DESIGN AND OPTIMIZATION OF PROCESSORS FOR ENERGY EFFICIENCY: A JOINT ARCHITECTURE-CIRCUIT APPROACH

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Omid Jalal Azizi

August 2010

This dissertation is online at: http://purl.stanford.edu/bv314tv4233

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christoforos Kozyrakis, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Stephen Boyd

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Power consumption limits have changed how processors are designed today: designers now need to be very careful to use their energy budgets efficiently. While a design option may be used to increase performance, it usually comes at an energy cost. Thus, to create a truly efficient processor, the designer must consider the space of all available options and parameters and make those design choices that offer the best rate of return in terms of performance per unit energy. Performing this efficiency optimization presents several challenges. First, the design space of a processor can be very large, and designers need a way of exploring this multi-dimensional space effectively. Second, the costs of the architectural units are directly dependent on the circuits that are used to implement them; these circuits, however, have a design space of their own and can internally trade off speed for energy. As a result, to properly evaluate the cost of an architectural feature, designers need to be aware of the circuit-level design spaces as well.

This work presents an integrated optimization framework that addresses these challenges by performing a co-exploration of the architectural and circuit-level design spaces. In this framework, we model large architectural design spaces by using statistical sampling and fitting techniques, and we characterize circuit design trade-offs for underlying units which we store in a library. We then link these two design spaces together to create a joint architecture-circuit model. By using posynomial functions for our models, we are then able to form a geometric program optimization problem

that we can solve efficiently with convex solvers.

The resulting optimization framework enables a designer to optimize large design spaces for various design objectives and under different resource constraints, including both area and energy. It identifies, for a given design problem specification, the optimal design parameters in both the architectural and circuit domains. The tool can also be used to map out cost-performance trade-offs within a design space, allowing the designer to select the design that best meets his needs.

We apply this framework to study energy-performance trade-offs in general purpose processor design. We consider six different high-level architectures, from a simple single-issue in-order processor to a quad-issue out-of-order processor. Optimizing across this space for different performance targets, we identify the order in which high-level architectural features should be considered as one seeks more performance. Starting from a single-issue in-order processor for low energy budgets, our results show that to increase performance efficiently, a designer should consider first increasing issue width to two, then adding an out-of-order execution engine, and ending with a further increase in issue width to end with the quad-issue out-of-order processor for very high-performance targets.

Adding voltage scaling changes these results dramatically. Our results show that architecture and circuit design techniques have a rapidly changing marginal cost profile, with many options having either very low or very high marginal costs. Since the marginal cost of obtaining additional performance through voltage changes more slowly, we find that when we optimize the system jointly with voltage, the set of efficient architecture and circuit design features become confined to a small sweet spot for a large part of the design space. Thus, when optimized with voltage as a parameter, two high-level architectures—the dual-issue in-order processor and the dual-issue out-of-order processor—are efficient over almost the complete range of performance targets.

Acknowledgments

As my time as a student at Stanford comes to an end, I have had the chance to reflect on the journey and think back to the many people who have supported me along the way. The journey has been full of learning and growth, and I owe a great deal to the support and friendship of a large number of people who have made this time so special and enriching.

First, and foremost, I have the deepest gratitude to my advisor, Professor Mark Horowitz. Working under the supervision of Mark has been a true blessing, and I thank Mark for all his guidance and patience during my research. I have learned a great deal from Mark, not only in the field of my research, but in so many other ways as well. I could not have hoped for a better advisor, and I will forever be indebted to him.

I would also like to thank my reading committee members. Professor Christos Kozyrakis was my first faculty contact at Stanford. I took a number of his classes, which I thoroughly enjoyed, and I want to thank him for all his feedback on this work. To Professor Stephen Boyd, I am equally thankful. Stephen is a wonderful teacher, and I am thankful for my various discussions with him regarding the optimization side of my research. I have always found my discussions with him insightful.

I am also grateful to a number of collaborators over the years. I had the pleasure of working with Aqeel Mahesri and Professor Sanjay Patel from the University of Illinois at Urbana-Champaign. Their support helped tremendously in moving my research

forward. I also had an internship at Intel Corporation. For this opportunity, I thank Hong Wang, who brought me on board, and Jamison Collins, who encouraged me to publish some of my early research ideas.

Next, I want to thank Benjamin Lee. I first met Ben at Intel, and then again when he came to Stanford as a post-doc. I shared an office with Ben at Stanford, and always enjoyed our discussions. Ben has been a good friend, and I wish him the best of luck as he embarks on his own journey as a professor.

I also owe a lot to the members of the Horowitz research group during my years at Stanford. This includes Dinesh Patil, with whom I worked closely as a new graduate student, and the members of the chip generator group: Ofer Shacham, Megan Wachs, Sameh Galal, Zain Asgar, Wajahat Qadeer, Rehan Hameed, Pete Stevenson, Amin Firoozshahian, Alex Solomatnikov and Stephen Richardson. More than being research collaborators, they have been good friends.

There are also a number of support staff and funding organizations to which I am grateful. First, I must thank Teresa Lynn who managed the administrative affairs of the Horowitz research group. She was always there to help, and did so with a smile. Second, I must thank Charlie Orgish for his administration of the computer systems. Lastly, I must also thank the FCRP's Center for Circuits & Systems Solutions (C2S2) for funding my research.

I cannot forget the many friends outside my research field who have contributed equally to life during this period. There are too many to adequately thank in this small space, so I must restrict myself to naming only a few. I thank Nikhil Ravi and William Wu for all our stimulating and enriching discussions over the years, Prashant Loyalka for his wisdom, Nong Thavisomboon for always listening and being a good friend I could count on, Avni Morjaria for her kind heart and for being a true friend, Chris Cardoso for his friendship and our stimulating debates. To all of these friends—and the many more I have left unnamed—I am thankful for their friendship

and support.

Furthermore, I would be remiss not to thank the Stanford Baha'i community. Here, in particular, I am grateful to the Talebi family. Their hospitality and generosity knows no bounds. The home was always open, and they always made sure that I felt at home. I thank them sincerely.

My final year at Stanford has also been a very special time for me as it blessed me with meeting Roxana. She has made my life truly full of happiness during this time, to a point that I did not think was possible. I love her for always being there for me, for all the wonderful adventures we have together, and for her bringing out the best in me. I thank her for all her support during the writing of this thesis.

Finally, I truly owe everything that I have achieved to my family. There are really no words to adequately express my gratitude for all that they have done. I thank my parents—my mother Shohreh and my father Fereydoon—from the bottom of my heart for their never-ending love and their endless support, for raising me and teaching me wrong from right, for providing me with an education, for instilling a love for science in me, for always gently expecting the best from me, and for so much more. I am equally full of love and gratitude towards my brothers, Navid and Paymon. They are, as brothers, the closest people to me. I owe a great deal academically to Navid for paving the way to graduate school for me, from elementary school through to the PhD. As for Paymon, and his joyful and energetic spirit, I wish the best for him as paves his own path through higher education. To both, I thank them for being the ones who I know I can always to turn to, and who will always be there for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer system design is currently undergoing a significant shift. Historically, chip design was focused, for the most part, on maximizing performance within a constrained die area. However, changes in device characteristics brought on by continued technology scaling—in addition to the emergence of a large mobile computing market—have caused a fundamental shift in the design constraints: power consumption constraints now play a major role in defining how we design the silicon chips that drive our electronics.

This change in design constraints has had a significant impact on system design. With power considerations now limiting the achievable performance, designers can no longer apply all the aggressive design techniques they once used to; achieving a desired performance target requires careful management of the power budget. Moreover, with voltages no longer scaling with technology nodes as they once did, power constraints will become even more stringent in the future. Thus, regardless of whether one is designing low-power embedded mobile devices or high-performance servers, power consumption is now a critical factor in determining the system's overall performance.

In this new power-constrained era, the principal design objective is to achieve *energy efficiency*. Designers need to find ways to make the most of their power

budgets, and—in addition to finding new, more energy-efficient design techniques—this requires ways of exploring existing design spaces to enable designers to tune their systems for efficient operation.

The process of optimizing a design for energy efficiency requires that designers perform a systematic trade-off analysis; they need to consider the cost-benefit trade-offs of all design options, choosing those with the best returns. By choosing design features with low marginal costs (i.e. energy cost per unit performance) and staying away from design options with high marginal costs, a designer can produce a more efficient system that best uses the available power budget.

While performing this optimization is simple in theory, it is quite difficult to perform in practice because of the size of the design space at hand. The number of design options at a microarchitecture level alone can be very large, and if the designers truly want to optimize the whole system, they need to consider the circuits and technology parameters as well; each of these domains, by itself, is challenging to explore and optimize.

In the past, optimization efforts have focused mostly on each of these design spheres independently. Years of research in each of these fields have produced various optimization tools and studies, with the most recent of these focusing on power-performance optimization, but these tools typically optimize for a given layer only. To truly optimize the whole system, however, a designer needs to consider all levels of the hierarchy. The energy consumption of a microarchitecture, for example, depends directly on the energy consumption of the circuits that it uses, and any thorough optimization of the microarchitecture needs to consider the circuit design. Unfortunately, communication of design possibilities and constraints between the layers of the hierarchy are typically very limited, and current tools are generally unable to expose the entire realm of possibilities from one domain to the higher level domain. Thus, we find that existing microarchitectural power estimation tools generally only

use fixed energy costs per circuit (based on a single design) to perform their analysis, and are oblivious to the different circuit implementations available.

In this dissertation, we close this architecture-circuit modeling gap by creating the first general *circuit-aware* approach to architectural analysis, presenting an integrated architecture-circuit modeling and optimization framework. Our approach brings together and leverages recent advances in modeling methodologies to create a powerful yet flexible framework for optimizing digital systems: we apply statistical regression techniques to build models of large architectural spaces, and we leverage existing circuit optimization tools to characterize circuit trade-offs. We then integrate these models together to characterize a large joint architectural and circuit design space. Finally, by using posynomials—mathematical functions with a special form—for creating our models, we enable the use of powerful convex optimizers to search the joint architecture-circuit system.

The resulting framework allows the designer to systematically analyze design trade-offs in a large microarchitecture-circuit design space, evaluating marginal costs of design options and identifying the most attractive design features. Furthermore, because it applies a sample-and-fit approach to creating models, the framework is general and flexible. Existing system simulators—which should exist in any architectural design space exploration environment—are used to extract a relatively small set of design space samples from the very large space of possible designs, which the framework then uses to create models for large architectural design spaces. By joining these architectural models with circuit trade-offs—which can be generated using the designer's tool of choice—the framework is able to explore the joint architecture-circuit space of energy-performance trade-offs to find the most energy-efficient designs.

We use this framework to optimize processor designs for energy efficiency. In our study, we consider various high-level architectures, from a simple in-order processor to an aggressive quad-issue out-of-order processor, optimizing not only the numerous

microarchitectural design knobs within each of these designs, but optimizing down to the circuit level as well.

Our results show that performing a joint optimization results in more energy-efficient designs over traditional architecture-only optimizations. While a fixed-cost approach is constrained to a single design and cannot adapt to different design objectives, a circuit-aware approach allows the optimizer to select the appropriate circuit based on the particular needs of the architecture and the design objective specified by the user. Slower, lower-energy circuits are used in parts of the design where the marginal cost savings warrants it, and faster, higher-energy circuits are used where performance is critical. By empowering the optimization with more circuit choices, the optimization framework is able to achieve significant energy savings and performance benefits over the entire energy-performance space.

Finally, we study the designs in the overall processor design space to show how the marginal cost profile of the joint architecture and circuit design space changes fairly rapidly: many design options turn out to be either very cheap or expensive, and the transition between these two extremes occurs quickly. Contrasting this behavior to the behavior of voltage scaling—an equally powerful means of trading-off energy and performance—we see that achieving performance through voltage has much more stable marginal costs. Because of voltage scaling's attractive marginal cost profile, this result suggests that voltage scaling is often an efficient means of trading-off energy for performance. Thus, we have found that applying the principles of marginal costs in a disciplined fashion results in a surprisingly small sweet spot of architecture/circuit design points being interesting for energy efficient operation when voltage scaling is available as a design knob.

We begin our discussion in the next chapter with some background on the power crisis that has led to the importance of energy efficiency, and modeling and optimization efforts, especially in the processor design space. We then strengthen our

case for a systematic framework for processor optimization that can look at multiple layers of the design hierarchy simultaneously. In Chapter 3 and Chapter 4, we subsequently present the details of our circuit-aware architectural modeling and optimization framework. Finally, in Chapter 5 we present the results of applying this framework to the processor design space, examining the advantages of performing joint circuit-architecture optimization and evaluating marginal costs in the processor design space.

Chapter 2

Background

The design of a digital electronic system, like the design of any engineered system, is an optimization problem: the designer has various resources (many of which may be limited and/or costly), design constraints, and a design objective that needs to be maximized or minimized. Faced with this problem, the task of the designer is to find a solution that best achieves the design goals.

Historically, the optimization objective in chip design was to produce chips that provided the most performance at reasonable production costs (i.e. chip area). More recently, however, changes in technology scaling have resulted in more restrictive power constraints that are changing the optimization problem. Whereas power dissipation used to be a secondary concern for high-performance designs, designers now face hard power constraints that significantly limit the system. These constraints directly affect the amount of computation a design can perform, and, therefore, the performance that can be achieved. In this new era of design, power is no longer a consideration for only low-power embedded devices; rather, design for energy efficiency is critical for all designs.

In this chapter, we examine the general problem of designing and optimizing systems for energy efficiency. We first discuss the fundamental causes of the power

constraints, showing why design for energy efficiency is so important today. We next examine the general principles of design in such a power-constrained world. We then consider the particular challenges in modeling and optimizing a complex digital system such as a processor, ending with an overview of our optimization framework that proposes to solve these challenges.

2.1 Power-Constrained Design & Energy Efficiency

In the past, chip designers could traditionally count on technology scaling to continually improve their designs. With each successive technology generation, technology scaling produced better devices that achieved gains on virtually all fronts: transistors got smaller, transistors got faster, and transistors also consumed less energy per switching activity. In addition to all these benefits, by following Dennard's constant-field scaling [19]—in which one scales supply voltage along with feature sizes—designs were able to maintain constant power densities throughout these scaling generations. The future of digital design—although not without challenges—was, at the time, looking good.

As scaling continued into feature sizes below 130 nm, however, technology scaling faced some new challenges in rising leakage currents. The fundamental problem, which Dennard had noted in his paper, was that the thermal voltage, kT/q, does not scale with technology nodes; this means that transistor leakage currents grow exponentially as threshold voltages are lowered [35]. Unfortunately, below the 130 nm node, technology scaling had finally reached the point where transistor leakage currents threatened to become a considerable source of power dissipation, and device technologists had little choice but to dramatically reduce the scaling of transistor thresholds in order to keep leakage power in check. This, however, also prevented further supply voltage scaling, as lowering the voltage without changing thresholds

would reduce the voltage overdrive, resulting in significant performance losses. Today, as we move to the 32 nm technology node, supply voltages remain at around 1 V, roughly where they were for the four previous technology generations.

The inability to practically scale voltages has considerable consequences for power consumption in designs when scaling. Traditionally, scaling by a factor α (where α is a unitless scaling factor; $\alpha < 1$) would cause the energy to perform an operation to scale with α^3 ($E = CV^2$; C and V scale with α), but with voltages no longer scaling, that energy now only scales with α. As area continues to scale with α^2, and the frequency of a fixed design now scaling at a factor between 1 and $\frac{1}{\alpha}$ [28], power densities are expected rise, scaling by between $\frac{1}{\alpha}$ and $\frac{1}{\alpha^2}$ (power density $= \frac{E \times f}{A}$), a significant increase over the constant power densities that came with constant-field scaling. The resulting situation is that to continue scaling—which is still driven by the economics of chip production—designing within power dissipation limits will become increasingly difficult.

The situation becomes even more dire when one considers that, even before the changes in technology scaling, designers were already facing rising power consumption envelopes. As Figure 2.1 shows, processor power consumption has been steadily rising over the years; ever since the early 2000s, commercial microprocessors have hit a "power wall" and have been operating at the limits of air cooling. What is interesting to note is that this figure shows increasing power levels despite Dennard's scaling rules, which suggest that processors should maintain constant power densities. The reason for this apparent contradiction is that Dennard scaling assumes that a design remains unchanged through scaling generations; a fixed design is scaled, resulting in a smaller area, lower power, but constant power density. Designers, however, have not been content with relying on scaling alone; rather, they have continually advanced their designs, using resources such as extra area and power to add more features and create more complex designs. Because of these more aggressive designs, we have witnessed

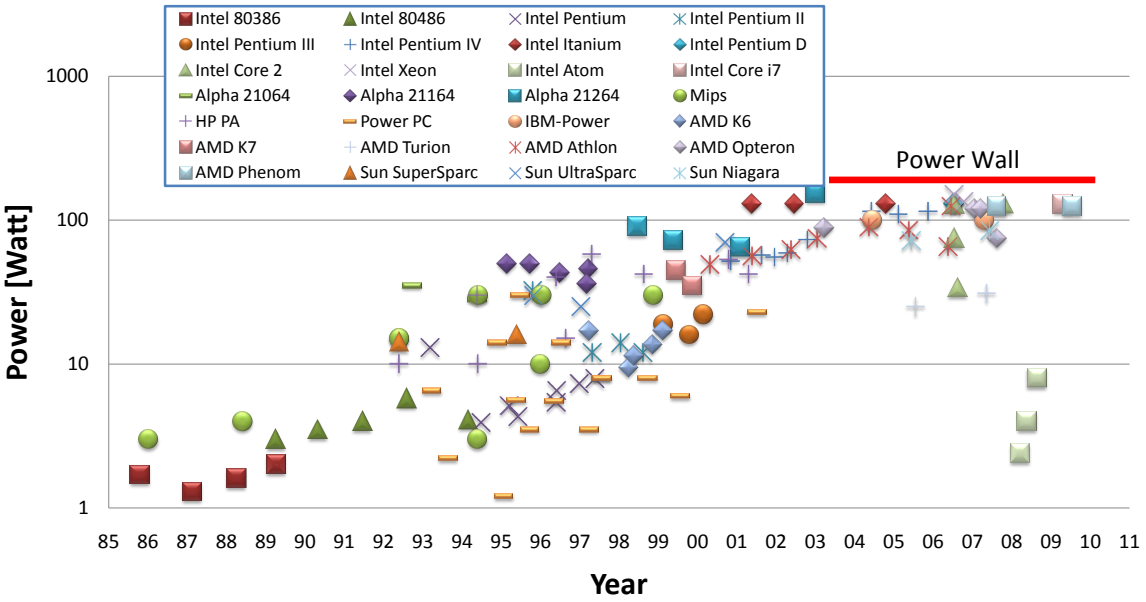


Figure 2.1: Plot of the power consumption of various commercial processors over the past 25 years. Power consumption has been steadily rising, hitting a "power wall" at approximately 130 Watts. High-performance designs today are now constrained by this power dissipation limit.

a relative increase in the number of transistors switched per cycle; the use of deeper pipelines with higher frequencies than what Dennard predicted, and more aggressive design features, in addition to some growth in die area, has led to a steady increase in power. While this effect is separate from the increase in power that is caused by the slow-down in voltage scaling—which is a more recent phenomenon—it makes the problem of designing within power limits even more difficult as we continue to move forward.

2.1.1 Power and Energy

With power constraints being so stringent, designers today need to manage their power levels carefully. Examining the definition of power,

$$\text{Power} = \frac{\text{energy}}{\text{second}} = \frac{\text{energy}}{\text{operation}} \times \frac{\text{operations}}{\text{second}} \tag{2.1}$$

we find that there are two primary means by which a designer can reduce power consumption. The first is to reduce the number of operations per second. This approach, however, simply reduces performance to save power. It is analogous to slowing down a factory's assembly line to save electricity costs; although power consumption is reduced, the factory output is also reduced and the electricity used per unit of output remains unchanged. If, on the other hand, a designer wishes to maintain—or even improve—performance under a fixed power budget, a reduction in the fundamental energy per operation is required. It is this reduction in energy, not power, that represents real gains in efficiency.

This distinction between power and energy is an important one. Even though designers typically face physical *power* constraints that stem from power supply and dissipation issues (e.g. heat dissipation), to increase efficiency (i.e. performance per unit power) requires that the fundamental *energy* of operations be reduced. While one could consider power-performance trade-offs, this can be misleading because power is a rate of energy consumption (watts = joules/second) and is directly affected by the performance; what may seem like a trade-off may just be a modulation in performance resulting in changes in power consumption. In fact, for this reason, it is often easy to achieve some trade-off between performance and power; simply decreasing the clock frequency (without changing voltage),[1] for example, results in both less power and

[1] If one also scales voltage as is done in dynamic voltage and frequency scaling (DVFS), then this *does* represent an increase in energy efficiency. The increased energy efficiency, however, essentially comes from the voltage scaling.

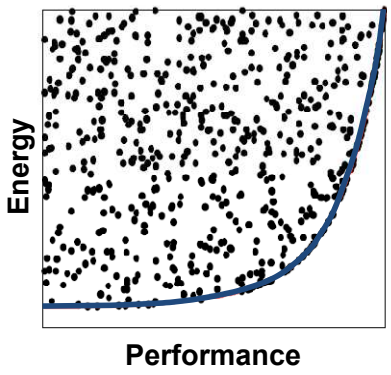


Figure 2.2: Illustration of the energy-performance trade-off space. As a designer considers all possible design implementations, an efficient frontier can be identified. Designs not on the frontier are inefficient, because there exist other designs that can achieve the same performance using less energy. Designs on the frontier, on the other hand, are optimal for their given level of performance. This frontier also demonstrates the fundamental trade-off between energy and performance, with higher performance designs requiring higher energy. In optimization for energy efficiency, the designer seeks to find the design configurations that lie on this frontier.

performance, but this does not reflect any improved efficiency. Thus, even though the designer may be facing a power constraint, it is energy per operation that is the more meaningful metric to use when evaluating the efficiency of a design.

2.1.2 Energy-Performance Trade-offs & the Efficient Frontier

With energy per operation (not power) as the primary cost metric for energy efficiency, design then becomes a process of evaluating the trade-offs between energy per operation and the achievable performance. To find a good design, a designer must consider different design alternatives; each potential design the designer evaluates will offer a certain amount of performance and will also consume a certain amount of energy. By comparing how well the different designs meet the design objectives, the designer can gradually move towards a more optimal design.

The resulting design optimization problem is illustrated in Figure 2.2. Here, the

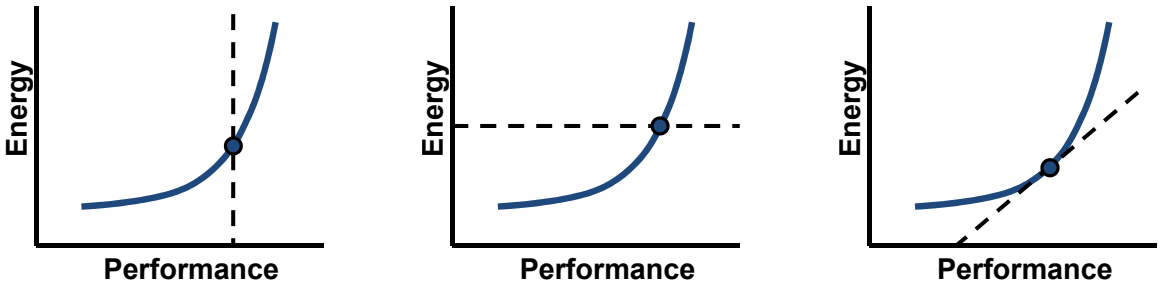
design space exploration process is extended over a large number of designs, and the performance and energy per operation of each design is plotted in the performance-energy space. In the context of this figure, the goal of the designer is to choose a design as far to the right (higher performance) and to the bottom (lower energy) as possible.

Examining the figure, we first note that many of the design points in the space are inefficient: there exist other design points which have at least the same performance, but with lower energy. If performance and energy are the only considerations, these designs should never be used. If we ignore all these inefficient points, then what remains is the set of design points that form the energy-efficient frontier (also referred to as the Pareto-optimal curve), which is represented in the figure with a curve. The designs on this frontier constitute the most energy efficient design points for different performance targets.[2]

This frontier demonstrates the fundamental trade-off between energy per operation and performance: to get higher performance, one needs to use more aggressive designs that require more energy. One can use a higher energy, higher performance design or choose a lower performance design with a lower energy cost. There is no single optimal design, but rather an entire set of optimal designs. In this context, it should be noted that using a higher energy design point does not necessarily mean that it is inefficient; as long as it is the best design for its performance level, it is still referred to as energy-efficient.

Determining the particular design point that is best for a given design problem is a matter of design objectives and possibly the energy cost one is willing to pay to get performance. If the design problem is to achieve a performance target with the least energy, the optimal design will be at the intersection of the frontier and a vertical line

[2]Equivalently, one can view these design points as those that maximize performance for different energy budgets.



(a) Performance Target (b) Energy Budget (c) ED^n Metric

Figure 2.3: Different optimization problems and their relation to the energy-efficient frontier: (a) represents the case of minimizing energy for a given performance requirement (vertical dotted line); (b) represents the case of maximizing performance for a given energy budget (horizontal dotted line). (c) ED^n metrics are popular alternative optimization metrics that minimize the design space with respect to the ED^n cost function. In a log-log plot, these cost function contours map to lines of slope n, as shown in this figure. ED^n metrics such as ED and ED^2 typically yield balanced designs, but do not directly map to specific performance targets or energy budgets.

representing the performance target; similarly, if the design problem is to maximize performance under an energy budget, then the optimal design is at the intersection of the frontier with the horizontal energy budget line. These scenarios are shown in Figure 2.3a and Figure 2.3b respectively.

One can also optimize for other metrics. For example, ED and ED^2 metrics are two commonly used metrics in design for energy efficiency. These ED^n metrics try to find a balanced design point by essentially setting a cost ratio between energy and performance; neither is allowed to be sacrificed too much for the other. The choice of n depends on how much one favors performance versus the energy cost. Graphically, optimizing for ED^n objectives means minimizing against a set of cost contours, which in a log-log energy-performance plot correspond to lines of slope n. As a higher value of n is chosen, higher sloped lines will cause a higher performance point to be tangent to the minimum cost line. One example is shown schematically in Figure 2.3c.

Practically, ED and ED^2 metrics are useful because they usually yield balanced designs. Strictly speaking, however, they can be viewed as somewhat arbitrary metrics in the sense that they do not consider a specific performance target or energy budget that needs to be met. Thus, it is unclear which design point an ED^n metric will produce with respect to these design specifications. It is also unclear which value of n the designer should use, since this is not restricted to be 1 or 2, but can be any real number (e.g. $n = 1.5$ could work as well).

For these reasons, we rely on the basic energy-performance curves to provide a more complete picture of the design space. Whereas optimizing for a particular metric produces only a single design, a trade-off curve exposes the entire space of possibilities to the designer and allows for a more complete evaluation of the trade-offs in the design space. Using performance requirements and design constraints, the designer can then choose which point on the curve best meets his needs. This is the approach we take when we examine energy-performance trade-offs in processors designs in Chapter 5.

We end this section by considering the energy-performance space of real microprocessor designs in Figure 2.4. This figure plots the energy per operation and performance of various historical processors—first normalizing them for voltage and technology scaling to make the comparison fair—and showing the energy-efficient frontier for real processors.

With respect to this energy-efficient frontier, design for efficiency involves two distinct, albeit related, challenges. On the one hand, designers are always seeking to *innovate* new design techniques that push this frontier out to achieve lower energy and higher performance. For example, one can view techniques such as power and clock gating, at the time that they were introduced, as increasing the design space of possibilities and extending the lower bound of the frontier to lower energy design

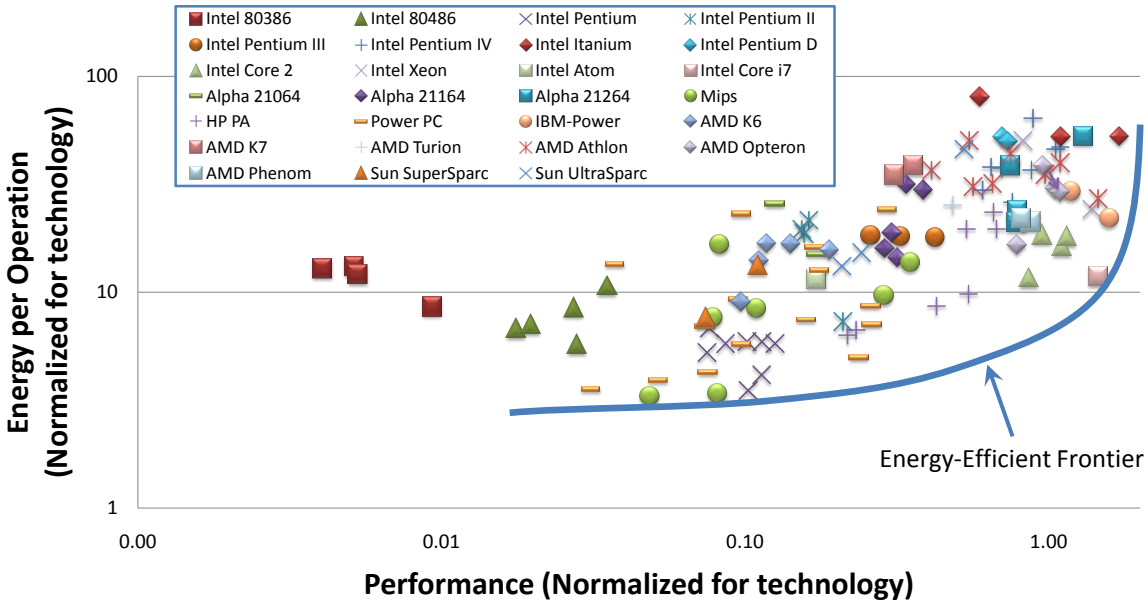


Figure 2.4: Plot of historical processors in the energy-performance space. Designs that maximize performance (to the right) and minimize energy (to the bottom) are preferred, resulting in an trade-off curve. Finding the designs on this energy-efficient frontier is the goal of design optimization. The particular design of choice depends on the designer's particular needs (i.e. energy budget or performance target).

points. Complementing innovation, we have on the other hand *optimization*.[3] This challenge, which is just as important as the first, is to search across an existing, often large, design space to actually identify how to achieve a design on the frontier. In this process, a designer needs to consider the space of all different design decisions and features. Then, by evaluating their performance and energy characteristics, and comparing them to each other, a designer can gradually optimize a design for more efficiency. It is this optimization challenge that we focus on in this dissertation. We examine this topic further in the next section.

2.2 Modeling and Optimization

To achieve energy efficiency in a design, a designer needs to consider all the different design options and features and must then evaluate them for their benefits and costs. By applying a trade-off analysis, where the benefits of a design decision are weighed against its costs, the more efficient design options can be identified, and a designer can begin to optimize his design.

Formalizing this process, we can denote the various different design parameters available to a designer with variables x_i. In the case of a processor system, for example, these x_i could represent design parameters such as cache sizes, pipeline depth and the issue-width (to name only a few). The critical metric for performing optimization is then the marginal cost, MC, of changing a design parameter, x_i:

$$MC(x) = \frac{\frac{\partial E}{\partial x_i}}{\frac{\partial P}{\partial x_i}} \tag{2.2}$$

Here, $\frac{\partial E}{\partial x_i}$ is the energy cost (or savings) of changing x_i, while the $\frac{\partial P}{\partial x_i}$ is the performance

[3] Many designers like to use the term "optimization" to mean innovation, but in this work, we distinguish the two terms and use "optimization" in the sense of design space exploration.

benefit (or loss) with respect to the same design decision.

Equipped with marginal costs, a designer can systematically optimize a system by selecting those design features with lower costs, and even exchanging those design features that have a higher cost for others that are cheaper. By following these optimization rules, a designer will methodically move towards a more efficient design. An important corollary of applying the principle of marginal costs is that all marginal costs should always match in an optimal design,[4] with an arbitrage opportunity otherwise existing in which the designer can *sell* the more expensive feature and then replace the lost performance by *buying* the feature with the lower cost. This observation will be critical in some of our analysis in later chapters.

While performing such a marginal cost analysis is conceptually simple, the real challenge lies in determining how a design change actually affects the performance and cost metrics. Thus, at its core, the evaluation of marginal costs and the optimization of a system is a modeling problem: we need models $P(..., x_i, ...)$ and $E(..., x_i, ...)$ that can predict how much the total performance and energy of the system will be affected by changing any design parameter, x_i. For example, if a designer wants to identify the optimal cache size for his system, then he needs models that can tell him how the performance and the energy of the entire system change with changing cache sizes. He can then use these model to evaluate different cache sizes.

Thus, at a high level, the entire design exploration process can be divided into two separate phases: modeling and optimization. In the first phase, the designer needs to develop models that accurately describe the system characteristics (e.g. performance, energy, area) as a function of the design space parameters and options. These models can be in analytical form or—as is often the case in system design—in the form of simulation models. Then, once the designer has acceptable models of the system, the

[4] Unless a design parameter is constrained such that the optimal marginal cost cannot be achieved; in this case, the design parameter should be at its limit.

designer needs to search the design space described by the models to find an optimal design; it is during this phase that the designer needs to evaluate the cost-benefit trade-offs.

Although the basic process is straightforward, there are challenges in both these phases. First, creating models of complex systems can be hard, especially when one considers that the space of designs that need to be modeled can be large and that the model evaluation time can become a critical bottleneck [34]. For example, while simulation models are common for system design, these are long-running (taking several hours to days to evaluate a single design) and can hinder the entire optimization process. We examine this problem and possible solutions in more detail in Chapter 3.

The second phase—searching the space to find an optimal design—can be as challenging as the first. The design space is often high-dimensional, growing exponentially with the number of parameters. Finding a good design can, therefore, be hard, the exponentially-growing size of the space making an exhaustive search intractable for most practical design spaces. Compounding this problem, the space may also have local optima, making the search more complex and making it difficult to know whether one has truly found the best design.

2.2.1 Optimization and Geometric Programming

To help with the design space exploration problem, numerous heuristic optimization techniques have been developed that try to find good designs. Algorithms such as simulated annealing and genetic optimization [17], for example, search the space while trying to avoid getting stuck in a local optimum. These approaches have shown to be effective for many problems—particularly problems with discrete design spaces—but still take a long time to perform their analysis and only probabilistically find the optimal design.

Fortunately, in certain cases, the form of the design space can be such that performing the optimization or design space exploration can be much more efficient. For example, one well known class of optimization problems which can be solved efficiently are linear programs [16], where the system can be described as a set of linear constraints. Using mathematical techniques that take advantage of the form of the problem, linear programs can be solved very quickly, even for very high-dimensional spaces.

Over the past several decades, optimization research has shown that linear programs are, in fact, only a subset of much larger class of optimization problems, termed convex optimization problems [9], that can be solved efficiently to find the global optimum. If a problem characterization can be shown to map to a convex optimization problem, then the search phase can be solved rapidly, with even problems with thousands of or more variables being solvable in minutes or hours.

Further optimization research has been able to expand the applicability of convex optimization to different types of problems. One important optimization class for various design problems, including this work, is the *geometric program* (GP) [8]. A GP is a kind of optimization problem with a special form that uses particular functions known as *posynomials* to describe the characteristics of the system. A posynomial is a mathematical expressions consisting of the sum of any number of positive monomial terms, where monomials are the product of powers of variables. For example, $kx^a y^b z^c$ is a monomial in the variables x, y and z (with k, a, b, and c as constants), while $k_1 x^a y^b z^c + k_2 x^d$ with $k_1 \geq 0$, $k_2 \geq 0$, is a posynomial in x, y and z because it is the sum of two positive monomials.

Having defined posynomials, the GP is an optimization problem that allows for posynomial constraints and objective functions. More formally, the basic form of a

GP is

$$\begin{aligned} &\text{minimize } f_0(x) \\ &\text{subject to } f_i(x) \leq 1, \quad i = 1, ..., m \\ &\qquad\qquad g_i(x) = 1, \quad i = 1, ..., n \end{aligned}$$

where f_i are posynomial functions, g_i are monomial functions, and x_i are the design optimization variables. Although, strictly speaking, GPs are not convex optimization problems, GPs can be mapped to a convex space through a log-transformation, meaning that GPs can be solved just as efficiently as regular convex optimization problems.

To give the reader a sense of how GPs are applied, we present a simple optimization problem for the energy-performance design problem of the previous section.

$$\begin{aligned} &\text{minimize } E(x), \\ &\text{subject to } T(x) \leq T_{target} \\ &\qquad\qquad x_{min} \leq x \\ &\qquad\qquad x \leq x_{max} \end{aligned}$$

Here x is a vector of design variables in the design space, and we are trying to minimize total energy, E, while achieving an execution time, T, that meets the minimum execution time specification, T_{target}. Each of the design variables is also constrained to values between the vectors x_{min} and x_{max}. In this problem, to increase performance (reduce execution time), we must change some design variable x_i, but this also affects the total energy. Conversely, if we try to reduce total energy, we must do so through a design knob, and this will affect total performance. Thus, changing a design knob has an effect on both energy and performance, which results in an energy-performance

trade-off. It should be noted that while the constraints have not been written in the form of $f_i(x) \leq 1$ they can be easily manipulated to be in such form. In the general case, there is no guarantee that the functions $E(x)$ and $P(x)$ will be posynomial, but if it can be shown that these functions can be modeled with posynomials, then GPs can be used find the optimal values of the design parameters in x.

While it may be ineffective to use any of these optimization methods if the problem intrinsically lacks the appropriate form, numerous design problems have been shown to be convex [6, 43, 67, 63, 62, 11, 57, 32, 66, 18, 10]. When one can identify a design problem as being convex, then formulating convex optimization problems enables convex solvers to globally optimize the system in a quick and robust fashion. In Chapters 3 and 4, we show how most of the energy and performance models for both the architecture and circuit design spaces of digital systems can actually be captured well with posynomial functions. For example, many architectural design knobs have a monotonic, diminishing returns profile; changing parameters such as instruction window sizes, cache sizes or functional unit latencies typically either smoothly increase or smoothly decrease performance. These characteristics—and even some other more complex effects—can be modeled well by posynomials. We take advantage of this property to formulate the system optimization problem as a geometric program. This allows us to rigorously optimize digital systems for energy efficiency, and identify the energy-efficient frontier of Figure 2.2.

2.3 Processor Modeling & Optimization

The previous section discussed the modeling and optimization of systems from a general perspective. We now examine the particular problem of modeling and optimizing digital electronic systems, with a specific focus on microprocessors, since they represent one of the most important, best-studied and most complex of all digital

systems. While the discussion here and in the rest of this work focuses on processors, the challenges and proposed solutions apply, for the most part, to digital systems in general.

The primary challenge in modeling and optimizing a digital system such as a processor arises from the fact that digital systems are complex, hierarchical systems with large design spaces. At each level of this hierarchy there exist numerous design decisions. At the lowest level, there are various transistor parameters that can be tuned to provide devices with different delay and energy characteristics. At the circuit level, each circuit can be implemented in various different ways with different characteristics. Similarly, the design of the architecture also involves a large space of design possibilities. Despite the complexity of the problem, each of these levels of the design needs to be engineered properly as each can have a potentially significant impact on the characteristics of the whole system.

While a designer would like to explore the space of all possibilities to optimize the whole system, the sheer size of the design space that needs to be considered can quickly make this analysis unwieldy, especially when one notes that each parameter or design choice in the design space causes the space to grow exponentially. Trying to explore the space of possibilities for a single level of the design hierarchy, such as the circuit design or architecture, can be daunting by itself, let alone trying to consider all aspects of the design simultaneously.

To simplify the design problem, designers rely on design decomposition and abstraction across the layers of the hierarchy to enable the design of each layer in relative isolation. By identifying the critical parameters at the level of interest and abstracting away the lower level details, a designer is able to focus on his particular domain of the hierarchy, making the design problem more manageable.

The use of hierarchies and abstractions have been a powerful means of managing the complexity of the design problem; without design decomposition, designing

large systems such as processors would be virtually impossible. A side effect of applying design layering, however, is that the communication of design constraints and capabilities between different layers of the hierarchy sometimes become limited. In particular, we find that, even though the modeling of circuits and architectures are very developed areas, the architecture and circuit design spaces are usually optimized separately: there exists a gap between the two domains, with architectural studies sometimes only loosely considering the circuit design space. This design abstraction into layers, along with the inter-layer gap it sometimes creates, has effects on both the performance and power modeling aspects of the design.

2.3.1 Performance Modeling

We first examine current architectural performance modeling methods, and how they connect to the lower level layers of abstraction. Years of work in the architectural domain has led to the development of very mature performance modeling methodologies. The most important and primary modeling tool used by the architect to perform performance evaluations is the architectural performance simulator [24, 7, 42, 4, 59, 68, 69, 70, 46]. In these simulators, to abstract away the lower-level circuit details, the clock cycle is established as the basic unit of delay, and all events occur around this basic unit of time. Thus, the latencies of the various different architectural units that form the processing pipeline are all expressed in terms of the number of clock cycles. While it is understood that the clock cycle time is a value that is dependent on the delays of the underlying circuits that implement the architectural functionality, these details are typically abstracted away for the sake of simplicity. This approach frees the architect to more easily perform studies of new architectures and features.

This use of design abstraction, however, also requires that care be taken to model the right system. For one, because cycles within the software simulator are abstract

delay entities, there is no real connection to the delays of the underlying units. This means a pipeline stage in a simulator can potentially contain arbitrary amounts of logic, and there is not much to prevent a careless or novice architect from creating an unrealistic design with too much logic in a given stage of the pipeline. Thus, it becomes important for an informed architect to always ensure that all circuits are accounted for, and that the right system is being modeled.

As a related issue, because architectural simulators operate using cycles, the total execution time (performance) of a given benchmark application is reported as the total number of cycles, or, in its normalized form, the number of cycles per instruction (CPI). It is well-known, however, that true performance is not the number of cycles, but rather the execution time in seconds. Using the instruction-normalized form, the total time per instruction, TPI, is

$$TPI = CPI \times T_{cycle} \tag{2.3}$$

where T_{cycle} is the clock cycle time. While architectural studies target CPI, it is again the duty of the architect to take care to account for the clock frequency, which is a critical parameter to total performance. For example, there can often be architectural design innovations that may improve CPI, but which may also increase T_{cycle}, making them undesirable. Here again, since there is no systematic approach to incorporating these effects into the architectural simulation models, an informed architect simply has to do his best to ensure that his study evaluates any potential effect on the clock cycle time.

Despite this gap between the architecture-level models and the lower-level circuit delays, the cycle-based simulation approach to performance modeling has been effective. The simplicity offered by this abstraction allows for quick exploration of new designs, and this strength far outweighs the drawbacks of that come with the loose

modeling of circuit issues. The architect simply needs to separately account for circuit issues and their effect on the cycle time. Although this may sometimes lead to some small errors when certain circuit details are not accounted for properly, this has been an issue that has generally been manageable. Thus, we find that simulators are powerful tools in performing architectural design evaluations.

2.3.2 Power Modeling

With power consumption becoming an increasingly important concern, we have, more recently, seen the development of various power estimation tools. Tools such as Wattch [13], SimplePower [64] and PowerTimer [12] are now important tools that help guide power-performance analyses and optimizations.

At a fundamental level, the operation of all these power estimation tools is fairly simple. As an application is simulated, these tools maintain activity counts for each time a unit is accessed (e.g. adder accesses, D-cache accesses, etc.). These activities are then multiplied by a per-access power cost for each unit to get the total power consumed in each unit. Summing the power in all the units then provides the total dynamic energy, to which a leakage component can then also be added.

In these power modeling approaches, the design hierarchy has again had an impact on how the modeling is performed. At the heart of the power modeling tools is a library of energy costs for each unit; the simulator references this library to determine the per-access dynamic energy cost during its computation. While each unit in this library is represented by a single energy cost, in reality, a circuit can be implemented in different ways. Thus, there is, in fact, no single cost; the cost, rather, depends on the desired circuit implementation. Including the whole space of circuit design options, however, would complicate the situation, as the power estimation tool would not know which cost to use. To maintain the simple abstraction model, therefore, these tools choose a single fixed energy cost, typically using the cost of an circuit

implementation in an existing chip.

While using a fixed cost has its advantages—particularly being a simple and easy to use modeling abstraction—the need to now rigorously evaluate energy-performance trade-offs when optimizing designs for energy-efficiency means that we need a tighter coupling between design decisions between the architecture and circuit levels. The energy spent in each of the underlying circuits can have a significant impact on the energy of the whole system, and designers now need to consider the different circuit implementations that offer different trade-offs between energy per operation and performance. For performing large-scale hierarchical optimization of a system, a single fixed energy cost does not capture the entire space of possibilities, and the single fixed circuit implementation is not necessarily the right choice for all systems and design objectives.

To truly optimize the architecture, a *circuit-aware* framework is required that takes into consideration the different possible circuit implementations and their characteristics. Building such a framework requires that existing modeling methodologies and infrastructure be extended to support better communication of design possibilities between layers to enable a multi-level optimization. In the next section, we provide an overview of how we build on current approaches to produce an architecture-circuit co-optimization framework that resolves these issues, and links the architectural models to the circuit domain. First, however, we end this section by examining some previous work in the areas of processor optimization and architecture-circuit co-optimization.

2.3.3 Processor Energy-Performance Optimization

Due to the importance of managing power in future systems, there have been numerous prior works examining various aspects of power-performance trade-offs in microprocessors. The emergence of the aforementioned architectural power modeling

tools has enabled an increasing number of studies that have focused on performing design space explorations under power considerations. Along these lines, several works have examined the optimal pipeline depth from a power-performance perspective [61, 31, 71], while more recent works have applied advances in modeling methodologies to explore larger parts of the space: Karkhanis and Smith used an analytical modeling approach to study power-performance trade-offs in processor design [41], while Lee and Brooks used regression-based models to explore an even larger part of the design space [45]. These works—and a large number of others—have focused on the architectural domain and provided plenty of insights on energy-efficient design. We use some of the approaches in these works to create the architectural models within our circuit-aware optimizer.

Research that explored joint architecture-circuit design has been fewer in number. Some early concepts of performing a hierarchical optimization of digital systems were proposed by Markovic et al. in their work examining methods for true energy-performance optimization [48]. In another work, Zyuban et al. proposed a hardware intensity metric—simply the relative marginal performance versus energy cost—to connect the architectural optimization to the circuit behavior [72]; Qi et al. generalized some of these concepts [56]. While these works were mostly theoretical in nature, they established a foundation for creating the practical framework for evaluating joint architecture-circuit trade-offs that we present.

Despite the lack of a concrete co-optimization framework in the digital design field, several works in other fields were able to show that co-optimization is possible. Markovic was able to apply co-optimization to the design of wireless MIMO detection systems [47], while Sredojevic did the same for high-speed links [60]. Although both these works focused specifically on a particular systems and included analog circuit design issues, they both showed significant benefit for the higher-level system, providing a case for creating a general framework for performing joint optimization for

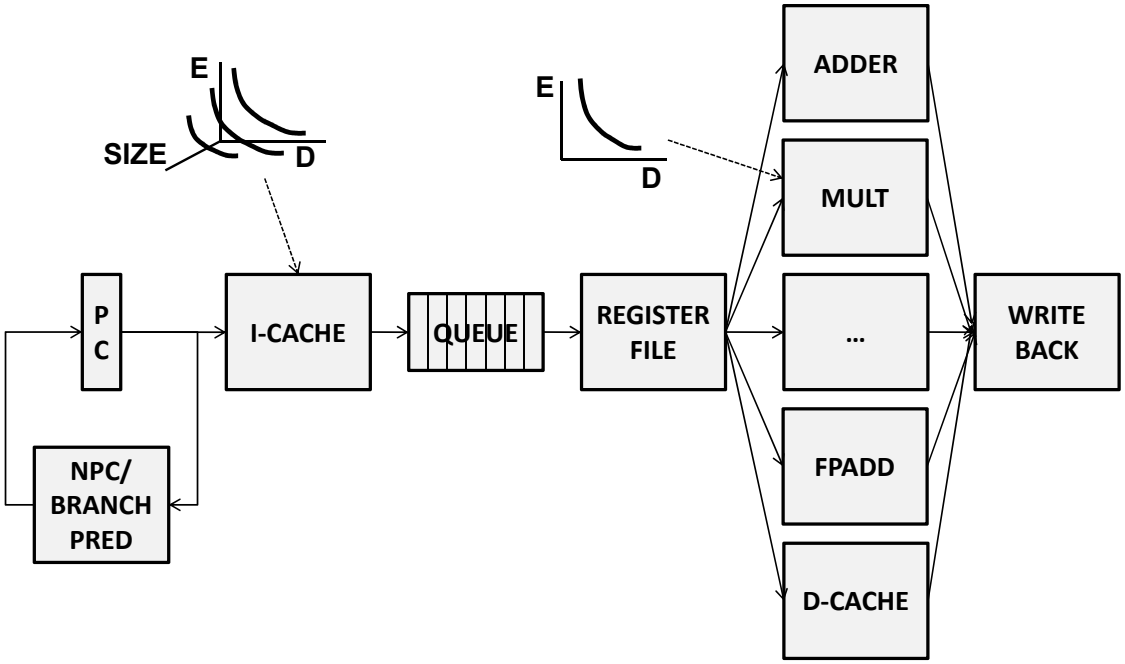


Figure 2.5: Example of a joint circuit-architecture optimization for a very simple in-order machine. For each architectural block, there is an associated circuit trade-off similar to the one shown for the multiplier. There are also microarchitectural design knobs, such as the sizes of queues, buffers and caches, which require a more complex trade-off characterization, as shown for the I-cache. The joint circuit-architecture optimizer uses these circuit trade-offs, along with models of the architecture and knowledge of dependencies between instructions, to explore the design space and find the optimal design.

digital systems.

2.4 Circuit-Aware Optimizer Overview

To provide an overview of the function of a circuit-aware optimization framework, Figure 2.5 shows a simple in-order processor system design problem. Even though this particular system is very simple, there are numerous design parameters that need to be explored and set by an optimization procedure. First, each of the architectural

blocks can be designed for different delays, with the chosen delay determining the energy per operation of that unit. This circuit-level trade-off is shown schematically in the figure for the multiplier block; similar trade-offs exist for other blocks as well, but are omitted for simplicity. For certain other units in the system, there are also higher level microarchitectural sizing knobs, such as the sizing of buffers, queues and caches, that affect the trade-off. In these cases, a more complex circuit trade-off characterization is required where the energy per operation of the block is dependent on both the size of the structure and its delay. This kind of trade-off is shown schematically in the figure for the I-cache block (but, again, exists for certain other blocks as well).

The task of a circuit-aware architectural optimizer is to determine the optimal values for each of the delays of the circuits, along with the sizes of any structures, when applicable. To accomplish this, however, the optimizer needs to know how a change in any one of these parameters translates into a change in the overall performance and energy of the whole system. Not all units are equally important to the architecture, and the optimizer, using application behavior, needs to determine the sensitivity of each parameter to the overall system characteristics during optimization. Some units, for example, may be used more frequently than others, making their effect on overall performance stronger; a designer may desire to allocate more of his energy budget to these units. As a more complex example, many units in the system can be pipelined, but whether the one should use a higher latency, pipelined version of a unit depends on many factors. First, the less aggressive circuitry may mean the energy per operation of the circuit will decrease, but one also needs to account for the additional energy in the pipeline registers. Second, one must consider how often that unit finds itself on a critical dependence loops. If the unit is not often found in critical dependence loops, then pipelining the circuit can save energy without sacrificing too much in performance; on the other hand, if a block is often essential to resolving critical data

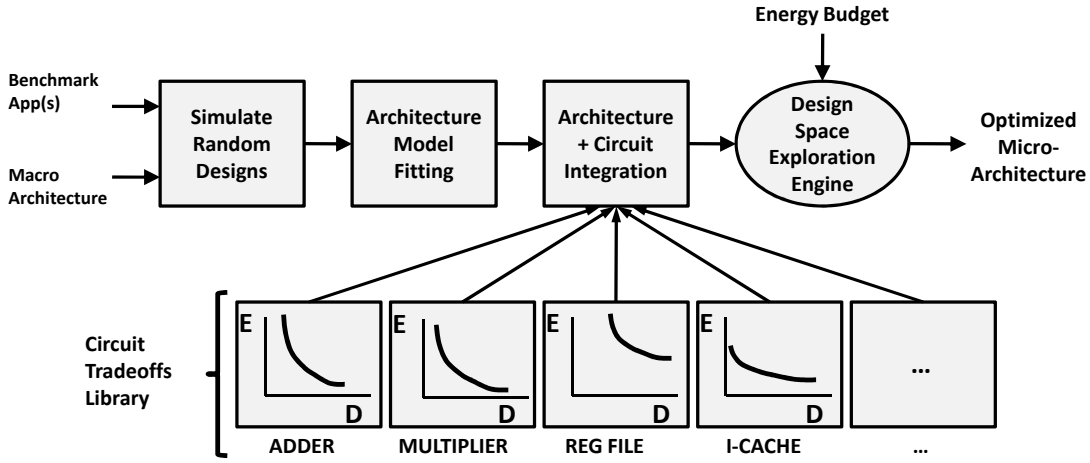


Figure 2.6: Overview of the optimization framework. Architectural models are generated using sampling and fitting methods. Energy-delay (E-D) trade-off curves are characterized for each circuit. These design spaces are integrated, and the design space exploration engine finds the optimized design.

or control dependencies frequently, then the performance cost of that design decision may not be worth any potential savings in energy. To produce a truly optimized design, all these effects—from the circuit-trade-offs to the architecture and the effect of the application behavior—need to be modeled in the optimization problem; this is the goal of our joint architecture-circuit system optimization framework.

Figure 2.6 shows an overview of our circuit-aware digital system optimization framework. There are four major components (representing four steps) in this framework. We first create circuit trade-off libraries that characterize the different energy-performance trade-offs for each of the underlying circuits that make up the system. These libraries, which essentially consist of the energy-performance points of different potential design implementations, are characterized by exploring the circuit design space. The circuit exploration process can be done with any existing circuit optimization tools, and then fed into the circuit-aware optimization framework. These circuit trade-off libraries replace the fixed energy cost libraries of existing power modeling

tools.

The second step involves creating architectural performance models that describe how changing architectural design knobs such as latencies and sizes of structures affect the overall architectural performance. In this step, the primary challenge is to model a large design space with numerous design parameters. To solve this problem, we leverage design space sampling and statistical inference methods to enable us to capture a large multi-dimensional space of microarchitectural parameters by sampling only a small subset of the entire design space. The application behavior is captured implicitly in the models generated in this step.

Once we have circuit libraries and architectural models characterized, the third step involves linking the circuit libraries to the architectural models. This step involves establishing the relationships between circuit delays, architectural pipeline latencies and the cycle time; computing total performance from CPI and the cycle time; and also linking circuit-level energy models to the overall energy. These constraints create a joint design space that not only makes the architectural latencies dependent on the real delays of the circuits, but which also exposes an entire space of circuit implementations (and their energy-performance trade-offs) to the overall system.

Throughout the modeling of the architecture, the circuits and their integration, we use posynomial characterization functions to fit the data. This enables the formulation of a geometric program to describe the behavior of the entire system. This joint architecture-circuit design space is then finally sent to an optimization/exploration engine—the last component—which, given an optimization objective and resource budgets, searches the space to find the most efficient design configuration.

A large part of this optimization framework is unavoidably centered around the modeling of the circuits and the architecture. Because we ultimately want to perform

optimization, creating mathematical characterizations through data-fitting is a common feature throughout the modeling phases: After collecting circuit data points, a data-fit is performed to characterize the energy-delay trade-offs mathematically. Similarly, the statistical inference modeling of the architecture is also fundamentally a data-fitting process. While they can initially be time-consuming, building the circuit libraries and architectural models are generally one-time costs unless new design spaces are being explored (e.g. systems requiring new circuit blocks, new architectures or new applications). Once the characterization is complete, the convex optimizer of the design space exploration engine can produce optimized designs relatively quickly, in under 30 seconds for complex processor designs on a desktop computer.

The combination of these techniques creates a framework that is both powerful and general. It is powerful since it enables a rigorous study of marginal performance benefits and energy costs of any design decision; with this framework we can identify the parameter values that yield the most energy efficient design for a performance target, or the highest possible performance design for a given energy target. It is general because we can construct architectural models for any system simply by extracting simulation samples from the designer's simulator of choice, and we can include circuit trade-offs from a broad range of tools.

In the following two chapters, we examine each of the components of this optimization framework in more detail. Chapter 3 discusses the problem of creating models of large architectural design spaces to drive this framework, discussing why traditional simulation-based approaches are insufficient and showing how statistical inference using posynomial functions can be effective in capturing large design spaces. Chapter 4 then examines the characterization of the circuit trade-offs, how they are integrated into the architectural models and the resulting optimization problem formulation for the joint design space.

Chapter 3

Architectural Modeling

The process of designing and optimizing a system at the architectural level requires significant effort and analysis. The designer needs to evaluate various design alternatives, features and ideas, while also optimizing the numerous microarchitectural design knobs in each design he considers. To help guide this design process, the designer relies on modeling tools as a means of evaluating different design configurations. Architectural performance models are needed that can predict how performance changes as design parameters change; these must then be complemented with appropriate cost models—whether area, power or some other metric—that estimate the cost of each design. Armed with performance and energy models, the designer can evaluate the performance benefits and associated costs of design choices to make informed design decisions, ultimately deciding on what design to implement.

In this chapter, we examine this architectural modeling problem. We begin by considering architectural performance modeling, where we examine traditional simulation approaches and show how regressions/fit-based techniques can be used in conjunction with simulation to help characterize large multi-dimensional spaces. We then present how posynomials functions can be applied to characterize the system performance. Then, in the second section in this chapter, we examine how energy

modeling is typically performed at the architectural level, and how it has to change in a joint architecture-circuit optimization framework.

3.1 Performance Modeling

Traditionally, architectural performance modeling has been done through simulation [7]. This has been primarily due to the fact that processors are complex pipelined systems in which performance depends on the input (i.e. the application). The performance of an architecture in executing an application depends on the sequences of instructions and dependencies in the instruction stream, and how well the architecture resolves those dependencies. Thus, performance is not a simple function of the architecture, but involves, rather, an interplay between how the given architecture meets the demands of the application. This means an architecture may perform well on one application class, but poorly on others.

To capture the application's effect on performance, any modeling approach requires, at minimum, some trace simulation/analysis to examine the application behavior. Characteristics such as instruction mix, data dependency patterns and control dependencies—which can vary greatly from one application to another—play a significant role in defining what aspects of the system should be optimized.

While simulation-based approaches are valued for their ability to model complex systems, simulation run times are typically very long. Whereas real high-performance processors can execute billions of instructions or more per second (BIPS), industrial simulators run at about tens of thousands of instructions per second (KIPS) [34]—a 100000 times speed gap. Thus, running even short simulations can take on the order of several hours and more complete simulations can take multiple days. This presents a serious challenge when trying to explore large design spaces, and so there have been considerable research efforts to find more efficient modeling techniques.

To help solve this problem, researchers have proposed, throughout the years, various different analytical processor performance models that predict performance directly from application characteristics and design parameters [23, 39, 51, 25, 26, 40, 5]. Having mathematical forms, these solutions offer the significant advantage that the design evaluation time is very quick. This is particularly important for design space exploration and makes searching large spaces considerably more efficient. Despite these advantages, analytical models have not been widely adopted by the architectural community. This has been due to a combination of reasons: analytical models are less flexible and can be hard to to modify when investigating new features, they are often restricted to a smaller set of design parameters that they can model, they often have lower precision than simulation models, and they are generally a less trusted form of performance modeling (having, in fact, to be calibrated and/or validated against the more trusted simulation models). Moreover, depending on how accurately an analytical approach wants to account for application characteristics, the analytical models still need to process long simulation traces, which diminishes its run-time advantage over simulation. In the end, while analytical models are useful as first-order models for performing high-level design space explorations, the flexibility that simulators provide in modeling complex systems, in addition to the fact that they are easy to understand and work with, has meant that simulators are still the preferred method of performance modeling for the architect.

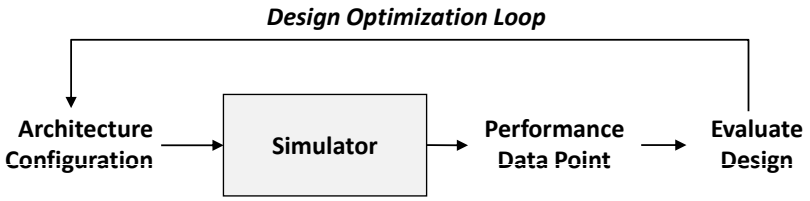
As simulators remain the most widely used modeling tools, there has been considerable research into finding ways of reducing simulation times. Here there are two orthogonal problems: First, one needs to reduce the simulation time per simulation run, and, second, one needs to reduce the total number of simulations that need to be run to explore a design space. On the first front, there are several approaches a designer can use. Tools such as SimPoint [29] and others [22, 65], for example, have been effective in identifying short, representative simulation segments that can serve

as a proxy for how the whole application would behave. Others have investigated the use of FPGA emulation to speed up simulation runs [3, 14, 55, 52]. Even with these approaches, however, there can still be a very large number of design points in the design space, with the space growing exponentially with the number of parameters. For example, if one considers a design space with even only 15 parameters, each with 4 possible values, then we already have a space with over a billion possible design configurations. Thus, the second approach to reducing total simulation time, which is complementary to the first, seeks to find some way of exploring these large spaces effectively.

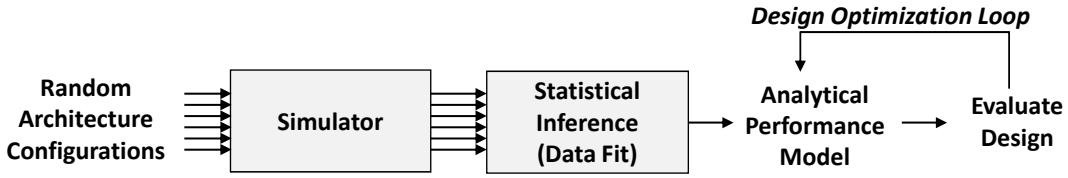
3.1.1 Fit-Based Performance Modeling

To help explore large design spaces effectively, recent works have proposed the use of statistical sampling and inference to create fitted models from a relatively small sample of design points [44, 36, 37, 20]. The basic idea behind these approaches is that one does not necessarily have to simulate every single design point to get a good indication of how different parameters in the system behave and interact. One can learn a lot about the behavior of the system simply by simulating a relatively small number of designs with different configurations and observing the output. Essentially, these techniques take advantage of the fact that the performance surface is usually fairly smooth; thus, they work by sampling the surface and then interpolating for missing data.

In these approaches, the design (i.e. the simulator) is treated as a black box with various configurable/tunable design parameters and a user-provided, but fixed, application workload. By randomly setting the design parameters to create random designs, and then running simulations for a handful of the possible designs in the design space, one can obtain enough data to develop a fitted model of how the system



(a) Traditional Design Optimization



(b) Inference-Based Design Optimization

Figure 3.1: Inference-based design optimization. In traditional optimization, the simulator directly serves as the system model, so all design points need to be simulated to determine their performance. This results in a design optimization loop around the simulator. In inference-based methods, an analytical model is generated from design space samples (acquired from the same simulator). Typically, several hundred to a few thousand samples are sufficient to capture a large space of billions of design points. Using the generated mathematical model then results in a much more efficient exploration and optimization of the system.

behaves as a function of the design parameters for the given workload.[1]

The result of these fitted models is a predictive mathematical model that can be quickly evaluated to predict the performance in any part of the captured space. The number of samples required to create the models can be dramatically smaller than the number of designs in the space, with several hundred to a few thousand samples typically being sufficient to model complex processor systems with spaces that span billions of possible design configurations. Thus, these approaches have shown to be powerful tools for design space exploration. This approach, and how it contrasts to traditional simulation based optimization is in shown in Figure 3.1.

This fit-based modeling offers several advantages. First, because it still relies on simulators to generate the initial samples, the approach offers a lot of flexibility; one can model any system so long as an appropriate performance simulator for the system exists (assuming an appropriate fit function is found). The designer simply uses the simulator to obtain design space samples from which he then creates fitted model. At the same time, because this approach generates mathematical models of the system, it can be very effective in quickly exploring the space and evaluating different design configurations. In this way, fit-based/inference approaches can be viewed as a hybrid of direct analytical modeling and purely simulation-based methods, achieving the strengths of both methods.

There are, of course, limitations to what kinds of design choices one can model with fit-based models. Fitting assumes that the surface of the space being fitted is smooth; if the data is not smooth, then it is hard to produce a good fit. Generally, this means the technique lends itself well to capturing tunable design parameters (such as sizes of resources and latencies of units). On the other hand, discrete architectural

[1]To model mixed application workloads, one can either develop a single model directly by using the mixed workload as the simulator input, or one can develop individual models for each of the different benchmarks from which an average performance can be computed later. We use the latter approach, so we generally have fitted models for each application in a suite.

changes—such as moving from in-order to out-of-order execution, or changing the instruction scheduling algorithm—represent parts of the space that are less natural to capture with fits. Being more numerical in nature, the tunable design parameters are also more amenable to mathematical optimization.[2] Thus, we partition architectural parameters into two groups: discrete design features and tunable design knobs. We capture the tunable knobs in the models we create, and generate separate models to capture discrete architectural changes.

After deciding to use a fitting approach to model an architecture, the next challenge is to find a fitting function that can serve as a good model for the system as suggested by the data. Previously, several different functional forms have been suggested to produce models from the simulation data. Lee and Brooks used cubic splines [44] to produce models of the fitted data, while Ipek et al. used artificial neural networks [36]; both were shown to produce good fits. The problem with these forms, however, is that they can result in spaces that can be "bumpy" (i.e. have local minima) and can therefore be hard to explore. The space, however, can usually be captured just as effectively with simpler functions that can make the search/optimization process more efficient. Thus, we explore the use of posynomial modeling functions next.

3.1.2 Performance Modeling using Posynomials

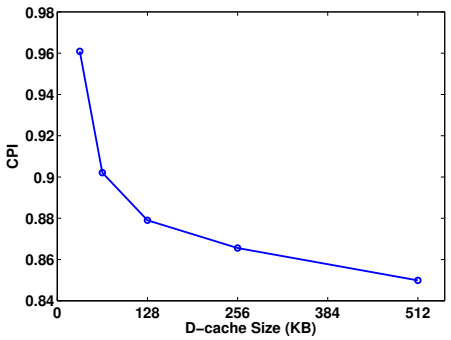
Because our ultimate goal is to optimize the system, we choose to create our architectural models using posynomial functions instead of the previously suggested functional forms. Using posynomial functions offers the advantage that, as log-convex functions,

[2] One could always map the discrete changes to numeric labels, but this presents its own set of difficulties: first, the combined space could become harder to model given that each architecture may have a different set of underlying design parameters; even common parameters may behave differently depending on the discrete choice. Second, since the ultimate goal is optimization, it is hard to interpret what fractional results really mean for a discrete option. For example, it is not clear what an instruction scheduling algorithm of 1.5 represents when there may be 3 scheduling algorithms available (not to mention that the result depends on the arbitrary labeling used).

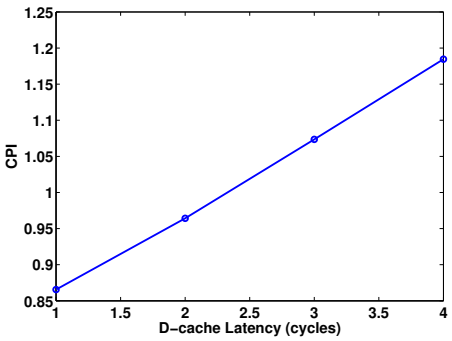
they can be optimized very efficiently. The potential drawback, on the other hand, is that posynomial functions are less general than other forms such as cubic splines; posynomials, for example, can have difficulty capturing spaces with hills and valleys. It would, of course, be hopeless to try to capture a space with a restricted form such as a posynomial if the space was intrinsically more complex. However, if the space does not require more complex mathematical forms, then it makes sense to use the simpler posynomial form: the simpler form makes over-fitting less of a concern, and—more importantly—the use of posynomials enables the application of some powerful mathematical techniques for performing optimization.

In examining the architectural performance space, many of the design knobs exhibit behavior that seems to be well-suited for posynomial modeling. Specifically, a large number of tunable design knobs have a smooth, monotonic profile that can typically be captured well by posynomials. For example, reducing a unit's latency or increasing the size of a queue, buffer or memory structure typically only improves CPI. To demonstrate this kind of behavior, Figure 3.2 plots CPI versus a few design parameters as obtained through simulation sweeps.[3] While only a few parameters are shown here, there are numerous other parameters that exhibit this smooth, monotonic profile—which also covers the frequently observed case of diminishing returns—including cache sizes, the reorder buffer size, reservation station sizes and instruction queues to name only a handful. Even in cases where a parameter exhibits more complex behavior that results in a peak or valley—for example, performance as a function cache block size—posynomials may sometimes still be able to capture the effect by using the sum of multiple different monomial terms (e.g. $\frac{1}{x} + x$ is a posynomial that results in a single valley) to model the more complex behavior.

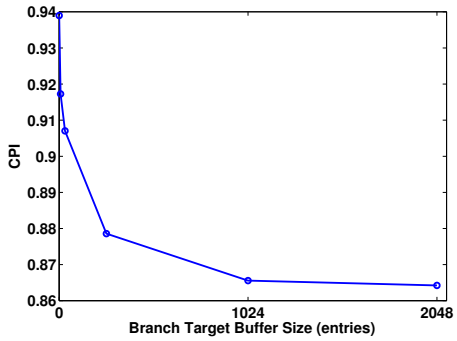
[3]It should be noted that these plots sweep parameters independently sweep of parameters, but we ultimately want to model the entire space as a function of all parameters simultaneously (including their interactions). Nevertheless, they still demonstrate the general monotonic nature of design parameters in the space.



(a) I-cache size



(b) D-cache latency



(c) BTB size

Figure 3.2: A sample of design parameters and their effect on CPI. Most architectural parameters have a smooth, monotonic profiles. As posynomials can have negative exponents, fitting parameters with both positive and negative correlations to CPI typically do not pose any difficulties.

Given that posynomials seem like good candidate for fitting the space we seek to characterize, we use a fit-based posynomial modeling approach to create our architectural performance models. As inputs to the model, we have the various tunable design knobs within the system. Thus, the models we generate predict CPI as a function of the latencies of units and sizes of structures like caches, buffers and queues:

$$CPI = f(..., latency_i, ..., size_j, ...) \tag{3.1}$$

While we have specified the design knobs and we know we want to use a posynomial to model the system, we still need to select the right posynomial function f to produce a good fit. This process involves choosing the right set of monomial terms to include in the model, and requires some attention. If some important terms are not included, the model will not be able to capture the space; if too many superfluous terms are included, then finding the fit coefficients becomes harder and takes longer, over-fitting the data becomes an issue (even though over-fitting is less likely to occur with posynomials because of their already restricted form), and the optimization is needlessly slowed down as it evaluates a more complex function. Thus, we aim for a relatively simple function that can capture the space well.

$$f(x_1, x_2, ..., x_n) = \sum_{i=1}^{n} a_i (x_i)^{d_i} + \sum_{(j,k) \in S} b_{jk} (x_j)^{e_{jk}} (x_k)^{g_{jk}} + c$$

Here $x_1, x_2, ..., x_n$ are the architectural parameters, and all other variables are fitting constants. The first summation term includes a monomial to account for each architectural parameter's independent influence on the output. It is always included for each parameter. The second summation term serves as a way to capture interactions between pairs of parameters (j and k), with the set S denoting parameter pairs in the model. For example, one important interaction between parameters involves the L1 cache size and the latency of the L2 cache; L2 access times become more important

as L1 miss rates increase. Thus, we would want to have this parameter pair in the set S.

As part of this fitting process, therefore, it is important to identify the right parameter interactions. Without capturing certain critical interactions, the fit could fail to produce any meaningful model; other smaller interactions can still improve the fits by several percentage points. One could use domain knowledge to manually specify expected interactions, but this requires that the system already be well understood. Instead, a more automated process is preferred. To determine the interactions terms in an automated fashion, we can simply perform an exhaustive search. We first create a base fit with no interaction terms. We then initialize S to be empty and iterate over all possible pairs. For each visited pair, we attempt a new fit with that term. If the resulting fit improves the model by some threshold, the interaction terms are added to S.

The interaction terms we have discussed have been limited to parameter pairs, but one can also look for more complex interaction terms that involve three or more parameters. The number of terms to consider when looking at three parameters, however, is exponentially larger, making it more difficult to identify such terms. Nevertheless, one way to check whether the exclusion of any term (not only triplets, but any single parameter or otherwise) is adversely affecting the fit is to check for residual errors in the fit that are correlated to the parameters under consideration. For the systems we have studied, we have performed some limited checks to look for errors correlated to triplets, but have been unable to identify any such cases. While it is certainly no guarantee that these terms may not be important, for the processor systems we have examined, using only interaction pairs still yields fairly good accuracy and seems to be sufficient for modeling the systems.

To validate and check the accuracy our fits, we set aside a fraction of the simulation samples specifically for the purpose of checking our fits; these samples are not used

in producing the fit. We can then compare how well the fitted model would predict the performance of a design configuration in the design space which it has not seen before. To measure error, we use the same metric as in [44]:

$$error = \frac{|predicted - actual|}{actual}$$

This metric is applied to each validation sample, and the median error achieved is reported.

Generally, the number of samples required to generate a good fit depends on the size and complexity of the system. For example, we have found 200 samples often enough for simple in-order processors with 11 parameters, and 500 samples to be sufficient for a complex superscalar out-of-order processor with 18 parameters. The average of median errors over different benchmarks range from less than 1% to 6%, with more complex high-level architectures such as out-of-order processors tending to be harder to fit.

Figure 3.3 shows some sample fits, which are scatter plots of the actual performance of a design configuration versus the performance predicted by the posynomial model. Thus, the closer the points are to the diagonal, the better the fit is. Each plot represents the result of running a particular application on a given architecture. The figure shows results of varying qualities: a very accurate fit, a typical fit, and a worse fit. Even in the worst case, the fitting error is below 10%. The cumulative distribution functions (CDFs) of these three fits are also shown to give the reader a sense of the distribution of errors in each of these generated models.

Although these results are only specific examples, in most cases we are able to achieve fits with behavior similar to the "typical" case. This means that our posynomial fits typically perform well in capturing the architectural performance space. In certain cases, however, our posynomial fits do yield slightly worse results as in the

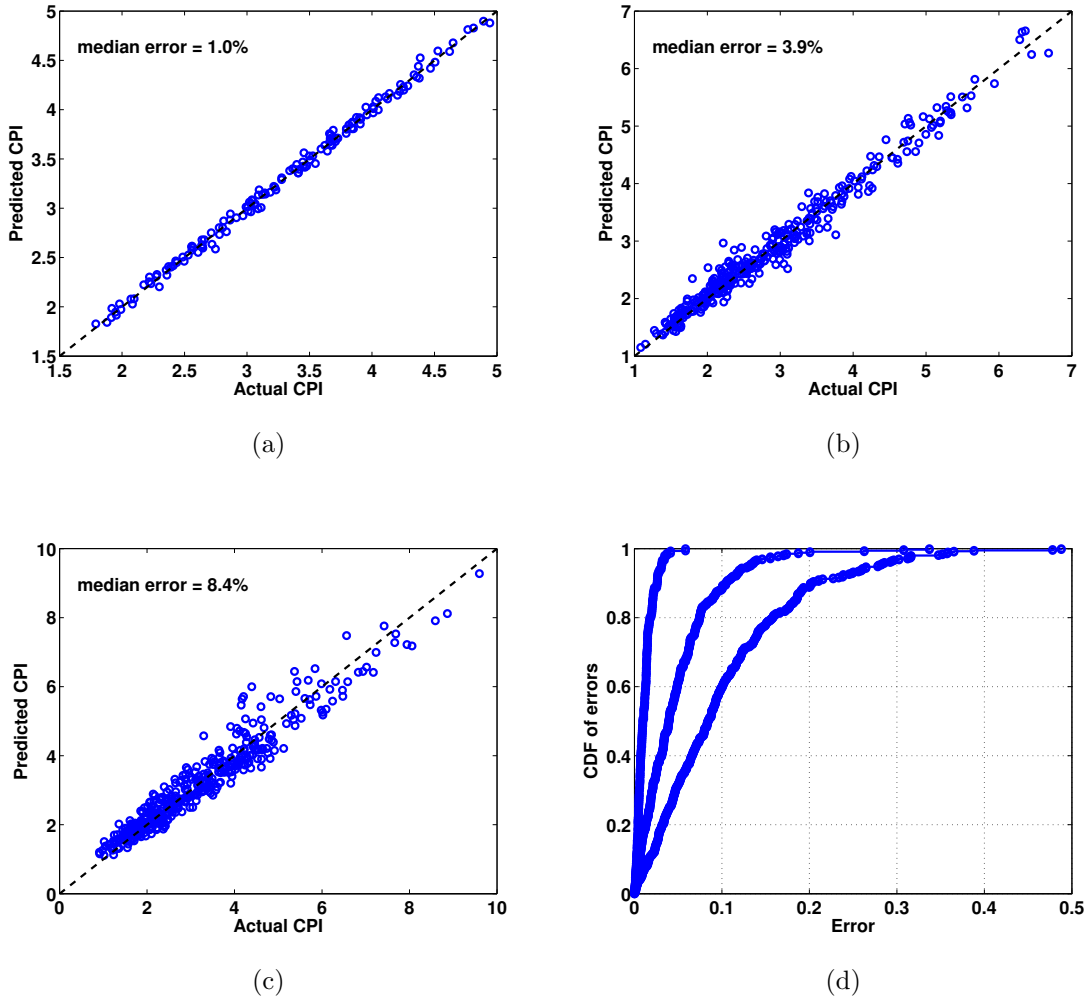


Figure 3.3: Validation of three architectural models generated through posynomial design space fitting. (a)-(c) compare model predictions to the results of the simulator. Each fit is for a particular application running on a given processor architecture. (a) is a very accurate fit generated for a single-issue in-order machine, (b) is a typical fit, in this case generated for a quad-issue out-of-order machine, and (c) is a worse fit, also for the quad-issue out-of-order machine (but using a different application). (d) shows the cumulative distribution of errors for these three models. Even in the worst case, the median error is less than 10% for a performance range that spans around 10x.

"worse" example.

To check the degree to which our posynomial fits may be restricting the quality of the models we generate, we compare our fits with other fits that use a more flexible form; this checks to see whether more freedom would produce a better fit. Along these lines, we have compared our fits to cubic spline fits and more relaxed versions of our posynomial functions with the positive constraints on coefficients removed. In some cases, we find that we could produce a fit that may be two or three percentage points more accurate (e.g. 6% accuracy instead of 8% accuracy) by using the more flexbile forms. In these cases, these results indicate that our particular choice of posynomial function is restricting the quality of the fit somewhat. Investigating these cases shows that the loss in accuracy can be attributed to smoothness issues. For example, we sometimes find that a parameter, though still monotonic, has a sharp change in curvature or slope. Our posynomial function often fits this data in a smoother way, resulting in some increased error at the point of sharp curvature change.

Despite these occasional fitting effects, in most cases, we find the results to be of comparable accuracy to the other, more flexible fitting functions. Thus, the use of our posynomial functions is not usually a restricting factor in the models we produce. Moreover, even in the "worse" cases, the fits still achieve a median error of less than 10%, which represents a good fit for performing the large-scale optimization that we are interested in; a second optimization iteration around the identified area of interest can always be used to refine the optimization results and get around these minor fitting effects. Finally, when optimizing over a suite of benchmarks, averaging makes the effects of these occasional fitting issues less significant. For further evaluation of how well this posynomial fitting approach works for different benchmarks and architectures, we refer the reader to the results in Section 5.1, where we create models and optimize different processor architectures.

3.2 Architectural Energy Modeling

To evaluate the marginal costs of different design decisions, a designer needs energy models to complement his performance models. With appropriate energy models, the designer can associate costs to the performance gained, and can determine whether a design decision is worth the effort from an energy efficiency perspective.

As power considerations have become more important over the years, simulation infrastructures have thus been extended to provide power estimates in addition to performance numbers. The use of architectural power analysis tools like Wattch, SimplePower and PowerTimer is now common. Fundamentally, these tools all operate in a similar fashion. To perform their analysis, these tools divide the total energy consumed into the energy consumed in different blocks, each of which can have dynamic and leakage components. The total dynamic energy consumed is therefore just the sum of the dynamic energy consumed by activity in each circuit block. To compute the dynamic energy of a particular block, the average cost of exercising that unit is simply multiplied by the number of times that the unit was used. Formalizing this, and converting it to a per instruction basis, the dynamic energy per instruction, $EPI_{dynamic}$, is computed as follows:

$$EPI_{dynamic} = \sum_{i} (\alpha_i \times E_i) \tag{3.2}$$

Here, i iterates over all the units in the system, E_i is the average energy cost of a unit i and α_i is the activity factor of that unit as obtained from simulation. This value can then be converted into power by dividing by the average time per instruction. Total leakage power, $P_{leakage}$, is easier to handle—at least from an architectural modeling

perspective—as it does not depend on any activity factors:

$$P_{leakage} = \sum_{i}(P_i) \tag{3.3}$$

Here, P_i is the leakage contribution from each unit. Finally, the dynamic and leakage components can be added together to compute the total power consumption, P_{total}:

$$P_{total} = \frac{EPI_{dynamic}}{TPI} + P_{leakage} \tag{3.4}$$

In this analysis, the average per-access energy consumption cost and leakage power of each unit needs to be characterized before-hand; these numbers are typically extracted from real circuit implementations, and are stored in a library that the simulator can access. The activity counts are then determined by the simulator as an application is simulated.

Since these integrated simulators now report power in addition to performance, one might expect that we can build energy models in exactly the same way we modeled performance—by fitting functions to the energy numbers obtained from the simulation runs; indeed, this method works and has been used in the past [45].

The situation, however, becomes more complicated in a joint circuit-architecture optimization. When the energy costs of each circuit are known, the simulator can easily report final power numbers and creating the models is straightforward. When the circuits themselves can change, however, then the costs, E_i, are not known until after an implementation is determined by the optimizer. As a result, we cannot multiply the activity factors by the energy costs at modeling time (as the costs are unknown); instead, we need late, optimization-time binding of this information.

To allow for this late-time binding, we need to extract the right information from the architectural simulator that will allow us to compute total energy later. Thus,

instead of directly characterizing the power of the different architectures, we must characterize the activity factors of each of the units.

To first order, the activity factors of most units are determined by the application instruction mix, and can be determined by a trace analysis. For example—assuming a very simple ISA—the number of times the multiplier is used depends on the number of *multiply* instructions in the execution trace.

Of course, this is not quite a complete characterization. First, speculative execution generally means that activity factors for units will be higher than indicated by a trace analysis; speculation implies that the processor is doing work that may be discarded. Second, and perhaps more significant, is that the activity factors for certain units can be affected by caches. For example, the number of L2 cache accesses depends on the miss rates—and therefore the sizes—of the L1 caches below it.

The basic solution to both these problems is the same: we need more characterization of the system. We need to model what the actual activity factors are, and how they change with design parameters. Fortunately, there is no need to run additional simulations to extract this data; we can use the same simulation samples as used for the performance modeling to obtain activity factors. We can then build mathematical equations that can predict how activities change with design parameters.

3.2.1 Characterizing Activity Factors

While we needed complex equations that were dependent on numerous parameters to generate the performance models, it is often possible to characterize activity factors using much simpler equations. To capture the primary effects on cache miss rates, for example, we need only characterize how these miss rates are affected by the cache size. While other parameters may cause miss rates to change slightly (due to changes in the speculative activity), these are smaller scale effects. Thus, for a system with

separate instruction and data L1 caches, we want to characterize miss rates as follows:

$$mr_{IL1} = f_{mr\_I}(size_{IL1}) \tag{3.5}$$

$$mr_{DL1} = f_{mr\_D}(size_{DL1}) \tag{3.6}$$

$$\alpha_{L2} = mr_{IL1} + mr_{DL1} \tag{3.7}$$

Here, mr is a global miss rate with respect to total number of instructions (not per cache access), and f_{mr} is a function that characterizes cache behavior for the application; separate characterizations are made for the L1 I-cache, $IL1$, and the L1 D-cache, $DL1$. Then, α_{L2} is the activity factor of the L2 cache, which depends on the number of misses below it.

We still need to determine a function f_{mr} that can fit cache miss rates well for different applications. Because we want to leverage geometric programming techniques to perform the overall system optimization, we would like this function to be a posynomial. In practice, we have found that different cache miss rate profiles for different applications require different fit forms. We have found one of the two following approaches often works well: either a simple function of the form

$$a \times (size)^b + c \tag{3.8}$$

where a, b and c need to be characterized, or a piecewise fit with n pieces of the form

$$max(a_1 \times \frac{1}{size} + c_1, ..., a_n \times \frac{1}{size} + c_n) \tag{3.9}$$

can be used. In both of these fits, a and c need to be positive for the function to be a posynomial. Of course, these are not the only posynomial functions that one could use; any other posynomial function that produces a good fit could be used as well. Nevertheless, in the various different applications we have examined, we have found

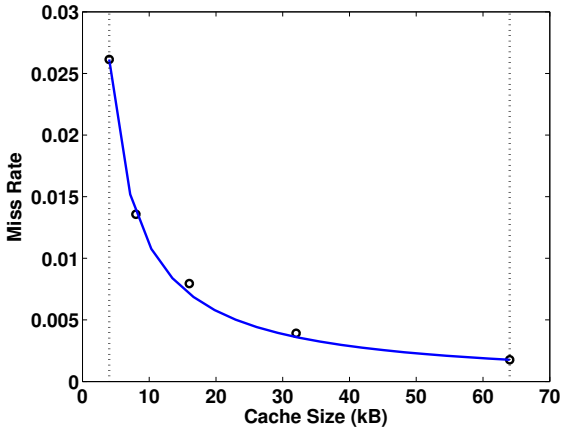


Figure 3.4: Characterization of cache miss rates as a function of cache size for one particular application. The dots are data extracted from simulation, while the solid line is the fitted posynomial function. The dotted vertical lines specify the boundaries between which this model can be trusted.

that at least one of these two forms produces a good fit.

Figure 3.4 shows an example of this characterization step. In this figure, the black dots are the data points extracted through simulation, while the curve is a fit of the data that will be used for optimization purposes. The dotted vertical lines specify the trust region of the data—since we do not have samples outside of this range, we want to restrict cache sizes to be within this region.

Unfortunately, it is not always possible to create such a good posynomial fit for a given set of data points. Figure 3.5, for example, shows cache miss rate behavior for a different application. In this case, we have first tried to fit the data using the same posynomial forms as before, but a good fit is not produced because the y-intercept (or asymptote), c, is restricted to being positive by the posynomial requirements. To rectify the fit, we can relax the posynomial form, allowing c to be negative and allowing the function to go into negative territory. While this means that the miss rate function could return negative values for some inputs, this will never happen over the trusted region of the fit, which is enforced by the constraints of the optimizer. A

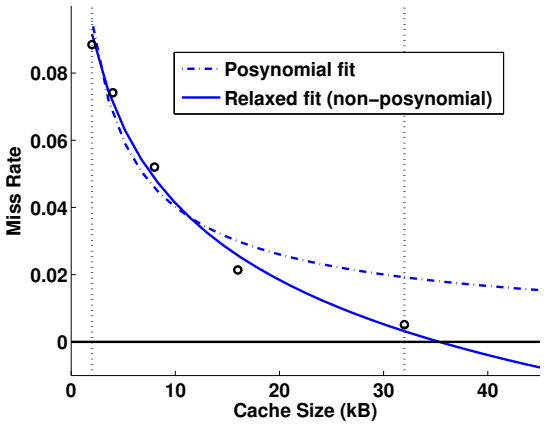


Figure 3.5: Cache fitting example where a posynomial fits poorly. By relaxing the posynomial and allowing a constant shift into negative values, we can produce a good fit. However, this new fit is no longer a posynomial, making the application of a pure geometric program for optimization impossible. This makes the optimization phase more complicated.

problem, however, is that the functional form has changed, and the fit is actually no longer a posynomial.

This is an important example, because it shows that it may not always be possible to use posynomials to characterize all aspects of the design space. In general, when such situations arise, it means some additional effort needs to be made during the optimization phase to find the optimal design. For example, in this case, we might have to try manually iterating over different cache sizes, while using geometric programming to optimize for all other parameters in the design space.

In certain special cases, there are also specific ways to get around such issues. In this particular case, for example, we have been able to avoid this issue by shifting the cache miss rate functions up by fixed amounts to make them posynomials. This causes the activity factor experienced by the L2 cache to increase; if the L2 cache has a fixed cost, however, this additional activity can be translated into a fixed energy overhead which can be added to the total energy budget. This artificial overhead

is removed after optimization. Unfortunately, this trick only works when the L2 access energy is fixed; it does not work if the L2 size is also a design parameter, as we would not be able to know how much additional energy to allocate prior to optimization. To optimize cache hierarchies, therefore, we must fall back to the more general approaches.

There can potentially be other posynomial fitting issues. For example, if an application has a very pronounced working sets, then it may be hard to capture the cache behavior through posynomials. This becomes less of an issue when optimizing for a suite of benchmarks, where such effects disappear through averaging effects. If optimization of a single application is absolutely necessary, then it is still possible to create separate posynomial models for the different working sets. This may require that both models be searched to find the right optimization, but is always possible.

3.3 Discussion

In this chapter, we examined the problem of modeling the architectural properties of a system for use within an optimization framework. To this end, we relied on a regression-based modeling methodology that uses posynomial functions to perform the fits. This approach has two key properties that make it attractive. First, being regression-based, it is a general method that can use existing architectural tools to create mathematical models of the architectural space; as such, it should be applicable to a wide variety of different digital systems (not only general-purpose processors). Second, by using posynomial forms, we open the door to the application of convex optimizations tools for later performing the design space exploration.

Through this chapter, we showed that for many of the system characteristics, posynomials do a good job of modeling the space. On the performance side, we are able to create accurate models of very large, high dimensional spaces. Errors with this

kind of approach were typically under 10% and at about 6% in the median case. On the energy side, we also showed that we are usually able to characterize the necessary activity factors that ultimately determine the total dynamic energy. The errors for these fits are typically in the same range or better.

Posynomials are thus often good functions to use to describe the system behavior. The reason for this is that we are typically dealing with tunable, numerical parameters that affect the system characteristics in a smooth way. When such a smooth behavior is encountered and the values that one is dealing with are positive, then posynomials are often promising candidates. As it turns out, this applies to a large number of architectural-level parameters, such as resource sizes, latencies and miss rates.[4] Thus, we find that many of the system parameters that exhibit monotonic, diminishing returns profiles are modeled very effectively by posynomials (although posynomials can model more complex behavior as well).

While posynomials are quite effective in many cases, they cannot be applied everywhere, and a designer who wishes to use this approach should be aware of these cases and the appropriate alternatives. First, there are often discrete parameters in the design space of interest. For example, in Chapter 5, we will look at several high-level processor architecture decisions such as whether to use in-order versus out-of-order execution engines; as another example, a designer may wish to explore the decision of whether to use a distributed or centralized instruction window for instruction scheduling. In the presence of such large discrete design decisions, we generally prefer to create separate models. This makes the modeling easier and produces better fitted models.

While smooth, tunable, positive-valued data is often fit well by posynomials, these

[4]Even though these parameters may require integer value assignments, we consider them to be of the tunable, numerical type. The fits we produce will be continuous functions that go through the integer-valued data points; this helps with optimization, where continuous valued functions are preferred. Then, some post-optimization analysis can be performed to snap results back their closest acceptable value.

attributes are not, by themselves, a guarantee that posynomials will be able to capture the space. As we showed with both the example of the "worse" architectural fit in Figure 3.3 and the example of the cache fit in Figure 3.5, posynomials can sometimes still have difficulty in capturing certain characteristics. For the architectural fit, our particular posynomial function sometimes had difficulties in capturing abrupt changes in the data; for the cache miss rate fit, there were issues because the fitted curve needed to go into negative territory (even though the data itself was positive and region of the fitted curve we were interested in would also be positive). Fortunately, in practice, cases like this have been less common for the systems we have examined. Moreover, it is not too difficult to deal with these cases, with a number of potential approaches being possible. While we had a very specific fix for the cache miss rate case, one can also simply generate segmented models that fit different parts of the curve, switching between models as necessary. Alternatively, one can use a broader, less accurate fit as a starting point, refining the fit through several iterations as one identifies the region of interest. These last two approaches both take advantage of the fact that creating fits of smaller regions of a curve is easier. Finally, one may even be able to reason, in some cases, that inaccuracy in a particular part of a fit will have a negligible effect on the overall optimization. For example, it may be the case that when dealing with very low cache miss rates near zero, the miss rate may not have a dominant effect on either performance and energy; if that were the case, then slightly over-estimating those cache miss rates may not be a serious issue.

Despite these occasional fitting issues for certain specific benchmarks, in the majority of cases, we have found that the characteristics we need to model tend to fit well with posynomials. This is encouraging, because it means we can use this fit-based approach to effectively model and optimize complex systems. In the next chapter, we examine how to incorporate circuit trade-offs into these models to produce a circuit-aware optimization framework.

Chapter 4

Integrated Architecture-Circuit Optimizer

In Chapter 3, we showed how we can model a system at the architectural level. In this chapter, we examine how we can make these models circuit-aware so the space of different circuit implementations becomes known to the architectural optimization. In Section 4.1, we first look at how to explore and characterize the circuit space to create our circuit trade-off libraries. Then, in Section 4.2, we look at how we can integrate these circuit trade-offs into the architectural models, establishing the appropriate links between the two characterizations and creating the full-system model; during this integration, we incorporate pipelining into the analysis, and we also include voltage scaling. Finally, in Section 4.3, we look at how we can search the joint space to find an architecture-circuit co-optimized design.

4.1 Circuit Trade-offs

Just as with architectural design, the design of a circuit involves the analysis of many options and design parameters. A given circuit can be implemented in various ways

that trade-off energy, area and delay. The design space at this level includes the choice of circuit topology (e.g. ripple-carry adders, carry-lookahead adders, etc.), logic synthesis mappings, circuit styles (e.g. static, dynamic, etc.) and the sizing of gates, yielding a large space of possible designs. Because the energy, area and delay characteristics of these circuits directly affect the energy, area and performance of the higher-level system, we need to explore and characterize this trade-off space for each circuit block. Ultimately, our goal is to make these circuit trade-offs known to the higher-level architecture. Thus, after exploring and characterizing this space, we store it in a circuit trade-offs library so it can be referenced when optimizing the overall system (Section 4.2).

While the space we need to explore can involve a large number of parameters, only the final delay, energy and/or area of the circuit matter to the higher level system. Thus, we may need to explore a high-dimensional space, but we only produce characterizations that summarize the trade-offs between these primary metrics. For example, in the overview presented in Figure 2.6, we had simple energy-delay trade-offs for different units; although simply an illustration, these curves would have been generated from searching a large space of circuit designs. Once a certain circuit delay point is selected from such curves, back-referencing is used to determine the specific circuit implementation. This summarization of the circuit space into its primary metrics is an important characteristic of our modeling approach; it reduces the number of variables that need to be considered by the next level of the hierarchy and makes the optimization of the higher-level system much more tractable.

There are many tools that can help explore the circuit design space [15, 27, 54, 38]. Given a circuit topology, many of these tools can automatically generate energy-delay trade-off curves. By trying different discrete circuit topologies and circuit styles with these tools, one can then create a large trade-off space for a circuit [53]. Because we want to keep our framework general, we require only a set of design data points

annotated with delay and energy per operation (or area) characteristics. These design samples are then used to create fitted models in a manner similar to the approach used to create the architectural models. The extent to which a circuit's design space is explored and the accuracy of those characterizations are left to the discretion of the user. Larger circuit trade-off spaces and more detailed circuit data will produce better, higher fidelity results, but more easily obtained, approximate circuit data may be appropriate for high-level rapid design space exploration.

The requirements of the higher-level optimization affect the circuit-level modeling in several ways. First, because pipeline depth and operating frequency are optimization parameters that are not known until after performing the optimization, the circuit characterizations we make are for unpipelined logic. In the integrated models of Section 4.2, we determine how deeply each of the circuits needs to be pipelined, and we model the energy, area and delay overheads of inserting these pipeline registers. This allows the optimizer to automatically pipeline circuits as needed to quickly explore a space of different pipeline depths.

Secondly, because the higher-level architecture views each circuit as a single entity with a single trade-off curve (as in Figure 2.5), each circuit is characterized by a single delay parameter. This means that circuits are assumed to have fixed boundaries with aligned inputs and outputs. Thus, it is important that the logic in the circuit is properly balanced. In cases where logic is not balanced (e.g. two distinct cones of logic with different timing characteristics), the logic can be split into two separate circuit blocks.[1]

Although this approach sets certain restrictions that the designer needs to take into account, it results in a simpler architectural modeling abstraction: the top level

[1]This approach is meant for large circuit blocks, and assumes the logic is separable. For example, a branch target buffer and the PC incrementer are two sub-blocks that work in parallel to compute the next PC; these can be separated. If outputs are staggered in a more fine-grained way (e.g. a ripple-carry adder), it may be harder to separate the block into parallel components. In these cases, the delay of the unit is set by the critical path.

architecture links to these trade-offs during model integration by importing the circuit blocks and connecting them together to create the architectural organization. The architecture then becomes aware of each of the trade-offs of the different attributes—delay, energy and area—and can determine system performance given the operating point of each circuit.

Finally, in addition to modeling trade-offs in the logic components, we also account for the effects of wires at this level of modeling. While the delay and energy of local wires in a circuit are built directly into the circuit characterizations that circuit tools produce, the delay and energy of global wires that transfer data across large portions of the die need to be added explicitly. Because global wires may grow in length with the chip area, we also need to model this effect when expected to be significant. We thus create models for these global wires—treating them as energy-delay units like any other logic circuit—and include them in the system model that we use for optimization.

4.1.1 Characterizing Trade-offs

Given a set of design points, our goal is to create a model that characterizes the cost of a unit as a function of performance (delay). This cost could be area, energy per use, or something else; here, we consider energy, but the principles remain the same for other cost metrics. We examine multiple cost metrics in more detail later in this section.

To create the circuit energy-delay trade-offs, we use a tool-based data-fitting approach that parallels, in many ways, the approach used to create the architectural models described in Chapter 3. Just as we relied on architectural simulators to obtain architectural design samples, we use existing circuit design tools to explore the characteristics of different circuit implementations. We then use this data to create fitted mathematical trade-offs for each circuit.

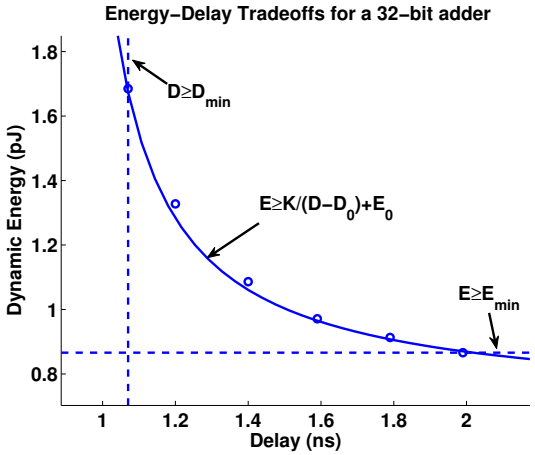


Figure 4.1: Characterizing a circuit trade-off space.

For purely logic units, the trade-off we need to characterize is between energy per operation and delay. For storage structures such as caches, buffers and queues, the delay and energy also depend on storage capacity. Thus, the trade-off characterizations that we produce generally take the form of:

$$Energy = f(Delay), \text{ for logic units}$$
$$Energy = f(Delay, Size), \text{ for memory structures}$$

Here, the energy per operation can be for random or prespecified input vectors, but should represent the average use case as closely as possible. Because we want to formulate the optimization problem as a geometric program, we use posynomial functions for f to produce these characterizations.

To produce the fits, we first filter out any non-Pareto optimal design points in the data sets. These are points which do not contribute to the optimal frontier, and which would never be used because there exist other designs points that are more efficient. Following this pre-filtering step, we then perform a data fit of the remaining

data points; this creates a mathematical characterization of the efficient frontier of the circuit space. The fit is produced using least squares, and we have found that a linear interpolation of the Pareto-frontier points prior to fitting often improves the fit, especially if the data points are not evenly distributed across the frontier. To produce a characterization, we have found the following three GP constraints to be effective in capturing the 2-D trade-off curves for our circuits:[2]

$$E \geq \frac{K}{D - D_0} + E_0 \tag{4.1}$$

$$E \geq E_{min} \tag{4.2}$$

$$D \geq D_{min} \tag{4.3}$$

Here, D is the delay, E is the cost (energy or area), and the rest are fit parameters. These equations represent a diminishing returns trade-off, where E_{min} is the minimum energy (or area), and D_{min} is the minimum delay of the circuit. The first constraint is produced through fitting. The other two constraints are important to ensure that the model is restricted to a trusted region; unfeasible design points could otherwise be selected.

Figure 4.1 shows one particular fit for the energy-delay trade-off of a 32-bit adder. The round dots represent data points produced using a synthesis tool, while the three lines show the mathematical model used to represent this space. The figure shows that the fitted model is able to track the synthesis data very closely. This result is representative of the general case; the trade-off curve fits we produce for a large number of circuits often fit the data within 5% error or less.

[2]Strictly speaking, the first constraint is not a GP constraint. However, it can be decomposed into one as follows: $t_D \leq D - D_0$, $t_E \leq E - E_0$ and $t_D \times t_E \geq K$.

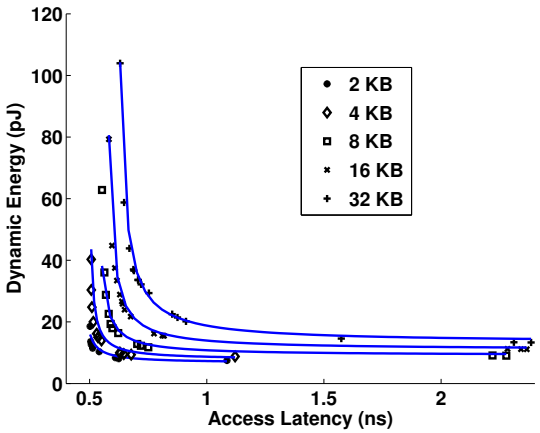


Figure 4.2: Characterization of cache access energy as a function of both size and delay. Data points extracted from CACTI [49] for a 2-way cache.

For structures such as caches and buffers, delay and energy also depend on the storage capacity. To capture this dependency in the circuit library, we need a more complex fitting function. We start with the same basic trade-off curve, but parametrize each of the fit parameters with monomials:

$$K \rightarrow K' \times \text{size}^{\beta_K} \tag{4.4}$$

$$D_0 \rightarrow D_0' \times \text{size}^{\beta_{D_0}} \tag{4.5}$$

$$E_0 \rightarrow E_0' \times \text{size}^{\beta_{E_0}} \tag{4.6}$$

$$D_{min} \rightarrow D_{min}' \times \text{size}^{\beta_{D_{min}}} \tag{4.7}$$

$$E_{min} \rightarrow E_{min}' \times \text{size}^{\beta_{E_{min}}} \tag{4.8}$$

where the primed variables and various βs are new fit parameters. In this way we produce a function that defines the cost as a joint function of its size and delay.

Figure 4.2 shows an example of applying such a fit for a cache over a range of sizes suitable for an L1 cache. Five sets of data are shown for cache sizes between

2KB and 32KB, with solid lines representing the fitted functions. Not shown are the boundary conditions, D_{min} and E_{min} of which there would also be five sets.

Generally, creating fitted energy models as a joint function of size and delay can be harder to produce than with the simple energy-delay trade-offs since there are more data points that need to be simultaneously captured. As with any fitting process, there is a fitting trade-off between accuracy and the size of the space captured. To achieve acceptable accuracies in the cases of memory structures, we often restrict the range of sizes to a limited set of values—4 or 5 sizes often performs well. This is not a serious restriction, however, since, after optimizing, one can always identify whether the domain of values that was considered was appropriate or not (if an optimized value is at its maximum or minimum, then it suggests that one should move the domain of values up or down respectively). Alternatively, one can use an iterative approach, where a first pass uses broad characterizations to find the local region of interest, followed by a second pass which focuses in on that region with higher accuracy characterizations.

Regardless of the level of detail used, each of these joint energy-size-delay trade-off curves is stored in the circuit library, along with all the other basic trade-off curves. The optimizer can then access these trade-offs to create an integrated architecture-circuit model.

4.1.2 Multiple Cost Metrics

In the previous section, we considered how to characterize dynamic energy as a function of delay, and sometimes, size. However, dynamic energy is not the only cost attribute of a circuit that we may be interested in. For example, when optimizing for total power, we must also take into account the circuit's leakage. Moreover, area is another important cost metric in chip design.

In general, trying to simultaneously capture the space of all these metrics requires

a higher dimensionality characterization. For example, there could theoretically be trade-offs between dynamic energy and leakage power, where the designer would need to select whether he prefers a circuit implementation with higher dynamic energy costs but lower leakage, or an alternative design with lower dynamic energy but higher leakage. This would require a multi-dimensional trade-off between energy, leakage and delay. While it is possible to create such trade-offs—we showed how to do so with energy, delay and size in the last section—we would like to avoid these situations whenever possible, as the creation of higher dimensionality trade-offs requires more effort.

Fortunately, in practice, relationships between parameters often allow us to create simpler, lower dimensional characterizations of the design space. For example, a faster circuit requires more resources, which typically results in a higher dynamic energy per use, more leakage and more area all at the same time (in fact, leakage and area are usually very highly correlated). This often allows us to simplify the modeling problem, by characterizing separate costs as a function of the same input (delay and, if applicable, size).

Figure 4.3 shows an example of such a situation using a floating point multiply accumulate (FPMAC) circuit data produced from a synthesis tool. For each of the three cost functions—dynamic energy, leakage power and area—we find the same physical design points (round dots) on the Pareto-optimal frontier. From a modeling and optimization perspective, this means we can can separately produce analytical fits of each of the data sets. Then, by turning a single delay knob, we can cause the three cost functions to change simultaneously.

Although we would like to use such simplifications when possible, they are not always applicable. For example, if building a circuit library that has both dynamic and static CMOS circuits, we can potentially run into situations where there are trade-offs between the dynamic energy and area costs: dynamic circuits have higher

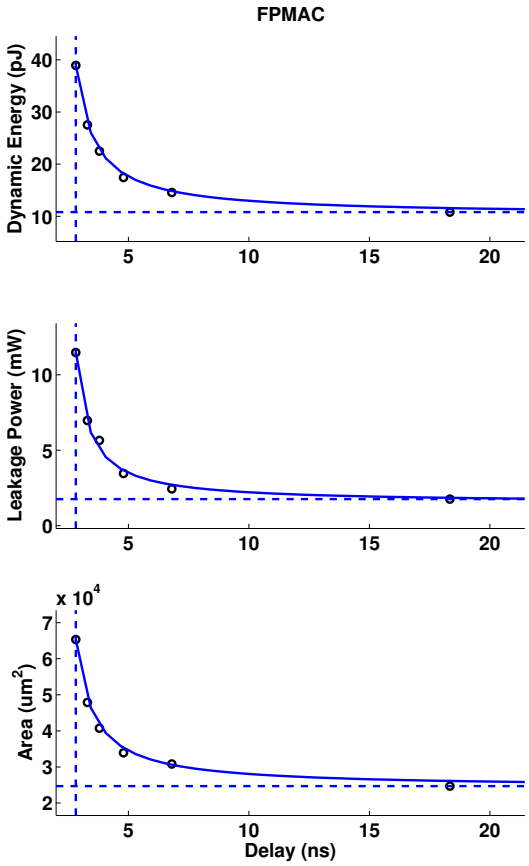


Figure 4.3: The high correlation between dynamic energy, leakage power and area of designs produced by a synthesis tool makes it possible to decompose the multi-dimensional trade-offs into independent functions of delay.

energy because of their clock power (the clock transistors transition frequently), but may have lower area because of the reduction in the number of transistors. As another example, changing the threshold voltages of devices affects a circuit's area, dynamic energy and leakage in different ways. In scenarios such as these, one must either find a way to break the characterization into separate components (for example, finding equations that describe how threshold voltages affect leakage and dynamic energy)[3] or one must fall back to creating multi-dimensional trade-offs. When we examine the use of dynamic circuits in the results of Chapter 5, we actually choose to create separate library modules for the static and dynamic implementations of each circuit, explicitly specifying which library to use based on the study.

4.1.3 Global Wires

From the perspective of the higher level system, global wires are treated logically like any other energy-delay unit. As there is not much of a design space for wires, global wires are in some ways easier to model than the circuit trade-offs. In fact, if it were not that global wires change in length with chip area, they could simply exist in the circuit library as single energy-delay points. Global wires, however, do grow in length with chip area, so to properly account for the delay and energy of wires, this effect needs to be modeled.

Establishing this relationship between energy and delay of wires versus chip area is not difficult. In the simplest case, one can simply scale the energy and delay of

[3]We will see this kind of approach applied later in this chapter when examine how to model supply voltage. We separately characterize how supply voltage affects the delay, dynamic energy and leakage of circuits, and we then use this characterization to specify how the basic trade-off curves scale with the supply voltage parameter.

wires linearly with wire length:

$$D_i = D_{wire} \times L \tag{4.9}$$

$$E_i = E_{wire} \times L \tag{4.10}$$

$$L = \gamma \times \sqrt{A_{total}} \tag{4.11}$$

Here D_i and E_i represent the delay and energy of the wire i respectively. D_{wire} and E_{wire} represent per unit length delay and energy costs, while L is the length of the wire. L, itself, is dependent on the fraction, γ, of the total chip length the wire needs to cross and the full length of one side of the chip, $\sqrt{A_{total}}$ (A_{total} is total area, and this formulation assumes square aspect ratio).

One can easily develop variations on this model as needed. For example, this model assumes that the wire length grows proportionally with the area of the entire chip. If, however, a medium-range wire grows with a more localized area of the design, then creating such relationships is not difficult. One simply needs to replace A_total with the area of interest.

Finally, it should be noted that performing such an evaluation assumes that the circuit libraries one generates have area components built in as described in the previous subsection. This then allows the optimizer to dynamically compute the delay and energy costs of global wires as parameters such as cache size change the total die area and cause the length of wires to grow. Having mentioned this, we note that in the results we present in Chapter 5, we mostly use fixed global wire costs.

4.2 Integrated Architecture-Circuit Model

Having shown how to create analytical characterizations of the architectural spaces and how to generate circuit libraries, the next step is to integrate the two models

together to create a joint model. This step involves not only making the energy models aware of the different potential implementations and their costs, but it also involves making the architectural models—which usually operate on cycles—aware of the physical delay of the circuits. This prevents the architect from inadvertently placing unrealistic amounts of logic into a single pipe stage, holding him accountable for architectural changes that require more circuit evaluation time.

4.2.1 Cycles, Circuit Delays and Pipelining

As mentioned in Chapter 2, in simulation-based architectural evaluations, the basic unit of time is the cycle. All the various events and processing latencies within the simulator are described in terms of the number of cycles they require, and simulations report final performance in terms of the total number of cycles (or, more frequently, the cycles per instruction, CPI). Any experienced architect, however, also knows that real performance depends not only on the CPI, but also the cycle time, T_{cyc}, that can be achieved. What we really need to measure, therefore, is the total execution time. Normalized to a per instruction basis, the desired performance metric is thus the time per instruction, TPI:

$$TPI = CPI \times T_{cyc} \tag{4.12}$$

To model the effect of both CPI and the cycle time, we need to simultaneously consider the architecture and also how underlying circuits affect the pipeline structure. By linking to a library of circuit implementations, our circuit-aware modeling approach is able to more accurately model these components of TPI; we can associate architectural blocks to physical circuit implementations, and since we know exactly how long each circuit block takes, we can determine how the cycle time or pipeline structure will have to change to accommodate different circuits.

Thus, one of the functions of our integrated model is to make the architectural model aware of the real delays. More formally, we do this by linking the physical delays, D_i, of units in the circuit library to the cycle-based latency, N_i, in the architectural model. The number of pipe stages in a unit is essentially the delay of the underlying circuit divided by the clock period T_{cyc}. This simply represents cutting the logic into stages. Adding delay overheads for pipeline registers, we get the following relationship

$$N_i = D_i/(T_{cyc} - T_{ff}). \tag{4.13}$$

where T_{ff} includes the delay overheads of registers caused by register setup, clock-to-q times and clock skew.

In this equation, the cycle time, T_{cyc}, is a design space variable (along with D_i and N_i). Any, or all, of the design parameters can change. For example, by changing T_{cyc} while holding D_i constant, we can explore different pipeline depths of a unit (N_i) for a fixed logic implementation. Alternatively, if we vary D_i, while holding T_{cyc} fixed, we can again explore different pipeline depths, but this time causing the underlying circuit implementation to change to be faster or slower. Finally, we also can hold the number of stages, N_i, fixed, meaning that when the frequency changes, the underlying circuit needs to speed up or slow down accordingly.

In the context of optimization, the inclusion of the cycle time parameter means we can explore the trade-offs of pipelining to different depths, and the ideal clock frequency can be identified along with all other circuit and architectural parameters. Of course, we need to account for the energy and area overheads of pipeline registers. For this, we need to know the approximate number of pipeline registers in each unit, R_i. We characterize this as a simple function of the number stages, N_i, and the

average logic width, W_i:

$$R_i = (N_i)^{\eta} \times W_i. \tag{4.14}$$

The parameter η allows for modeling super-linear pipeline register growth, and can be unique for each architectural block.

We can also constrain how deeply a unit can be pipelined. For example, control units such as next-PC logic or a processor's instruction scheduler often need to compute once a cycle and cannot be pipelined. In these cases, we simply add a constraint that $N_i = 1$.

As a final note, in this formulation, there is an implicit assumption that perfect pipelining can be achieved when pipelining a circuit (i.e. the logic can be evenly cut into the N_i stages); in reality, this is not always true. There are a couple cases to consider here. In cases where latches with transparency windows can be used or intentional clock skewing can be applied, some degree of uneven logic distribution can, in fact, be managed as long as there are no hold-time violations. On the other hand, when flip-flops with fixed clock edges are being used, one may want to add overheads to account for non-ideal pipelining. This can be done by characterizing an additional overhead term, $T_{overhead}$, that represents the maximum overhead across all stages due to uneven pipeline stages. Then, Equation 4.13 would remain unchanged, but $T_{overhead}$ would be added to T_{cyc} before its use in the TPI calculation of Equation 4.12. While modeling the pipeline overhead in this fashion is possible, the results we present in Chapter 5 assume perfect pipelining.

4.2.2 Cost Functions

In addition to tying the architecture to the circuit delays, a circuit-aware approach is able to determine the energy cost of a unit based on the circuit implementation used; by linking to the trade-off curves discussed earlier in this section, costs can

be computed based on the performance required from the circuits. Our next task is now to create the overall energy models for the entire system from the energy of the components that come from the circuit trade-offs. While this section focuses on energy models, area models follow the same principles. In fact, area modeling is generally simpler to handle since area is a cost that is determined completely at design time,[4] whereas energy costs depend not only on the design, but also on how applications exercise the different parts of the design (i.e. activity factors). Energy costs are also more complicated in that they have several different components—dynamic energy and leakage—that need to be considered.

For the purposes of creating the energy models within the optimization framework, we break total energy into several components: energy spent in the logic (including global wires), in pipeline registers, and in the clock distribution network. For the energy within the logic and registers, we further split the energy into dynamic and leakage components.

We decompose the total energy into these components because of how they relate to different design choices. The energy spent in logic components depends on the aggressiveness of the circuits, and links to the trade-offs in the circuit libraries. The energy spent in pipeline registers depends on the cycle time and how deeply we pipeline the design. Finally, the energy spent in distributing the clock also depends on the number of registers, but has a different activity factor than the internal power of the data transitions within the registers.

For the logic component, the total dynamic energy depends on the average energy consumed per use of each circuit, E_i, multiplied by its activity rate α_i:

$$EPI_{logic,dynamic} = \sum_i (\alpha_i \times E_i) \tag{4.15}$$

[4]We assume here that area of the different components are additive, and that packing/floorplanning issues are not a concern. This is more or less true for synthesized designs that use modern place and route tools.

The energy cost E_i links directly to the circuit trade-offs. Thus, using faster circuits will naturally cause an increase in total logic energy. The activity factor α_i represents how often a unit is used per instruction and depends on the application characteristics as discussed in Chapter 3. This characterization of dynamic energy assumes that the design is clock gated: the activity factors represent only real work being performed by the circuit. At all other times, the circuit is assumed to be idle, with no switching activity and no dynamic energy consumed.

Each circuit also has a rate at which it leaks, P_i, from which we can determine the total leakage power dissipated in the logic components:

$$EPI_{logic,leakage} = TPI \times \sum_i (P_i) \tag{4.16}$$

Like its dynamic energy counterpart, the P_i values link to the circuit trade-offs (assuming we have created circuit libraries with leakage components as described in Section 4.1.2). Thus, trying to use faster circuits will cause increases in leakage power in addition to the extra dynamic energy). Since leakage power is a rate of energy consumption (watts = joules/second) and is independent of whether a circuit is being used or not, we have to convert the leakage power to an energy per instruction basis. We accomplish this by multiplying by the average time per instruction, TPI. This essentially distributes the leakage cost across all instructions. Thus, if the rate of instruction processing is lower, more energy will be wasted in leakage relative to the number of instructions, and leakage will have a larger contribution to the total EPI.

There are a few points that should be noted in this formulation of leakage power. First, this formulation assumes that circuits are always on, and therefore always leaking. Power gating could be used to virtually eliminate leakage in a circuit when that unit is expected to be inactive for long periods of time, but this ability to apply power

gating depends on the application behavior. With more application characterization, one can adjust these models to determine how often a circuit can be power gated, and then include gating factors into the analysis. As another issue, it should be noted that leakage is a property that can change considerably with temperature; this is not included in the current characterization, but could be added with more modeling. The leakage power values in the circuit libraries we have generated come from the underlying tools used to produce the data (in our case, synthesis tools), and thus use the same assumptions as those tools.

We account for the energy spent in registers separately from the logic because it depends on pipeline depth and the number of pipeline registers, which changes at optimization time with the cycle time. To compute total dynamic register energy, we take the average energy cost of a single register, E_{ff}, and multiply by the total number of pipeline registers in a unit, R_i, times the unit's activity factor, α_i. We also have a leakage component that is based on the average leakage power of a single register, P_{ff}:

$$EPI_{reg,dynamic} = \sum_i (\alpha_i \times R_i \times E_{ff}) \tag{4.17}$$

$$EPI_{reg,leakage} = TPI \times \sum_i (R_i \times P_{ff}) \tag{4.18}$$

So while increasing pipeline depth will improve performance, it also results in increased energy because of a larger value of R_i.

The last component of dynamic energy is clock power. Energy spent in the clock distribution network depends on the clock load, which is dependent on both the number of pipeline registers that are driven and also an intrinsic component that represents the distribution network itself (wires). This latter component is a function

of the area of the chip. Putting these two components together, we get:

$$EPI_{clk} = CPI \times (E_{ffclk} \times \sum_i (R_i) + f(\sum_i (A_i))) \tag{4.19}$$

Here, E_{ffclk} is the average clock energy contribution per register per cycle, which we extract from real designs. The summation $\sum_i (A_i)$ represents the total area as computed from the area of each of the constituent circuits. A function f is used to account for how chip area affects the energy spent in the clock network; this function is obtained empirically by extracting energy numbers for designs of different sizes. Both these components are summed and then multiplied by the number of cycles per instruction, CPI, to convert the final energy value to a per instruction basis.

Putting everything together, we get

$$EPI_{logic} = EPI_{logic,dynamic} + EPI_{logic,leakage} \tag{4.20}$$

$$EPI_{reg} = EPI_{reg,dynamic} + EPI_{reg,leakage} \tag{4.21}$$

$$EPI = EPI_{logic} + EPI_{reg} + EPI_{clk} \tag{4.22}$$

where the overall EPI is the final optimization metric used when optimizing for energy efficiency.

4.2.3 Voltage Scaling

So far, we have focused primarily on design knobs in the circuit and architecture domains. Voltage, however, is a powerful knob that can produce significant trade-offs between speed and energy. It is therefore important to include voltage into the optimization analysis.

One way to incorporate voltage scaling would be to include it directly into the circuit trade-offs. However, since the underlying design details are encapsulated when

creating circuit trade-offs—only the primary delay, energy and area metrics are exposed to the architecture—there would be no way to control the voltage parameter. This would mean each unit could operate at its own preferred voltage.

Instead we characterize delay and energy scaling factors as a function of voltage and introduce scaling functions to allow the optimizer to scale the basic circuit trade-off curves at optimization-time. The scaling functions we use have the following form:

$$M_D = a_D \times (V_{DD})^{b_D} + c_D \tag{4.23}$$

$$M_E = a_E \times (V_{DD})^{b_E} \tag{4.24}$$

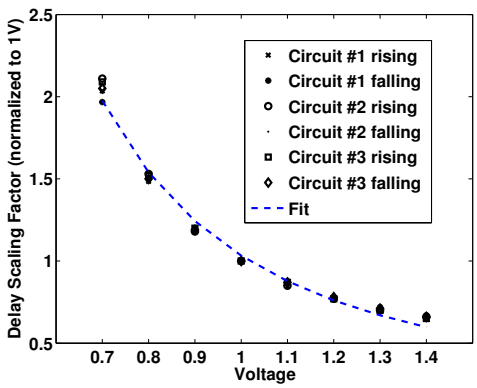
$$M_P = a_P \times (V_{DD})^{b_P} + c_P \tag{4.25}$$

In these equations, V_{DD} is the supply voltage, while M_D, M_E and M_P are delay, energy and leakage scaling factors respectively. The remaining variables are constants that need to be characterized.

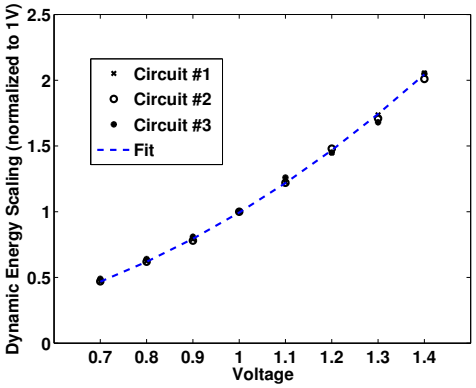
Figure 4.4 shows how the delay, dynamic energy and leakage scale with voltage on several small circuits simulated in SPICE (data is normalized to $1V$). The dotted line shows the mathematical characterization of the data. In these characterizations, energy scaling follows an expected V^2 profile, while leakage and delay also change as one would expect. The choice of circuit has little effect on the scaling behavior, meaning that we can apply these results to circuits in general to create the effect of voltage scaling.[5]

We therefore use these scaling factors to scale the basic circuit trade-off curves and other energy and delay values. This scaling is done before the trade-off curves are

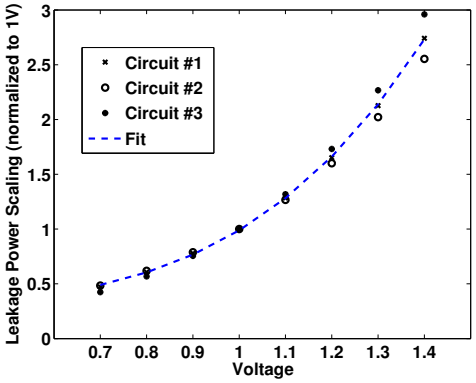
[5]There are certain cases where these scaling characteristics cannot be applied, most notably where full-swing signaling is not used. One kind of circuit in which this is often the case is large on-chip memories such as the L2 cache. The bit-lines in such large memories typically do not completely discharge, and instead sense amplifiers are used to detect the signal. In these cases, one must be careful to use the correct scaling characteristics. Fortunately, in most circuits, including static CMOS logic and dynamic logic, full-swing signaling is used, and the standard scaling characteristics can be safely applied.



(a) Delay Scaling



(b) Energy Scaling



(c) Leakage Scaling

Figure 4.4: Characterization of how delay, energy and leakage scale with voltage. Various circuits were simulated in SPICE, and then fits were produced.

linked with the architectural models, and is represented mathematically as follows:

$$D' = D \times M_D \tag{4.26}$$

$$E' = E \times M_E \tag{4.27}$$

$$P' = P \times M_P \tag{4.28}$$

These new "primed" variables are used instead of the original variables for the equations throughout this section.

Having voltage explicitly modeled using these forms gives us more control over the voltage scaling assumptions in the design. By using a single voltage variable that scales all circuit trade-offs, we can restrict the design to be a single-V_{DD} design. On the other hand, by using multiple voltage variables, we can still model multi-V_{DD} designs if we so choose.

4.3 Optimization

Having modeled the architectural space, the circuit trade-offs and their interactions, the final step is to search the design space defined by these models to find a design configuration that best meets the designer's needs. While one could use a variety of heuristics or other algorithms to link to the models and perform the search, our approach is to formulate a formal geometric program (GP) optimization problem; using a GP allows us to leverage powerful mathematical optimization techniques that search the design space efficiently and robustly to find the optimal design.

The first step in this process is to describe the system behavior in the GP. This involves taking each of the equations from this chapter and Chapter 3, and then molding them into a GP form. The equations we have are already in posynomial form—we specifically ensured ensured we used posynomials in each of our modeling steps, and

it is for this reason that we are looking to use a GP optimization solver in the first place—but this process still requires some minor transformations to the equations. The primary issue consists of our use of equalities to describe system characteristics; although we represented all the system characteristics as equalities, a geometric program requires that most constraints be in the form of inequalities. Thus, most equations must be modified to use "≥" instead of "=" with only a monomial being allowed on the left side of the inequality.[6] This needs to be done carefully, to ensure that the wrong problem is not being solved. Fortunately, for the performance-cost characterizations that we make, it turns out we can do this by setting the performance metrics and energy costs on the left (e.g. $E \geq f(...)$ or $CPI \geq f(...)$). The pressure to reduce waste during optimization will then cause these inequalities to be tight.

The next step is to provide the top-level design objective and constraints to the geometric program as defined by the designer. These constraints typically apply to the high-level characteristics of the system, such as overall performance, energy and area. For example, the optimization problem can be to *minimize* EPI for a given performance target ($TPI \leq TPI^{max}$) in the case of a high-performance design or to *minimize* TPI (maximize performance) for a given energy budget ($EPI \leq EPI^{max}$) in the case of an embedded/low-power design. Alternatively, one can optimize for other metrics such as the energy-delay product, *minimize* $EPI \times TPI$. If the circuit libraries include area models, then it is also possible to optimize for more complex metrics such as performance per unit area, which is important in CMP designs.

Figure 4.5 shows pseudo-code for a geometric program formulation that demonstrates the formulation of the optimization problem. A constraint is listed for each particular aspect of the system. Together, all the constraints describe a complex system with interconnected parameters. For example, changing a delay parameter D_i in a given circuit affects N_i and T_{cyc}; N_i then affects CPI; and both T_{cyc} and

[6] For example, $x \geq y + z$ is a valid GP constraint, but $y + z \geq x$ is not

CPI affect the final performance metric TPI. At the same time, a change in D_i also affects the energy cost of the circuit, E_i, which similarly causes a chain reaction that ripples all the way to the final energy metric EPI. Moreover, the change in N_i will cause additional energy in the pipeline registers which also affects EPI. One could take this example even further, as there are other interactions as well, but our point is simply to show the degree of analysis required to evaluate trade-offs in a complex system. Fortunately, because we are using a GP solver, we can simply describe all these constraints as an optimization problem; the optimizer then accounts for all the complex interactions and trade-offs between parameters in its analysis.

Thus, having described all the design space equations and constraints, the last step is to execute the GP solver. The GP solver transforms the problem to a convex optimization problem by performing log-transforms on the equations, and then uses a convex solver to automatically search the space. The optimizer essentially evaluates the cost-benefit sensitivities of each parameter to quickly find the optimal set of design knob values. We use GGPLAB [50] as our GP solver, although any GP solver will do. Using a desktop computer system with a 3.2 GHz Pentium 4 CPU, this optimization phase takes only about 30 seconds per optimization run for optimizing a fairly large out-of-order processor, with simpler designs taking even less time. By sweeping the design target—for example, by iteratively solving the problem with increasing energy budgets—one can map out the overall trade-off space of the entire system.

The output of the optimization is the set of values for each parameter in the design space. This includes values such as the latencies of units, the sizes of memories, the clock frequency and the supply voltage. The costs of these parameters and their sensitivities can also be easily accessed. An additional back referencing step through the circuit library provides the internal circuit design configuration that achieves the requested circuit specifications.

It should be noted that the results produced by this framework will be continuous

Circuit trade-offs library
for each basic circuit i

Dynamic energy

$$E_i \geq \frac{K_{E,i}(size_i)}{D_i - D_{0,i}(size_i)} + E_{0,i}(size_i)$$
$$E_i \geq E_{min,i}(size_i)$$

Leakage power

$$P_i \geq \frac{K_{P,i}(size_i)}{D_i - D_{0,i}(size_i)} + P_{0,i}(size_i)$$
$$P_i \geq P_{min,i}(size_i)$$

Area

$$A_i \geq \frac{K_{A,i}(size_i)}{D_i - D_{0,i}(size_i)} + A_{0,i}(size_i)$$
$$A_i \geq A_{min,i}(size_i)$$

$$D_i \geq D_{min,i}(size_i)$$

Architectural models with posynomial characterization $f()$

$$CPI \geq f(..., N_i, ..., size_j, ...)$$

Link circuit delays to architectural latencies
for each linked pair (i,j) (circuit i, architectural latency j)

$$N_j \geq \frac{D_i}{(T_{cyc} - T_{ff})}$$

Model pipeline registers
for each architectural stage i

$$R_i \geq (N_i)^{\eta} \times W_i$$

Logic energy

$$EPI_{logic,dynamic} \geq \sum_i (\alpha_i \times E_i)$$
$$EPI_{logic,leakage} \geq TPI \times \sum_i (P_i)$$
$$EPI_{logic} \geq EPI_{logic,dynamic} + EPI_{logic,leakage}$$

Pipeline register energy

$$EPI_{reg,dynamic} \geq \sum_i (\alpha_i \times R_i \times E_{ff})$$
$$EPI_{reg,leakage} \geq TPI \times \sum_i (R_i \times P_{ff})$$
$$EPI_{reg} \geq EPI_{reg,dynamic} + EPI_{reg,leakage}$$

Clock energy

$$EPI_{clk} \geq CPI \times (E_{ffclk} \times \sum_i (R_i) + f_{clk}(\sum_i (A_i)))$$

Total performance
$TPI \geq CPI \times T_{cyc}$

Total Energy
$EPI \geq EPI_{logic} + EPI_{reg} + EPI_{clk}$

User-defined objective and constraints
(In this case to minimize TPI for an energy budget EPI^{max})
minimize TPI
subject to
$EPI \leq EPI^{max}$

Figure 4.5: Simplified pseudo-code of the final optimization problem constraints. Some effects such as voltage scaling are left out for simplicity. The final optimization objective and constraint in this example is to minimize TPI subject to an energy budget, but it could also be to minimize EPI for a performance target, or any other objective and constraints from the user.

in nature because of the analytical modeling and optimization. This may not be a serious issue for certain parameters, but has to be managed for others such as the number of pipe stages in a functional unit.[7] Some post-optimization snapping of the results to discrete values is therefore needed. This snapping can be done in various ways. Greedy algorithms that consider the sensitivities of each of the parameters could be used, possibly with re-optimizations as parameters are locked down. As we will see in the results presented next, however, voltage scaling makes this discretization issue less significant.

4.4 Discussion

The framework discussed in the last two chapters brings together a number of techniques to produce a general, yet powerful, tool for evaluating design decisions and

[7]Intentional clock skewing and register retiming, however, may sometimes allow a designer to work with non-integer values of pipe stages

their cost-performance trade-offs. It applies a systematic approach to design, intrinsically relying on marginal cost analysis to help rigorously optimize decisions at both the architectural and circuit levels.

At its core, the framework is driven by models that describe the system behavior, characterizing how metrics such as performance, energy and area change with design parameters. The general method used in creating these models is primarily a sample-and-fit approach. To explore large architectural spaces, for example, the framework uses statistical inference on a relatively small set of simulation samples to create regression models of the entire architectural space. At the circuit-level, we see a similar approach of using data-fitting to characterize the design trade-offs.

While perhaps a simple idea, this general approach of fitting data points enables the modeling of a wide range of systems. In theory, this approach can be applied to virtually any kind of system; to create architectural models, all that is required of the designer is a system simulator, a tool which must be developed in any design process regardless. By turning the different design knobs in the simulator and extracting performance samples of random design configurations, it becomes possible to create powerful models that describe a much larger space of designs. The models can then be used to determine the importance of each design knob and how design knobs interact.

For generating the circuit-trade-offs, the approach was once again quite general. The designer is free to use the circuit optimization tool of his choice—of which there are many—simply providing the energy-delay points into the circuit library. These data points are then processed and fitting is used to create a mathematical characterization of the design trade-offs.

While applying a mathematical characterization is, in itself, powerful, the framework goes further by applying posynomial functions to produce the fits. By doing so, the framework restricts the space to being log-convex, enabling the use of geometric programming solvers to search the large, multi-dimensional space to find the optimal

design.

This approach assumes that log-convex models of desirable accuracy can be produced. If a good architectural model for the entire design space cannot be fitted, however, then it may be necessary to segment the design space and model the segments separately. This, of course, only works to a certain degree, otherwise degenerating into a manual search. Alternatively, lower-accuracy models can be used to locate the general area of a result and a second iteration of the methodology focused on that area (and with higher-resolution models) can be used to produce a finer-grained optimization result.

While these issues could theoretically complicate matters, in practice, we have found that for most design knobs in a system—excluding discrete design decisions—the use of posynomial functions is generally quite effective in modeling the system. In the next chapter, we apply the framework to study energy-performance trade-offs in the processor design space, in process of which we also demonstrate the effectiveness of the modeling and optimization methodologies used by the framework.

Chapter 5

Processor Optimization

In Chapters 3 and 4 we showed how we can model and optimize digital systems, from the high-level architecture down to the circuit trade-offs. In this chapter, we use this framework to study the energy efficiency of general-purpose processors. We examine various different high-level architectures—from a simple in-order core to an aggressive multiple-issue out-of-order core—in addition to examining pipeline depth, various lower-level microarchitectural knobs and circuit design trade-offs for each of these architectures. We first consider this design space without voltage scaling, and then study how the introduction of voltage scaling changes the basic results. Before presenting the results, however, we look at the experimental setup of our study: the design space we consider, the benchmarks we use, and the accuracy of the models we generate.

Parts of the work in this chapter were performed in collaboration with Aqeel Mahesri and Sanjay Patel at the University of Illinois at Urbana-Champaign.

Table 5.1: Microarchitectural design space parameters.

Parameter	Range
Branch predictor	0-1024 entries
BTB size	0-1024 entries
I-cache (2-way) size	2-32KB
D-cache (4-way) size	4-64KB
Fetch latency	1-3 cycles
Decode/Reg File/Rename lat.	1-3 cycles
Retire latency	1-3 cycles
Integer ALU latency	1-4 cycles
FP ALU latency	3-12 cycles
L1 D-cache latency	1-3 cycles
ROB size	4-32 entries
IW (centralized) size	2-32 entries
LSQ size	1-16 entries
L2 cache latency	8-64 cycles
DRAM latency	50-200 cycles
Cycle Time	unrestricted
Supply Voltage	0.7-1.4 V

5.1 Experimental Methodology

To study the processor design space, we examine six different high-level processor architectures: single-issue, dual-issue and quad-issue designs, each with both in-order and out-of-order execution. This covers a large range of the traditional architecture space, from a simple lower-energy, low-performance single-issue in-order processor to an aggressive higher-energy, high-performance quad-issue out-of-order processor.

For each of these high-level architectures, there are then various tunable microarchitectural parameters that trade-off energy and performance. Table 5.1 lists these parameters for the design space we explore. This microarchitectural space consists of billions of possible design configurations for each high-level architecture, and this is without even taking into account the circuit design space we explore.

For our study, we use a large 8MB L2 cache with a fixed access time. The large

Table 5.2: Percent errors of architectural performance (CPI) models generated through statistical inference.

	1-issue in-order	**2-issue in-order**	**4-issue in-order**	**1-issue ooo**	**2-issue ooo**	**4-issue ooo**
bzip2	0.49	0.55	0.41	3.76	4.43	4.66
crafty	6.18	6.70	8.25	6.56	7.74	7.77
eon	3.93	9.25	7.94	6.37	7.50	7.27
gap	1.10	1.13	1.30	4.58	5.01	4.77
gcc	4.45	2.31	6.58	4.93	6.30	5.54
gzip	1.25	1.24	6.25	3.48	4.10	4.24
mcf	2.33	5.69	5.78	5.83	6.42	8.99
parser	0.63	1.39	0.87	3.41	4.13	3.80
perlbmk	3.37	2.07	3.62	5.62	7.38	5.98
twolf	2.36	3.40	3.14	4.08	5.97	5.27
average	**2.61**	**3.37**	**4.42**	**4.86**	**5.90**	**5.83**

L2 cache was used because it reduces costly accesses (both energy and delay-wise) to the main memory and lets us focus on the energy-efficiency of the processing core. Because the clock cycle time of the core is an optimization parameter, the relative access latency in cycles can still vary, and so the L2 access latency is included in the design space. The DRAM latency is likewise included in the design space because the core frequency can change.

We considered two branch prediction schemes for our processors: a simple table of 2-bit saturating counters, and a YAGS branch predictor [21]. We tried both predictors on all architectures, and found that from an energy-performance perspective, the performance return of the YAGS predictor was usually worth the small increase in total energy. Thus, we use the YAGS predictor in all architectures except the single-issue in-order design, where we use simple 2-bit counters instead. We found the 2-bit counters were still useful for very low energy budgets, so we use this simpler predictor for the single-issue in-order design. The lower CPI of these simple designs also means that more aggressive predictors are not as important.

As benchmark suites, we use a subset of SPEC CPU benchmarks. We simulate 500 randomly generated design configurations per benchmark, from which we then generate our architectural models through statistical inference. We set aside a percentage of these these samples for validating our models. Median fitting errors are listed in Table 5.2. The most complex out-of-order architectures exhibit errors that are a bit higher because of the additional complexity of the system being modeled. Two of the benchmarks, crafty and eon, also show higher errors because they have certain parameters that cause sharp changes in performance; our particular posynomial functions fit this data more smoothly, resulting in some additional error. Neverthelss, averaged over all benchmarks, our models have errors of 6.0% or less.

For this study, we use a CMOS 90 nm technology. This technology's leakage current is relatively low, so for the results we present, we consider dynamic energy consumption only. We discuss how the inclusion of leakage currents would affect the results in Section 5.6.

To create the circuit energy-delay characterizations for our circuit libraries, we use a mixed approach. For logic units and small memory structures (queues, register files, etc.), we build Verilog implementations, and use Synopsys Design Compiler to synthesize each of these blocks. By sweeping the timing constraint on these blocks, Design Compiler produces different logic topologies, synthesis mappings and gate sizings that trade-off energy and delay. Designs with tighter delay constraints use more aggressive mappings and larger gates, resulting in higher energy per use.

For larger memories such as the memory caches and the BTB, we use CACTI 6.0 [49] to characterize the energy-delay trade-off space. CACTI searches the space of possible SRAM memory organizations to evaluate access time and power characteristics of design points. We use CACTI to extract all energy-delay points in this search space, which we then use to construct energy-delay trade-off models.

Because we use synthesis tools, most of the circuits trade-offs we produce are

static CMOS circuits, although CACTI models dynamic circuits in parts of the SRAM memories. We consider dynamic circuits by using alternate circuit libraries that we discuss in Section 5.4.

We use this approach to characterize the energy-delay trade-offs for all the major blocks in the processor: the ALUs, the caches, the reorder buffer, the instruction window, etc. While we have taken care to include all major components in a processor, there are often numerous smaller units and state registers that are present in commercial designs that we are not including. Moreover, while characterizing energy-delay trade-offs for individual circuit blocks is straightforward, accounting for the communication in a processor is more difficult, and is often done with empirical data. We have created first-order models of these effects by using wireload models in the circuits that we synthesize, but we expect others, with more data to draw on, to improve our models in the future. While the detailed results will change as the underlying models improve, we believe that the general trends and conclusions in our study will still hold true.

5.1.1 Validation Methodology

To validate our results, we take a two-pronged approach. We first consider the validation of the individual models that we have created, and then we also compare our overall results to real industrial processors as a more global validation. At the modeling level, we need to validate each of the performance and energy models we create. This has already been discussed for our performance models; we achieve median errors of 6% or less when compared to architectural performance simulators. On the energy side, the process is analogous: we ensure that the energy-delay trade-off characterizations we create fit the energy-delay points from the synthesis tools. Here, again, we achieve good results, with most energy models typically being within 5% error.

While performing validation of the individual models is useful, this approach is unable to verify the extent to which we have modeled all the important units. For this reason, we also compare our designs to real processors. For our comparisons, we consider two ARM Cortex A-Series processors: a low-power Cortex-A5 [1]—a single-issue in-order processor—and a performance-targeted Cortex-A9 [2]—a multi-issue, superscalar out-of-order processor with speculative execution. These two processors are good candidates for performing our comparisons because they map very well to the range of processors we examine. Normalizing the available performance and efficiency data [1, 2] to our technology and voltage, the Cortex-A5 consumes about 175 pJ per instruction while the more aggressive Cortex-A9 consumes about 415 pJ per instruction. These values are in the same range as the 80 to 340 pJ per instruction that we see in our optimized designs in the next section; they are a bit higher, but this is not surprising since there are many smaller components within a commercial design that we have not modeled. On the performance side, the Cortex-A9 achieves a instruction processing rate of approximately 1800 MIPS (normalized using delay scaling factor half-way between 1 and $\frac{1}{\alpha}$), which is also close to the high-end performance of around 1900 MIPS the optimization framework produces. This comparison reassures us that our models are reasonable and cover the dominant performance and energy issues.

5.2 Base Results

Having described the experimental set up, we apply the optimization framework to each of our high-level architectures. The resulting energy-performance trade-offs for these architectures are shown in Figure 5.1. These Pareto-optimal curves show the entire range of trade-offs. As performance is pushed, each architecture uses more aggressive structures and circuits, causing the energy consumed per instruction to

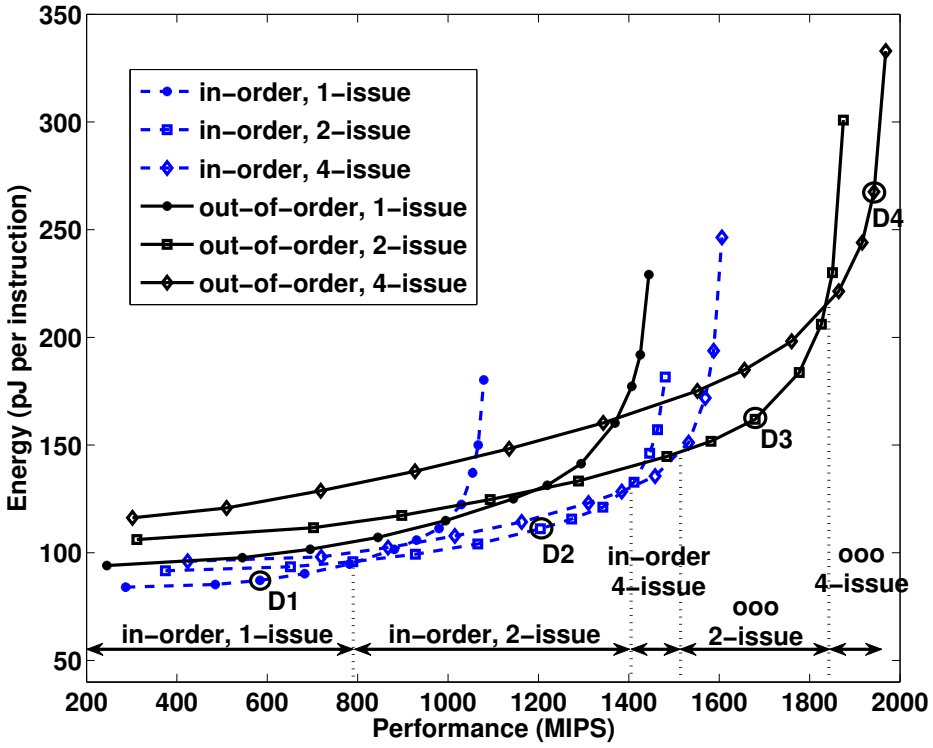


Figure 5.1: Overall energy-performance trade-offs of our six macro-architectures for a 90 nm CMOS technology, produced by jointly optimizing microarchitectural and circuit parameters. As the performance is pushed, the optimal choice of macro-architecture changes to progressively more aggressive machines. Design details for the circled design points are shown in Table 5.3.

increase. Given an energy budget or performance target, a designer can use these curves to identify the most appropriate design.

We remind ourselves that these Pareto-optimal trade-off curves between performance and energy are more general than commonly used metrics like ED or ED^2. As was mentioned in Chapter 2, ED^n metrics essentially set an exchange ratio between energy and performance, with higher powers of n favoring more performance; in this sense, they can be somewhat arbitrary. Optimizing for ED^n with a particular value of n would correspond to a particular point on the Pareto-optimal curve. Since one generally wants to design for a specific performance target or energy budget, neither of these points is necessarily the desired answer. Representing the results as a trade-off curve between energy per operation and performance provides a more complete picture of the design space to designers.

The overall trade-off space spans approximately 6.5x in performance—from about 300 MIPS to 1950 MIPS—and 4.25x in energy—from about 80 pJ/op to 340 pJ/op. The various architectures contribute different segments to the overall energy-efficient frontier. As one would expect, the single-issue in-order architecture is appropriate for very low energy design points, while the quad-issue out-of-order is only appropriate at very high performance points. In between these two extremes, we find that the dual-issue in-order and out-of-order processors are efficient for large parts of the design space. Thus, when starting from a basic single-issue in-order design, the order in which high-level architectural features should be considered is, first, superscalar issue, and then—if more performance is still needed—out-of-order processing. From the perspective of the marginal energy cost per unit performance, the move to a superscalar design is cheaper than investing in out-of-order processing. The quad-issue in-order design is only efficient for a small performance range, not being as energy-efficient as the dual-issue in-order design at lower energy points, and being outmatched at high performance points by the dual-issue out-of-order design. Finally, the single-issue

Table 5.3: Design Configuration Details For Selected Design Points.

	D1	**D2**	**D3**	**D4**
In-order vs out-of-order	in-order	in-order	out-of-order	out-of-order
Issue width	1-issue	2-issue	2-issue	4-issue
Cycle time (FO4)	27.5	16.9	17.2	16.3
Branch pred size (entries)	264	600	1024	870
BTB size (entries)	64	90	554	1024
I-cache size (KB)	21	32	32	32
D-cache size (KB)	8	11	14	42
Fetch latency	1.0	1.6	2.2	2.1
Decode/Rename latency	1.0	1.7	2.4	3.0
Retire latency	N/A	N/A	2.0	2.2
Integer ALU latency	1.0	1.0	1.0	1.0
FP ALU latency	3.0	4.0	3.9	4.1
L1 D-cache latency	1.0	1.1	1.1	1.1
ROB size	N/A	N/A	22	32
IW size	N/A	N/A	11	9
LSQ size	N/A	N/A	16	16

out-of-order design is never efficient and does not contribute to the overall efficient frontier. This architecture represents a design that is out of balance. Being able to issue only a single instruction becomes a bottleneck to the out-of-order processor, resulting in wasted effort.

We can also examine how the various underlying parameters are changing throughout the design space. In Table 5.3, we examine these parameters for design points D1 through D4 as marked on Figure 5.1. Not surprisingly, as we push for more performance, the frequency and structure sizes generally increase, while latencies generally decrease. Some of the latencies show fractional values which would need to be snapped to discrete values, although techniques such as time borrowing and register retiming can also be used to work with the results. We highlight a few points from these results that demonstrate the advantages and capabilities of using a systematic optimization framework.

First, both the I-cache and D-cache tend to stay away from small sizes, even when targeting lower-performance points. Across the design points, the D-cache reaches a minimum of 8KB even though a 4KB cache is available, and the I-cache never goes below 20KB. This behavior is the result of two counteracting forces that "fight" in the optimization of the cache size. Although smaller caches result in less expensive individual accesses, a larger cache potentially saves energy by reducing the number of misses that incur a more expensive access to higher level caches. Making an L1 cache big would reduce L2 cache access energy, but would be wasteful because of the high access cost to the L1 cache itself; conversely, making an L1 cache small would reduce the L1 cache access energy, but would be wasteful because of an increased number of expensive accesses to the L2 cache. Thus, for lower power design points, the optimizer determines that the marginal savings it can achieve by reducing misses outweighs the access cost of the larger caches, ultimately finding the right balance and settling on the chosen values. In these results, the I-cache tends to have larger sizes than the D-cache; the I-cache has higher hit rates which means that the marginal cost of increasing its size (per unit performance offered) is lower. Generally, these results show the importance of caches in energy-efficient designs as a way to both save energy and increase performance.

Secondly, we note that the instruction window (IW) is relatively small compared to the maximum available IW of size 32. In this case, we once more see an optimization "fight". While a larger IW improves the architectural performance and increases CPI, a larger IW also increases the complexity (and delay) of the instruction dispatch logic. Since the dispatch logic in a traditional out-of-order machine must execute every cycle[1]—the previously dispatched instruction(s) need to be removed from the

[1]While it is possible to create more complex dispatch schemes that operate over multiple cycles, we do not consider those in this example. It should be noted that multi-cycle dispatch logic schemes bring with them their own set of trade-offs, as they would generally require either segmentation of the instructions into dispatch banks, which may limit ILP extraction, or speculation with correction logic, which is also expensive. The fact that we have not considered these machines is simply because

instruction window before selecting the instructions to be dispatched for the given cycle—the delay of the dispatch circuitry can adversely affect the clock frequency. The optimizer realizes this trade-off and finds the right balance between architectural performance through a higher CPI and pipeline performance through a higher cycle time. Moving from design point D3 to D4, the instruction window size backs off to accommodate frequency scaling. We see a similar effect in the branch predictor size, another structure that needs to execute once per cycle.

This example demonstrates one of the major advantages of using a circuit-aware evaluation of architectural trade-offs: the consideration of circuit delays prevents the designer from creating unrealistic architectural evaluations that could be misleading (even if unintentional). As architectural performance simulators are not generally linked to the actual delay of different circuits, such simulators cannot identify situations where the delay of a circuit would affect the frequency of a pipeline. Thus, it is very easy, for example, for an architectural simulator to model systems with very large instruction windows with hundreds of entries, ignoring the fact that these designs are not realistically implementable.

The need for the consideration of circuit-level issues is not specific to the instruction window, but affects all circuits and structures in the system. When adding any new architectural feature, the designer needs to account for the additional logic in the design. The clock frequency may need to be reduced to allow that feature to execute within the given pipeline stage (this may also require that faster circuits be used, resulting in an additional energy cost), or a new pipe stage may need to be created, which potentially has an adverse effect on the system's CPI whenever data or control dependencies are present. Nevertheless, the effect is most pronounced where critical dependence loops are present—such as the dispatch logic, next-PC

it is not our purpose to explore every possible design, but rather to demonstrate the abilities of the framework and perform a high level study of energy-efficient machines. The optimization framework would have no problem modeling these kinds of systems if the system simulators for them exist.

logic or data forwarding logic—because these circuits must complete once a cycle and otherwise adversely affect the clock frequency.

Returning to the results in Table 5.3, we finally note that the optimizer generally ensures that the delays of units such as the integer ALU and the D-cache fit into one clock cycle. This is because these units are critical to resolving data dependencies. Thus, it is usually worth the energy cost to ensure these units fit into one clock cycle. Of course, this is not a surprising fact, and is confirmed by current design practices. It is important to note, however, that the delays of these units are changing with the cycle time, so it is not the case that the same implementations are being used throughout the design space. Machines with more aggressive cycle times use faster, higher energy versions of these circuits, whereas the lower power design points use lower-energy circuit implementations.

5.3 Circuit Trade-offs

The circuit-aware approach that we use integrates delay and energy information into the optimization, exposing different energy-delay design points to the design space exploration. This extends most architectural design space tools and studies which typically use fixed energy costs for each circuit (e.g. Wattch [13], others [71, 41]). The additional fidelity allows us to trade-off the energy and delay within a circuit to find the optimal circuit operating point. For example, the optimizer can choose to slow down a circuit to save energy when it does not need to run as fast, or it can allocate more energy to a circuit to speed it up if it finds that circuit to be critical to the system performance. This is an important consideration in the optimization space, especially when we consider that different circuits will be optimal at different performance targets.

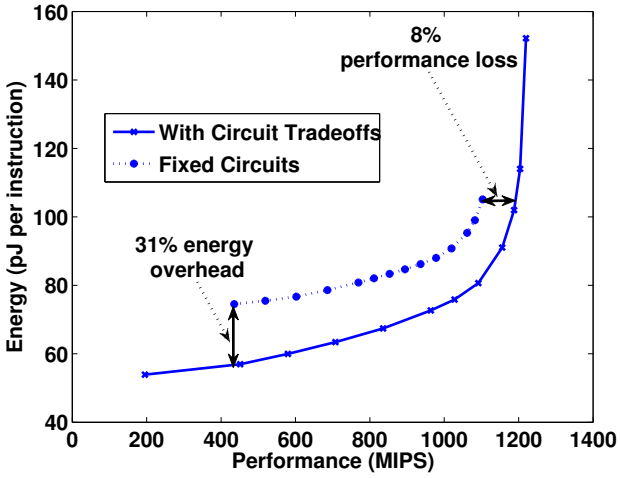


Figure 5.2: Co-optimization with circuit trade-offs versus optimization with fixed circuit data. In this experiment, fixed circuits operate at 15% back-off from their minimum delay. By restricting the circuit operation to these fixed points, we see a 31% energy overhead at the low energy design points, and an 8% performance loss at high-performance points. A single circuit implementation is unable to meet the demands of the entire design space, and each circuit needs to be tuned different for efficient operation.

To evaluate the advantages of including circuit trade-offs in the design space analysis and optimization, we compare our approach to a fixed energy cost approach. We perform this study by restricting our circuit libraries to single energy-delay points (instead of curves) for each circuit. This causes the optimizer to consider only a single implementation per circuit, thus mimicking—from an energy modeling perspective—the functionality of current architectural tools. Using these fixed circuit libraries, we then run a complete architectural optimization, comparing the optimal energy efficiency achieved using the the fixed circuits to energy efficiency of a joint circuit-architecture optimization.

In performing this study, we first have to select design points for our fixed circuit libraries. To maintain a fair comparison, we desire points that represent a reasonable balance between the high energy costs of running circuits at their maximum speeds and running circuits too slow. To achieve this balance, we set each of the circuits in our libraries to be operating at 15% back-off from their maximum speed. We have found that this approach causes most circuits to be operating approximately at the "knee" of their energy-delay trade-off curves.

In Figure 5.2, we compare a 15% back-off fixed-circuit architectural optimization to a circuit-aware optimization that includes the full circuit design space in the optimization. What this figure shows is a significant amount of inefficiency over the entire design space caused by the use of fixed circuit libraries. At the low-energy points—where the lowest energy circuits would normally be preferred—the architecture is restricted to more costly circuits, resulting in a 31% energy overhead. On the other hand, at high-performance points—where some circuits should be operating at their peak performance—the architecture is restricted to slower circuits, resulting in an 8% performance loss. Moreover, the inefficiency does not occur only at the extreme data points, but exists over the entire range of the trade-off space; at no point does the fixed-circuit optimization achieve the same energy efficiency of the circuit-aware

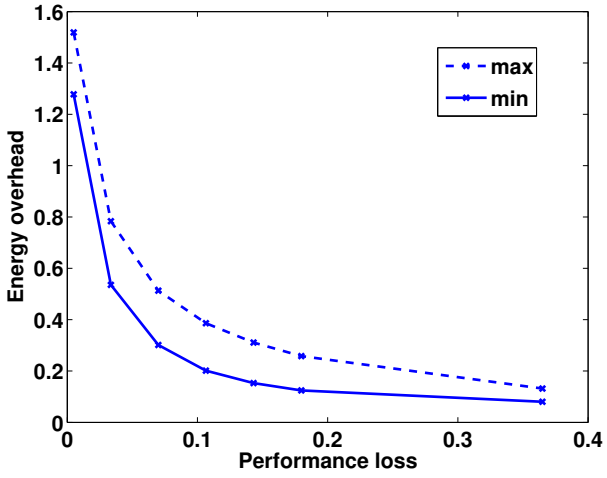


Figure 5.3: Plot showing inefficiency caused by using fixed circuit data. Fixed circuit libraries both reduce the achievable peak performance and require more energy than a jointly optimized design. Each fixed library is placed along the x-axis by the performance loss of its fastest solution, so a circuit library at 1% off the minimum delay is located at near 0 performance loss and circuits at 50% off from the minimum delay cause the overall system to slow down 37%. For each library we then find the energy overheads compared to the jointly optimized design. The dotted line is the max overhead throughout the architectural design space, while the solid line is the min overhead. No fixed circuits provide good performance with low energy overhead.

optimization.

To ensure that these results are not an outcome of the particular fixed circuit libraries that we have used, we extend this study to other fixed libraries. We thus sweep the fixed circuit points to be at 1, 5, 10, 15, 20 25 and 50% of the minimum delay, and repeat the previous study. Figure 5.3 shows the resulting inefficiency of using each of these different "fixed" circuit points, plotting the maximum and minimum energy efficiency losses of each these circuit libraries against the performance loss experienced at the highest performance point (for our 15% circuit libraries, for example, the high-end performance loss was 8%).

In this figure, we see that running all the circuits at near maximum speeds (left-most point, circuits at 1% of their min delay) allows the machine to run at near-peak performance, but comes with significant energy overheads of 130% or more. These large overheads are the result of uniformly running all circuits very fast, since only performance critical units should run at their maximum speed; non-critical units should be slowed down to save energy. On the other hand, if slower circuits (right-most points) are used in an effort to reduce energy, energy overheads reduce to about 15%, but maximum performance is sacrificed.

The problem, of course, is that in the fixed library approach, the circuit designs are restricted to a single implementation, and the architecture has no choice but to use the provided circuit designs. Unfortunately, no single operating point can possibly cover the needs of architecture across the entire range of the design space. Circuits which are appropriate for high-performance targets are different than circuits for low-energy objectives, and if one wants to map out the energy efficient frontier of the entire design space, it becomes necessary to include the entire space of circuit trade-offs in the analysis.

Even if one is not trying to explore the entire energy-efficient frontier, but is rather targeting a single design objective, the inclusion of circuit trade-offs in the optimization is still important. In a system with many sub-units, each of the underlying circuits needs to be tuned differently according to how important they are to the system as a whole; certain circuits that are critical to the system performance should be sped up, while less critical circuits should be slowed down to save energy. Of course, to perform this analysis, a circuit-aware architectural optimizer is required that can evaluate the system-level marginal costs of making circuit changes. The optimizer can then tune the circuit design and select the most appropriate circuit. Without making the system optimization aware of the the circuit design space, the appropriate circuits cannot be determined, ultimately resulting in some degree of energy inefficiency in

the system.

5.4 Dynamic Circuits

The use of a circuit-aware optimization framework also allows us to perform other studies that cross the architecture-circuit boundary that were difficult to perform before. One such study, that we present here, explores the effect of using different circuit styles.

Our base circuit libraries use static CMOS energy-delay trade-offs which include the space of gate sizings and logic synthesis mappings. There are various other circuit styles, however, each of which have their own set of characteristics and trade-offs; here, we examine the trade-offs of using dynamic circuits. Dynamic circuits were once commonly used in high-performance processors and can be significantly faster that static CMOS circuits. However, because of their clock power and high activity factors, dynamic circuits also come at a significant energy cost.

Our first task is to create circuit libraries with energy-delay trade-off curves for dynamic circuits. To characterize the performance gains and energy overheads of using dynamic circuits, we compare a dynamic dual-rail domino adder to a static implementation using a circuit optimization tool [53]. These results indicate that the dual-rail domino circuit achieves 0.67x the delay of the static circuit at 4x the energy. Since we do not have a complete dynamic circuit library, we use this adder scaling data as a proxy for all circuits (except memories[2]) to generate dynamic circuit trade-offs.

We examine two dynamic designs using these new libraries. In the first, we use dynamic circuits for certain performance-critical components, speeding up the integer ALU and the out-of-order issue logic. While the ALU does not strictly stand in the

[2]Most memories already use some dynamic circuits internally, so we do not change their performance in this experiment.

way of the cycle time—the ALU is pipelinable—the performance loss of pipelining the ALU due to data dependency stalls means it is usually not an energy-efficient design choice. The optimizer generally prefers a 1-cycle ALU with a longer cycle time over a higher frequency design with a multi-cycle ALU. In the case of the issue logic, it cannot be pipelined in our design, and actually limits the use of shorter cycles times.

Dynamic logic has a second potential advantage. Since every dynamic gate already has a clock, it is possible to build an entire system without including any explicit flops or latches. Furthermore, the overall design can be constructed to be tolerant of skew on the clock lines [30, 58]. This type of design essentially removes all clocking overheads that are found in conventional designs. For this method to work, all logic must be a monotonic function of its input (domino logic), so this generally requires one to create dual-rail gates, which compute both true and complement outputs from true and complement inputs. Thus, for our second design, we also explore the performance trade-offs of a complete dual-rail design.[3]

Figure 5.4 shows the results of using these circuit styles on our dual-issue out-of-order architecture. Also shown is the original static version. As expected, both dynamic designs push performance to new limits, but come with some added energy overheads. The partially dynamic design provides more performance because it can now achieve higher cycle times. The faster issue logic now also allows for larger instruction windows of up to 20 entries; this in contrast to the small 8 entry instruction windows we saw in the static design. These performance benefits, of course, come at a somewhat high energy cost: the transition from the static design to the partially dynamic design comes at a marginal cost of 2.3% in energy for 1% in performance. The fully dual-rail, skew-tolerant design offers an even larger performance gain; it virtually eliminates clocking overheads. However, it comes with an even larger energy cost

[3]While the memories would need to be modified to work in this system, the changes would be small and would not cause major changes in memory power or delay.

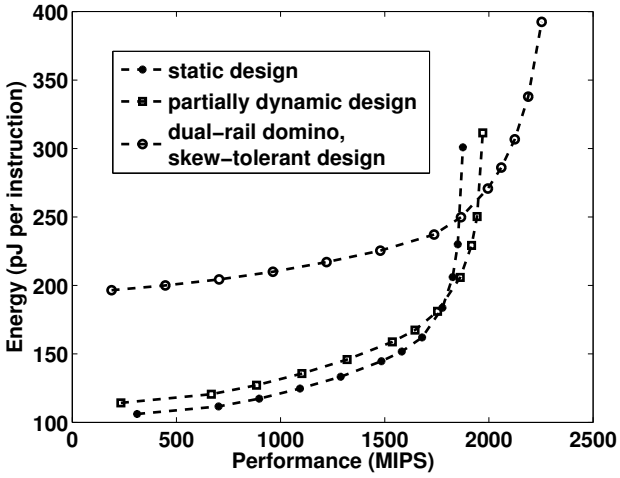


Figure 5.4: Trade-offs of dynamic circuits. A partially dynamic design (with a dynamic ALU and issue logic) increases maximum performance by enabling shorter cycle times. A fully dynamic, skew-tolerant design offers even greater performance by removing clocking overheads. Both designs, however, come at significant energy costs which place them on a steep part of the trade-off curve.

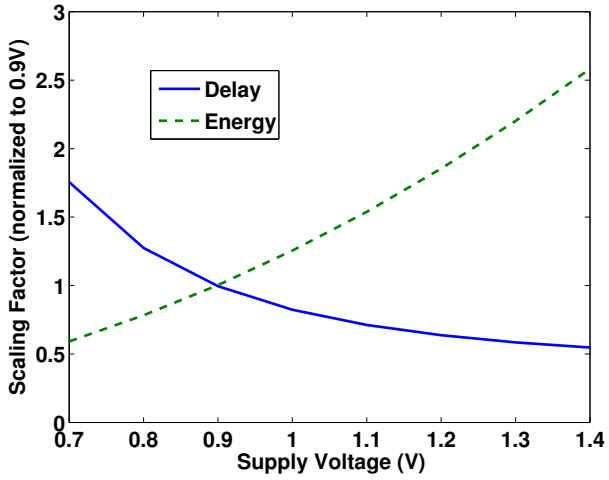
since the entire machine must be implemented in dynamic logic. This design option represents a more expensive choice at about 2.7% in energy for 1% in performance.

While neither of these options are cheap, a designer may be willing to pay the cost if the added performance is truly needed. As we will see in the next section, however, voltage scaling can often offer better marginal costs and should be considered first.

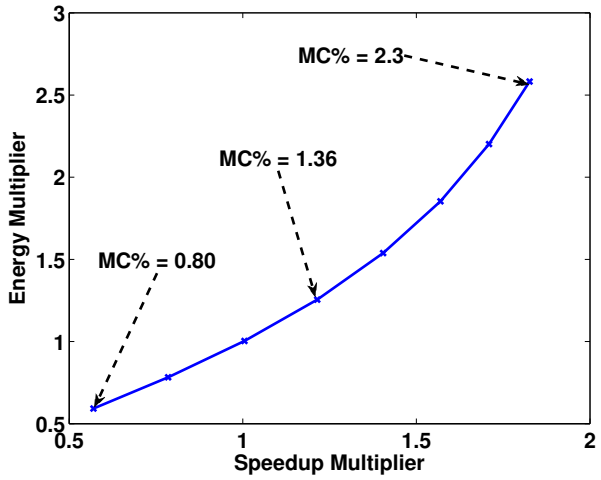
5.5 Voltage Optimization and Marginal Costs

It is well-known that an important consideration in energy-efficient design is the choice of operating voltage. One needs only to scale voltage by a few tenths of a volt to see significant increases in both performance and energy consumption. Thus, it becomes important to optimize the design along with the supply voltage.

Figure 5.5a shows the energy and delay scaling characteristics of circuits as a



(a)



(b)

Figure 5.5: Effect of voltage on delay and energy. (a) shows the delay and energy scaling as a function of supply voltage (normalized to 0.9V). (b) shows the corresponding energy-performance trade-off curve. Percentage marginal costs (MC%) is the percentage energy cost required to increase performance by 1%. For a wide range of energy and performance, marginal costs do not change much, making voltage a powerful knob for energy-efficiency.

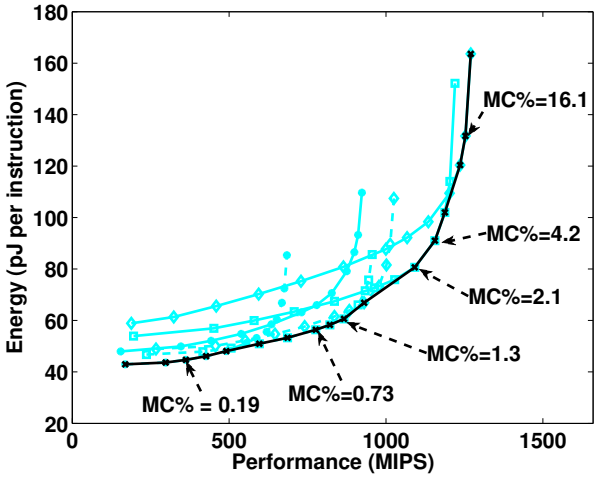


Figure 5.6: Marginal costs of the joint architecture and circuit design space. In light are the energy-performance curves of our six architectures. In dark is the overall, composite energy-performance frontier. Data is normalized to 0.9V to allow for comparison to the voltage marginal costs. Marginal costs in the architecture/circuit space vary considerably more than voltage marginal costs. For most practical design objectives the optimal architecture should be selected from the narrow band of designs with a marginal cost of 0.80%-2.3% to match voltage marginal costs.

function of voltage as obtained through SPICE simulations. The energy curve follows an expected Vdd^2 profile; the delay shows an inverse relationship proportional to $\frac{1}{Vdd^{3.325}} + 1$ (empirical fit, normalized to 0.9V). Composing these two relationships, we get the energy-delay scaling trade-offs of the supply voltage parameter in Figure 5.5b.

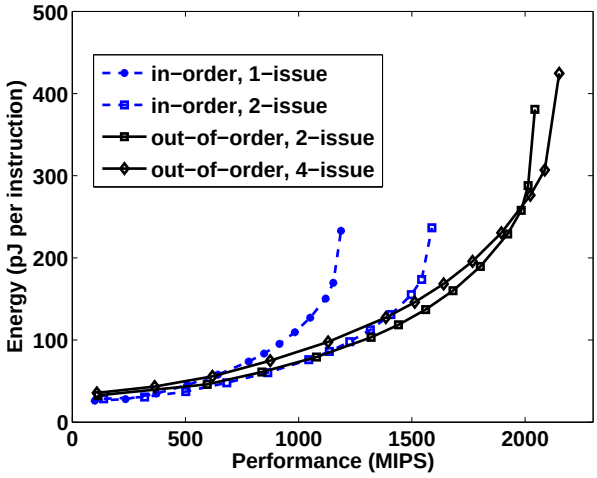
This data shows that, by itself, voltage tuning from 0.7V to 1.4V provides a range of about 3x in performance and 4x in energy. More importantly, the profile of the energy-performance curve is relatively shallow throughout this entire range. This means that the marginal cost of increasing performance through voltage scaling does not change much as we continue to increase the voltage parameter. At low voltages, the marginal cost is at about 0.80% in energy for 1% in performance; at the high end, this marginal cost reaches 2.3% in energy for 1% in performance.

We can contrast this marginal cost profile against the marginal cost profile of achieving performance through circuits and architecture, shown in Figure 5.6. This space shows a much larger range of marginal costs. At the low performance points, the marginal costs are very cheap, while at the high performance points, the marginal costs are very expensive.

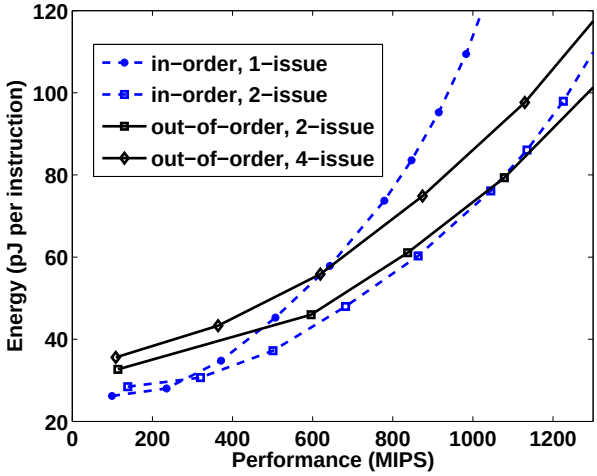
We recall that to optimize a design, the marginal costs of all parameters should be equal. If this were not the case, then an arbitrage opportunity would exist, and the more expensive parameters could be exchanged for cheaper parameters: selling the expensive parameter would cause some performance loss, but this performance could be recovered at a lower cost through the cheaper parameter. Comparing the marginal costs of voltage versus architectural parameters, this suggests that, unless we are trying to achieve the very extremes of performance or low power (to the point that the voltage knob is constrained by its maximum or minimum voltage, respectively), the optimal set of designs should lie roughly in the range of marginal costs from 0.80% to 2.3% in order to match the marginal costs of voltage scaling.[4] This results in a narrow band of architectural and circuit designs being optimal when the voltage scaling parameter is available.

Figure 5.7 shows the optimization results when the supply voltage parameter is included in the design space. Confirming the marginal cost analysis, we see that a

[4]Strictly speaking, this is an approximate statement as it requires that all components in the system scale uniformly with voltage. While this is mostly the case, in reality, components like L2 caches use low-swing bitlines that change their scaling behavior. This means that one cannot simply look at the marginal cost profile of the whole system when considering voltage scaling; one must, rather, separate out the L2 cache component and treat it differently. This effect applies to our case study as well, since we assume L2 caches and main memory have a fixed access cost and physical latency (i.e. we specifically exclude these components from scaling with voltage because they are outside the design space). Nevertheless, the rule of matching marginal costs still applies to a high degree and is useful as a rule of thumb. It is particularly so in our study because the activity factors of the L2 cache and main memory are not high, and the energy spent in these components is not a dominating factor. Regardless, in the results we present next in Figure 5.7, these effects are all modeled correctly, as the optimizer is aware of which components scale with voltage, and it therefore evaluates the correct set of energy-performance trade-offs to find the energy-efficient frontier.



(a)



(b)

Figure 5.7: (a) Energy-performance trade-offs for the processor design space with voltage scaling. The dual-issue out-of-order design now dominates an even larger part of the design space; the dual-issue in-order design is optimal at low energy points. The quad-issue in-order design and the single-issue out-of-order design are not shown to simplify the plot; they are never efficient. (b) Same results zoomed in on low energy points.

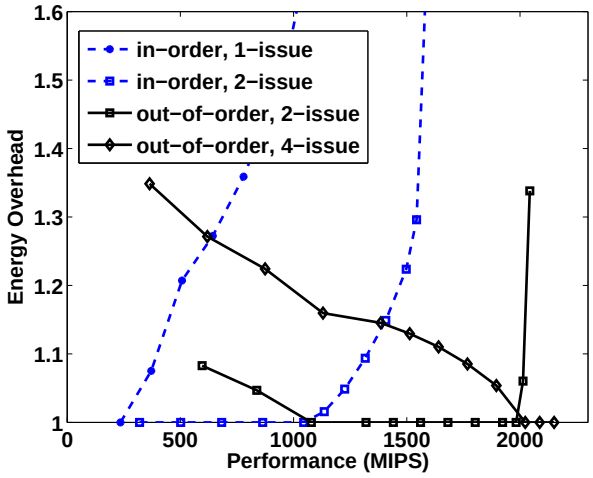


Figure 5.8: Results of Figure 5.7 plotted as the energy overhead over the lowest energy available. A value of 1 means that the design is the most energy-efficient available; values greater than 1 represent the inefficiency.

smaller set of architectures cover a larger part of the energy-efficient frontier. The dual-issue out-of-order processor is energy-efficient for a large part of the design space. At low performance targets, the dual-issue in-order processor takes over, although the dual-issue out-of-order processor is still not overly inefficient. Only at the very extremes, when the voltage knob becomes capped, do the single-issue in-order and quad-issue out-of-order designs play a role, and these represent designs with very low and very high marginal costs respectively. Figure 5.8 provides an alternative view of the same data, plotting the energy overhead versus the overall energy-efficient frontier.

This result suggests that a small number of properly tuned designs can cover most of the overall energy-performance frontier at near optimal efficiencies simply by voltage and frequency scaling. We pick one dual-issue in-order processor and one dual-issue out-of-order processor with fixed microarchitectural and circuit parameters, and evaluate these fixed designs under voltage and frequency scaling. Figure 5.9 shows the

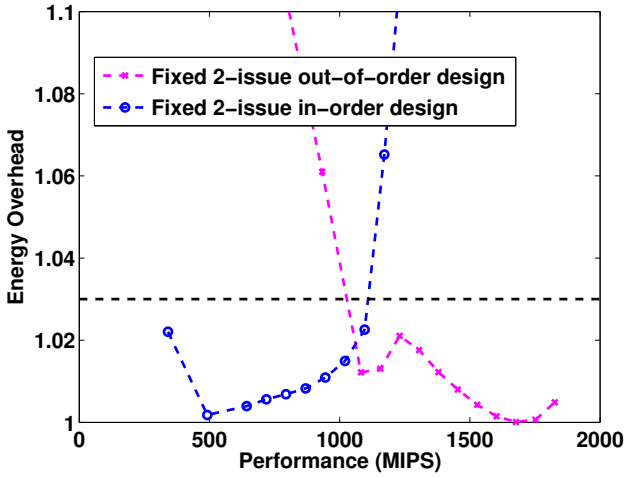


Figure 5.9: By using two carefully selected designs—a fixed dual-issue in-order design and a fixed dual-issue out-of-order design—voltage scaling can be used to cover a large performance range within 3% of the optimal energy-efficiency.

energy overheads of scaling these fixed designs as compared to the fully optimized designs which also tune the architecture and circuits. Because design parameters are fixed, we see some inefficiency; the lines deviate from the normalized optimal value of 1. This result is expected because the marginal costs of all parameters in the system are no longer equal. Yet, we see the resulting inefficiency is small—under 3%. Of course, this result requires that we start with the right two designs in the architecture/circuit space sweet spot. Thus, with two carefully selected designs and voltage scaling, we can operate at near optimal energy-efficiencies over a broad performance range.

5.6 Discussion

In design for efficiency, a designer needs to consider the cost-performance trade-offs of all available design options, using this information to make the best decisions. This

process needs to be done in a disciplined, systematic way, with the designer always adhering to the principles of minimizing marginal costs and ensuring that marginal costs of all design options match.

The results in this chapter have shown the potential pitfalls of ignoring these principles. One needs to be careful to neither over- nor under-design any aspect of the system; all components need to be in balance, with all design decisions having matching marginal costs. Without applying this principle, inefficiencies arise, with a cheaper sources of performance being left untapped. The importance of this principle was particularly clear when considering voltage scaling versus architectural/circuit design decisions as a means of achieving performance. The results of the last section showed that it makes little sense to over-aggressively design the system when the marginal costs of voltage—an equally powerful design parameter—changes relatively slowly over its range. Since the marginal costs of voltage vary between 0.8 to 2.3% energy for 1% in performance (over the range of 0.7-1.4V), this implies that a potential rule of thumb should be that the marginal cost of any design decision should fall within this range to be acceptable, with lower or higher marginal costs only being considered if the voltage parameter becomes constrained, and the designer has no other feasible options.

As a result of the steady marginal cost profile of voltage versus the rapidly changing marginal costs of architectural and circuit design, we found that the optimal architecture/circuit design was limited to a small sweet spot; most other designs fell outside the 0.8 to 2.3% marginal cost range. Some design features landed on a very cheap part of the trade-off curve, meaning they should virtually always be used, while many other design options came at very expensive rates, meaning they should be avoided. In between these two extremes, the set of design knobs did not vary much. This suggested that with a few fixed designs from within this sweet spot, voltage scaling could be a very effective means of acheiving different design objectives, a

hypothesis that our results seemed to confirm.

It should be stressed that this result should not be mis-interpreted as indicating that the architecture and circuit design are irrelevant. In fact, the conclusion is the opposite. There are many ways to build an inefficient design, and the designer has to make a concerted effort to find a design within this sweet spot. If the initial system is inefficient or lies outside this sweet spot, voltage scaling cannot make up for the initial inefficiency. It becomes critical to tune the design to include the right set of features with the right marginal costs.

At this point, it is important to consider how these results would change with higher leakage future technologies. There are at least a couple ways that the results would be affected by leakage. First, as leakage is highly correlated to area, one would expect that structures with larger area would be penalized somewhat during the optimization; this is especially true if the structures are less frequently used (i.e. have a lower activity factor), because then the amount of leakage energy per instruction rises, requiring a higher increase in performance per instruction to make the structure attractive from an energy-performance perspective. Second, in cases where the chip can power down to a low-power idle mode, optimizing with a high leakage technology can actually favor more aggressive, higher performance features. This is because leakage is a rate of energy consumption that gets multiplied by the execution time; with leakage considerations, the whole system would like to "run" to the finish line of a task, and then power down, reducing the leakage contribution which would otherwise be incurred over a longer time period. We have confirmed these results with simple experiments for our optimizations. Including leakage causes the trade-off curves rise due to the additional leakage energy, but in a skewed way—the design points at the left-hand side (lower performance), rise more as a percentage of their original power. This has two consequences: first, the low-energy tails of the trade-off curves get cut off—at some point it does not make sense to run any slower

because the small savings in dynamic energy per operation are offset by the increase in leakage energy—and second, because the trade-off curves rise in a skewed fashion, the more aggressive architectures become somewhat more important from an energy-efficiency perspective and end up covering a larger area of the energy-efficient frontier. Thus, we saw that the dual-issue out-of-order design was optimal over an even larger percentage of performance targets.

While this chapter has presented the results of optimally tuned processors, the real advantage of this type of design framework is the insight it can give a designer. For example, we initially had each of our high level architectures fetch instructions according to the width of the machine (e.g. one word for the single-issue machines, etc.). This led to wider machines being more energy efficient than single issue machines even at low performance. Clearly, because of high instruction locality, it makes sense to have all architectures fetch multiple instructions at a time to amortize the cost of going to cache. As with any tool, the task of examining and interpreting the results to find new design directions lies with the user.

This framework is thus also an important tool for exploring new architectures and designs. Without having a means of tuning the design knobs within a new architecture, it becomes difficult to compare it versus previous designs. Comparing single design instances of two architectures can be misleading, because it is unclear how their internal design trade-offs have been selected, and where within their energy-performance space they lie. Instead, one should be comparing energy-efficient frontiers as a proper comparison. It might be the case that one architecture is always more efficient than another—in which case the idea is clearly a good one—or the result might be that the frontiers cross at some point, similar to how the different architectures of Figure 5.1 are each optimal at different parts of the design space. In either case, to make a fair analysis, the energy-efficient trade-off curves need to be compared, and this requires a systematic optimization framework that can perform

this analysis.

In addition to exploring new architectures, the designer may desire to explore different optimization objectives. The results presented in this chapter have focused on performance-energy trade-offs, and not considered area (die cost) or what happens for threaded or data parallel applications. It is easy to include these effects using our framework. For example, in the case of multi-core designs for highly parallel workloads, we need to change the performance objective. The number of cores that we can fit on a die is critical to performance, and we must consider both the performance and area of the cores.

Under the assumption of infinite parallelism, using more cores to increase performance is always an energy-efficient choice; the performance scales linearly with the number of cores, while the energy per instruction remains constant.[5] Because performance can be achieved very cheaply through more cores, the design objective to optimize for in this case is performance per mm^2.

Figure 5.10 shows optimization results under this new design objective. In this case, the area overheads of implementing out-of-order processors outweigh their performance benefit, and so the dual-issue in-order design is always optimal. To account for workloads with more realistic amounts of parallelism, one just needs to change how the performance scales with the number of cores [33], which will just change the performance/area function that needs to be optimized.

We present this simple example only to show that it is possible to consider various different optimization objectives. In this case, the optimization metrics were changed http://padworld.myexp.de/index.php?filesto include a combination of area, energy and performance. In other cases, the designer may wish to optimize for other metrics, such as total fabrication plus operating cost (in dollars); given the right set of

[5]Overall power increases, but energy per operation is constant because each executed instruction requires the same amount of energy.

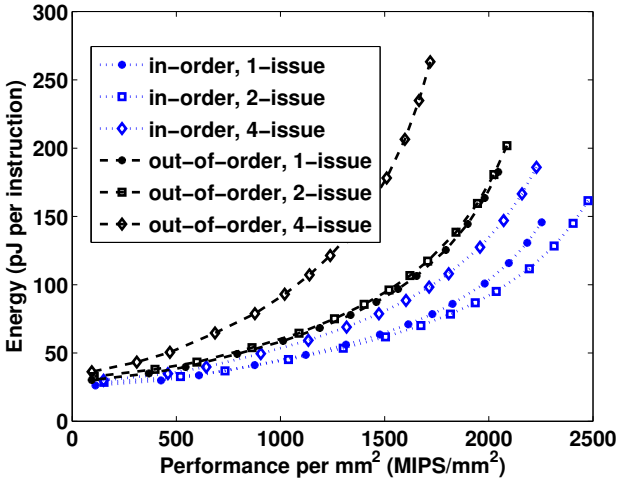


Figure 5.10: Energy per operation vs performance/mm^2 for a highly parallel workload. The area overheads of out-of-order processing make it a less attractive option.

data, such studies would also be possible. In analogous ways we can optimize whole systems that contain SIMD units, external high-power accelerators (i.e. GPUs) or other components that affect energy or performance.

Chapter 6

Conclusion

The optimization of digital electronic systems requires a cost-benefit analysis of many design parameters, a process that is made challenging because of the large hierarchical nature of the design spaces that need to be explored. Current modeling and design simulation tools have done a good job of guiding the design optimization process by providing performance and cost predictions by abstracting away lower levels, but they have typically relied on a modeling approach that communicates only a single design point between the levels of hierarchy. Thus, in current architectural evaluations, the architectural optimization relies on the characteristics of fixed circuit designs, and is unaware of the design space of possibilities at this lower level. The emergence of strict power constraints, however, has made the design of energy efficient systems critical, and this requires that designers examine energy-performance trade-offs more thoroughly.

In this dissertation, we extended current modeling and optimization methodologies to create a hierarchical optimization framework that could evaluate design space trade-offs in the joint architecture-circuit space. Through this framework, we were able to show two key advancements over current methodologies. First, we were able

to demonstrate that it is, in fact, possible to create a hierarchical modeling and optimization framework that communicates design space information between levels of the hierarchy without greatly increasing the complexity of the resulting optimization. In particular, we were able to expose the space of circuit design implementations to the architectural level by using circuit trade-off libraries. This extension of current modeling tools provided the optimizer with the flexibility to choose the correct circuit implementation based on the needs of the architecture and marginal costs, producing significantly more energy efficient designs. Second, we were able to show that it is possible to model a large part of the design space using posynomial models that enable the application of convex optimization to find the optimal design. This property allowed us to create a framework that could easily search large multi-dimensional design spaces with numerous architectural and circuit design parameters to find the optimal design quickly and reliably. Moreover, the basic sample-and-fit approach at the core of this framework proved to be very effective, and we showed one can easily characterize different aspects of complex processor systems—from circuit trade-offs to multi-dimensional architectural design spaces—with high accuracy using fitted posynomial forms. The generality of using a fitting-based approach makes the framework powerful, theoretically being applicable to a diversity of systems, provided that the appropriate simulators exist from which data samples can be extracted.

The creation of this framework enabled a disciplined analysis of energy-performance trade-offs in the processor design space centered around the systematic application of marginal costs: the well-known principle that design features with lower energy costs per unit performance should always be preferred over higher cost alternatives (and its related form, that in an optimal design, marginal costs of all parameters must match, arbitrage opportunities otherwise being available). Applying this methodology to the general-purpose processor space enabled a study of energy-efficient processors, with the framework identifying the design parameters with the best marginal costs and

ultimately yielding a Pareto-optimal set of designs. Through this study, the importance of always adhering to a marginal cost analysis became critically apparent when voltage was considered in addition to the architecture and circuit design. With voltage scaling's marginal cost profile being relatively steady—the cost of buying more performance through voltage scaling does not change much as one moves from low voltages to high voltages—and the marginal cost profile of architectural and circuit techniques changing much more rapidly, this suggested that only a small number of designs within a design "sweet spot" were optimal over a large part of the design space. In particular, since the marginal cost of voltage scaling between 0.7 to 1.4 V ranges from 0.8% to 2.3% in energy for 1% performance, a good rule of thumb one can use is that unless a proposed design feature attains a marginal cost of 2.3%, it is probably not worth implementing (voltage scaling being a more efficient alternative). As a result of these marginal cost profiles, our results showed that the dual-issue out-of-order design, properly tuned, was an efficient design over a large range of performance tragets when used in conjunction with voltage scaling; the dual-issue in-order design was suitable for lower energy points.

6.1 Future Work

This focus of this dissertation has been to show how to create an optimization framework for analyzing trade-offs in complex hierarchical systems, but it is important to clearly define the scope of this work, and acknowledge areas where work still needs to be done. The first point of consideration relates to the optimization framework itself. Although most of the design space that we examined was convex, we mostly considered design knobs that were tunable in nature. There is still, however, a question of how one can practically explore discrete design decisions in an effective manner. For

example, we explored the space of different high-level architectures by creating separate models—a decision that made sense in our circumstances given the significant changes between those architectures—and the optimization framework was instrumental in allowing us to compare these different architectures in a fair manner by tuning each design and producing Pareto-optimal curves we could compare. Nevertheless, if the number of discrete options increases, the space grows exponentially and this potentially presents a problem. In this case, it may become necessary to wrap some higher-level heuristic optimization around the framework to handle the discrete parameters, with the inner core using convex optimization to optimize internal parameters.

A related matter is how one can handle modeling issues that arise when one produces fits that may not be as accurate as desired. In these cases, it would be interesting to investigate the possibility of extending the framework to perform a multi-pass optimization: using a high-level optimization with crude models to find the local region of interest, then dynamically refining the optimization by refitting the models with a focus on the new area of interest. By using this kind of approach, one may be able to create an even more accurate and robust optimization framework that could consider even larger design spaces.

On a different front, while the work in this dissertation examined creating a closer coupling of the architectural and circuit design spaces, one can consider extending this approach to more levels of the hierarchy. Going deeper down the stack, one should be able to include trade-offs in the design of the underlying transistor; this design space could be encapsulated into the circuit libraries, ultimately rippling all the way to the architectural design space. At the other end of the spectrum, one could consider incorporating design trade-offs in the software stack, creating a joint hardware-software co-optimized system. Since the application characteristics can have a significant influence on the energy and performance of systems, including this

layer in the optimization could potentially have a large impact on system energy efficiency.

Finally, the study of energy efficient processors presented in this work covers only one particular type of system optimized for a specific environment, namely single-threaded applications. While general principles like the application of marginal costs will always hold true, it should be expected that the results would change under different circumstances. As such, it would be interesting to apply this optimization framework to different kinds of digital systems. There are many different types systems one could consider; graphics processors are one example. Throughput computing systems, in general, are a particularly interesting area to explore. Not only are throughput computing systems becoming more common, they also present new modeling challenges. They introduce an additional level of hierarchy—covering not only the circuits to the core design, but up to the multi-core system as well. Applying our hierarchical optimization framework to these systems, the top-level architecture would be the multi-core system, with the individual processing cores being like the circuit libraries that we used in this work; just as we built circuit trade-off libraries, one can imagine a library of processor core trade-offs being used in the optimization of the full multi-core system. Generating the system-level models could again be simulation-based, but would model the interactions between cores, including any communication and memory contention patterns. Thus, while the particular details of modeling such systems may need to be explored further, one should be able to use the current optimization framework to evaluate trade-offs in these important systems.

Bibliography

[1] ARM Ltd., "Cortex-A5 processor – ARM," Aug. 2010. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a5.php?tab=Performance

[2] ARM Ltd., "Cortex-A9 processor – ARM," Aug. 2010. [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php?tab=Performance

[3] Arvind, K. Asanovic, D. Chiou, J. C. Hoe, C. Kozyrakis, S.-L. Lu, M. Oskin, D. Patterson, J. Rabaey, and J. Wawrzynek, "RAMP: Research accelerator for multiple processors - a community vision for a shared experimental parallel HW/SW platform," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-05-1412, Sep 2005.

[4] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[5] O. Azizi, J. Collins, D. Patil, H. Wang, and M. Horowitz, "Processor performance modeling using symbolic simulation," in *ISPASS '08: Proceedings of the ISPASS 2008 - IEEE International Symposium on Performance Analysis of Systems and software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 127–138.

[6] A. Ben-Tal and A. S. Nemirovskiaei, *Lectures on modern convex optimization: analysis, algorithms, and engineering applications*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2001.

[7] P. Bose and T. M. Conte, "Performance analysis and its impact on design," *Computer*, vol. 31, no. 5, pp. 41–49, 1998.

[8] S. Boyd, S.-J. Kim, L. Vandenberghe, and A. Hassibi, "A tutorial on geometric programming," 2004.

[9] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[10] S. P. Boyd and S. J. Kim, "Geometric programming for circuit optimization," in *ISPD '05: Proceedings of the 2005 international symposium on Physical design*. New York, NY, USA: ACM, 2005, pp. 44–46.

[11] S. P. Boyd, S.-J. Kim, D. D. Patil, and M. A. Horowitz, "Digital circuit optimization via geometric programming," *Oper. Res.*, vol. 53, no. 6, pp. 899–932, 2005.

[12] D. Brooks, P. Bose, V. Srinivasan, M. K. Gschwind, P. G. Emma, and M. G. Rosenfield, "New methodology for early-stage, microarchitecture-level power-performance analysis of microprocessors," *IBM J. Res. Dev.*, vol. 47, no. 5-6, pp. 653–670, 2003.

[13] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 83–94, 2000.

[14] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-accelerated simulation technologies (FAST): Fast, full-system, cycle-accurate simulators," in *MICRO 40: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.

[15] A. R. Conn, I. M. Elfadel, J. W. W. Molzen, 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 '99: Proceedings of the 36th ACM/IEEE conference on Design automation*. New York, NY, USA: ACM, 1999, pp. 452–459.

[16] G. B. Dantzig, *Linear programming and extensions*. Princeton University Press, Princeton, N.J.,, 1963.

[17] L. Davis, *Genetic Algorithms and Simulated Annealing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987.

[18] M. del Mar Hershenson, S. S. Mohan, S. P. Boyd, and T. H. Lee, "Optimization of inductor circuits via geometric programming," in *DAC '99: Proceedings of the 36th annual ACM/IEEE Design Automation Conference*. New York, NY, USA: ACM, 1999, pp. 994–998.

[19] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *Solid-State Circuits, IEEE Journal of*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[20] C. Dubach, T. Jones, and M. O'Boyle, "Microarchitectural design space exploration using an architecture-centric approach," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 262–271.

[21] A. N. Eden and T. Mudge, "The YAGS branch prediction scheme," in *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1998, pp. 69–77.

[22] L. Eeckhout, J. Sampson, and B. Calder, "Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, 6-8 2005, pp. 2 – 12.

[23] P. G. Emma and E. S. Davidson, "Characterization of branch and data dependencies on programs for evaluating pipeline performance," *IEEE Trans. Comput.*, vol. 36, no. 7, pp. 859–875, 1987.

[24] P. G. Emma, J. W. Knight, J. H. Pomerence, T. R. Puzak, and R. N. Rechtschaffen, "Simulation and analysis of a pipeline processor," in *WSC '89: Proceedings of the 21st conference on Winter simulation*. New York, NY, USA: ACM, 1989, pp. 1047–1057.

[25] B. Fields, S. Rubin, and R. Bodík, "Focusing processor policies via critical-path prediction," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2001, pp. 74–85.

[26] B. A. Fields, R. Bodík, M. D. Hill, and C. J. Newburn, "Using interaction costs for microarchitectural bottleneck analysis," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 228.

[27] J. P. Fishburn and A. E. Dunlop, "TILOS: A posynomial programming approach to transistor sizing," in *IEEE Int. Conf. Computer-Aided Design*, 1985, pp. 326–328.

[28] D. J. Frank, R. H. Dennard, E. Nowak, P. M. Solomon, Y. Taur, and H.-S. P. Wong, "Device scaling limits of si mosfets and their application dependencies," *Proceedings of the IEEE*, vol. 89, no. 3, pp. 259 –288, mar. 2001.

[29] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program analysis," in *Journal of Instruction Level Parallelism*, 2005.

[30] D. Harris, *Skew-tolerant circuit design*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[31] A. Hartstein and T. R. Puzak, "The optimum pipeline depth considering both power and performance," *ACM Trans. Archit. Code Optim.*, vol. 1, no. 4, pp. 369–388, 2004.

[32] A. Hassibi, J. How, and S. Boyd, "Low-authority controller design via convex optimization," in *Decision and Control, 1998. Proceedings of the 37th IEEE Conference on*, vol. 1, 1998, pp. 140 –145 vol.1.

[33] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[34] J. C. Hoe, D. Burger, J. Emer, D. Chiou, R. Sendag, and J. Yi, "The future of architectural simulation," *IEEE Micro*, vol. 30, pp. 8–18, 2010.

[35] M. Horowitz, E. Alon, D. Patil, S. Naffziger, R. Kumar, and K. Bernstein, "Scaling, power, and the future of CMOS," in *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, Dec. 2005, pp. 7 pp.–15.

[36] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 195–206.

[37] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 161–170.

[38] S. Joshi and S. Boyd, "An efficient method for large-scale gate sizing," *Circuits and Systems I: Regular Papers, IEEE Transactions on*, vol. 55, no. 9, pp. 2760 –2773, oct. 2008.

[39] N. P. Jouppi, "The nonuniform distribution of instruction-level and machine parallelism and its effect on performance," *IEEE Trans. Comput.*, vol. 38, no. 12, pp. 1645–1658, 1989.

[40] T. S. Karkhanis and J. E. Smith, "A first-order superscalar processor model," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 338, 2004.

[41] T. S. Karkhanis and J. E. Smith, "Automated design of application specific superscalar processors: an analytical approach," in *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2007, pp. 402–411.

[42] E. Larson, S. Chatterjee, and T. Austin, "MASE: A novel infrastructure for detailed micro architectural modeling," in *Proceedings of the 2001 International Symposium on Performance Analysis of Systems and Software*, 2001, pp. 1–9.

[43] H. Lebret and S. Boyd, "Antenna array pattern synthesis via convex optimization," *Signal Processing, IEEE Transactions on*, vol. 45, no. 3, pp. 526 –532, mar 1997.

[44] B. C. Lee and D. M. Brooks, "Accurate and efficient regression modeling for microarchitectural performance and power prediction," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 185–194, 2006.

[45] B. C. Lee and D. M. Brooks, "Illustrative design space studies with microarchitectural regression models," *High-Performance Computer Architecture, International Symposium on*, vol. 0, pp. 340–351, 2007.

[46] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *In Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009.

[47] D. Markovic, B. Nikolic, and R. W. Brodersen, "Power and area efficient VLSI architectures for communication signal processing," in *Proceedings of the IEEE International Conference on Communications, Vol. 7*, June 2006, pp. 3323–3328.

[48] D. Marković, V. Stojanović, B. Nikolić, M. A. Horowitz, and R. W. Brodersen, "Methods for true energy-performance optimization," *IEEE Journal of Solid-State Circuits*, no. 8, Aug 2004.

[49] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0," in *Proceedings of the 40th Annual International Symposium on Microarchitecture*, December 2007.

[50] A. Mutapcic, S. Kim, K. Koh, and S. Boyd, "GGPLAB version 1.00 a Matlab toolbox for geometric programming," May 2006. [Online]. Available: http://www.stanford.edu/~boyd/ggplab/

[51] D. B. Noonburg and J. P. Shen, "Theoretical modeling of superscalar processor performance," in *MICRO 27: Proceedings of the 27th annual international symposium on Microarchitecture*. New York, NY, USA: ACM Press, 1994, pp. 52–62.

[52] T. Oguntebi, S. Hong, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "FARM: A prototyping environment for tightly-coupled, heterogeneous architectures," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 221–228, 2010.

[53] D. Patil, O. Azizi, M. Horowitz, R. Ho, and R. Ananthraman, "Robust energy-efficient adder topologies," in *ARITH '07: Proceedings of the 18th IEEE Symposium on Computer Arithmetic*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 16–28.

[54] D. Patil, S. J. Kim, and M. Horowitz, "Joint supply, threshold voltage and sizing optimization for design of robust digital circuits," Department of Electrical Engineering, Stanford University, Tech. Rep. [Online]. Available: JointSupply, ThresholdVoltageandSizingOptimizationforDesignofRobustDigitalCircuits

[55] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick performance models quickly: Closely-coupled partitioned simulation on FPGAs," *Performance Analysis of Systems and Software, IEEE International Symmposium on*, vol. 0, pp. 1–10, 2008.

[56] Z. J. Qi, M. Ziegler, S. V. Kosonocky, J. M. Rabaey, and M. R. Stan, "Multi-dimensional circuit and micro-architecture level optimization," in *ISQED '07:*

Proceedings of the 8th International Symposium on Quality Electronic Design. Washington, DC, USA: IEEE Computer Society, 2007, pp. 275–280.

[57] 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," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 12, no. 11, pp. 1621 –1634, nov 1993.

[58] J. Silberman, N. Aoki, D. Boerstler, J. L. Burns, S. Dhong, A. Essbaum, U. Ghoshal, D. Heidel, P. Hofstee, K. T. Lee, D. Meltzer, H. Ngo, K. Nowka, S. Posluszny, O. Takahashi, I. Vo, and B. Zoric, "A 1.0-GHz single-issue 64-bit PowerPC integer processor," *Solid-State Circuits, IEEE Journal of*, vol. 33, no. 11, pp. 1600–1608, Nov 1998.

[59] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in computer architecture evaluation," *Computer*, vol. 36, no. 8, pp. 30–36, 2003.

[60] R. Sredojević and V. Stojanović, "Optimization-based framework for simultaneous circuit-and-system design-space exploration: a high-speed link example," in *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 314–321.

[61] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. N. Strenski, and P. G. Emma, "Optimizing pipelines for power and performance," in *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 333–344.

[62] L. Vandenberghe, S. Boyd, and A. El Gamal, "Optimal wire and transistor sizing for circuits with non-tree topology," in *Computer-Aided Design, 1997. Digest of*

Technical Papers., 1997 IEEE/ACM International Conference on, 9-13 1997, pp. 252 –259.

[63] L. Vandenberghe, S. Boyd, and A. El Gamal, "Optimizing dominant time constant in RC circuits," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 17, no. 2, pp. 110 –125, feb 1998.

[64] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower," *SIGARCH Comput. Archit. News*, vol. 28, no. 2, pp. 95–106, 2000.

[65] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "SimFlex: Statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, pp. 18–31, 2006.

[66] B. R. Woodley, J. P. How, and R. L. Kosut, "Direct unfalsified controller design-solution via convex optimization," in *American Control Conference, 1999. Proceedings of the 1999*, vol. 5, 1999, pp. 3302 –3306 vol.5.

[67] S.-P. Wu, S. Boyd, and L. Vandenberghe, "FIR filter design via semidefinite programming and spectral factorization," in *Decision and Control, 1996., Proceedings of the 35th IEEE*, vol. 1, 11-13 1996, pp. 271 –276 vol.1.

[68] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, pp. 22–29, 2006.

[69] J. J. Yi and D. J. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on Computers*, vol. 55, pp. 268–280, 2006.

[70] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," *Performance Analysis of Systems and Software, IEEE International Symmposium on*, vol. 0, pp. 23–34, 2007.

[71] V. Zyuban, D. Brooks, V. Srinivasan, M. Gschwind, P. Bose, P. N. Strenski, and P. G. Emma, "Integrated analysis of power and performance for pipelined microprocessors," *Computers, IEEE Transactions on*, vol. 53, no. 8, pp. 1004–1016, Aug. 2004.

[72] V. Zyuban and P. Strenski, "Unified methodology for resolving power-performance tradeoffs at the microarchitectural and circuit levels," in *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*. New York, NY, USA: ACM, 2002, pp. 166–171.