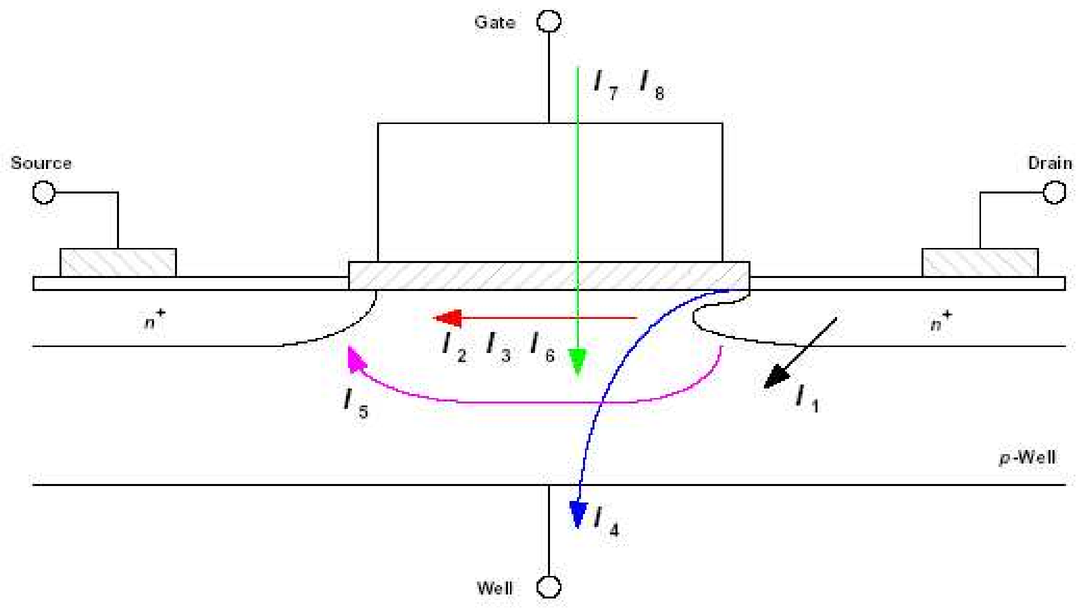


Figure 1.2 Components of leakage current

2. **Subthreshold Leakage Current (Weak Inversion)** (I_2)

   One of the dominant sources of leakage current, subthreshold leakage is the current which flows from drain to source even when $|V_{gs}| < |V_t|$ (but above the weak inversion point) and the transistor is supposed to remain turned-off. Similar to charge transport across the base of a bipolar transistor, the carriers move by diffusion between the drain and the source. The subthreshold current, under some mild assumptions, is modeled by Equation 1.2. As it can be seen, the subthreshold current has an exponential and inverse dependence on the threshold voltage, V_t. The lower the V_t, the higher the subthreshold leakage current.

3. **Drain-Induced Barrier Lowering (DIBL)** (I_3)

   This leakage current from drain to source results when there is a high-enough voltage applied to the drain such that the depletion region of the drain interacts with the source, thereby causing the source potential barrier near the channel surface to lower. Due to this effect, the carriers can now be injected into the channel surface by the source without any influence from the gate.

4. **Gate-Induced Drain Leakage (GIDL)** (I_4)

   Carriers which flow into the substrate and drain due to the high electric field between the gate and the drain overlap region.

5. **Punchthrough** (I_5)

   This occurs when the depletion regions of the drain and the source electrically touch each other.

6. **Narrow Width Effect** (I_6)

   For transistors using trenched-isolated technology, their effective V_t decreases for channel width $W \leq 0.5\mu m$ [3]. The lower the V_t, the higher the leakage current.

7. **Gate Oxide Tunneling** (I_7)

   Another major source of leakage, this is the current which flows due to the direct tunneling of electrons through the gate, or Fowler-Nordheim (FN) tunneling through the oxide bands. This current is a function of the oxide's electric field. Thus, as T_{ox} scales down, the thinning of the gate oxide leads to an exponential increase of the gate leakage. In current 90nm technology, the magnitude of this leakage component is on-par with that of the subthreshold leakage. However, unlike the subthreshold leakage, there are currently no good solutions to this problem at the circuit level. Most solutions exist at the device level (ie. high-K dielectric), but even they are still under heavy research. Thus, this leakage component is expected to be a major source of problem in future designs.

8. **Hot Carrier Injection** (I_8)

   This current results from the injection of hot carriers (both electrons and holes) into the oxide.

Among the eight different leakage mechanisms listed above, most of the total leakage current in today's 90nm process can be attributed to the reverse-biased P-N junction leakage, the gate-induced drain leakage, the subthreshold weak inversion leakage, and the gate oxide tunneling leakage. Although there are some known solutions for dealing with the reverse-biased P-N junction leakage and the gate-induced drain leakage, these leakage components are still small in magnitude when compared to the gate and the subthreshold leakage. Since there are currently no good solutions for dealing with the gate tunneling leakage, this thesis will instead focus on the reduction of the subthreshold leakage component only, which is still the most dominant source of leakage current.

1.2 Manufacturing Process Variations

Beside leakage current/power, another challenge facing today's IC designers is manufacturing process variations, which worsens as device geometry shrinks. Manufacturing process variations can cause not only performance degradation, but also functional incorrectness if it is not properly accounted for in the design. For example, suppose the critical delay of a cloud of combinational logic gets altered due to process variations. If this cloud of logic sits in between two flip-flops, then the setup time and/or hold time of the flip-flops could be violated if the variation is large enough. Thus, delay variations can lead to functional incorrectness. Also, leakage power variations is quickly becoming a significant problem as well. In fact, it has been demonstrated by Intel that the leakage power in today's process could vary and increase by anywhere between 10x to 100x, depending on the degree of variations to the device channel length [4]. Clearly, this could lead to unacceptable power consumption and violation of the power budget. Thus, designers today must take careful consideration of the impact of manufacturing process variations. One way to do so is to add timing margins at the expense of performance in order to guarantee correctness (even in the presence of significant process variations) [5].

We now briefly elaborate on the sources of process variations, then conclude this section with a discussion on how traditional gate-sizing methods can worsen the impact of delay variations.

1.2.1 Sources of Process Variations

For integrated circuits, performance variability is mainly impacted by two sets of factors: environmental and physical. Environmental factors include variations in the temperature, power supply voltage, and coupling noise among nets. Physical factors include variations in device parameters, such as width and channel length. Physical factor variations are caused by processing, mask imperfections, and reliability-related degradation. Environmental factors are usually analyzed at the intra-die level, while physical factors are addressed at both inter and intra-die level.

Inter-die variation refers to the variation between different dies on a wafer only. Thus, the environmental/physical factors are assumed to be constant within a die. On the contrary, intra-die variation refers to the case where the factors vary even within a die. For inter-die variation, since the factors are constant within a die, they are independent of the layout of the design. Thus, it is common to treat intra-die variation as variability imposed upon the design [5]. Such variability can then be analyzed using either worst-case techniques, Monte Carlo simulations, or classical statistical analysis. On the other hand, for inter-die variability, analysis of the variation can be difficult due to the large number of varying entities. Nevertheless, there have been several analytical techniques proposed.

1.2.2 Impact of Traditional Gate-Sizing Methods on Delay Variability

One of the most common circuit optimization techniques for helping to achieve timing and power closure is gate-sizing. In addition to sizing up those gates which are on the critical paths to make them faster, traditional gate-sizing methods try to size down those gates which are on the non-critical paths in order to reduce their power consumption. This tuning process is typically applied iteratively until most of the paths become equally critical. Thus, the final path-delay distribution of a circuit tends to exhibit a ”wall-like” shape after gate-sizing [6], as can be seen in Figure 1.3. In older technologies where there was not too much process variations, this did not have any drawbacks at all. However, as process variations become more significant in the future, applying such traditional gate-sizing methodology can actually pose

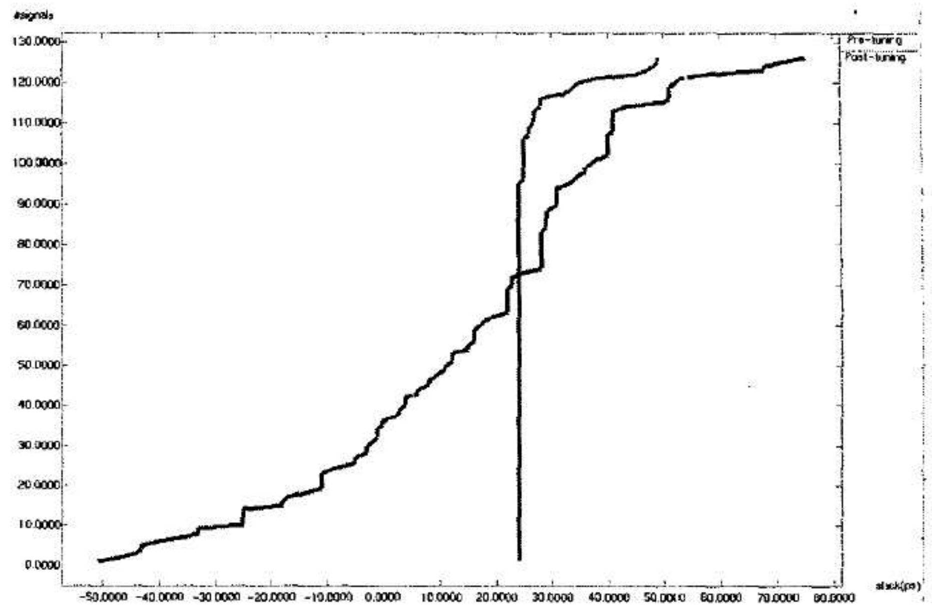


Figure 1.3 Path-delay distribution after traditional gate-sizing

a problem. This is because if a circuit has many critical paths, the probability that its critical delay gets increased post-process variations is high, since only one of the critical paths needs to be perturbed to cause the overall delay to fluctuate. Thus, applying traditional gate-sizing methods can exacerbate the impact of delay variability and potentially hurt the yield (due to resulting functional incorrectness).

Chapter 2

Background and Motivation

In Chapter 1, the looming challenges in future IC designs were introduced. In this chapter, we give a brief background review on some of the previous works done in gate-sizing and leakage current/power control. We also highlight the main flaws in these earlier works in order to justify the motivation behind this thesis work.

2.1 Previous Works on Gate-Sizing

It is well-known that optimization at the circuit-level can play a significant role in helping to achieve timing and power closure. Specifically, since varying a transistor's width can affect its delay, area, dynamic power, and even leakage power, experienced IC designers typically try to fine-tune the size of each transistor in a circuit individually such that the optimal design tradeoff point is achieved. However, given the size complexity of modern designs as well as the degree of freedom in this tuning process, it is not surprising that the cumbersomeness of this task can easily outweigh its benefits, especially if the designer has to adjust each and every transistor uniquely.

Gate-sizing, as opposed to transistor-sizing, is one approach by which the complexity of the tuning task can be alleviated. That is, rather than trying to tune each and every transistor individually, the sizes of the transistors in a gate are kept constant (with pmos-to-nmos ratio enforced) and tuning is done only on a gate-by-gate basis. Another alternative is to tune the drive strength of each gate such that the pre-supplied drive strength cell instances from a standard cell library can be used. The former approach is appropriate for full and semi-custom design methodologies, while the latter is more suited for the ASIC flow.

The importance of circuit tuning has been well-documented in the literature [7–11]. Due to stringent time-to-market requirements as well as the high complexity of the tuning process, most large semiconductor companies today automate the tuning process by have either their own internal tuning CAD tool or some external tool purchased from an EDA vendor. To be successful, these tools must not only be effective, but must also be runtime and memory-efficient. In general, circuit tuning tools fall into two categories: static and dynamic tuning. Static tuning is a method whereby the tuning is carried out irrespective of the input vector pattern. This can be done by using a worst-case cornering approach and considering all paths in the circuit simultaneously. Dynamic tuning, on the other hand, involves the use of transient simulations to aid the tuning process. Consequently, this procedure is heavily input vector-dependent and computationally expensive. On the other hand, by relying on detailed simulations instead of pessimistic worst-case estimates, dynamic tuning can arrive at a much more accurate tuning result than static tuning. Nevertheless, despite trading off accuracy for runtime and memory usage, static tuning is currently the preferred method of the two.

2.1.1 Gate-Sizing Using Mathematical Programming Techniques

It has been shown in past works [7,9–11] that the circuit tuning problem can be expressed as a large-scale nonlinear optimization problem subject to constraints and bounds. The constraints involve maximal arrival time, maximal total power, area, slew rate, etc., while the bounds apply to the lower and upper limits imposed on the tunable transistor width range. The objective is typically either delay, area, or a mixture of the two. Using the Elmore Delay model, both the delay and the area can be expressed as a function of the transistor width. Thus, the tunable parameters in the optimization problem are the transistors' widths as well as their pmos-to-nmos ratios. The problem can then be solved using any standard, numerical mathematical programming technique for nonlinear constrained optimization problems. If formulated properly, the problem will be convex and the solution will be guaranteed to be globally optimal. This is one of the key strengths to mathematical programming-based circuit tuning.

Among the well-known circuit tuning tools based on mathematical programming techniques, JiffyTune [12], EinsTuner [13], and Lagrangian Relaxation (LR)-based tools [7, 11] have all shown great promise in handling large-scale VLSI circuits. JiffyTune utilizes dynamic simulations to achieve high accuracy, while EinsTuner relies on static timing analysis with arrival time pruning to perform fast and efficient static tuning. Both methods use an Augmented Lagrangian function with penalty barriers to solve the optimization problem. LR-based tools, on the other hand, utilize only the regular Lagrangian function. The key to this approach lies in the pruning of the Lagrange multipliers and their subgradient adjustment procedure. This method has been experimentally proven to be fast, effective, and memory efficient.

2.2 Previous Works on Leakage Control

As was mentioned in Chapter 1, the four largest components of leakage current are the reverse-biased P-N junction leakage, the gate-induced drain leakage, the weak inversion subthreshold leakage, and the gate oxide tunneling leakage. Between these four, the reverse-biased P-N junction leakage and the gate-induced drain leakage are typically negligible in magnitude compared to the other two, so we do not discuss them in this work. As for the gate oide tunneling leakage, no good solution are known at present (high-K dielectric is still under research, and is being planned for introduction in 2006). Thus, this section will focus on only the background of subthreshold leakage current control, which still happens to be the largest component of leakage current in today's 90nm technology.

For subthreshold leakage control, many circuit-level and architectural-level solutions are available. Several of these techniques are based on the body effect of the threshold voltage, which is given by the following equation:

$$V_t = V_{to} + \gamma \left[\sqrt{2\Phi_b + |V_{sb}|} + \sqrt{2\Phi_b} \right] \tag{2.1}$$

The important thing to note from Equation 2.1 is that the threshold voltage is a function of the potential difference between the source and the bulk. This is the key to some of the known subthreshold leakage control techniques that have been proposed to-date, which we will discuss now.

2.2.1 V_t Hopping

One technique that has been proposed is V_t-hopping [14], whereby the threshold voltage is adjusted dynamically during runtime to suit the running applications' needs. For example, if an operation requires low-latency, then V_t can be lowered to speed up the performance. Otherwise, V_t can be elevated during high-latency operations and idle periods so that the subthreshold leakage current can be minimized over time. However, V_t can only take on certain discrete levels of voltage, hence the meaning of ”hopping” from one V_t to another.

2.2.2 Reverse Body-Biasing

Another technique for subthreshold leakage control is reverse body-biasing. The idea is that since the threshold voltage is affected by the body effect (Equation 2.1), reverse-biasing the bulk potential (with respect to the source) dynamically during runtime can lead to an elevated V_t, which results in lower subthreshold leakage current. Of course, this comes at the expense of performance, since the transistors will slow down due to having smaller drive currents.

2.2.3 MTCMOS (Sleep Transistors)

Yet another well-known subthreshold leakage control technique is Multi-Threshold CMOS (MTCMOS) [15], or sometimes referred to as sleep transistors. The basic idea is shown in Figure 2.1 [16], where it can be seen that two high-V_t sleep transistors are inserted on the top and bottom of a normal low-V_t block. These are commonly called the header and the footer. During idle or sleep mode, these sleep transistors are gated-off to reduce the Virtual Vdd and increase the Virtual GND. As a result, $|V_{sb}|$ increases for the connecting devices in the low-V_t block, thereby resulting in reduced subthreshold leakage current flow. This technique is one of

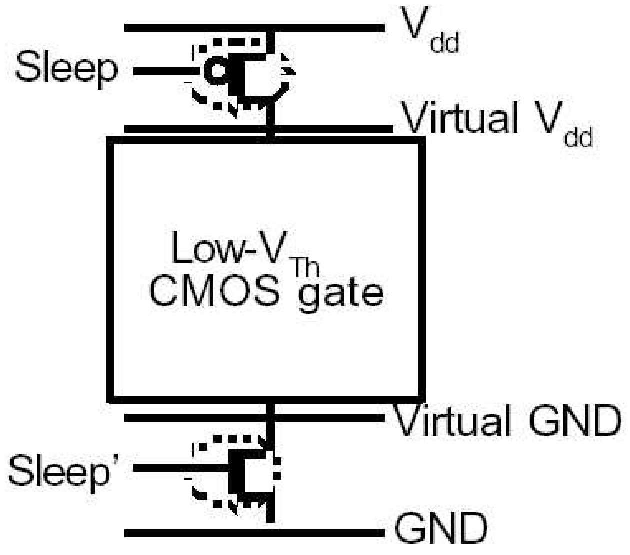


Figure 2.1 MTCMOS (sleep transistors)

the most effective leakage solutions available, but it incurs a heavy penalty in the delay, area, and switching power overhead due to the insertion of the extra sleep transistors.

2.2.4 Input Vector Control (IVC)

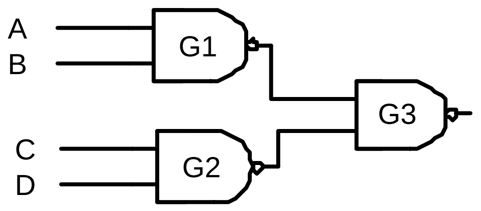
A technique that works similar in principle to MTCMOS but incurs less overhead is Input Vector Control (IVC) [17–20]. By selectively assigning the input pattern during sleep mode, the overall circuits's static leakage power consumption can be lowered during sleep periods. The reason that the subthreshold leakage current is input pattern-dependent comes from the fact that the transistors are naturally stacked in series in CMOS gates. This means that under different input values, different number of transistors in the series will be turned off. The more the off-transistors, the higher the $|V_{sb}|$ of the connecting devices, and thus the higher the V_t. This phenemonon is known as the stacking effect [21], and it has been shown that the stacking effect can be exploited to reduce the subthreshold leakage current by orders of magnitude. Thus, although input vector control is similar in principle to MTCMOS, it does not incur as high of a penalty as MTCMOS because no extra sleep transistors are inserted. The downside to this approach is that the power savings is not as good as MTCMOS because the effectiveness of this technique is strongly dependent on a circuit's logic depth as well as its logic structure arrangement. Another drawback is that the optimal input vector determination problem is NP-hard. Fortunately, many good heuristics have been proposed, such as one based on greedy algorithm with branch-and-bound [19] and one on satisfiability [20].

2.2.5 Dual-Vt Assignment

Perhaps the most well-known leakage control technique today is dual-V_t assignment [22]. In this technique, normal low-V_t devices in the non-critical paths are replaced by their high-V_t equivalent counterparts so that the leakage current can be reduced without impacting the overall critical delay. Many efficient algorithms for finding the replaceable low-V_t devices have been proposed, and this technique is now becoming a standard practice in even ASIC designs. In fact, most standard cell libraries today come with two (or sometimes even more) available sets of cells, one high-V_t and one low-V_t.

2.3 Motivation of this Thesis Work

There are many motivations behind the works proposed in this thesis. First, from the circuit tuning perspective, it is clear that Lagrangian Relaxation-based approaches are more efficient and just as effective as their Augmented Lagrangian counterparts. However, in all previous works on LR-based tuning, the accuracy of the result is hindered by the use of the Elmore delay model in the problem formulation. Furthermore, these earlier works employed a Guass-Siedel-like greedy and local optimization flow in the Lagrangian subproblem solving step, which again hurts the result's accuracy as well as the runtime. Last and most significantly, these previous works do not consider the impact of gate-sizing on leakage current/power and delay variability. Since the leakage current is a direct function of the transistor width and the delay variability is worst when there exists many critical paths in a circuit, applying these previously-proposed LR sizing methods would significantly degrade the design in today's technology. Also, as multiple



In	Leakage
00	93
01	507
10	378
11	1050

Figure 2.2 Example circuit demonstrating the inefficiency of Input Vector Control. Table on the right corresponds to the leakage current for a NAND2 under different input combinations

V_t usage starts to become mainstream, it would be beneficial to consider the optimization of transistor sizes and V_ts simultaneously as opposed to orthogonally.

From the leakage reduction mechanism perspective, it is clear that Input Vector Control is highly promising due to its low overhead cost. However, although IVC has shown some success in leakage control, nearly all of the findings to-date have been performed on small benchmarks, which are not really representative of the type of complexity and size one would see in modern circuits. This is actually a serious flaw because intuitively, one would expect the effectiveness of input assignment to wane with increasing circuit size. For example, consider the circuit shown in Fig. 2.2. The table on the right hand side show what each NAND2's leakage current value would be under different input patterns. As it can be seen, out of all 16 possible input combinations, the minimal-leakage input vector for this circuit is ABCD=$\{0000\}$. However, even with this vector, gate G3 still happens to be poorly controlled. This is unavoidable due to the way the gates are arranged, or the logical structure of the circuit. The larger the circuit, the higher the probability that interferences like this occurs. Therefore, the degree of effectiveness one can achieve with standard IVC will be ultimately bounded by how large a circuit is and by how favorably its gates are arranged. This is clearly a strong drawback to what would otherwise be a great leakage control technique.

Chapter 3

LARTTE: A Delay Variation-Aware Generalized Lagrangian Relaxation Tuning Tool for Fast and Effective Gate-Sizing and Multiple-V_t Assignment

In this chapter, we propose a novel method to perform efficient gate-sizing and multiple V_t assignment using LR and posynomial modeling. Our algorithm optimizes a circuit's delay and power consumption subject to slew rate and transistor width constraints, and can readily take manufacturing-induced delay variations into account. We first use SPICE to generate accurate delay and power models in posynomial form [23] for standard cells, then formulate a large-scale, convex optimization problem based on these models. Finally, we perform LR to solve for the globally-optimal (with respect to the posynomial-based optimization problem, without discretization) set of transistor sizes and V_ts (with discretization) for each gate. The key contribution of this work is that our LR-based gate-sizing method is delay variation-tolerant, can perform tuning in a ”generalized” or non-Gauss-Seidel manner, and can simultaneously and optimally carry out the V_t assignment procedure for leakage reduction. Experimental results show that our implemented tuning tool, LARTTE, exhibits linear runtime and memory usage requirement, can effectively tune a circuit with over 15,000 variables and 8,000 constraints in under 7 minutes, and can minimize the impact of delay variation by introducing a timing margin between the worst output arrival time and all other outputs' arrival times. Our experiments also show that LARTTE compares favorably with SNOPT [24], a state-of-the-art general-purpose optimization problem solver. LARTTE is over 250x faster than SNOPT, but can achieve the same quality of results.

This chapter is organized as follows. Background and posynomial modeling information are detailed in Section 3.1, followed by the main LARTTE algorithm description in Section 3.2. Modifications to LARTTE to guard against delay variations is detailed in Section 3.3. Experimental results and concluding remarks follow in Section 3.4 and Section 3.5. Lastly, in Section 3.6, we give a modified LARTTE algorithm that can be used in an ASIC flow.

3.1 Preliminaries and Posynomial Modeling

In this section, we first define the notations that we will be using throughout this chapter. Then, we provide some background information on posynomial functions and optimization problems in general. Finally, we describe the method that we used to accurately characterize the various attributes of a gate (ie. delay, dynamic power, leakage power, input slew, etc.) as posynomial functions.

3.1.1 Notations

The following notations are used throughout this paper. Given a combinational circuit, we first introduce two auxiliary nodes, a sink and a source (see Figure 3.1). The sink has all of its fan-ins from the primary outputs, and the source has all of its fan-outs to the primary inputs. The nodes in the circuit are labeled in reverse topological order, with the sink having the index of 0 and the source having the index of N (assume N total nodes). Let $input(i)$ and $output(i)$ be the set of node indices that connect directly to the input(s) and output(s) of node i. Define $\mathcal{D}$ and $\mathcal{G}$ to be the set of primary inputs and internal gate components in the circuit, respectively. For $i \in \mathcal{G}$, a_i is the arrival time at the output of gate i, W_{g_i} is the width of the NMOS and PMOS (adjusted by a γ ratio), V_{tn_i} and V_{tp_i} are the NMOS and PMOS threshold voltages, C_{L_i} is the loading capacitance (expressed as a function of the widths of the loading gates), and s_i is the slew rate of gate i. For simplicity of presentation, a_i and s_i are assumed to be the same for both the rising and the falling transition. Let T_i, D_i, $P_{dynamic_i}$, and $P_{leakage_i}$ denote the input slew rate, propagation delay, dynamic power, and leakage power posynomial functions of gate i, respectively. Lastly, define L_{w_i} and U_{w_i} to be the lower and upper bound of W_{g_i}, L_{tn_i} and

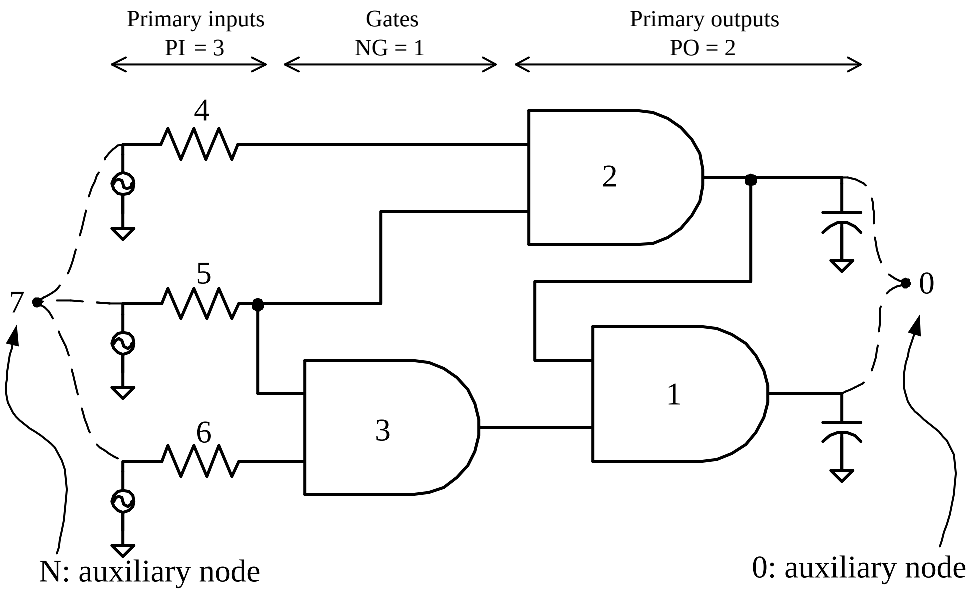


Figure 3.1 Notations used in this work

U_{tn_i} to be the lower and upper bound of V_{tn_i}, and L_{tp_i} and U_{tp_i} to be the lower and upper bound of V_{tp_i}.

3.1.2 Background: Nonlinear Optimization Problems and Posynomial Functions

In general, nonlinear constrained optimization problems [25] have the form:

$$\begin{aligned} \text{minimize} \quad & f_0(x) \\ \text{subject to} \quad & g_i(x) \le 0,\ i = 1, \ldots, m \\ & h_i(x) = 0,\ i = 1, \ldots, n \end{aligned} \tag{3.1}$$

where $x \in \Re^n$ is a n-vector of optimization variables and f_0, g_i, and h_i are the objective function, inequality constraints, and equality constraints, respectively. If f_0, g_i, and h_i are all convex functions, then the problem becomes a convex optimization problem. An important property of the convex optimization problem is that any locally-optimal solution is also guaranteed to be globally-optimal.

A posynomial function has the form:

$$f(x) = \sum_{j=1}^{k} c_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \tag{3.2}$$

where f is a real-valued function whose domain $x \in \Re^n$ is non-negative, $c_j \geq 0$, and $\alpha_{ij} \in \Re$. A posynomial is a sum of monomials. It is well-known [23] that under a simple exponential transformation, a posynomial function can be converted into a convex function. Hence, if an optimization problem is expressed in terms of posynomial functions, then a global minimum can be easily found by searching for a local minimum, which can be done with any formal mathematical programming technique [25]. Thus, this is the main reason to use posynomials to model gate characteristics.

3.1.3 The Posynomial Modeling Procedure

The posynomial modeling procedure is essentially done via least-square regression analysis on SPICE simulation data. Formally, we define the posynomial parametric regression problem as follows:

$$\begin{aligned} \textbf{Posyfit: } \text{minimize} \quad & \sum_{m=1}^{z} \left(\left(\sum_{j=1}^{k} c_j \prod_{i=1}^{n} x_i^{\alpha_{ij}} \right) - b_m \right)^2 \\ \text{subject to} \quad & c_j \geq 0,\ 1 \leq j \leq k \end{aligned} \tag{3.3}$$

where $x \in \Re^{m \times n}$ corresponds to m different sets of a n-vector of tunable parameter values, $b \in \Re^m$ is a vector of m different SPICE-simulated scalar results (each corresponding to one unique simulation run under the associated tunable parameter values in x), and k, $c \in \Re^k$, and $\alpha \in \Re^{k \times n}$ are the unknown parameters that we are trying to determine. The value of m is user-defined and corresponds to the number of SPICE simulations that will be run to generate the necessary b$_i$ values for posynomial-fitting. In general, a higher m leads to a greater accuracy in the final characterized posynomial, but in turn requires a longer pre-processing time (as SPICE simulations are inherently time-consuming). The value of n is the number of tunable parameters which affect the metric being approximated. For example, if the delay posynomial form is being determined, then n equals 5, for the delay of a gate depends on W$_g$, C$_L$, V$_{tn}$, V$_{tp}$, and s (slew rate). Note that W$_g$ here denotes the size of NMOS only. A pre-fixed γ ratio is used to automatically adjust the PMOS size during tuning.

The posynomial-fitting procedure works as follows. First, we select m different sets of n-tunable parameter values and simulate each individually to find m different b$_i$ values. Then, after plugging these terms back into equation 3.3, we are left with 3 unknowns, k, c, and α. To solve for these, we first guess a value for the vector α and its dimension k. Then, using α and k, we solve for the last remaining unknown, c, using CFSQP [26], a general-purpose unconstrained problem solver. If the resulting least-square value using the solved c is below an error tolerance level, we stop and return the characterized posynomial form (the inner summation term). Otherwise, we repeat the fitting-procedure for a different guess of α and k, and continue to do so until the least-square error is minimized.

To avoid excessive trial count in guessing the posynomial form, we employ the following heuristic when trying to find the right k, c, and α. First, we guess a dominant monomial term by exploiting well-known dependence relationships. For example, for the delay posynomial, we start with a term that has W_g^{-1} and C_L^1, since we know in general that the delay of a gate depends on its loading capacitance and its drive strength. Then, based on the resulting fitting error using this guess, we gradually adjust the power coefficients appropriately and add more monomial terms into the posynomial equation until we find a reasonably accurate approximation.

We give the following example to more clearly illustrate the posynomial-fitting procedure. Suppose that we are trying to determine the delay posynomial of a particular gate, say a CMOS inverter. Then, let m=2 and pick the following two sets of tunable parameter values: $\{$W$_g$=3, V$_{tn}$=0.7, V$_{tp}$=0.7, C$_L$=5, s=0.5$\}$ and $\{$W$_g$=4, V$_{tn}$=0.9, V$_{tp}$=0.8, C$_L$=2, s=0.7$\}$. Next, we simulate in SPICE the delay of an inverter under these two sets of parameters, and call the results b$_1$ and b$_2$. Assume for this example that b$_1$=15 and b$_2$=10. Given these data, the Posyfit problem is reduced to the following:

$$\begin{aligned}
\text{minimize} \quad & \left(\left(\sum_{j=1}^{k} c_j 3^{\alpha_{1j}} 0.7^{\alpha_{2j}} 0.7^{\alpha_{3j}} 5^{\alpha_{4j}} 0.5^{\alpha_{5j}}\right) - 15\right)^2 + \\
& \left(\left(\sum_{j=1}^{k} c_j 4^{\alpha_{1j}} 0.9^{\alpha_{2j}} 0.8^{\alpha_{3j}} 2^{\alpha_{4j}} 0.7^{\alpha_{5j}}\right) - 10\right)^2 + \\
\text{subject to} \quad & c_j \geq 0 \qquad (3.4)
\end{aligned}$$

It should be noted that the k, c, and α values are required to be the same across all m copies of the inner summation term, since we are trying to determine a posynomial model that would be accurate for any set of parameter values. With the reduced Posyfit problem, we can then carry out the iterative fitting procedure to find the unknown parameters (k, c, and α), and thus the delay posynomial function, for the inverter. The returned posynomial is expressed as a function of W$_g$, V$_{tn}$, V$_{tp}$, C$_L$, and s. For illustration purpose, the following is the actual inverter delay posynomial form found in this work:

$$\begin{aligned}
D_{inv}(W_g, V_{tn}, V_{tp}, C_L, s) = & 0.39 V_{tn} V_{tp}^{-1} + 2.14 W_g^{-1} C_L V_{tp} + 623 V_{tp}^{0.5} W_g^{0.5} + \\
& 12.2 V_{tn}^{3} + 29 W_g^{0.5} V_{tn}^{-1} V_{tp}^{0.5} + 0.14 s^{0.5} + \\
& 1.07 W_g^{-1} C_L V_{tn}^{2} V_{tp}^{-1}. \qquad (3.5)
\end{aligned}$$

In this work, we set the stopping criteria of the fitting procedure to be when 90% of the fitting samples, using the guessed posynomial form, agree numerically to within $\pm$ 10% of their corresponding SPICE results. Also, when generating the SPICE values, we assumed the worst case conditions (ie. for delay simulations, the input signal to the last transistor in the stack is set to arrive last, etc). Table 3.1 shows the model-fitting error mean and standard deviation for the characterized gates. Prefixes Inv, Na, and No in the table represent inverter, NAND, and NOR gates, and suffixes TP, PL, and PD represent delay, leakage, and dynamic power respectively. The unit for the entries in the table is the % difference (in either direction) between the samples' values using the final posynomial form and their corresponding SPICE results. For example, the leakage power posynomial of an inverter (InvPL) has a mean fitting

Table 3.1 Model-fitting error mean and standard deviation

Gate	Mn.	Dev.	Gate	Mn.	Dev.	Gate	Mn.	Dev.
InvPD	-0.1	3.5	Na6TP	-0.2	4.7	No4PL	-2.7	6.0
InvPL	-2.6	5.6	Na7PD	-0.2	4.4	No4TP	-0.1	3.2
InvTP	-0.1	2.5	Na7PL	-0.1	1.8	No5PD	-0.1	2.1
Na2PD	-1.2	6.5	Na7TP	-0.2	4.8	No5PL	-0.0	1.8
Na2PL	-0.0	1.6	Na8PD	-0.2	4.5	No5TP	-0.2	4.7
Na2TP	-0.1	3.4	Na8PL	-0.0	1.8	No6PD	-0.1	2.2
Na3PD	-0.4	6.7	Na8TP	-0.3	4.9	No6PL	-2.7	6.1
Na3PL	-0.0	1.8	Na9PD	-0.2	4.9	No6TP	-0.1	3.2
Na3TP	-0.2	4.2	Na9PL	-0.0	1.9	No7PD	-0.1	2.3
Na4PD	-0.3	5.6	Na9TP	-0.3	5.1	No7PL	-2.8	6.5
Na4PL	-0.0	1.8	No2PD	-0.8	6.6	No7TP	-0.1	3.0
Na4TP	-0.2	4.5	No2PL	-2.5	5.4	No8PD	-0.1	2.4
Na5PD	-0.2	4.9	No2TP	-0.1	3.2	No8PL	-2.8	5.6
Na5PL	-0.0	1.8	No3PD	-0.7	6.3	No8TP	-0.1	3.0
Na5TP	-0.2	4.7	No3PL	-2.6	5.6	No9PD	-0.1	2.6
Na6PD	-0.2	4.5	No3TP	-0.1	2.9	No9PL	-2.8	5.5
Na6PL	-0.0	1.8	No4PD	-0.2	4.5	No9TP	-0.1	3.1

error of 2.6%, and a standard deviation of 5.6%. For illustration purpose, the fitting error distribution for a NAND6 is also given in Figure 3.2. The unit for the x-axis in these figures is again the % difference.

3.2 Posynomial-Based Lagrangian Relaxation

In this section, we derive a generalized Lagrangian Relaxation tuning algorithm which incorporates the use of posynomial delay and power models. The section is organized as follows.

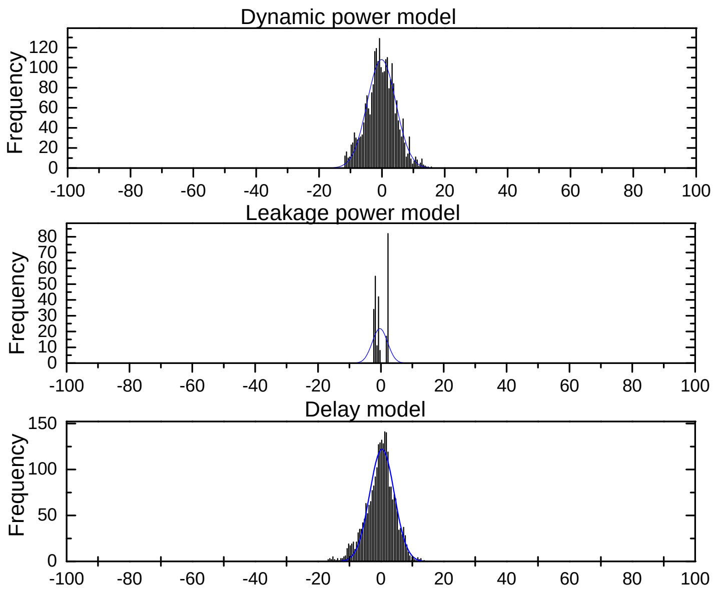


Figure 3.2 Model-fitting error distribution for a NAND6 gate.

In 3.2.1, we formally formulate the circuit tuning optimization problem, or the Primal Problem ($\mathcal{PP}$). 3.2.2 introduces the Lagrangian Subproblem, $\mathcal{LRS}/\lambda$. 3.2.3 states the first-order KKT condition which will be used in our algorithm to significantly speed up the tuning process. 3.2.4 outlines the Lagrange Multiplier adjustment scheme used in this work, while 3.2.5 describes the method by which $\mathcal{LRS}/\lambda$ can be solved optimally, efficiently, and accurately in a generalized manner. This will in turn solve our original problem ($\mathcal{PP}$) as well. Finally, in 3.2.6, we discuss the necessary post-tuning V_t discretization heuristic as well as give a summary of LARTTE.

3.2.1 Primal Problem Formulation

In general, the problem of minimizing the maximum delay and total power consumption (dynamic + leakage) subject to arrival time/width/slew constraints can be formulated as a large-scale, nonlinear programming problem. We call the following the Primal Problem ($\mathcal{PP}$):

$$
\begin{aligned}
\mathcal{PP}: \quad & \text{minimize} \quad && \alpha_1 a_0 + \alpha_2 P_{leakage}(Wg, Vtn, Vtp, s) \\
& && + \alpha_3 P_{dynamic}(Wg, C_L, Vtn, Vtp, s) \\
& \text{subject to} \quad && a_j \leq a_0, \ j \in input(0) \\
& && a_j + D_i \leq a_i, \ i \in \mathcal{G} \cap \forall j \in input(i) \\
& && D_i \leq a_i, \ i \in \mathcal{D} \\
& && T_i \leq s_i, \ i \in (\mathcal{D} \cup \mathcal{G}) \\
& && L_{w_i} \leq W_{g_i} \leq U_{w_i}, \ i \in \mathcal{G} \\
& && L_{tn_i} \leq V_{tn_i} \leq U_{tn_i}, \ i \in \mathcal{G} \\
& && L_{tp_i} \leq V_{tp_i} \leq U_{tp_i}, \ i \in \mathcal{G}
\end{aligned} \tag{3.6}
$$

where α_1, α_2 and α_3 are the normalized weighting factors to the maximum delay of the circuit, a_0 (arrival time of the artificial sink node), total leakage power, $P_{leakage}$, and total dynamic power, $P_{dynamic}$ (For simplicity of presentation, the activity factor is not shown in the parameter list of the $P_{dynamic}$ term because it is not a tunable parameter). The sum of α_1, α_2, and α_3 is 1. The weighting factors are user-assigned based on the operating condition of the target application, such as how much time spent in idle mode, how critical is the timing of the design, etc. (Alternatively, a more sophisticated α assignment scheme could be applied, ie. iteratively invoking LARTTE and adjusting the α factors along the way based on the previous iteration's tuned results). For all other notations, see Section 3.1.1.

From simple rearrangement, equation 3.6 can be transformed into the following:

$$\begin{aligned}
\text{minimize} \quad & \alpha_1 a_0 + \alpha_2 P_{leakage}(Wg, Vtn, Vtp, s) \\
& + \alpha_3 P_{dynamic}(Wg, C_L, Vtn, Vtp, s) \\
\text{subject to} \quad & \frac{a_j}{a_0} \leq 1,\ j \in input(0) \\
& \frac{a_j + D_i}{a_i} \leq 1,\ i \in \mathcal{G} \cap \forall j \in input(i) \\
& \frac{D_i}{a_i} \leq 1,\ i \in \mathcal{D} \\
& \frac{T_i}{s_i} \leq 1,\ i \in (\mathcal{D} \cup \mathcal{G}) \\
& L_{w_i} W_{g_i}^{-1} \leq 1,\ W_{g_i} U_{w_i}^{-1} \leq 1,\ i \in \mathcal{G} \\
& L_{tn_i} V_{tn_i}^{-1} \leq 1,\ V_{tn_i} U_{tn_i}^{-1} \leq 1,\ i \in \mathcal{G} \\
& L_{tp_i} V_{tp_i}^{-1} \leq 1,\ V_{tp_i} U_{tp_i}^{-1} \leq 1,\ i \in \mathcal{G}
\end{aligned} \tag{3.7}$$

In general, $\mathcal{PP}$ is not in the form of a convex optimization problem. However, posynomials can be readily transformed into convex form by the following simple exponential transformation of the variables [23]: Let x represent the vector of all tunable parameters, and transform each entry x_i in x to a new variable y_i, where x_i = e^{y_i}. Now, y represents the vector of tunable parameters, and it is substituted into the $\mathcal{PP}$ equation for x to form a convex optimization problem. Applying LR to the transformed $\mathcal{PP}$ will give us an optimal solution in terms of y, but we can easily recover the desired x_is by exponentiating the y_is.

3.2.2 Lagrangian Relaxation with Logarithmic Transformations

From PP, after making the necessary exponential variable transformations (to both the posynomials as well as the arrival time terms), the next step is to make a Logarithmic transformation on the non-simple constraints by taking the natural log of both sides (We perform the Logarithmic transformation because empirically, we found that it resulted in exceptional runtime improvement). Since the logarithmic function is monotonically increasing, this can be done without affecting the final result. The newly transformed problem is the following:

$$\begin{aligned}
\text{minimize} \quad & \alpha_1 e^{a_0^*} + \alpha_2 P^*_{leakage}(Wg, Vtn, Vtp, s) \\
& + \alpha_3 P^*_{dynamic}(Wg, C_L, Vtn, Vtp, s) \\
\text{subject to} \quad & \ln\left(\frac{e^{a_j^*}}{e^{a_0^*}}\right) \leq 0,\ j \in input(0) \\
& \ln\left(\frac{e^{a_j^*} + D_i^*}{e^{a_i^*}}\right) \leq 0,\ i \in \mathcal{G} \cap \forall j \in input(i) \\
& \ln\left(\frac{D_i^*}{e^{a_i^*}}\right) \leq 0,\ i \in \mathcal{D} \\
& \ln\left(\frac{T_i^*}{e^{s_i^*}}\right) \leq 0,\ i \in (\mathcal{D} \cup \mathcal{G}) \\
& L_{w_i} W_{g_i}^{-1} \leq 1,\ W_{g_i} U_{w_i}^{-1} \leq 1,\ i \in \mathcal{G} \\
& L_{tn_i} V_{tn_i}^{-1} \leq 1,\ V_{tn_i} U_{tn_i}^{-1} \leq 1,\ i \in \mathcal{G} \\
& L_{tp_i} V_{tp_i}^{-1} \leq 1,\ V_{tp_i} U_{tp_i}^{-1} \leq 1,\ i \in \mathcal{G}
\end{aligned} \tag{3.8}$$

where parameters with a $*$ superscript represent those after an exponential change of variable.

From equation 3.8, we can form the general Lagrangian function [25] by introducing non-negative Lagrange multipliers to relax each arrival time and slew constraint into the objective function. Simple bounds on the transistor widths and V_ts are not relaxed. For example, for $j \in input(0)$, let λ^A_{j0} denote the multiplier for the constraint $\ln(\frac{e^{a_j^*}}{e^{a_0^*}}) \leq 0$. For $i \in \mathcal{G} \cap \forall j \in input(i)$, let λ^A_{ji} denote the multipliers for the constraint $\ln(\frac{e^{a_j^*}+D_i^*}{e^{a_i^*}}) \leq 0$, and for $i \in (\mathcal{D} \cup \mathcal{G}) \cap \forall j \in input(i)$, let λ^S_{ji} denote the multipliers for the constraint $\ln(\frac{T_i^*}{e^{s_i^*}}) \leq 0$. For $i \in \mathcal{D}$, let λ^A_{mi} denote the multipliers for the constraint $\ln(\frac{D_i^*}{e^{a_i^*}}) \leq 0$. Finally, let λ be the vector of all the multipliers introduced. Then, the general Lagrangian function can be written as:

$$\begin{aligned}\mathcal{L}(Wg, Vtn, Vtp, a, s, \lambda) &= \alpha_1 e^{a_0^*} + \alpha_2 P^*_{leakage}(Wg, Vtn, Vtp, s) \\ &+ \alpha_3 P^*_{dynamic}(Wg, C_L, Vtn, Vtp, s) \\ &+ \sum_{j \in input(0)} \lambda^A_{j0} \ln\left(\frac{e^{a_j^*}}{e^{a_0^*}}\right) \\ &+ \sum_{i \in \mathcal{G}} \sum_{j \in input(i)} \lambda^A_{ji} \ln\left(\frac{e^{a_j^*} + D_i^*}{e^{a_i^*}}\right) \\ &+ \sum_{i \in (D \cup G)} \sum_{j \in input(i)} \lambda^S_{ji} \ln\left(\frac{T_i^*}{e^{s_i}}\right) \\ &+ \sum_{i \in \mathcal{D}} \lambda^A_{mi} \ln\left(\frac{D_i^*}{e^{a_i^*}}\right) \end{aligned} \tag{3.9}$$

The Lagrangian relaxation subproblem associated with a particular fixed Lagrange multiplier value λ ($\mathcal{LRS}/\lambda$) is:

$$\begin{aligned} \mathcal{LRS}/\lambda : \qquad & \text{minimize} \quad \mathcal{L}_\lambda(Wg, Vtn, Vtp, a, s) \\ & \text{subject to} \quad L_{w_i} W_{g_i}^{-1} \le 1,\ W_{g_i} U_{w_i}^{-1} \le 1,\ i \in \mathcal{G} \\ & \qquad L_{tn_i} V_{tn_i}^{-1} \le 1,\ V_{tn_i} U_{tn_i}^{-1} \le 1,\ i \in \mathcal{G} \\ & \qquad L_{tp_i} V_{tp_i}^{-1} \le 1,\ V_{tp_i} U_{tp_i}^{-1} \le 1,\ i \in \mathcal{G} \end{aligned} \tag{3.10}$$

From the theory of the Lagrangian function, it is known that there exists a vector value of λ for which the optimal solution of $\mathcal{LRS}/\lambda$ is equal to the optimal solution of the original $\mathcal{PP}$ problem. Hence, if we can find this λ value, then we can find the optimal solution to the original problem (by solving $\mathcal{LRS}/\lambda$).

Before we discuss our strategy for finding the correct λ value, we first present a key part of our algorithm which is largely responsible for the excellent runtime of LARTTE.

3.2.3 First-Order KKT Necessary Condition For The Lagrangian Function Solution

For a given Lagrangian function that we are interested in solving, the theory of the Lagrangian tells us that for a particular vector value λ to be the correct, optimal solution multiplier, the first-order Kuhn-Karush-Tucker (KKT) necessary condition must hold. Under the first-order KKT condition, the gradient of the Lagrangian function with respect to all variable parameters must be equal to 0. That is, $\nabla_{W^*_{g_i}}\mathcal{L}_\lambda = 0$, $\nabla_{V^*_{tn_i}}\mathcal{L}_\lambda = 0$, and $\nabla_{V^*_{tp_i}}\mathcal{L}_\lambda = 0$ for $1 \le i \le$ NG+PO. Also, $\nabla_{a^*_i}\mathcal{L}_\lambda = 0$ and $\nabla_{s^*_i}\mathcal{L}_\lambda = 0$ for $1 \le i \le$ PI+NG+PO. Therefore, in trying to find out what the correct, optimal multiplier value λ should be, we need only consider cases where the above conditions are satisfied. This ”filtering” process is the key to dramatic runtime reduction.

By taking $\nabla_{a^*_i}\mathcal{L}_\lambda = 0$ and $\nabla_{s^*_i}\mathcal{L}_\lambda = 0$ to the Lagrangian, we obtain the following required optimality condition on the arrival time and slew constraint multipliers:

$$\begin{aligned}
&\sum_{j\in input(0)} \lambda^A_{j0} = \alpha_1 e^{a^*_0} \\
&\sum_{j\in input(i)} \lambda^A_{ji} = \sum_{k\neq 0\in output(i)} \frac{\lambda^A_{ik}\cdot e^{a^*_i}}{e^{a^*_i}+D^*_k},\ i\in(\mathcal{D}\cup\mathcal{G}) \\
&\sum_{j\in input(i)} \lambda^S_{ji} = \sum_{k\neq 0\in output(i)} \left(\frac{\lambda^A_{ik}}{e^{a^*_i}+D^*_k}\frac{\partial D^*_k}{\partial s^*_i} + \frac{\lambda^S_{ik}}{T^*_k}\frac{\partial T^*_k}{\partial s^*_i}\right) \\
&\qquad + \alpha_2\frac{\partial P^*_{leakage}}{\partial s^*_i} + \alpha_3\frac{\partial P^*_{dynamic}}{\partial s^*_i},\ i\in(\mathcal{D}\cup\mathcal{G})
\end{aligned} \tag{3.11}$$

Note that each line in 3.11 applies to an individual set of components of λ and is independent to the other lines. For example, if a particular vector value λ^* is to be deemed a candidate for the correct, optimal multiplier λ, then all of its outgoing primary output (PO) multiplier components must sum up to be $\alpha_1 e^{a^*_0}$. Furthermore, for all gates in $\mathcal{D}\cup\mathcal{G}$, all of their incoming multipliers (from fan-in gates) must sum up to their outgoing multipliers (multiplied by $\frac{e^{a^*_i}}{e^{a^*_i}+D^*_k}$). In considering only those values of λ^* which satisfy equation 3.11 as solution candidates for the correct, optimal multiplier λ, our tuning process can significantly cut down on runtime by avoiding unnecessary computation involving impossible λ candidates.

Using equation 3.11, we now present our method for solving for the correct, optimal λ value (and consequently the optimal solution of our original problem as well).

3.2.4 Iterative Multiplier Adjustment for Determining Optimal λ

We employ an iterative, modified sub-gradient method for finding the desired λ vector. First, we arbitrarily pick a starting lambda value which satisfies equation 3.11. For example, we can start by assigning each of the λ_{j0}^{A} to be $\frac{\alpha_1 e^{a_0^*}}{N}$, where N is the number of inputs to sink node 0. The assignment of all other multiplier components can be done in a similar manner (in reverse topological order). After forming an initial λ^* guess, we then iteratively update λ^* using a modified sub-gradient approach shown in Table 3.2, line 3, to form a new guess at every iteration. θ_k is a step size value which is initialized to 1 and gradually updated over iterations using a Trust-Region approach [27]. We continue to iterate and make new guesses for the correct, optimal value of λ until our $\mathcal{LRS}/\lambda^*$ value converges to that of the PP value. When this occurs, we will have found our desired multiplier λ, which is just equal to the λ^* at the stopped iteration.

3.2.5 Solving $\mathcal{LRS}/\lambda$ in a Generalized Manner

Our LARTTE algorithm terminates when the solution of $\mathcal{LRS}/\lambda$ converges to that of $\mathcal{PP}$. In order to do this, we must have a method for solving $\mathcal{LRS}/\lambda$ for the optimal (with respect to the given λ) tunable parameter vector, x. In previous works [7] [11], due to the use of the Elmore delay model, this procedure had to be carried out in a Gauss-Seidel-like or serialized manner. The reason is as follows. In the Elmore model, the delay at a node is characterized as a function of the resistance and capacitance values along the path, not as a function of the tunable parameters in the circuit (specifically, the widths of the gates). Therefore, the process of solving for the tunable vector x in $\mathcal{LRS}/\lambda$ cannot be done in one single iteration (in a generalized manner). This is why in [7] and [11], the authors resorted to a greedy, serialized approach, where the tunable parameters (x_is) are solved in topological order and one-at-a-time so that after solving for one x_i, its corresponding downstream resistance and capacitance values can be updated appropriately for the next gate's x_i to be solved correctly. The order-dependent and serial nature of this solving-procedure makes the tuning process inaccurate and time-consuming, and this is the key drawback to these previous works.

In this work, we overcome the above drawback by using posynomial-based models in the LR framework for the first time. Since the posynomials are characterized with respect to only the tunable parameters of the circuit, $\mathcal{LRS}/\lambda$ is expressed completely in terms of x, and the entire problem can be solved optimally in a generalized manner (in one single iteration) using any formal mathematical programming technique. Thus, the serialization/order restriction is removed entirely in our posynomial-based method, leading to a much more accurate and faster tuning process. The accuracy, efficiency, and elegance of our generalized solving-procedure is the key to LARTTE's performance, and is the main contribution of this work.

We resort to an off-the-shelf solver in L-BFGS-B [28] to solve $\mathcal{LRS}/\lambda$. L-BFGS-B implements the well-known, Limited-Memory BFGS method [25], which has been proven to be exceptional for handling large-scale, unconstrained problems. This method belongs to the class of quasi-Newton methods, which uses a Hessian approximation of the objective function (instead of the exact Hessian) to compute the Newton search direction for the minimum. However, unlike the standard BFGS method, the Limited-Memory approach uses only the curvature information from the most recent iterations to construct the Hessian approximation. This is beneficial for large problems whose Hessian matrices cannot be computed at a reasonable cost or are too dense to be manipulated easily. To avoid any confusion, we leave out the internal details of this method and refer interested readers to [25].

3.2.6 Vt Discretization and LARTTE Summary

Up to now, we have treated V_t as a tunable parameter in $\Re$. This was done because LR is a technique for optimizing continuously-differentiable problems. Obviously, this is a problem because in practice, there is usually only a limited number of V_t levels available for use. Hence, in order to rectify this situation, we must discretize our V_t solutions in the end to the nearest allowable V_t value. For example, if we find that after tuning, one of our transistors has an optimal V_t solution value of 0.176V, but we can only choose between a device with 0.24V V_t and a device with 0.16V V_t, then we would discretize this transistor's V_t solution to be 0.16V instead. This discretization step is carried out at the end of the tuning process for all transistors and their corresponding V_t solutions.

Since the discretization step is a heuristic, the quality/optimality of the solution after applying discretization seems questionable at first. However, we have empirically found that as long as the number of V_t levels available for use is around 4 or more, the solution after discretization will typically be not too far off from the original, un-discretized solution (as it will be shown in our experimental results section). Hence, our LR technique is still reasonable under mild assumptions.

A summary of LARTTE is given in Table 3.2.

3.3 LARTTE with Process Variation Guard

Recall that if process variation were not taken into account during the tuning process, then a ”good” tuning tool will size in such a way that many of the outputs end up having the same critical arrival time in the end. This in turn creates a high probability that the final delay value (subject to process variation) will differ from that calculated via static timing analysis, since any of the critical output's arrival time can increase from variation. Therefore, due to inaccurate timing estimates, functional incorrectness can result (ie. from setup and hold time violations). To improve the chance of high yield, delay variations can be taken into account during the LARTTE tuning process as follows: First, LARTTE is invoked normally and the critical-path delay and its corresponding output pin is recorded. Then, using this recorded information, the same critical output's arrival time constraint is modified/relaxed to derive a new $\mathcal{PP}$ formulation, which is then subsequently solved by LARTTE again. Thus, we have

```
**ALGORITHM** LARTTE:
**Output**: optimal gate-sizing and $V_t$ allocation solution
1. $k := 1$ /* iteration number */
   $\lambda$ := arbitrary initial vector of constraint multipliers satisfying (3.11)
   Initialize all optimization tunable parameters
2. Solve $\mathcal{LRS}/\lambda$ by calling L-BFGS-B to minimize $\mathcal{L}_{\lambda}(Wg, Vtn, Vtp, a, s, \lambda)$
   until optimal solution found and then compute $a_1, \ldots, a_{\text{PI+NG+PO}}$ and
   $s_1, \ldots, s_{\text{PI+NG+PO}}$
3. /* Adjust multipliers $\lambda$ */
   for $i := 0$ to PI+NG+PO do
       foreach $j \in input(i)$ do
           $\lambda_{ji}^{NEW} := \begin{cases} \lambda_{ji}^{A} * \left(\frac{e^{a_j^*}}{e^{a_0^*}}\right)^{\theta_k} & \text{if } i = 0 \\ \lambda_{ji}^{A} * \left(\frac{e^{a_j^*}+D_i^*}{e^{a_i^*}}\right)^{\theta_k} & \text{if } i \in \mathcal{G} \\ \lambda_{ji}^{A} * \left(\frac{D_i^*}{e^{a_i^*}}\right)^{\theta_k} & \text{if } i \in \mathcal{D} \\ \lambda_{ji}^{S} * \left(\frac{T_i^*}{e^{s_i^*}}\right)^{\theta_k} & \text{if } i \in (\mathcal{D} \cup \mathcal{G}) \end{cases}$
           Project $\lambda_{ji}^{NEW}$ to the nearest point satisfying (3.11)
4. $k := k + 1$
5. Goto step 2 until the cost functions of $\mathcal{PP}$ and $\mathcal{LRS}/\lambda$ converge to within
   a specified tolerance
6. Discretize the $V_t$ solutions
7. Solve $\mathcal{LRS}/\lambda$ by calling L-BFGS-B to find the optimal solution
```

Table 3.2 Algorithm summary for LARTTE

the following:

$$\begin{aligned}
\text{minimize} \quad & \alpha_1 a_0 + \alpha_2 P_{leakage}(Wg, Vtn, Vtp, s) \\
& + \alpha_3 P_{dynamic}(Wg, C_L, Vtn, Vtp, s) \\
\text{subject to} \quad & a_j \le a_0,\ j \in input(0),\ j \ne c \\
& a_c \le (1.0 + \eta) a_0,\ c = \text{critical PO}, 0 \le \eta \le 1 \\
& a_j + D_i \le a_i,\ i \in \mathcal{G} \cap \forall j \in input(i) \\
& D_i \le a_i,\ i \in \mathcal{D} \\
& T_i \le s_i,\ i \in (\mathcal{D} \cup \mathcal{G}) \\
& L_{w_i} \le W_{g_i} \le U_{w_i},\ i \in \mathcal{G} \\
& L_{tn_i} \le V_{tn_i} \le U_{tn_i},\ i \in \mathcal{G} \\
& L_{tp_i} \le V_{tp_i} \le U_{tp_i},\ i \in \mathcal{G}
\end{aligned} \tag{3.12}$$

It can be seen from equation 3.12 that the only thing that has changed from the original $\mathcal{PP}$ formulation is that the old constraint on the original critical output has been modified/relaxed by a user-specified ratio, η. This is done to explicitly introduce a margin of separation between the most critical arrival time and all other outputs' arrival times. By doing so, the impact of delay variations can be minimized. The value of η ranges between 0 to 1, with a larger η leading to a greater margin of separation.

3.4 Experimental Results

We implemented LARTTE in C++ and ran all of our experiments on a 1.0GHz Pentium 4 machine with 1.0Gb of RAM. The stopping criterion of LARTTE was set to when $\mathcal{PP}$ and $\mathcal{LRS}/\lambda$ agreed to within 1.0%. Lower and upper bounds of the transistor width were 0.2μm and 1.1μm, respectively. For V_t, the lower and upper bounds were 0.14V and 0.26V. V_{DD} was 1.0V. Input slew ranged from 30ps to 150ps. Four V_t levels were made available for discretization: 0.14V, 0.18V, 0.22V, and 0.26V. Appropriate activity factors were assigned to the posynomials throughout the circuit using PowerMill. All SPICE simulations were carried out in 0.1μm technology. We conducted our experiments on the ISCAS85 benchmark circuits, where the number of gates ranged from 214 to 3,512, and the total number of tunable parameters from 654 to 15,198. Tables 3.3 and 3.4 show the LARTTE optimization results. To illustrate the convergence property of LARTTE, we show in Figure 3.3 the convergence sequence for a 12-bit ALU controller. As it can be seen, the duality gap is closing each step along the way as desired. This behavior was observed in all of our experiments.

In Table 3.3, the ”optimize delay” columns show the maximum delay before and after tuning, with only timing involved in the objective function (α_1=1, α_2=α_3=0). All transistors have a nominal V_t value of 0.18V. After obtaining the best possible delay value from sizing optimization alone, we then try to optimize the total power consumption subject to that same delay value. Hence, the solution obtained from tuning the power consumption is guaranteed to have a critical path delay not exceeding the corresponding delay value shown in the ”optimize delay” column. For power tuning, the dynamic and leakage power terms were arbitrarily assigned equal weights. The resulting optimized-power solution from tuning both the transistor width and V$_t$ are shown in the ”optimize total power” columns of Table 3.4. Compared to the power consumption of the circuit with delay-tuning only, this shows an average of 58% improvement in total power reduction. The Tables also show that LARTTE exhibits linear runtime and memory usage requirement (see Figure 3.4 as well). Lastly, we show in Table 3.4 the leakage power consumption before and after V$_t$ discretization. As it can be seen, applying V_t discretization introduces only a trivial amount of error. This suggests that with 4 levels of V_t available, the discretization heuristic works reasonable well. This is also shown through Figure 3.5, which analyzes the degree of power reduction which can be achieved as a function of the number of V_ts available for use.

To gauge the effectiveness and runtime of LARTTE, we used a state-of-the-art, general convex problem solver in SNOPT to solve the same primal problem (with discretization as well). The runtime results are tabulated in Table 3.4, where it can be seen that our LR method is over 250x faster. Independently, we verified that our LARTTE solution agreed with the SNOPT solution to within 1% in all cases.

Table 3.3 LARTTE results: Part I

Circuit Name	# of Gates	# of Var.	# of Constr.	Optimize Delay (ps)			Memory (MB)
				Min. size nom.-V_t	Sizing nom.-V_t	%	
c432	214	654	473	1620	1230	24.1	1.0
c499	514	1716	1059	1060	895	15.6	1.5
c880	383	1665	987	1070	872	18.5	1.5
c1355	546	1908	1227	1070	914	14.6	1.5
c1908	880	3315	1781	1500	1220	18.7	2.5
c2670	1193	5397	2903	1860	1520	18.3	3.5
c3540	1169	7446	3824	2170	1800	17.1	4.5
c5315	2307	10656	5932	1900	1590	16.3	6.0
c6288	2416	8016	5120	6070	5170	14.8	5.0
c7552	3512	15198	8011	1520	1250	17.8	8.5

Table 3.4 LARTTE results: Part II

Circuit Name	Optimize Total Power (0.1mW)						Leakage Power	
	Sizing nom.-V_t	Sizing multi-V_t	%	Runtime (s)		Speed up	Before Discretize	After Discretize
				SNOPT	LARTTE			
c432	1.25	0.59	52.9	31	5	5.9	7.66e-6	7.67e-6
c499	3.49	1.46	58.3	290	10	29.7	1.71e-5	1.74e-5
c880	3.41	1.35	60.4	341	42	8.1	1.90e-5	1.91e-5
c1355	5.62	2.93	47.9	269	9	29.7	4.43e-5	4.47e-5
c1908	7.22	3.07	57.5	1316	57	23.0	4.21e-5	4.24e-5
c2670	10.7	4.09	61.9	7915	107	74.0	3.93e-5	3.95e-5
c3540	14.7	6.02	58.9	20773	222	93.6	5.44e-5	5.48e-5
c5315	19.8	8.42	57.4	64424	330	195.2	9.28e-5	9.32e-5
c6288	15.8	4.66	70.4	25326	299	84.7	1.85e-5	1.89e-5
c7552	27.8	12.6	54.6	117067	431	271.6	1.35e-4	1.36e-4

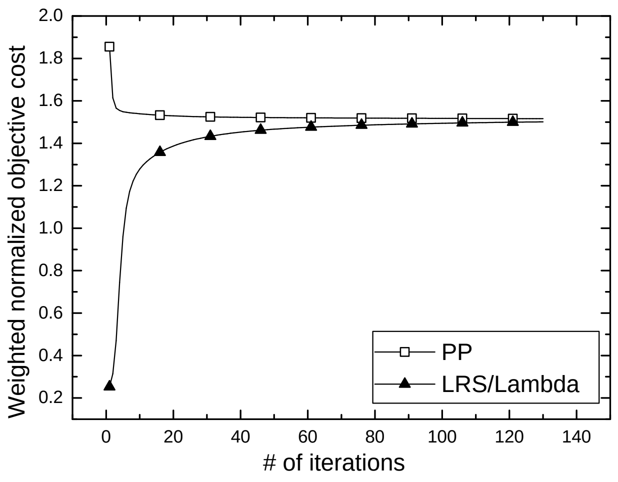
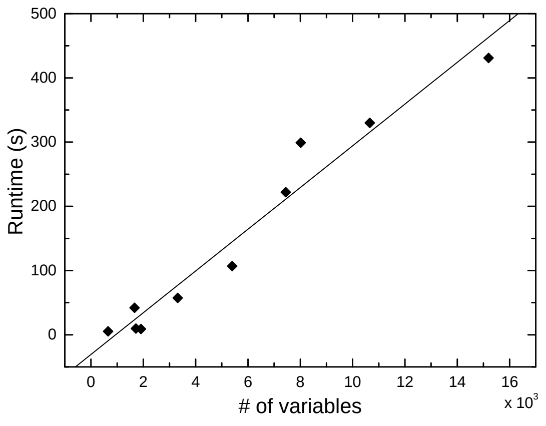


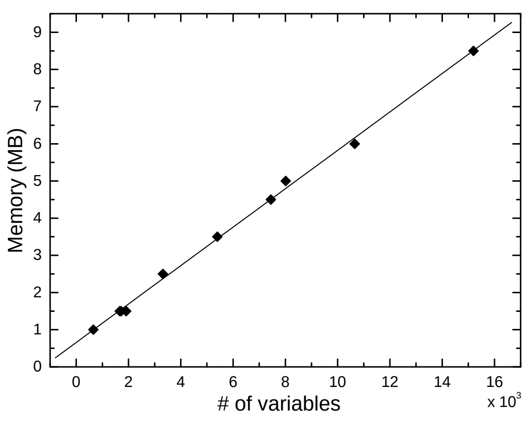
Figure 3.3 The convergence sequence for a 12-bit ALU and controller.

We next investigate LARTTE's effectiveness of guarding against delay variations. An η value of 0.5 was used in all of our tests. Shown in Table 3.5 are all the circuit's critical-path delay before and after re-invoking LARTTE with the process variation modifications. Also shown are the next closest output arrival times before and after re-tuning. As it can be seen, LARTTE successfully creates a distance separation between these two values after re-tuning. We also show the relationship between η's value and the resulting max frequency in Figure 3.6(a). For the circuit (c1908) in that figure, it can be seen that increasing the value of η bumps up the final critical delay value and decreases the maximum operating frequency. However, as the critical delay value rises, the number of 'potential' delay-violating paths directly decreases. A 'potential path' was arbitrarily defined as any non-critical path whose final delay value is within 10% of the final critical delay value. The tradeoff between max frequency and final delay variation probability is clear from this figure. Finally, for completeness, we also show the tradeoff between max frequency and total circuit area in Figure 3.6(b). Area was calculated by summing all transistors' widths.

Table 3.5 Delay Separation Before and After Re-Tuning with Process Variation Modifications

	Before Re-Tuning		After Re-Tuning	
Circuit Name	Critical Delay (ps)	Nearest Delay (ps)	Critical Delay (ps)	Nearest Delay (ps)
C432	1230	1165	1279	1165
C499	895	889	968	890
C880	872	847	924	847
C1355	914	914	994	914
C1908	1220	1207	1312	1204
C2670	1520	1519	1593	1520
C3540	1800	1784	1921	1786
C5315	1590	1561	1703	1558
C6288	5170	5170	5582	5170
C7552	1250	1248	1362	1249





(a) (b)

Figure 3.4 The (a) runtime and (b) storage requirements of LARTTE vs. number of variables.

3.5 Conclusion

In this chapter, we presented a novel, effective, and fast way to perform simultaneous gate-sizing and multiple-V$_t$ assignment using generalized Lagrangian Relaxation and posynomial modeling. Our technique is practical, versatile, and accurate. We also showed an easy way to modify the tuning algorithm to directly take process variations into account. This is one of the key contributions of this technique, along with the ability to reduce leakage through simultaneous V_t optimization.

3.6 Extension: LARTTE for ASIC Designs

It should be noted that the LARTTE algorithm presented up to now seems to suit full and semi-custom design flows only, since standard cell issues such as pin unateness, rise time/transition vs. fall time/transition, and pin-to-pin propagation delay/slew were largely simplified to the point of trivialness. However, in this section, we show that the LARTTE formulations can be easily modified to allow the algorithm to work within an ASIC standard cell flow. However, instead of sizing the width of the transistors within a gate, we are now

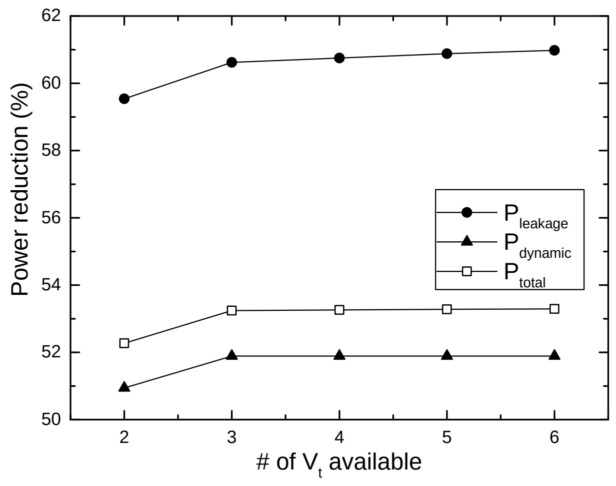


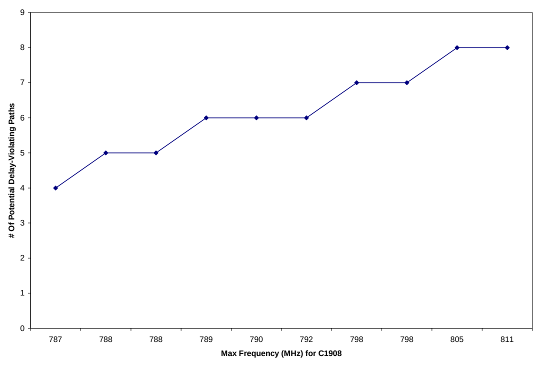
Figure 3.5 Effect of varying the number of V_t levels available on the potential of power reduction with LARTTE

finding the drive strength instance of the cell instead, which is characterized as a function of the gate's input capacitance.

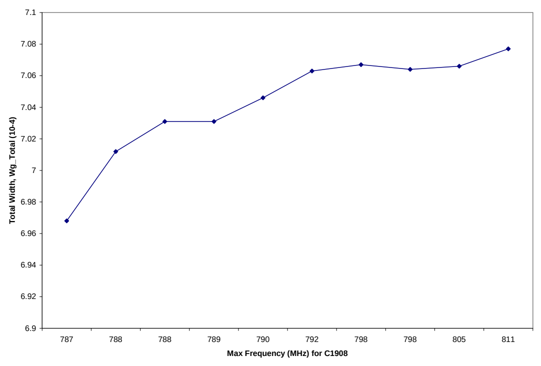
We begin by first making changes to the posynomial-fitting procedure. The posynomial regression problem remains unchanged, which is repeated here for convenience:

$$\textbf{Posyfit:}\ \text{minimize} \quad \sum_{m=1}^{z}\left(\left(\sum_{j=1}^{k} c_j \prod_{i=1}^{n} x_i^{\alpha_{ij}}\right) - b_m\right)^2$$
$$\text{subject to} \quad c_j \geq 0,\ 1 \leq j \leq k \tag{3.13}$$

Before, we solved this problem by iteratively "guessing" and fixing the unknown exponents (α_{ij}), then solving for the coefficients (c_j) using CFSQP. This strategy is no longer practical for the ASIC flow, since the sheer size of the standard cell library makes the time requirement of this guess-and-solve procedure excessive. Instead, what we do now is combine the exponents and coefficients into one single unknown vector, call it Z, and then let CFSQP solve for the complete value of Z directly (it should be noted that the number of monomial terms, k, is still guessed and fixed at the beginning of this procedure). Doing so gets rid of the exponent guessing step and makes the regression procedure viable once again.



(a)



(b)

Figure 3.6 The (a) Number of Potential Delay-Violating Paths Vs. Max Frequency, and (b) Total Wg Vs. Max Frequency.

Other than the posynomial-fitting procedure, another change that needs to be made to LARTTE is that the original, nonlinear contrained optimization problem needs to be re-formulated. Ignoring power and threshold voltage assignment for now, Equation 3.14 shows a new Primal Problem ($\mathcal{PP}$) formulation which is suitable for the ASIC flow. In this equation, I denotes the set of primary inputs, N the set of internal gates, and O the set of primary outputs. Also, L_j and $incap_j$ denote the loading capacitance and the input capacitance of gate j, respectively. From this new Primal Problem, a new Lagrangian Function can be derived as shown in Equation 3.15. Its KKT conditions follow in Equation 3.16 and 3.17. The subgradient multiplier adjustment step also gets changed, as is shown in Equation 3.18. Using these new derivations, Lagrangian Relaxation can be carried out thereafter in exactly the same manner as before to arrive at an optimal set of input capacitance values for each gate. These values can then be discretized to the nearest drive strength instance to find the desired optimal cell instance to use for each gate. Thus, we have shown that LARTTE can be easily modified to fit within an ASIC design flow.

$$\begin{array}{lll}
\text{minimize} & a_{sink}^{rise} + a_{sink}^{fall} & \\
\text{s.t.} & \forall j \in O & : a_j^{rise} \leq a_{sink}^{rise} \\
& & a_j^{fall} \leq a_{sink}^{fall} \\
& \substack{\forall j \in (N \cup O) \\ \forall i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \notin I} & : a_i^{rise} + D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) \leq a_j^{rise} \\
& & a_i^{fall} + D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) \leq a_j^{fall} \\
& \substack{\forall j \in (N \cup O) \\ \forall i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \notin I} & : a_i^{fall} + D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) \leq a_j^{rise} \\
& & a_i^{rise} + D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) \leq a_j^{fall} \\
& \substack{\forall j \in (N \cup O) \\ \forall i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \in I} & : D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) \leq a_j^{rise} \\
& & D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) \leq a_j^{fall} \\
& \substack{\forall j \in (N \cup O) \\ \forall i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \in I} & : D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) \leq a_j^{rise} \\
& & D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) \leq a_j^{fall} \\
& \substack{\forall j \in N \\ \forall i \in in(j) \wedge \\ i \in (+)unate(in(j))} & : T_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) \leq s_j^{rise} \\
& & T_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) \leq s_j^{fall} \\
& \substack{\forall j \in N \\ \forall i \in in(j) \wedge \\ i \in (-)unate(in(j))} & : T_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) \leq s_j^{rise} \\
& & T_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) \leq s_j^{fall} \\
& bound_j^{lower} \leq incap_j \leq bound_j^{upper} &
\end{array} \tag{3.14}$$

$$
\begin{aligned}
L_\lambda = {} & a_{sink}^{rise} + a_{sink}^{fall} + \sum_{j\in O} \lambda_{jsink}^{rise}(a_j^{rise} - a_{sink}^{rise}) + \sum_{j\in O} \lambda_{jsink}^{fall}(a_j^{fall} - a_{sink}^{fall}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\notin I}} \lambda_{ij}^{rise\_(+)unate}(a_i^{rise} + D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - a_j^{rise}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\notin I}} \lambda_{ij}^{fall\_(+)unate}(a_i^{fall} + D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - a_j^{fall}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\notin I}} \lambda_{ij}^{rise\_(-)unate}(a_i^{fall} + D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - a_j^{rise}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\notin I}} \lambda_{ij}^{fall\_(-)unate}(a_i^{rise} + D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - a_j^{fall}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\in I}} \lambda_{ij}^{rise\_(+)PI}(D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - a_j^{rise}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\in I}} \lambda_{ij}^{fall\_(+)PI}(D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - a_j^{fall}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\in I}} \lambda_{ij}^{rise\_(-)PI}(D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - a_j^{rise}) + \\
& \sum_{j\in(N\cup O)} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\in I}} \lambda_{ij}^{fall\_(-)PI}(D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - a_j^{fall}) + \\
& \sum_{j\in N} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))}} \lambda_{ij}^{rise\_(+)slew}(T_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - s_j^{rise}) + \\
& \sum_{j\in N} \sum_{\substack{i\in in(j)\wedge \\ i\in(+)unate(in(j))}} \lambda_{ij}^{fall\_(+)slew}(T_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - s_j^{fall}) + \\
& \sum_{j\in N} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))}} \lambda_{ij}^{rise\_(-)slew}(T_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - s_j^{rise}) + \\
& \sum_{j\in N} \sum_{\substack{i\in in(j)\wedge \\ i\in(-)unate(in(j))}} \lambda_{ij}^{fall\_(-)slew}(T_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - s_j^{fall}) \qquad (3.15)
\end{aligned}
$$

$$\sum_{j \in O} \lambda_{jsink}^{rise} = 1$$

$$\sum_{j \in O} \lambda_{jsink}^{fall} = 1$$

$$\forall j \in O: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{rise\_(+)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{rise\_(-)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{rise\_(+)PI} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{rise\_(-)PI} = \lambda_{jsink}^{rise}$$

$$\forall j \in O: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{fall\_(+)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{fall\_(-)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{fall\_(+)PI} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{fall\_(-)PI} = \lambda_{jsink}^{fall}$$

$$\forall j \in N: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{rise\_(+)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{rise\_(-)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{rise\_(+)PI} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{rise\_(-)PI} = \sum_{\substack{k \in out(j) \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{rise\_(+)unate} + \sum_{\substack{k \in out(j) \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{fall\_(-)unate}$$

$$\forall j \in N: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{fall\_(+)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \notin I}} \lambda_{ij}^{fall\_(-)unate} + \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{fall\_(+)PI} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j)) \wedge \\ i \in I}} \lambda_{ij}^{fall\_(-)PI} = \sum_{\substack{k \in out(j) \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{fall\_(+)unate} + \sum_{\substack{k \in out(j) \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{rise\_(-)unate} \tag{3.16}$$

$$\forall j \in N: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j))}} \lambda_{ij}^{rise\_(+)slew} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j))}} \lambda_{ij}^{rise\_(-)slew} =$$
$$\sum_{\substack{k \in out(j) \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{rise\_(+)unate} (\frac{\partial D_{jk}^{*rise}}{\partial s_j^{rise}}) + \sum_{\substack{k \in out(j) \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{fall\_(-)unate} (\frac{\partial D_{jk}^{*fall}}{\partial s_j^{rise}}) +$$
$$\sum_{\substack{k \in out(j) \wedge \\ k \in N \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{rise\_(+)slew} (\frac{\partial T_{jk}^{*rise}}{\partial s_j^{rise}}) + \sum_{\substack{k \in out(j) \wedge \\ k \in N \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{fall\_(-)slew} (\frac{\partial T_{jk}^{*fall}}{\partial s_j^{rise}})$$

$$\forall j \in N: \sum_{\substack{i \in in(j) \wedge \\ i \in (+)unate(in(j))}} \lambda_{ij}^{fall\_(+)slew} + \sum_{\substack{i \in in(j) \wedge \\ i \in (-)unate(in(j))}} \lambda_{ij}^{fall\_(-)slew} =$$
$$\sum_{\substack{k \in out(j) \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{fall\_(+)unate} (\frac{\partial D_{jk}^{*fall}}{\partial s_j^{fall}}) + \sum_{\substack{k \in out(j) \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{rise\_(-)unate} (\frac{\partial D_{jk}^{*rise}}{\partial s_j^{fall}}) +$$
$$\sum_{\substack{k \in out(j) \wedge \\ k \in N \wedge \\ j \in (+)unate(in(k))}} \lambda_{jk}^{fall\_(+)slew} (\frac{\partial T_{jk}^{*fall}}{\partial s_j^{fall}}) + \sum_{\substack{k \in out(j) \wedge \\ k \in N \wedge \\ j \in (-)unate(in(k))}} \lambda_{jk}^{rise\_(-)slew} (\frac{\partial T_{jk}^{*rise}}{\partial s_j^{fall}}) \qquad (3.17)$$

$$j \in O: \quad \lambda_{jsink}^{rise\ \prime} = \lambda_{jsink}^{rise} + (a_j^{rise} - a_{sink}^{rise})$$
$$\lambda_{jsink}^{fall\ \prime} = \lambda_{jsink}^{fall} + (a_j^{fall} - a_{sink}^{fall})$$

$$\substack{j\in(N\cup O)\wedge \\ i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\notin I}: \quad \lambda_{ij}^{rise\_(+)unate\prime} = \lambda_{ij}^{rise\_(+)unate} + (a_i^{rise} + D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - a_j^{rise})$$
$$\lambda_{ij}^{fall\_(+)unate\prime} = \lambda_{ij}^{fall\_(+)unate} + (a_i^{fall} + D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - a_j^{fall})$$

$$\substack{j\in(N\cup O)\wedge \\ i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\notin I}: \quad \lambda_{ij}^{rise\_(-)unate\prime} = \lambda_{ij}^{rise\_(-)unate} + (a_i^{fall} + D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - a_j^{rise})$$
$$\lambda_{ij}^{fall\_(-)unate\prime} = \lambda_{ij}^{fall\_(-)unate} + (a_i^{rise} + D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - a_j^{fall})$$

$$\substack{j\in(N\cup O)\wedge \\ i\in in(j)\wedge \\ i\in(+)unate(in(j))\wedge \\ i\in I}: \quad \lambda_{ij}^{rise\_(+)PI\prime} = \lambda_{ij}^{rise\_(+)PI} + D_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - a_j^{rise})$$
$$\lambda_{ij}^{fall\_(+)PI\prime} = \lambda_{ij}^{fall\_(+)PI} + D_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - a_j^{fall})$$

$$\substack{j\in(N\cup O)\wedge \\ i\in in(j)\wedge \\ i\in(-)unate(in(j))\wedge \\ i\in I}: \quad \lambda_{ij}^{rise\_(-)PI\prime} = \lambda_{ij}^{rise\_(-)PI} + D_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - a_j^{rise})$$
$$\lambda_{ij}^{fall\_(-)PI\prime} = \lambda_{ij}^{fall\_(-)PI} + D_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - a_j^{fall})$$

$$\substack{j\in N\wedge \\ i\in in(j)\wedge \\ i\in(+)unate(in(j))}: \quad \lambda_{ij}^{rise\_(+)slew\prime} = \lambda_{ij}^{rise\_(+)slew} + T_{ij}^{*rise}(s_i^{rise}, L_j, incap_j) - s_j^{rise})$$
$$\lambda_{ij}^{fall\_(+)slew\prime} = \lambda_{ij}^{fall\_(+)slew} + T_{ij}^{*fall}(s_i^{fall}, L_j, incap_j) - s_j^{fall})$$

$$\substack{j\in N\wedge \\ i\in in(j)\wedge \\ i\in(-)unate(in(j))}: \quad \lambda_{ij}^{rise\_(-)slew\prime} = \lambda_{ij}^{rise\_(-)slew} + T_{ij}^{*rise}(s_i^{fall}, L_j, incap_j) - s_j^{rise})$$
$$\lambda_{ij}^{fall\_(-)slew\prime} = \lambda_{ij}^{fall\_(-)slew} + T_{ij}^{*fall}(s_i^{rise}, L_j, incap_j) - s_j^{fall}) \qquad (3.18)$$

Chapter 4

Sectoral Partial Vector Control (SPVC) with Leakage-Aware Technology Mapping for Subthreshold Leakage Reduction

The previous chapter presented a technique for optimization of delay and power subject to process variation uncertainties. In this chapter, we focus further on the reduction of subthreshold leakage current/power. We present a novel, low-overhead technique which is similar in principle to Input Vector Control, but much more effective in practice on large, VLSI circuits.

As was mentioned earlier, the degree of effectiveness one can achieve with Input Vector Control (IVC) is ultimately bounded by how large a circuit is and by how structured its gates are arranged. In this chapter, we propose a new technique for leakage power reduction that aims to overcome these barriers such that the use of input assignment can be practical for future circuits of high complexity. Our method is called Sectoral Partial Vector Control, or SPVC, and is coupled with a leakage-aware technology mapping step. As it will be shown in our results, this method can achieve on average 28.3% leakage savings with SPVC alone, and 69.5% leakage reduction when SPVC is coupled with technology mapping.

The rest of this chapter is organized as follows. Background information is provided in Section 4.1, followed by the two key contributions of this work, SPVC and leakage-aware technology mapping, in Sections 4.2 and 4.3. Section 4.4 contains the experimental results. A summary is given in 4.5.

4.1 Background

In modern technology, the leakage current is dominated by the subthreshold leakage and by the gate tunneling leakage. The exact physical characteristics of these currents can be found in [29]. In this work, only a high-level picture is needed, which is that due to an important phenomenon known as the stacking effect [21], a CMOS gate's leakage value is highly sensitive to the input pattern it sees. For example, the leakage characteristics for several basic CMOS cells is tabulated in Table 4.1 using SPICE. As it can be seen, $I_{leakage}$ can vary significantly between different input patterns. It can be independently verified that this is true in general for most CMOS cells (the exceptions being inverters and those cells which use inverted inputs, such as XORs/XNORs). Based on this observation, it's easy to see why an input vector can be used to control the leakage state of a circuit.

4.2 SPVC: Sectoral Partial Vector Control

The degree to which a circuit's leakage power can be controlled via IVC is ultimately bounded by how large that circuit is. To overcome this limitation, we explore a novel strategy to use in tandem with input vector control: internal node control.

Our technique is called Sectoral Partial Vector Control, or SPVC, and is divided into 3 phases: sectoring, partial vector identification, and merging. The main approach is as follows: first, we divide a circuit into smaller and more tractable sub-circuits. Next, we independently find a low-leakage ”partial” input vector for each sector. Finally, we merge the partial vector solutions together using internal ”switches”. The end result is a set of control points inserted throughout the input nodes as well as the internal nodes, thereby guaranteeing a tighter degree of leakage management than that which is possible with input vector control alone. We will show how the additional overhead arising from internal node switches can be kept low through the concept of partial vector control, where we only control a subset of the inputs in achieving a low-leakage state.

We now explain each phase in more details.

Table 4.1 Leakage current (e-9) corresponding to different input combinations

INPUTS	NAND2	NAND3	NAND4	NOR2	NOR3	NOR4
0000	151	85	58	1293	1929	2552
0001	644	151	85	270	256	248
0010	513	152	85	386	267	253
0011	784	639	151	57	54	53
0100		147	85		381	265
0101		510	151		57	54
0110		496	151		56	54
0111		1174	634		29	29
1000			83			375
1001			144			57
1010			145			56
1011			507			29
1100			143			56
1101			493			29
1110			486			29
1111			1562			20

4.2.1 First Phase: Sectoring

We first partition a circuit into ”sectors” using the Multilevel Fiducia-Metheyes (MLFM) algorithm [30]. In MLFM, balanced partitions are sought using as few cuts as possible. This matches our desired goals, for by maintaining a balance amongst the different blocks, each resulting sector will be less likely to be too large still. Also, the fewer the cuts, the lesser the number of internal nodes to merge in the end, and therefore the smaller the overhead.

For MLFM, since balance is defined by the weights on the vertices, we heuristically assign each vertex or each node a weight proportional to its ”expected leakage”. That is, because each node is an output of some gate, call it gate Y, and Y will have different leakage values for different input patterns, then we can assign the weight of that node to be gate Y’s expected leakage, which is simply defined as that shown in Equation 4.1. However, in order to calculate this value, each node’s 0 and 1 probabilities must be computed first. Because this is itself an NP problem, we heuristically ignore the issue of fanout reconvergence for now in order to keep the problem simple.

$$\begin{gathered} \forall\, i \in \text{possible input combinations to gate Y :} \\ ExpectedLeakage(Y) = \sum_i Prob(i) \times (I_{leakage}(Y) \mid \text{input combination = i}) \end{gathered} \tag{4.1}$$

4.2.2 Second Phase: Partial Vector Identifi cation

Once sectoring is complete, the next phase of SPVC is to independently identify a low-leakage ”partial” vector for each sector. In general, it is possible to be able to control only a subset of the inputs and yet still achieve decent leakage control. This can best be explained by the data shown in Table 4.1, where we see that even though different input patterns can lead to different leakage values, this difference can sometimes be relatively trivial. The reason this occurs is because of the stacking effect. Due this phenomenon, *don’t cares* can be exploited among the similar leakage input patterns to arrive at a low-leakage partial input vector.

To begin partial vector identification, every combinational standard cell in the cell library must be pre-characterized like that shown in Table 4.1. This is necessary to introduce leakage

”tiers” for each cell. A leakage tier is defined as a small and distinct range of leakage values. For example, using Table 4.1, a NAND3 can be characterized to have 3 tiers: one with {’000’, ’001’, ’010’, ’100’}, another with {’011’, ’101’, ’110’}, and the last being {’111’}. In some cases, a cell will only exhibit 1 leakage tier(i.e., inverters, XORs/XNORs). Therefore, the leakage state of these cells are relatively insensitive to the input pattern and these cells cannot really benefit from input assignment.

The way in which a tier is defined or grouped affects how ”partial” the end resulting vector will be. The looser the grouping(lumping values which do not differ as little as we would have liked them to be), the more ”partial” the result will be. However, looser groupings will lead to a solution which is inferior in accuracy of leakage control to a solution which is found using tighter groupings. It is up to the user to pick the tradeoff point in accuracy vs. partiality. If the number of ways a circuit was partitioned was large, then it makes sense to group looser to incur less overhead. However, if only a small number of partitions were used, then one probably should group tighter to exercise better leakage control. For this work, we found that 3 leakage tiers is typically enough for most of the standard cells.

After characterizing the leakage tiers for every cell, the next step is to introduce a ”LeakageClause” for every gate that has more than one leakage tier, or those gates which are sensitive to input control. The LeakageClause models the leakage-input dependence of a gate and is formulated as follows: Suppose that gate$_i$ was pre-characterized to have N leakage tiers, T$^i_1$, T$^i_2$,... T^i_N. Let the following notation comb^j_i be used to denote an input combination that corresponds to leakage tier T$_i$, with j being just an index to distinguish between different combinations of the same leakage tier(i.e., comb$^1_1$, comb$^2_1$, and comb$^3_1$ are all input combinations that result in T$_1$). Then, the general form of a LeakageClause$_i$ can be written as follows:

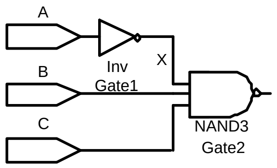


Figure 4.1 Example Circuit

$$
\begin{aligned}
LeakageClause_i &= \\
\left[comb_1^1 + comb_1^2 + ... + comb_1^j\right] (T_1^i)(\overline{T_2^i})...(\overline{T_N^i}) &+ \\
\left[comb_2^1 + comb_2^2 + ... + comb_2^j\right] (\overline{T_1^i})(T_2^i)...(\overline{T_N^i}) &+ \\
&. \\
&. \\
\left[comb_N^1 + comb_N^2 + ... + comb_N^j\right] (\overline{T_1^i})(\overline{T_2^i})...(T_N^i) &
\end{aligned}
\tag{4.2}
$$

An important thing to note is that in a LeakageClause, the input combination expressions comb$_i^j$ are formed from the previous gates' logic functions, which are recursively expressed down to only the input variables. For example, LeakageClause$_2$ for the circuit shown in Figure 4.1 is written as follows (Assuming the same NAND3 tier characterization as before):

$$
\begin{aligned}
LeakageClause_2 &= \\
\left[(A)(\overline{B})(\overline{C}) + (A)(\overline{B})(C) + (A)(B)(\overline{C}) + (\overline{A})(\overline{B})(\overline{C})\right] & \\
(T_1^2)(\overline{T_2^2})(\overline{T_3^2}) &+ \\
\left[(A)(B)(C) + (\overline{A})(\overline{B})(C) + (\overline{A})(B)(\overline{C})\right] & \\
(\overline{T_1^2})(T_2^2)(\overline{T_3^2}) &+ \\
\left[(\overline{A})(B)(C)\right] (\overline{T_1^2})(\overline{T_2^2})(T_3^2) &
\end{aligned}
$$

Notice that in the above equation, the first input to the NAND3, X, is implicitly replaced with ($\overline{A}$) in the input combination expressions. This is done to avoid the introduction of new variables besides those associated with the leakage tiers.

After deriving the LeakageClause for every gate with more than 1 leakage tier, we then proceed to AND them all together to form a conjunction called the CircuitLeakage.

$$CircuitLeakage = \Pi_{i=1}^{n}(LeakageClause_i) \tag{4.3}$$

It is easy to see that CircuitLeakage is always satisfiable because the values for the leakage tier variables T_j^i can be arbitrarily set to match the actual resulting tier states under a given input assignment.

Once CircuitLeakage is found for the target circuit, we can then construct a weighted, Reduced Ordered Binary Decision Diagram (ROBDD) [31] called the FinalBDD to represent CircuitLeakage. The weights are assigned to the arcs of FinalBDD as follows: For those nodes associated with the primary inputs, both their THEN and ELSE arcs are arbitrarily assigned a small(though non-trivial) value. In this work, we chose a value of 30 (The reason for this decision will be explained soon). For the leakage tier nodes T_j^is, their ELSE arcs all have a weight of 0 while their THEN arcs have a weight that is proportional to the leakage value represented by that tier. If a leakage tier is associated with more than one input combination, then its THEN arc's weight is computed as the average of those input combinations' resulting leakage values. For example, suppose we are dealing with a NAND3 gate. Then, using the same NAND3 leakage tier characterization as before, we would assign its T_1 node's THEN arc a weight of 134, which is the average between 85('000'), 151('001'), 152('010'), and 147('100'). Similarly, its T_3's THEN arc would have a weight of 1174('111'). As an illustration, Figure 4.2 shows the FinalBDD for the circuit of Figure 4.1 (with no reordering).

From the FinalBDD, a low-leakage partial vector can be determined by tracing a shortest path from root to 1, where shortest is defined as min(Σ(total path arc weights)) for all possible paths to 1. Through this path, by examining which arcs are taken from the input nodes, the values to control the different input signals can be determined. Furthermore, those inputs which do not need to be controlled can be identified. For example, in Figure 4.2, the shortest path is (A)($\overline{B}$)(T_1^2)($\overline{T_2}^2$)($\overline{T_3}^2$), so the optimal leakage partial input vector is simply AB = $\{10\}$. Notice that input C is not included in the solution because regardless of what C's value is, as long as AB = $\{10\}$, the leakage state of this circuit will be low.

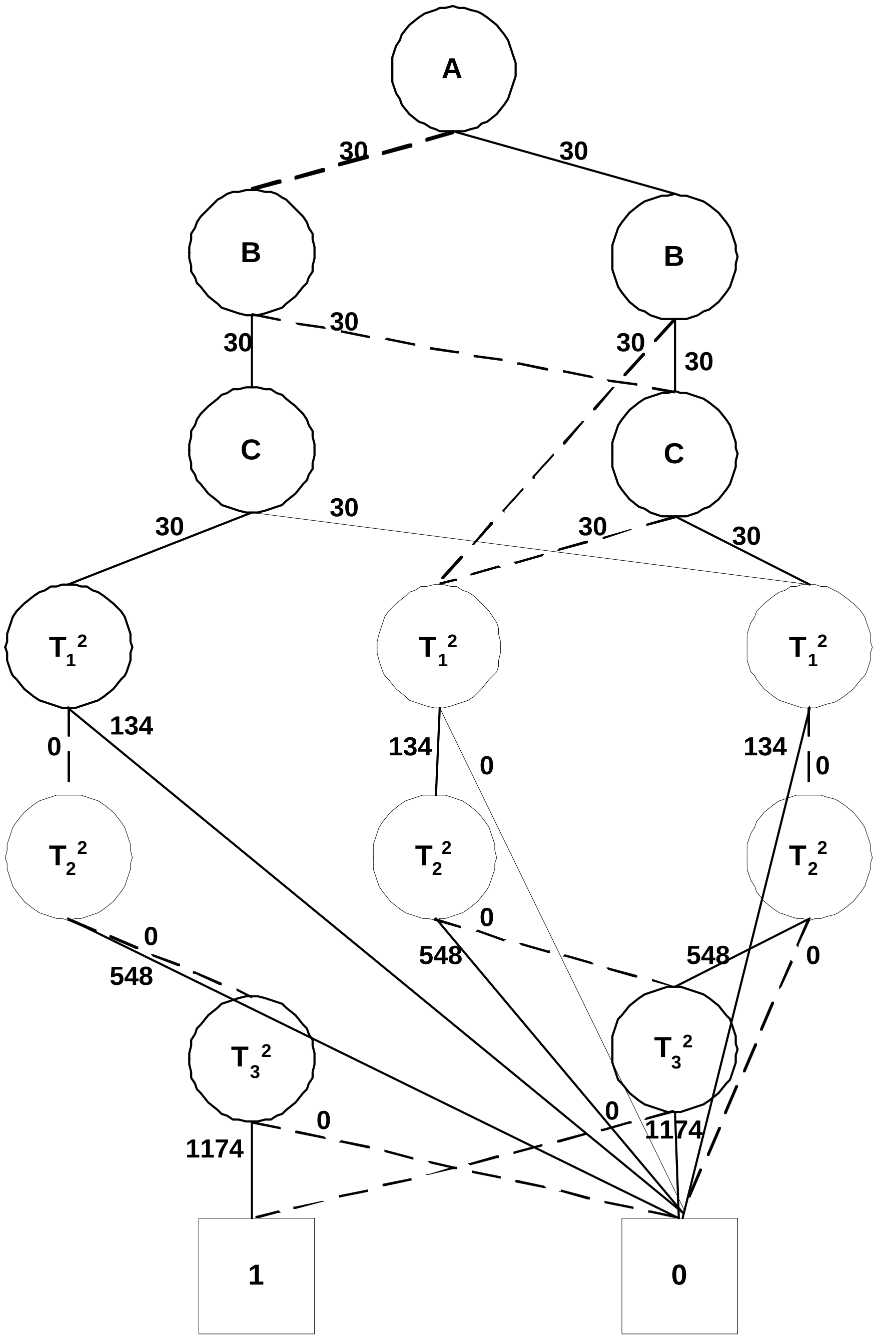


Figure 4.2 FinalBDD for Figure 4.1. Solid lines=THEN, dashed=ELSE

Although it’s not shown in this example, there are times when a shortest path found can be sub-optimal in the sense that even though the controlled leakage state may be low, the number of inputs required to be controlled may not be minimized. This is because depending on the way the node variables are ordered, a path from root to 1 can unnecessarily traverse a node which in another order may be avoided altogether. For instance, in the ROBDD of Figure 4.2, node A is the first node from root, so it must be traversed no matter what. However, it is possible that if the tree were ordered in a different way where A is not the first node from the root, then we could potentially bypass A in tracing a path to 1. Therefore, the order of nodes can play a large role in determining how partial we can get in our solution. Hence, this is the main reason why ROBDDs are used in this work, because of their dynamic reordering capabilities.

Previously, we said that we chose a small yet non-trivial value for the arc weights of the input nodes. The reason for this is that in assigning a small value in relation to the weights on the tier state nodes, we can allow the path-finding routine to prioritize for leakage reduction first over input minimization, because the weights on the tier state nodes will dominate in determining the ”shortest” solution. However, because the weights of these input nodes are non-trivial, it will allow the number of input nodes traversed in the final solution to be minimized once the tier state nodes’ values have been fixed.

The partial vector identification process is summarized in Algorithm 1. Some implementation details (ie. the data structure for LeakageClause) are now clarified. As a sidenote, we want to point out that this method for determining a partial vector was designed specifically for SPVC, and not for use by itself. If one were to try to use Algorithm 1 to find a partial vector for any circuit in general, one would find that often, the size of the ROBDD will grow quickly out of control. However, because we use this method only for SPVC where the target circuit is small(a sector), this method is viable.

```
Enable automatic dynamic reordering for the ROBDDs;
FinalBDD = the constant 1 BDD;
for all the gates in this circuit do
    /* Assume that the output of this gate is node i and its inputs are nodes a, b, ... , m */
    LogicBDD_i=Funct_i$(LogicBDD(a)..LogicBDD(m))$;
    if current gate has more than 1 leakage tier then
        LeakageBDD_i = BuildROBDD(LeakageClause_i);
        Assign arc weights to LeakageBDD_i;
        FinalBDD = $\prod$ (FinalBDD)(LeakageBDD);
    end
end
BestShortestPath = NULL;
$\alpha$ = user-specified threshold limit;
while $\alpha \geq 0$ do
    TempPath = ShortestPathTo1(FinalBDD);
    if Length(TempPath) $<$ BestShortestPath then
        BestShortestPath = TempPath;
    end
    $\alpha$ = $\alpha$ - 1;
    Dynamically reorder FinalBDD in preparation for the next iteration;
end
Return BestShortestPath;
```

Algorithm 1: Partial Vector Identification

4.2.3 Third Phase: Merging

After we have identified a low-leakage partial vector for each sector, the last phase in SPVC is to merge these results together and determine which internal node needs a ”switch”. A switch is a hardware that outputs a fixed value if the current mode is IDLE. In normal ACTIVE mode, a switch behaves like a buffer. For example, an AND gate with IDLE’ as one of the inputs is a switch for 0-forcing. Similarly, an OR gate with IDLE as an input is a mechanism for 1-forcing. Latches/Muxes can also be used as switches.

Switches should not just be inserted at every cut. To reduce the overhead of internal node control, we only insert a switch at a cut if its input and output values do not match. That is, consider a cut occurs on node x. Then, x will be an output in 1 sector, let’s say sector A, and an input in another, let’s say sector B. To determine whether a switch should be inserted at x, we first find out the logic value that x will have during IDLE while a partial vector is applied on its parent sector A. Then, we compare this value with that which is required by the child sector B’s partial vector. If the values match or there is a don’t care, then we don’t need to insert a switch at x because the values can naturally converge. If they don’t, then a switch is needed at x to force out its opposite value during IDLE. This is because the value of x needed to compose B’s partial vector is different from what is normally outputted by A during IDLE. The switch insertion strategy is summarized in Algorithm 2.

After the switches have been inserted, the original netlist must be modified to re-route the nets in accordance with which sector needs which internal node value. For those fanout sectors which required the use of a switch to enforce their partial vectors, their input terminal is re-routed to the output pin of the appropriate switch. This is shown in Figure 4.3.

Care must be taken when implementing a switch, for we have found that if the switches were implemented in normal CMOS configuration or as latches/MUXes, then the resulting leakage consumption of the switches themselves can quickly make up for any reduction benefit gained from internal node control. Hence, in this work, we used forced-stacking [21] AND/OR gates as our switches to reduce the internal leakage of the switches. Figure 4.4 shows our 0-forcing switch. The 1-forcing switch can be derived in a similar manner.

```
for all the internal nodes of this circuit do
    /* Assume current node is node j and it belongs to sector J */
    if current node is not a member of the cutset then
        /* current node does not have a fanout gate which belongs to a different sector */
        break;
    else
        for all of the fanout nodes of j which belongs to a different sector do
            /* A stores the Boolean output value of node j while sector J sees its optimal
            partial vector. If A=2, this value is not fixed under the partial vector */
            A = Restrict(LogicBDD$_j$, PartialVector(J));
            /* Assume that the current fanout node is node k, which belongs to sector K.
            Let B store the required Boolean input value of k for which is needed to form
            sector K's optimal partial vector. Note that if k does not need to be
            controlled, then B=2 for don't care */
            B = Extract k's value from PartialVector(K);
            switch A, B do
                case A=B or B=2
                    No internal switch inserted at node k;
                case (A=0 or A=2) && B=1
                    Insert internal switch at k for forcing out a logic 1 during IDLE;
                case (A=1 or A=2) && B=0
                    Insert internal switch at k for forcing out a logic 0 during IDLE;
        end
    end
end
```

Algorithm 2: Internal Switch Insertion

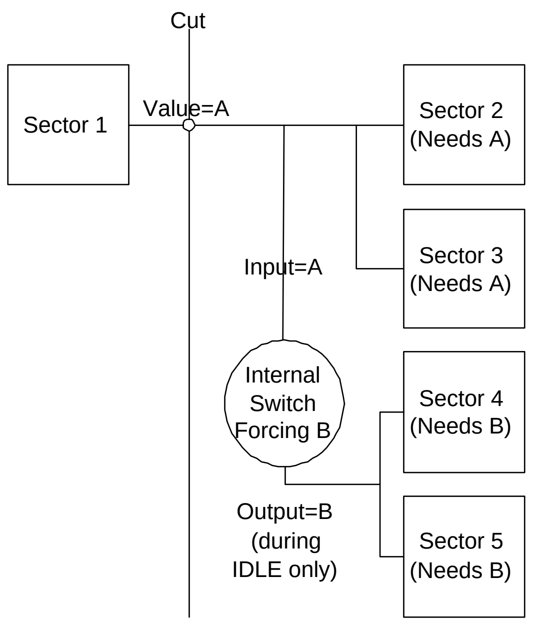


Figure 4.3 Re-routing of Nets After Switch Insertion

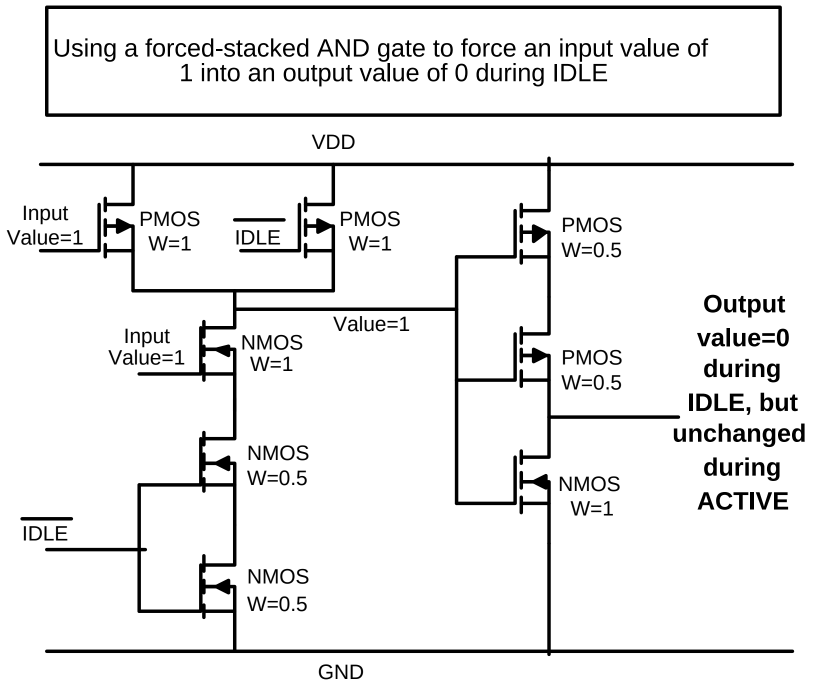


Figure 4.4 Low-Leakage 0-Forcing Switch

The description of SPVC is now complete. Algorithm 3 summarizes the entire technique.

```
Compute and store all node's 0-1 probabilities;
Assign each node a weight proportional to the expected leakage at node;
MLFM(weighted circuit hypergraph);
for each of the sectors do
    PartialVectorIdentification(sector);
end
for each node in the cut set do
    Perform input-output compatibility analysis. Insert switch if needed and re-route
    nets afterwards;
end
Add in the final hardware for controlling the inputs as well;
```

Algorithm 3: SPVC

4.3 SPVC with Leakage-Aware Technology Mapping

Through SPVC, the circuit size limitation is moderated. However, there is another factor which strongly determines the potential of input assignment, and that is a circuit's logical structure. Therefore, to achieve the highest potential of input control, we propose a technique to simultaneously apply technology remapping while performing SPVC. Before we begin on this subject, we assume the reader to be familiar with the basics of technology mapping, in particular the work done in [32].

4.3.1 Localized Technology Remapping

Since our circuit to apply SPVC on may come from the output of some logic synthesizer which targets for a classical objective (delay, area, dynamic power), this circuit may have an initial structure which is poor to input vector control. Therefore, we propose that after SPVC has completed on the original circuit, we first store the results for future comparison purposes, then go through each sector and perform ”local” technology remapping on that sector. The goal is to avoid remapping an entire circuit all at once and instead work on one sector at a time so as to limit the effect of errors which were inadvertently introduced during node probability computations(which we will use again in remapping). With this localized approach, we can accept some remappings and reject others, thereby achieving a higher level of quality in the final solution.

The order in which we perform localized remapping is done starting from the sector with the worst leakage consumption state(while being controlled by its partial vector) and down to the least leakage-costly sector. The reason for this is because in-between the end of a sector's remapping procedure and the start of another's, some update has to occur like slack changing, switch insertion/deletion, etc., and this can introduce dependencies which can degrade a future remapping result for another sector. It is impossible to know which order of remapping leads to the best result, so we heuristically try to improve on the sector with the worst initial result first, then gradually work backwards towards the sectors with the better initial results.

4.3.2 DC Curve-Based Remapping Using Expected Leakage

The method that we use to perform technology remapping is actually well-established [32]. It is a DC curve procedure for performing technology mapping under two constraints, delay and cost (in [32], area was used as the cost metric). Therefore, due to space limitation, we will skip most of the details on the mapping process and instead focus on how to integrate simultaneous remapping with SPVC.

Given a target sector for remapping, we follow the exact procedure which is done in the DC-curve method. For the forward traversal or matching phase, the cost metric we use for each match at a node is based on the previously-discussed notion of expected leakage, which is computed on the fly for each match. The backward tree traversal phase picks for each output node a DC-point that is lowest in total expected leakage and is $\leq$ the required time of that node. Our goal is to find a remapping which does not violate the original delay of this mapping (post SPVC with switches inserted), thereby bounding the final delay penalty to that of the SPVC phase. The details of the rest of the procedure can be found in [32].

4.3.3 DAG mapping, Fake Inverter Bubble, and Mixed-V_t Inverters

We performed DAG mapping in this work instead of tree mapping. Our intuition was that the severing of DAGs into trees can significantly degrade the quality of the final result, so we stuck with the original DAG and followed heuristics from [32] while performing DAG mapping.

A heuristic which we found to boost the quality of the final remapping solution significantly was the use of fake inverter bubbles at every node in conjunction with the inclusion of a mixed-V_t inverter cell in the mapping cell library. The reason for this is because with fake bubble insertion, the flexibility in remapping improves substantially, but only if these bubbles are used in the final covering phase. Unfortunately, because inverters are inherently high in leakage cost, this will frequently prevent the covering process from choosing a DC-point that uses any more inverters than what it originally used. Hence, to remove this bottleneck in remapping, a low-leakage inverter cell must exist in the cell library. However, rather than using a high-V_t inverter cell, we used a mixed-V_t inverter instead due to the reasons documented in [33]. In short, it's because mixed-V_t inverters have a much lower delay penalty than that of a high-V_t inverter, but they still maintain a decent level of leakage reduction over that of the regular low-V_t version.

4.3.4 Re-identifi cation of a Partial Vector for the Remapped Sector

In remapping, we have found a new circuit structure which is statistically better than before. However, there are no guarantees when working with statistics, so in order to really ascertain that the remapped structure should be accepted over the old structure, we perform partial vector identification again, but this time on the remapped structure, then check to see if the resulting leakage state of this new structure with its new partial vector is better than the initial stored leakage state result. If yes, we accept this remapping and modify the netlist before moving on to the next sector. If not, we reject this remapping decision, keep the original structure for this sector, and move on to remap the next sector right away.

4.3.5 Node Updating and Switch Insertion/Deletion Post-Remapping

If a remapped decision is accepted, then we must not only modify the netlist, but we must also update various timing information on different nodes (requiredTime, arrivalTime) so that when the remapping for the next sector begins, it will have an up-to-date and correct delay information to base its final DC-point covering decision on. Furthermore, with each remapping acceptance, some internal switches could be added or deleted to/from the original netlist. This is because in using a new remapped structure with its new partial vector, the logic value at the output nodes of that remapped sector can change from before, and so since we may have previously inserted a switch for one or more its fanout sectors during SPVC, we must now check to see if these switches are still needed with the new logic value at that node. Similarly, we may need to insert new internal switches given that the logic values at the outputs may have changed.

Previously, we said that during covering, we would pick a DC-point at an output node which is $\leq$ the requiredTime of that node. However, because after remapping we may need to introduce new switches at that node, this can change and possibly violate the delay-constraint we set out to achieve. Therefore, to accommodate for this source of perturbation, we make a slight modification of our original method. That is, for each output node, we pick a DC-point which is $\leq$ (requireTime - ϵ), where ϵ is an delay adjustment factor to take into account possible insertion of switches post-remapping. Taking ϵ into account allows us to find a remapped solution which is guaranteed to not violate the critical delay constraint of the post-SPVC circuit. If no point on the curve satisfies this constraint, then we keep the original structure.

The entire technology-remapping algorithm has now been described. We summarize it in Algorithm 4.

4.4 Experimental Results

We conducted our experiments on a machine with 320MB of Ram and running on an Intel Pentium2 600Mhz processor. 70nm technology was used in our SPICE simulations. All of

```
Apply SPVC on the target circuit;
Sort the sectors into remappingQueue;
/* The queue is ordered from worst to best-controlled */
while remappingQueue is not empty do
    currentSector = pop(remappingQueue);
    generateAllDCMatchPoints(currentSector);
    /* Use fake inverters, mixed-Vt, DAG mapping */
    pickBestDCCover(matchList, originalDelayConstraints, $\epsilon$);
    newVect = partialVectorIdentification(remappedStructure);
    if newLeakageState(remappedStructure, newVect) $<$ originalLeakageState then
        modifyNetlist();
        insertDeleteSwitches(fanoutSectors);
        updateTiming(allNodes);
    else
        keepOriginalStructure();
    end
end
```

Algorithm 4: Simultaneous Technology Remapping with SPVC

the SPICE decks were generated as a single whole circuit with the internal switches embedded within, the pins re-routed, and the global IDLE signal set to TRUE. Two off-the-shelf packages, a BDD [34] and a MLFM partitioner [35], were used in this work. We implemented all of our algorithms as well as our own technology mapper in C. The benchmarks used in this work were taken from the ISCAS and MCNC suites. Our data will show why our technique is scalable beyond these small circuits.

We begin by tabulating in Table 4.2 the leakage reduction effectiveness of the two techniques presented in this paper. α and ϵ were picked judiciously. Due to space limitation, only one arbitrary choice of the sector count is shown. In the third column, we see the degree of leakage control which can be exerted by SPVC alone. The numbers indicate that on average, SPVC can achieve 10-30% reduction, which is not significant but certainly not trivial either. However, what we want to emphasize is the extremely fast runtime of SPVC. The importance of this speed cannot be stressed enough, especially when taking into account the data shown in the last two columns. As it can be seen, technology remapping with SPVC is a very powerful technique for leakage reduction, especially considering the fact that no delay penalty is incurred in upgrading from SPVC. However, because this technique involves finding a partial vector for each sector twice, its runtime, and therefore its general practicality, critically depends on the speed of SPVC. Therefore, the direct observation Table 4.2 provides for the low runtime requirement of SPVC boosts the viability of our proposed technique of technology remapping + SPVC.

No technique is ever perfect, and the drawbacks to our approach is that in inserting the internal switches, additional overhead is incurred in area, delay (if it's on the critical path), and dynamic power (There will also be some perturbations due to the remapping routine, but this can be either positive or negative). These penalties are now shown in Table 4.3 for the same set of parameters used in Table 4.2 (# of sectors, α, etc.). Due to space limitation, we only show the penalties associated with the combination of SPVC and technology remapping and for only the larger circuits. This penalty naturally encompasses the penalty incurred via SPVC alone. The values are computed from a well-calibrated library model file, ”lib2.genlib”,

Table 4.2 Leakage Reduction Results using SPVC with Technology Mapping

	i_{leak} Original Circuit (uA)	# of Sect.	i_{leak} SPVC (uA)	Run-Time SPVC (secs)	i_{leak} SPVC + Tech. Remap. (uA)	Run-Time SPVC + Tech. Remap. (secs)
i1	19.73	5	13.18	2	1.96	3
i2	70.66	10	21.51	10	8.69	21
i3	62.70	10	36.22	3	34.10	11
i4	162.34	10	115.09	12	28.55	34
i5	98.21	10	69.25	2	12.19	9
i6	235.78	15	185.27	23	58.49	58
i7	310.97	20	178.86	41	76.07	95
i8	596.69	40	386.90	93	113.88	186
i9	245.02	25	131.44	23	41.11	66
i10	1110.00	80	800.57	190	347.57	511
c432	111.92	10	78.87	3	7.65	10
c499	249.76	10	222.75	9	102.18	27
c880	199.17	10	144.51	16	67.29	48
c1355	252.93	10	224.96	10	99.57	29
c1908	231.58	20	192.97	7	78.79	22
c2670	309.82	20	237.98	20	160.30	85
c3540	530.82	40	398.12	61	187.33	158
c5315	715.96	50	572.99	109	327.26	265
c6288	1430.00	80	1260.00	90	699.83	317
c7552	1080.00	50	928.56	79	537.48	214

from SIS [36] using the standard load-based delay model. Dynamic power consumption was found with SPICE over a ten cycle period. As it can be seen, some tradeoffs must be made in choosing to apply SPVC.

Some heuristics could probably be used to reduce the penalty shown in Table 4.3. For example, one could try to insert a switch only on the non-critical paths. However, in doing so, the runtime would be burdened, not the mention the fact that many complications will be introduced during the remapping process (since each sector will no longer be able to be remapped independently of each other). Therefore, we leave the strategy as it is.

Next, we examine the impact that varying the number of sectors formed has on the the quality of the final result. We also study the correlation between the sector count and the overhead fluctuation. This data is shown in Table 4.4. Only SPVC is shown because its effects propagate to SPVC + technology remapping. As it can be seen, the behavior reflects what we expect to see: the greater the # of partitions, the more granular the sub-circuits will be, and consequently the better the leakage control and runtime will be. The price to pay is that there will be a greater overhead addition due to more internal switches being inserted.

The data in Table 4.4 back our claim that our technique is scalable for circuits of any size. This is because by reasoning from the correlation observed between choosing a sector count and the resulting degree of leakage control effectiveness, we argue that no matter how large a circuit gets, we can simply introduce more partitions if we need to in order to achieve a satisfactory level of leakage control. However, as shown in Table 4.4, even though we may have to introduce more partitions for larger circuits, this does not naturally imply that we will incur more delay penalty. This is because unlike the area and power penalty, which is directly proportional to the number of internal switches inserted, the delay is a complex function of where the switch is inserted (on critical path or not) and how the circuit is partitioned, which can in turn dictates whether natural convergence of values takes place or not. Therefore, just because we are partitioning to more sectors doesn't mean that our critical delay will inevitably suffer. In fact, as we see from the data, sometimes a larger number of sectors can actually lead to better performance (with respect to a previously chosen smaller sector count).

Table 4.3 Area, Delay, and Dynamic Power Overhead of SPVC

	Area of Original Circuit	Area with SPVC + Tech. Remap.	Delay of Original Circuit	Delay with SPVC + Tech. Remap.	Dynamic Power of Original Circuit	Dynamic Power w/ SPVC + Tech. Remap.
i6	592528	599488	9.11	11.2	7.22e-4	7.34e-4
i7	691360	734512	9.95	12.18	9.28e-4	1.00e-3
i8	1309872	1485264	15.78	19.22	2.23e-3	2.54e-3
i9	629184	747504	16.10	21.29	8.62e-4	1.07e-3
i10	2730176	3126896	49.01	60.78	3.74e-3	4.44e-3
c2670	876496	960016	23.67	30.48	1.41e-3	1.55e-3
c3540	1328432	1556720	40.71	49.28	2.44e-3	2.84e-3
c5315	2072688	2294016	33.53	42.72	3.60e-3	3.99e-3
c6288	4069280	4319840	112.91	137.63	1.43e-2	1.47e-2
c7552	2895824	3163088	36.98	45.47	6.78e-3	7.25e-3

Table 4.4 Impact of varying the sector count on SPVC

	# of Internal Switches Added	i_{leak} with SPVC	Runtime of SPVC	% of Area Increase From Original Area	% of Delay Increase From Original Delay	% of Dynamic Power Increase From Original Power
50	196	549.68 uA	117 secs	9.4 %	20.28 %	7.14 %
60	210	525.18 uA	104 secs	10.1 %	13.35 %	7.65 %
70	238	500.46 uA	101 secs	11.4 %	21.76 %	8.67 %
80	262	499.01 uA	100 secs	12.5 %	19.38 %	9.54 %
90	275	484.23 uA	96 secs	13.2 %	18.79 %	10.01 %
100	292	463.58 uA	94 secs	14.0 %	21.17 %	10.63 %

4.5 Summary

In this chapter, we presented two novel techniques in SPVC and leakage-aware technology mapping for alleviating the size and logic structure limitations of input assignment. Our results confirm the effectiveness of the two techniques, but some tradeoffs must be made in area, delay, and dynamic power consumption.

Chapter 5

Conclusion

In this thesis, we discussed two of the biggest challenges facing tomorrow's IC designers in leakage current/power and process variations. For process variations, a novel, variation-aware technique for simultaneous gate-sizing and multiple-V_t assignment was proposed based on generalized Lagrangian Relaxation. For leakage control, a method called Sectoral Partial Vector Control with leakage-aware technology mapping was introduced. Both proposed works showed promising results when tested on benchmark circuits. We believe that together, these two techniques can serve as a comprehensive and effective circuit optimization framework for designs in future technologies.

LIST OF REFERENCES

[1] A. Grove in *International Electron Devices Meeting (IEDM)*, September 2002.

[2] S. Stiffler, “Optimizing performance and power for 130 nanometer and beyond,” in *IBM Microelectronics Technical Report*, June 2003.

[3] J. A. Mandelman and J. Alsmeier, “Anomalous narrow channel effect in trench-isolated buried-channel p-mosfets,” *IEEE Electronic Devices Ltr.*, vol. 15, no. 12, 1994.

[4] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De, “Parameter variations and impact on circuits and microarchitecture,” in *Design Automation Conference*, June 2003.

[5] S. Nassif, “Modeling and forecasting of manufacturing variations,” in *5th International Workshop on Statistical Metrology*, June 2000.

[6] X. Bai, C. Visweswariah, P. Strenski, and D. Hathaway, “Uncerty-aware circuit optimization,” in *Design Automation Conference*, pp. 58–63, 2002.

[7] C. P. Chen, C. C. N. Chu, and D. F. Wong, “Fast and exact simultaneous gate and wire sizing by lagrangian relaxation,” *IEEE Transactions on Computer-Aided Design*, vol. 18, pp. 1014–1025, July 1999.

[8] J. Cong and L. He, “An efficient technique for device and interconnect optimization in deep submicron designs,” in *International Symposium on Physical Design*, pp. 45–51, 1998.

[9] J. P. Fishburn and A. E. Dunlop, “Tilos: A posynomial programming approach to transistor sizing,” in *International Conference on Computer-Aided Design*, pp. 326–328, November 1985.

[10] S. S. Sapatnekar, V. B. Rao, P. M. Vaidya, and S. M. Kang, “An exact solution to the transistor sizing problem for cmos circuits using convex optimization,” *IEEE Transaction on Computer-Aided Design*, vol. 12, pp. 1621–1634, November 1993.

[11] H. Tennakoon and C. Sechen, “Gate sizing using lagrangian relaxation combined with a fast gradient-based pre-processing step,” in *International Conference on Computer-Aided Design*, pp. 395–402, 2002.

[12] A. R. Conn, P. K. Coulman, R. A. Haring, G. L. Morrill, and C. Visweswariah, “Jiffytune: circuit optimization using time-domain sensitivities,” *IEEE Transactions on Computer-Aided Design*, vol. 17, pp. 1292–1309, December 1998.

[13] A. R. Conn, I. M. Elfadel, W. W. M. Jr., P. R. O’Brien, P. N. Strenski, C. Visweswariah, and C. B. Whan, “Gradient-based optimization of custom circuits using a static-timing formulation,” in *DAC*, pp. 452–459, 1999.

[14] K. Nose, M. Hirabayashi, H. Kawaguchi, S. Lee, and T. Sakurai, “Vth-hopping scheme to reduce subthreshold leakage for low-power processors,” *IEEE Journal of Solid-State Circuits*, vol. 37, pp. 413–419, March 2002.

[15] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, “1-v power supply high-speed digital circuit technology with multithreshold-voltage cmos,” *IEEE Journal of Solid-State Circuits*, vol. 30, pp. 847–854, August 1995.

[16] E. Macii, “Leakage power minimization in deep-submicron cmos circuits,” www.sti.uniurb.it/bogliolo/ didattica/progel/Seminario1.pdf.

[17] Y. Ye, S. Borkar, and V. De, “A new technique for standby leakage reduction in high-performance circuits,” in *Intl. Symp. VLSI Circuits Dig. Tech. Papers*, pp. 40–41, 1998.

[18] D. Lee and D. Blaauw, “Static leakage reduction through simultaneous threshold voltage and state assignment,” in *Design Automation Conference*, pp. 191–194, 2003.

[19] M. C. Johnson, D. Somasekhar, and K. Roy, “Models and algorithms for bounds on leakage in cmos circuits,” *IEEE Transaction on Computer-Aided Design*, vol. 18, pp. 714–725, June 1999.

[20] F. Aloul, S. Hassoun, K. Sakallah, and D. Blaauw, “Robust sat-based search algorithm for leakage power reduction,” in *International Workshop on Power and Timing Modeling, Optimization and Simulation*, 2002.

[21] S. Narendra, S. Borkar, and etc., “Scaling of stack effect and its application for leakage reduction,” in *Intl. Symp. Low Power Electronics and Design*, pp. 195–200, 2001.

[22] T. Karnik, Y. Ye, and etc., “Total power optimization by simultaneous dual-vt allocation and device sizing in high performance microprocessors,” in *Design Automation Conference*, pp. 486–491, 2002.

[23] K. Kasamsetty, M. Ketkar, and S. S. Sapatnekar, “A new class of convex functions for delay modeling and their application to the transistor sizing problem,” *IEEE Trans. CAD*, vol. 19, pp. 779–788, July 2000.

[24] P. E. Gill, W. Murray, and M. A. Saunders, “Snopt: An sqp algorithm for large-scale constrained optimization,” numerical analysis report 97-2, Department of Mathematics, University of California, San Diego, La Jolla, CA, 1997.

[25] J. Nocedal and S. J. Wright, *Numerical Optimization*. Heidelberg, Berlin, New York: Springer Verlag, 1999.

[26] Lawrence, C., Zhou, J. L., Tits, and A. L., “User’s guide for cfsqp version 2.4: A c code for solving (large scale) constrained nonlinear (minmax) optimization problems, generating iterates satisfying all inequality constraints,” tech. rep. tr-94-16r1, Institute for Systems Research, University of Maryland, College Park, MD, 1996.

[27] A. R. Conn, N. Gould, and P. L. Toint, “Global convergence of a class of trust region algorithms for optimization with simple bounds,” *SIAM J. Numerical Analysis*, vol. 25, pp. 433–460, 1988.

[28] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu, “A limited memory algorithm for bound constrained optimization,” technical report nam-08, Northwestern University EECS, 1994.

[29] K. Roy, S. Mukhopadhyay, and H. Mahmoodi-Meimand, “Leakage current mechanisms and leakage reduction techniques in deep-submicrometer cmos circuits,” in *Proceedings of the IEEE*, vol. 91, pp. 305–327, February 2003.

[30] S. Sait and H. Youssef, *VLSI Physical Design Automation*. World Scientific, 1999.

[31] R. E. Bryant, “Graph-based algorithms for boolean function manipulation,” *IEEE Transaction On Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[32] K. Chaudhary and M. Pedram, “Computing the area versus delay trade-off curves in technology mapping,” *IEEE Transaction on Computer-Aided Design*, vol. 14, pp. 1480–1489, December 1995.

[33] Q. Wang and S. Vrudhula, “An investigation of power-delay trade-offs for dual vt cmos circuits,” in *International Conf. on Computer Design*, pp. 556–562, 1999.

[34] CUDD: http://vlsi.colorado.edu/∼fabio/CUDD.

[35] hMetis: http://www-users.cs.umn.edu/∼karypis/metis/hmetis.

[36] SIS: http://www-cad.eecs.berkeley.edu/Software/software.html.