Fast Design Space Exploration for Low-Power Configurable Processors

by

PETER HALLSCHMID

M.A.Sc., University of British Columbia, 2003
B.Eng., University of Victoria, 1999
B.Sc., University of Victoria, 1996

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS OF THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Electrical and Computer Engineering)

UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)

July 2008

Abstract

Customizable and extensible processors (commonly known as "configurable processors" or ASIPs) can provide the flexibility of off-the-shelf processors with a performance closer to that of custom logic. Manual configuration of an ASIP requires highly-specialized knowledge of computer architecture and typically results in sub-optimal architectures leading to poor performance and higher costs. Ideally, the ASIP flow should be entirely automated; however, optimal solutions are only guaranteed with an exhaustive search of the design space. Unfortunately, an exhaustive search is computationally prohibitive and so the research community continues to study ways to find "good" solutions within a reasonable time.

This dissertation presents new methods of design space exploration and fast architecture evaluation. These methods are intended to improve the automation and usability of ASIPs. Design space exploration is conducted using a novel approach where the design space is modeled using a small sample of points. Each sample point evaluation is expensive; however, the design space model can then be used to quickly estimate all other points in the space. Non-parametric statistics are used to construct the model and, consequently, the precise nature of the design space need not be specified *a priori*. This approach provides a computationally-efficient alternative to existing optimization heuristics with additional benefits that provide easy discovery of architectural trends and tradeoffs.

Experiments were conducted using the proposed modeling approach to configure both the branch prediction unit (BPU) and the cache hierarchy of an embedded processor. Results showed that the approach could achieve a 100x speedup while providing near optimal configurations.

In addition, a fast performance estimation approach is proposed for evaluating configurations of instruction-set extensions. This approach considers pipeline effects and consequently improves the quality of results over existing approaches. This improvement is achieved while maintaining constant run-time complexity.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ADL	Architecture Description Language
AMBA	Advanced Microcontroller Bus Architecture
ANN	Artificial Neural Network
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
BPU	Branch Prediction Unit
CAD	Computer-Aided Design
DAG	Directed Acyclic Graph
DSE	Design Space Exploration
DT	Design Technology
EP	Embedded Processor
EPIC	Explicitly Parallel Instruction Computing
FPGA	Field Programmable Gate Array
FU	Functional Unit
GPP	General Purpose Processor
HDL	Hardware Description Language
ILP	Instruction Level Parallelism
IP	Intellectual Property
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
ITRS	International Technology Roadmap for Semiconductors
LOESS	LOcally wEighted Scatterplot Smoothing
LRM-DSE	Local Regression Modeling-based Design Space Exploration
NRE	Non-Recurring Engineering
PHT	Pattern History Table
PT	Process Technology
SoC	System-on-Chip
SoC-PE	Power-Efficient System-on-Chip
TLB	Translation Lookaside Buffer
VLIW	Very Long Instruction Word

Acknowledgements

This thesis would not have been completed without the help of many great people who made my life at UBC an enjoyable and enlightening experience.

I would first like to thank my supervisor, Dr. Resve Saleh, for the incredible opportunity. His passion for research, dedication to teaching, and wisdom in dealing with others, has mentored me throughout my degree. On countless occasions, he has gone far beyond the call of duty as a supervisor.

I am also grateful to all the other professors of the SoC research Lab for learning something from all of them. I would especially like to thank Dr. Steve Wilton, Dr. Guy Lemieux, Dr. André Ivanov, Dr. Mike Feeley, and Dr. Alan Hu, for serving on my committee and for their support throughout the years.

I would also like to thank past and present student members of the SoC lab for their support. I especially thank Victor Aken'Ova, Brad Quinton, Sohaib Majzoub, Julian Lamoureux, Pedram Samini, Dipanjan Sengupta, Mohsen Nahvi, Marvin Tom, David Yeager, and Zahra Ebadi. I would also like to thank the always helpful staff of the SoC lab, Roberto Rosales, Roozbeh Mehrabadi, and Sandy Scott.

Last, but certainly not least, I would like to thank my loving wife May and my children who have been extremely supportive throughout all of my work. I would also like to thank my father, mother and the rest of my family for the tremendous love and support they have shown me my entire life.

In Loving Memory of my father

Dr. Claus August Hallschmid

17/10/41 – 17/05/06

Chapter 1: Introduction

1.1. Motivation

Embedded systems are special-purpose designs intended for a specific application or set of applications and with specific design constraints (i.e., speed, power consumption, cost, reliability, usability, etc.). Such systems are typically embedded in mechanical or electrical hardware as part of a system deployed in the field. Applications for embedded systems include PDAs, cell phones, MP3 players, traffic control systems, guidance systems and medical instrumentation, to name a few. With a growing market demand for communications and portability, the use of embedded systems has grown correspondingly.

The cost of designing embedded systems and their constituent components has increased in a similar manner for several key reasons. First, the size of circuits has grown according to Moore's law which states that the number of transistors in an integrated circuit doubles every 18 months [1]. Also, deep-submicron effects are contributing to the rising costs because they make design more difficult on a per transistor basis. It has been predicted that if it were not for the CAD innovations developed to date, the average design costs of a *power-efficient system-on-chip (SoC-PE)* would be in the order of \$900M rather than the current \$20M [2]. The International Technology Roadmap for Semiconductors (ITRS) emphasizes that the current pipeline of CAD innovations must continue otherwise design costs will quickly become prohibitive [2].

While designs have become more complex, engineers have also become more productive in part due to increasingly more sophisticated CAD tools and methodologies.

In spite of this, a design trend known as the *productivity gap* continues to widen where design complexity continues to outpace the ability of engineers to meaningfully design them. To make matters worse, market pressures have steadily reduced the acceptable time-to-market. Designers have a reduced design window and a lowered tolerance for design revisions. This forces companies to incur the costs of additional engineers to help meet deadlines.

One approach to reducing development costs is to use System-on-Chip (SoC) methodologies. An SoC integrates pre-verified and pre-qualified intellectual property (IP) components onto a single chip. Design time is reduced through the use of off-the-shelf IP while performance is greatly improved over multi-chip solutions.

A central component of an SoC is the embedded processor which adds programmability to the system. With programmability, the system can be reused for a variety of applications, thus amortizing the cost of design and manufacturing. A design with programmability provides a lower risk and shorter time-to-market implementation path; however, increased flexibility generally comes with a power, speed, and area/cost overhead.

The likely trend for future SoCs will be an increase in the number of on-chip processors and other forms of programmability (i.e., Field Programmable Gate Arrays (FPGAs)). Figure 1.1 illustrates some of the key differences between processors, FPGAs and their fixed-function equivalent known as an *Application Specific Integrated Circuit (ASIC)*. Processors and FPGAs are both configurable devices, but they achieve configurability in very different ways. At one extreme, processors are configurable platforms that implement "temporal" computing and are programmed via software.

Processors are more suitable for complex state machines. At the other extreme, FPGAs implement "spatial" computing and are programmed via programmable look-up tables and programmable interconnect. FPGAs are more suitable for data-path applications where speed, logic density, and power are more important. A typical rule-of-thumb is that an FPGA is an order of magnitude faster, more dense, and more power efficient than the equivalent software running on a processor [3].

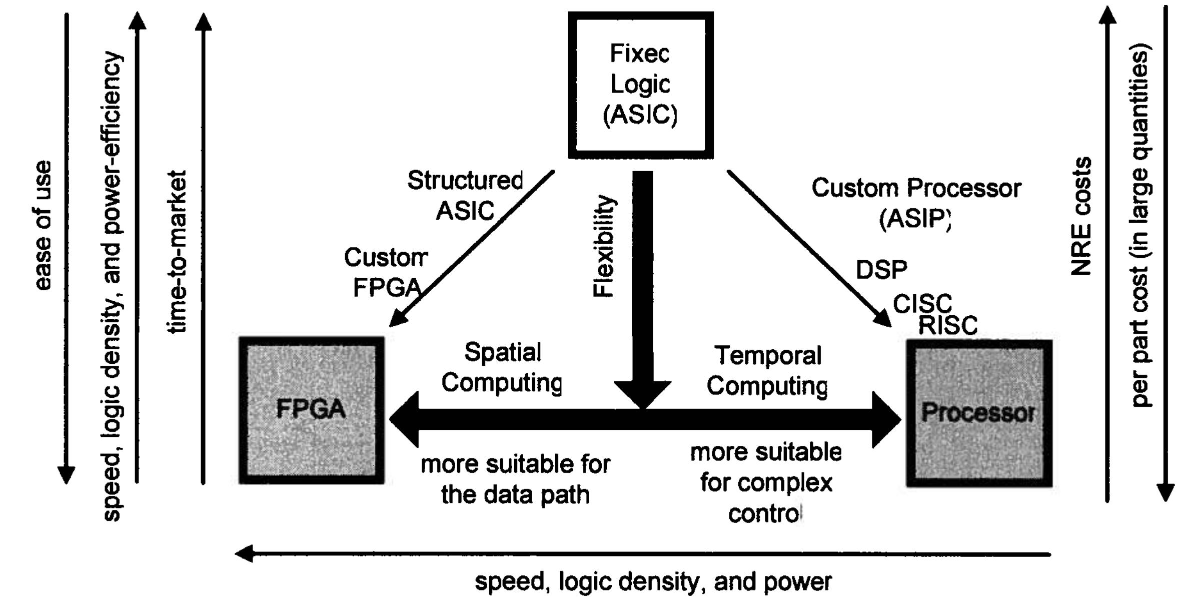

ease of use
speed, logic density, and power-efficiency
time-to-market
Fixec Logic (ASIC)
Structured ASIC
Custorr FPGA
Flexibility
Custom Processor (ASIP)
DSP
CISC
RISC
NRE costs
per part cost (in large quantities)
FPGA
Processor
Spatial Computing
Temporal Computing
more suitable for the data path
more suitable for complex control
speed, logic density, and power

Figure 1.1: Comparison of Processors, FPGAs, ASICs, and custom devices.

The vertical axis in Figure 1.1 represents the degree to which the device is configurable (i.e., flexibility). Programmable devices are generally easier to implement, have a faster time-to-market, lower risk, and lower NRE costs than the equivalent ASIC implementation [4]. On the other hand, they are slower, less dense, consume more power, and have a higher per part cost in larger quantities.

A *customizable device* is inherently programmable but has some aspects of its architecture tailored to a specific application or application domain. In doing so, these devices maintain many of the advantages of programmability such as reduced costs, reduced risk, and faster time-to-market. At the same time, they benefit from improved performance thus closing the gap between programmability and fixed implementations. This compromise between programmable implementations and fixed implementations is evident in Figure 1.1 where customizable processors and customizable FPGAs have been included.

A disadvantage to customizable devices is that they are no longer suitable for as wide a range of applications as for general purpose processors (GPPs). This is illustrated in Figure 1.2 where the set of all applications is shown with several sub-domains labeled for applications with particular speed, density, power, and reliability requirements. In this example, a device has been customized to work best for a specific domain within lower-power applications (i.e. a hypothetical set of similar compression algorithms).

Thus far, we have discussed customizable devices in general; however, the scope of this dissertation is on customizable or configurable processors, so it is useful to understand the key difference between embedded processors (EPs) and GPPs. EPs are intended to be run on a few applications which are known at design time whereas GPPs run a wide variety of applications which are chosen by the end-user. Because we have knowledge of the intended applications at design time, significant performance gains can be made by customizing the EP to the applications. As a result, the system designer only adds what is needed to the processor and can tailor components to precise specifications in order to meet area, speed, and power goals. In fact, customizable processors have been shown to result in performance gains of 1.12x to 11.3x over general-purpose processors [5].

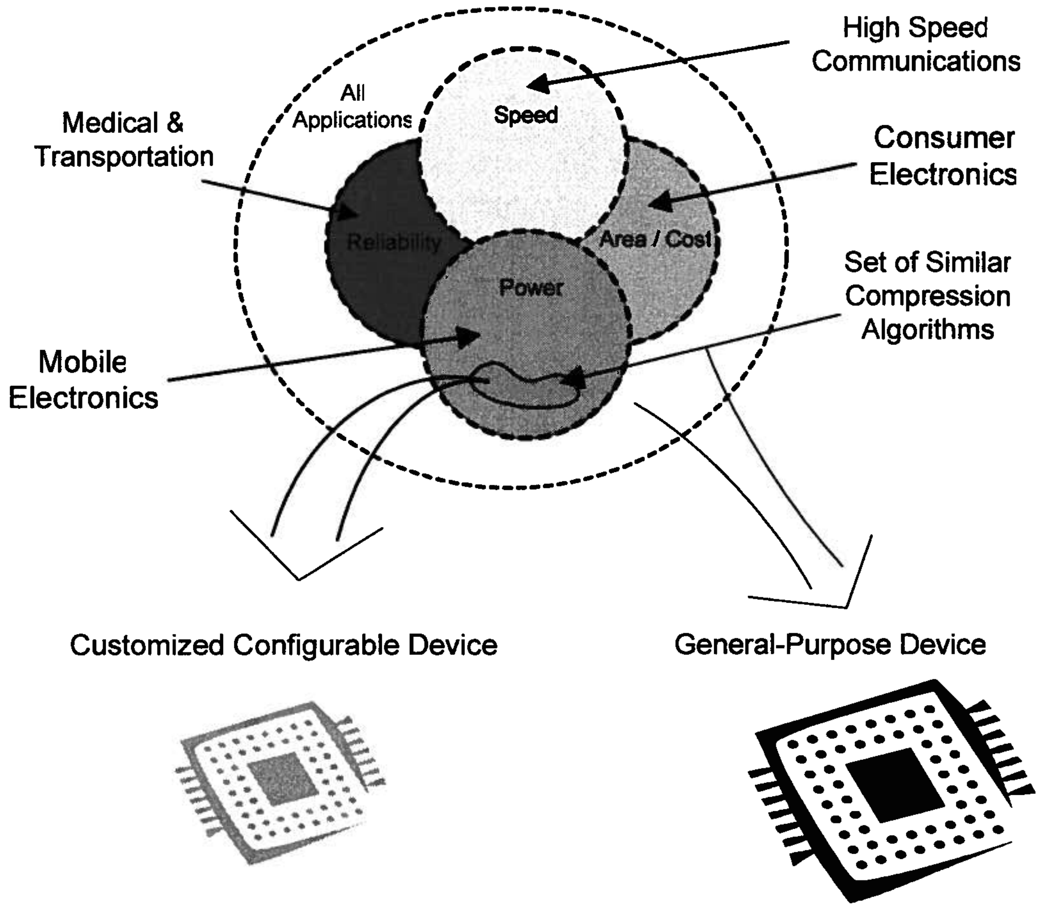


Figure 1.2: Hypothetical application domains for custom and general-purpose devices.

In the industry, customizable processors are commonly referred to as "configurable processors", or Application-Specific Instruction-Set Processors (ASIPs). To date, industrially-available ASIPs such as Tensilica Xtensa [6], ARCtangent [7], MIPS32 M4K [8], and Altera Nios/Nios II [9] have a base architecture with only a limited set of customizable parameters. Aside from these parameters, many other aspects of the architecture have a significant impact on performance and energy-efficiency but are normally fixed as prescribed by the base architecture. One goal of this research is to determine the proper settings of key parameters of the processor front-end to customize for reduced power.

Currently-available ASIPs are difficult to use because effective customization requires advanced knowledge of processor architecture. Even if a user were to have in-depth knowledge of computer architecture, it is difficult to manually find an optimal configuration due to the complex manner by which parameters interact. With this in mind, *the work in this research focuses on methods of automatically tuning the architectural parameters to meet specific design goals* while adhering to design specific design constraints.

A major difficulty in automatic customization of ASIPs is that a large solution space may be defined by the parameter set. Even with only a few parameters, exhaustive methods for exploring the solution space quickly become intractable due to a "combinatorial explosion." There are three orthogonal approaches to solving this. First, the solution space can be "pruned" using design constraints. Second, sophisticated heuristics can be used to minimize the time required to explore the design space. Third, fast evaluation techniques can be use to minimize the time needed to evaluate each point visited during exploration. The latter two approaches will be addressed within this dissertation.

This dissertation focuses on customizability of the front-end of the processor because of its impact on *instruction level parallelism (ILP)* and its significant contribution to both power and area. Henceforth, the combined fetch, decode, and memory hierarchy shall be referred to as the "front-end" of the processor. Improvements in ILP through front-end improvements typically come with a penalty of increased hardware (cache, buffers, and history look-up tables), all of which add significant power consumption in addition to that of the main memory. The trend for memory hierarchy is towards increased size because of the growing gap between memory and processor performance [10]. It is for these reasons that this dissertation focuses on the configuration of several key aspects of the front-end; in particular, it focuses on cache and branch prediction.

The primary design objective in this research is to reduce power. In both academia and industry, power has become the main focus of research and development for both embedded and high-performance processors. For embedded processors, reduced power consumption extends the battery life of mobile devices such as laptops and cell phones. For high-performance processors, reduced power consumption helps reduce power-density problems and related cooling costs.

1.2. Research Goals and Challenges

Having described the motivation and background for the research, a succinct thesis statement is as follows: *to investigate methods to improve the automation of customizable processors in order to make them more accessible to system-level designers*. The following research objectives where identified for this work:

1. To develop an efficient methodology for design space exploration. This method would be used for both optimization and assessing trade-offs and trends between parameters and design objectives. Further, this method should require only a minimal set of simulations and the user should not require in-depth knowledge of the design space.

2. To develop fast estimation techniques for evaluating candidate configurations when extending the instructions of a base instruction set.
3. To apply the proposed methods to the customization of industry-standard embedded processors using well-recognized benchmarks suites. The primary design objective will be power reduction. All techniques will be incorporated into an existing compiler and simulation flow to demonstrate the advantages and limitations of the new methods.

1.3. Thesis Organization

In Chapter 2, a brief overview is provided on embedded processors and ASIPs. In addition, an overview of previous work is described for key components of the front-end including branch prediction, cache management, and instruction-set extensions. Background is given on the design goals and base architecture used to perform experiments. Research described in this thesis has some overlap with previous work from the high-performance general-purpose processor community, the low-power embedded processor community, and the configurable processor community.

In Chapter 3, a Design Space Exploration (DSE) optimization methodology is proposed based on modeling of the design space and is called Design Space Modeling. First, motivations for modeling the design space are discussed. Second, a comparison is made between Design Space Modeling and existing path-oriented heuristics and instruction trace sampling approaches. Third, the methodology is outlined in detail with a comparison of four modeling variants. The first of these variants (called *Manual Decomposition*) is a first attempt at modeling the design space. This variant provides excellent results; however, construction of the model requires considerable manual effort and is therefore not scalable. The shortcomings of this method help to motivate the second modeling variant that also offers accurate models but is fully automated. This variant is a major contribution of the thesis and is discussed in detail in Chapter 4. The last two variants were developed by other research groups concurrently with this dissertation. To conclude the chapter, Manual Decomposition is applied to an example problem where the processor's branch prediction unit (BPU) is configured. This example is small enough to illustrate many of the concepts but complex enough to gain insight into the difficulties of the approach.

Chapter 4 proposes a non-parametric regression-based variant of Design Space Modeling. The advantage to this approach is that it is fully automated and that the class of functions required to perform the statistical fit need not be specified *a priori*. A detailed description of the approach is discussed. It is applied to the configuration of a 2-level cache hierarchy.

Chapter 5 discusses ways to improve the speed of customizing an Instruction-Set Architecture (ISA) using fast performance estimation. In this chapter, a general methodology for adding custom instructions (Instruction Set Extensions or ISEs) is introduced and includes a description of the current speed, power, and area estimation techniques. Because existing approaches do not account for data hazards, a novel "hazard-aware" estimation approach is proposed. This approach is evaluated by comparing it to existing approaches through experiments conducted using a modified ISE experimental flow.

Chapter 6 provides conclusions and directions for future work.

Chapter 2: Background

2.1. Overview

In this chapter, background information is provided beginning with a description of ASIPS in Section 2.2. Then, Section 2.3 discusses the base processor architectures used. Section 2.4 discusses the primary design objectives which include *instruction-level parallelism (ILP)* and power efficiency. Section 2.5 provides background information on the architectural parameters to be configured. Finally, Section 2.6 provides background information on the two methodologies addressed which includes design space exploration and instruction set extensions.

2.2. Customizable Processors / ASIPs

2.2.1. Configurability

A detailed classification of configurable platforms is shown in Figure 2.1. It is composed of the three dimensions of *Compute Abstraction Level*, *Binding Time*, and *Reconfigurable Feature*. Compute Abstraction Level is the level of configurability presented to the user and can vary from system-level configurability to logic-level configurability. This dimension is shaded on the y-axis in Figure 2.1.

The binding time refers to the time at which the platform is configured. For some platforms such as Tensilica Xtensa [6] and Actel Axcellerator [11], the configuration is set at production/fabrication time. Other platforms, such as the FPGAs available through Xilinx [12] and Altera [9], are configured at start-up. Still others, such as GARP [13] and Chimaera [14], can be configured as many times as needed at run-time. Platforms with production, start-up and run-time configurability are often referred to as configurable, reconfigurable, and dynamically reconfigurable, respectively. In this thesis, platforms with production-time binding are referred as *customizable platforms*. This dimension is shaded along the y-axis in Figure 2.1.

The configurable dimension is shaded along the x-axis in Figure 2.1 and includes communication elements, storage elements, and computational elements. This dimension represents the manner in which the platform supports configurability.

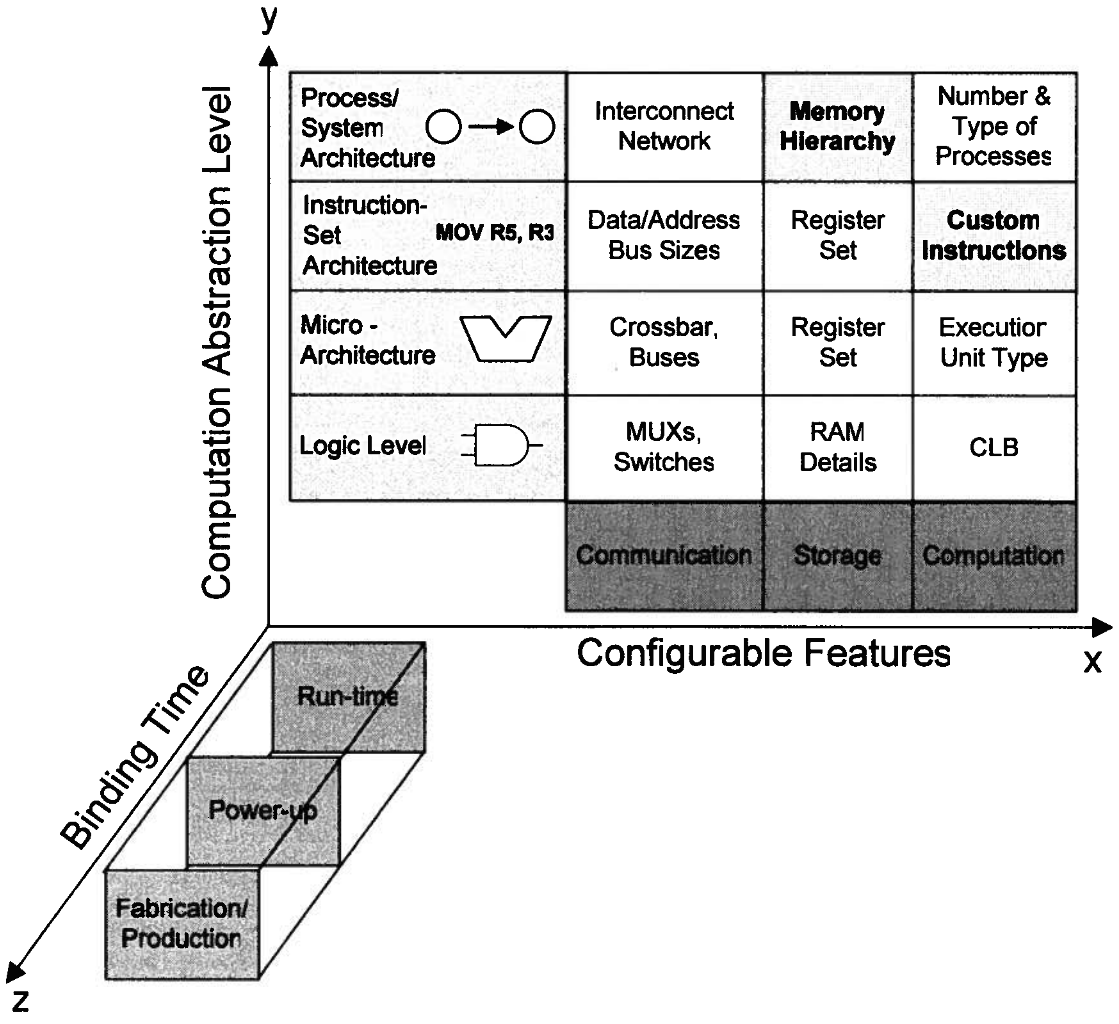


Figure 2.1: Classification of configurable platforms (adapted from [15] and [16]).

Example architectural features are shown in Figure 2.1 for the space defined by the compute abstraction level and reconfigurable features dimensions. Two examples in this table that will be discussed later in this dissertation are the bold blocks. One of these is the memory hierarchy which is a system level issue when sizing is performed. The other example is instruction customization which is an ISA level issue. As will be discussed in subsequent chapters, the work in this dissertation focuses on configurable devices with binding times at fabrication/production.

2.2.2. Customizable Processors

An *application-specific instruction set processor (ASIP)* is a processing element that can be customized for a particular application domain [17,18]. The ASIP design flow allows system designers to build processors that can execute code faster while using less power than off-the-shelf processors. This avoids expensive and risky development of a fully custom processor or Application-Specific Integrated Circuit (ASIC). Previous studies have shown improvements from 1.12x to 11.3x over general-purpose processors [5]. Using these processors, designers can tune the instruction set architecture, optimize the pipeline, or add new data components for critical code segments in specific applications.

ASIPs come either in the *soft* or *hard* form. A soft ASIP is implemented on an FPGA and its binding time is at power-up. Industrially available soft ASIPs include Altera Nios/Nios II [9] and Xilinx MicroBlaze [12]. A hard ASIP is one for which the binding time is at fabrication time and is implemented using a standard cell flow like an ASIC. Industrially available ASIPs of this form include Tensilica Xtensa [6], ARC tangent [7], MIPS32 M4K [8] and Improv Jazz [19].

ASIP methodologies use compiler analysis, directed profiling, and design automation techniques to transform a base architecture to one that is optimized for a particular application or application domain. This allows the processor to take advantage of particular regularities in control and data behavior, and common clusters of operations.

2.2.3. ASIP Methodologies

The primary focus of most research in ASIPs is the study and development of algorithms to automate the configuration flow. Many methodologies have been suggested [20,21,22,23,24,25,26]; most methodologies include the following five steps of typical ASIP synthesis [27] which are illustrated in Figure 2.2:

1. *Application Analysis*: statically or dynamically analyzes the application and input data.
2. *Architecture Design Space Exploration*: enumerates all possible parameterizations of the architectures given the design constraints. Power, speed, and area are estimated and used to identify the candidate architecture that minimizes a user-defined metric. This phase will be discussed further in Section 2.6.1.
3. *Instruction Set Generation:* extends the Instruction-Set Architecture (ISA) of the processor to include new instructions that minimize power, speed and area. Newly-defined operations can be of type Fused, Vector, or Very Long Instruction Word (VLIW) [5]. Fused operations are the combination of several simple operations with the outcome of reduced code size, reduced fetch and issue bandwidth needs, and possible reduced register file port requirements. Vector operations increase throughput by operating on more than one data

element in parallel. VLIW operations contain multiple independent operations that are encoded into the instruction in "slots". VLIW operations add the potential of significant performance improvements through software pipelining and instruction scheduling techniques [28]. Each of these new instruction types vary in the amount of additional hardware needed. As shown in Figure 2.2, this phase of synthesis can be broken down further into *Pattern Enumeration*, *Pattern Selection*, and *Pattern Mapping*. Each of the sub-phases will be discussed further in Section 2.6.2.

4. *Code Synthesis*: either generates a compiler suitable for the new architecture or directly emits the executable code for the application.
5. *Hardware Synthesis*: synthesizes the processor based on ASIP templates written in a hardware description language (HDL) using standard tools.

This research is chiefly concerned with design space exploration and instruction-set enumeration and selection (shaded blocks).

The Mescal research project [17] developed a set of five *elements*, as shown in Figure 2.2, which together provide a coherent approach for the development and deployment of programmable platforms. These five elements are as follows:

1. *Judiciously Using Benchmarks:* Benchmarks used to perform ASIP synthesis should be chosen such that results are comparable across various system implementations. Further, benchmarks should be representative of the application domain, should be indicative of the real-world application performance, and should be well-specified. The selection of benchmarks used within this thesis will be discussed for each set of experiments separately.

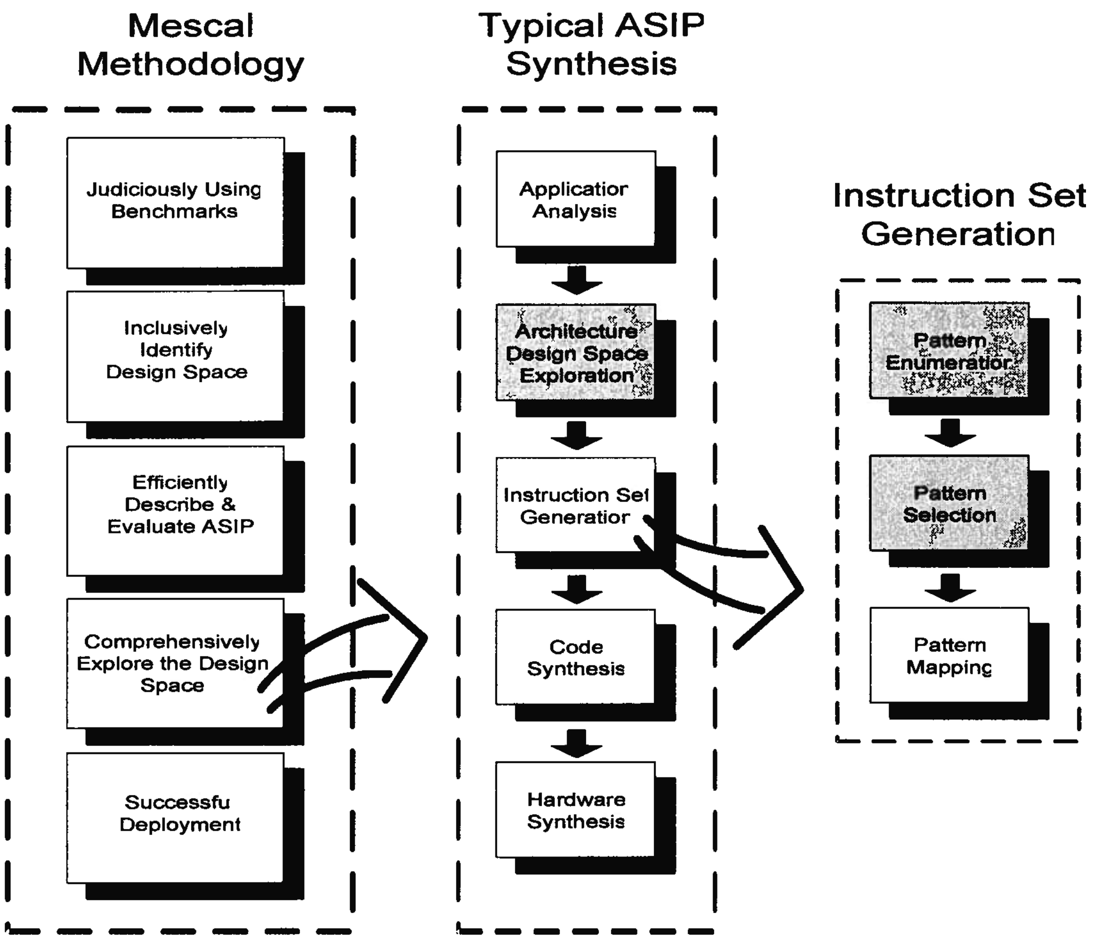

Mescal Methodology
Typical ASIP Synthesis
Instruction Set Generation
Judiciously Using Benchmarks
Inclusively Identify Design Space
Efficiently Describe & Evaluate ASIP
Comprehensively Explore the Design Space
Successfu Deployment
Applicatior Analysis
Architecture Design Space Exploration
Instruction Set Generatior
Code Synthesis
Hardware Synthesis
Pattern Enumeratior
Pattern Selection
Pattern Mapping

Figure 2.2: General ASIP Configuration Methodology [17, 27 , 29].

2. *Inclusively Identify the Design Space:* In the past, the design space explored has been relatively small and often limited to only those parameters with which the designer is familiar. Defining a broad set of parameters increases the likelihood of discovering more optimal configurations. For this reason, the design space should be *inclusive*. The focus of this thesis is on novel methods for DSE; consequently, an assumption is made that the design space has been correctly identified and is inclusive.

3. *Efficiently Describe & Evaluate the ASIP*: In order to evaluate a broad range of architectures, each architecture must be easy to describe and evaluate. Efficient methods must be developed to efficiently map the application benchmark(s) onto each candidate architecture. What is necessary is a re-targetable software environment for mapping and evaluation. The development of this environment was an important part of the experimental platforms used in this research work.
4. *Comprehensively Explore the Design Space:* DSE involves two orthogonal issues: first, how each point in the design space should be evaluated, and second, how much of the design space should be covered during exploration. These questions are important because the sheer size of most design spaces makes exhaustive searches computationally infeasible. As a result, intelligence must be built into exploration to minimize the number of points visited in order to find optimal points and to minimize the cost of evaluating point each visited. This represents the central theme of this thesis. Chapters 3 and 4 propose a novel DSE approach that models the design space while only having to evaluate a small sample of points. Chapter 5 proposes a new method to speed-up the evaluation of candidate architectures while performing instruction set generation.
5. *Successfully Deploy the ASIP:* ASIP synthesis may provide an excellent match between the architecture and the intended application while at the same time could result in a useless end product because of poor deployment. Successful deployment implies that the device must be easy to program, debug, and simulate. To easily program the ASIP, it must be at a sufficiently high level to

make it practical; however, it must also be low enough to take advantage of ASIP architectural features. This element is outside the scope of this thesis.

Examples of ASIP flows that are available from industry include the Embedded Processor Designer by CoWare [30], the Sx000 product line from Stretch [31], the Synfora PICO Express processor array [32], and Clarity by Mimosys [33]. The CoWare ASIP flow emits a custom processor based on the LISA processor design platform [34] and a custom compiler using the CoSy compiler development system by ACE [35]. CoWare also emits a configurable instruction-set simulator (ISS) and debugger. Unfortunately, candidate architectures must be specified manually using the LISA architecture specification language.

Synfora's PICO Express is based on the PICO processor array originally developed by Hewlett Packard laboratories [36]. PICO is based on an *explicitly parallel instruction computing (EPIC)* processor developed by Hewlett Packard and Intel which has its origins in *very long instruction set (VLIW)* processors. The PICO flow includes significant innovations in automatic processor configuration and will be discussed further in Section 2.6. The Trimaran compiler infrastructure is used for PICO and is also used for the experimental framework developed in Chapter 5 of this thesis.

The Stretch processor line is based on Tensilica technology but adds automatic customizability of the instruction set. Included with Stretch products are a customizable compiler, simulator, and debugger. Mimosys Clarity also customizes the instruction set automatically but does so for a number of target architectures including ARC, MIPS, Xilinx, Altera, Tensilica, IBM, ARM, CoWare, and Toshiba.

Academia offers several ASIP flows including LISA [34], ASIP Meister [37], and Expression [38]. Each of these tools allows the system designer to specify the desired architecture of the processor manually. They do not provide any form of automatic configurability; however, a compiler flow and simulator is generated. Using these tools, the user can iterate through various architectures until the desired performance is met. For LISA and Expression, the user specifies candidate architectures using an *architecture description language (ADL)*; for ASIP Meister, the user specifies the architecture through a graphical user interface. In the case of LISA and ASIP Meister, a *hardware description language (HDL)* description of the processor can be emitted.

2.3. Base Architectures for ASIPs

2.3.1. Embedded Processors

An *embedded processor (EP)* differs from a *general-purpose processor (GPP)* in that an EP is designed for one or a few specific applications. Because the EP is designed for specific task, it can be optimized and tailored to increase performance, reduce cost, and increase reliability. EPs are often produced in mass quantity thus taking advantage of *economies of scale*. Typical uses for embedded processors include phones, DVD players, HDTV sets, photocopiers, GPS navigation, printers, routers, automobiles, personal digital assistants (PDAs), and MP3 players.

With an increase in market demand for communications and portability, EPs have continued to grow at a faster rate than GPPs. In 1998, 2.5 billion embedded processors were installed, compared to 100 million general-purpose machines [39]. In 2002, 98% of all processors were embedded processors [40].

2.3.2. ARM7TDMI

Experiments conducted throughout this thesis make use of the most popular embedded processor called ARM7TDMI which is a well-known member of the ARM7 family [41]. The ARM7TDMI is a 32-bit reduced instruction set computer (RISC) on a Princeton architecture designed by ARM [42]. Although ARM7 is over 14 years old, it is still seeing extensive use in the market. Because of its popularity, it is also frequently used for academic experimental flows. Therefore, it is a natural choice for use as a base architecture for the research in this thesis.

In addition to the configurability discussed in Section 2.2.2, ARM has incorporated one form of configurability through the use of its *Advanced Microcontroller Bus Architecture (AMBA)* [43] bus. AMBA is an open bus standard designed by ARM for use in *System-on-Chip (SoC)* platforms. AMBA provides a technology-independent solution that can easily be used for modular *Intellectual Property (IP)* design. ARM also offers *PrimeCell Peripherals* [42] which are re-usable, pre-verified, AMBA-compliant macrocells. The combination of AMBA and PrimeCell, to some extent, provides a configurable SoC infrastructure based on reusable modular design.

2.3.3. StrongARM SA-110

In 1995, the StrongARM SA-110 processor [44] was launched by ARM and Digital Equipment Corporation (sold later to Intel in 1998). The StrongARM was based on the ARM architecture but was intended for higher-speed applications. One key difference is that StrongARM has a separate data and instruction cache rather than the unified cache of the ARM7TDMI. The processor has also been prevalent in academia and industry and so it is also used as a base architecture for the research outlined in this thesis.

2.4. Design Objectives

The primary design objective used throughout this dissertation for optimization experiments is energy. The total energy dissipated by the processor to complete a task is a combination of the power it dissipates and how long it takes to execute the task. Achieving a low-energy solution typically requires a balance between reducing architecture size and complexity to reduce the *power* dissipated by the processor and increasing architecture size and complexity to improve Instruction Level Parallelism (ILP) (which consequently reduces the execution time, thus saving energy). An in-depth discussion of both ILP and power is discussed in the following sections.

2.4.1. Instruction Level Parallelism (ILP)

ILP is a measure of how many instructions in a program can be executed in parallel. With increased ILP, the processor can complete program execution within a shorter number of cycles, thus consuming less energy as shown in the next section.

Program execution is inherently sequential; however, there are several mechanisms that provide ILP including pipelined execution, superscalar execution, out-of-order execution, and VLIW techniques [45]. All of these methods can increase ILP; however, there is a significant amount of overhead circuitry and memory that contribute additional power. In most cases, the benefits of ILP must be carefully weighed against the power and area penalties associated with the overhead.

A basic block is a sequence of instructions with only one entry point, one exit point, and no jump instructions contained within. Techniques have been developed to improve ILP within blocks and between blocks. Within blocks, traditional techniques include *register renaming* [46] and *aliasing analysis* [47]. Across block boundaries, *branch prediction* [48], *loop unrolling* [49], *software pipelining* [28], and *trace scheduling* [50] are traditional techniques.

The theoretical limit to ILP was explored by Wall [51] using what he called an "impossibly good" architecture that has effectively unlimited resources with perfect branch prediction and speculation. In that work, Wall found that such architectures could achieve an ILP of 7 at best, with 5 being more common. Lam et al. [52] attributed limitations of ILP primarily to control flow and then explored how this can be resolved by speculatively executing multiple flows. Processors with the ability to speculatively execute multiple flows are relatively complex and are not considered to be suitable for embedded systems.

In this dissertation, ILP is regulated by tuning a parameter in the branch prediction unit (BPU) called the pattern history table (PHT) and through several parameters in a 2-level cache hierarchy.

2.4.2. Power

Power consumption for CMOS logic is typically expressed as:

$$P = \alpha C V_{DD}^{2} f + \tau \alpha V_{DD} I_{short} f + V_{DD} I_{leak} \tag{2.1}$$

where the terms represent dynamic power, short-circuit power, and leakage power, respectively [53]. α is the gate activity, C is the total capacitance of the gate, V_{DD} is the supply voltage, f is the operating frequency, τ is the short-circuit switching time, I_{short} is the short-circuit current, and I_{leak} is the leakage current. Further, the following proportionality relationship holds, where V_T is the threshold voltage:

$$f_{\max} \propto (V_{DD} - V_T)^2 / V_{DD} \tag{2.2}$$

In Equation (2.1), the first term represents a dynamic contribution due to capacitive charging and discharging, and has traditionally been the dominant term due to increases in f over the years. On the other hand, scaling supply voltages and threshold voltages has increased the contribution of static power dissipation from the last term in Equation (2.1) which is based on leakage current[1]. The second term in Equation (2.1) represents a relatively small contribution due to switching current and is often incorporated into the first term.

At the architectural level, two important aspects that affect power consumption are the level of parallelism and the amount of memory used for main memory, cache, branch prediction, and the translation lookaside buffer (TLB). At a fundamental level, the effect of parallelism is evident from Equations (2.1) and (2.2). Parallelism can reduce the intrinsic power needed to complete a task because it permits the supply voltage to be scaled. For a system with a doubling of parallelism, ½ the voltage is needed to maintain throughput; thus, only ¼ of the dynamic power is needed. With overhead, leakage, and short-circuit power neglected, it has been shown that the general equation for a circuit with N levels of parallelism is:

[1] Leakage current is composed of both subthreshold leakage and gate-oxide leakage.

$$P_N / P_1 = V_N^2 / V_{ref}^2 = 1 / N^2 \tag{2.3}$$

where V_N is the supply voltage for an N processor system and V_{ref} is the reference supply voltage for a system implemented with one processor [54]. This equation represents an upper bound on the reduction in power dissipation due to parallelism.

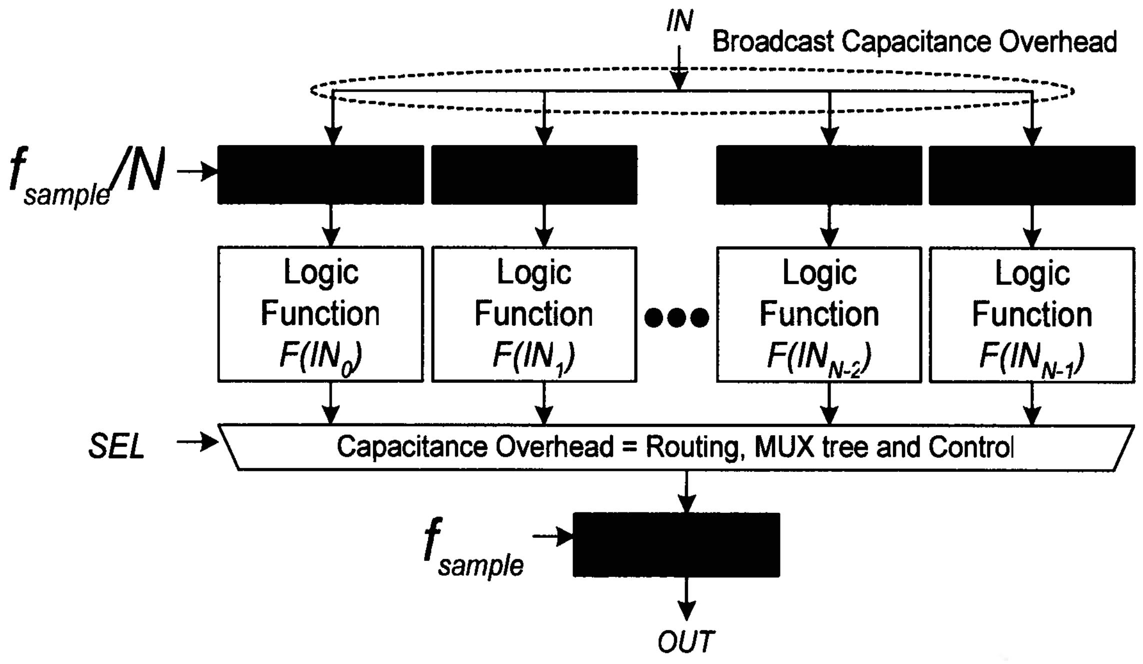

IN
Broadcast Capacitance Overhead
f_{sample}/N
Logic Function $F(IN_0)$
Logic Function $F(IN_1)$
●●●
Logic Function $F(IN_{N-2})$
Logic Function $F(IN_{N-1})$
SEL
Capacitance Overhead = Routing, MUX tree and Control
f_{sample}
OUT

Figure 2.3: General parallelization of a logic function.

Overhead power must be accounted for in the extra circuitry needed to facilitate parallelism. Figure 2.3 illustrates the general case where a logic function, F, sampling the input at a rate of f_{sample} is parallelized such that there are N logic functions, each sampling input at a rate of f_{sample}/N. In the figure, overhead circuitry is needed to broadcast the inputs and to combine the outputs. With overhead circuitry considered, Equation (2.3) becomes [54]:

$$P_N / P_1 = \left(1 + \frac{C_s(N)}{N \cdot C_{\text{ref}}} + \frac{C_{ns}(N)}{C_{\text{ref}}}\right) \cdot \left(\frac{V_N}{V_{\text{ref}}}\right)^2 \tag{2.4}$$

where $C_s(N)$ is overhead that scales the operating frequency with increased parallelism and $C_{ns}(N)$ is overhead that does not scale the frequency with increased parallelism. As the operating voltage is scaled in Equation (2.4), capacitance due to overhead circuitry may grow to the point that there is no longer a benefit to increase parallelism. It is for this reason that the degree of parallelism must be carefully selected to optimally reduce power dissipation.

Equation (2.4) does not consider power dissipation due to leakage current. Static leakage power is heavily dependent on the form of parallelism used whether it be logic function duplication or pipelining [55]. If parallelism is achieved by duplicating logic functions, then static power dissipation is effectively duplicated. Alternatively, parallelism can be achieved by pipelining the logic function. Other than inter-stage latches, no other circuitry is added for this solution; thus, static power is minimized. Parallelization of a logic function through pipelining is usually the more power-efficient solution. Unfortunately, the power benefits of pipelining may also be offset by data and control hazards.

For most applications, parallelism is used not only to reduce power dissipation but also to reduce the number of cycles of execution. Both of these objectives have motivated aggressive instruction-level parallelism techniques in the design of processors. As described in the previous section, it is difficult to extract parallelism from most applications. In fact, most modern processors include significant overhead hardware in an attempt to maintain very modest levels of parallelism. This overhead hardware dissipates both dynamic and static power. Examples of this include the cache hierarchy and the branch prediction unit.

In order to minimize power dissipation, processor designers must find the correct balance of the size and complexity of extra hardware used to improve parallelism. Large and complex hardware improves parallelism (which reduces intrinsic power and reduces run-time) while at the same time dissipates excess dynamic and static power. At the same time, reduced run-time also reduces the static power from all other components of the processor [56]. Figure 2.4(a) shows power dissipation for a processor with *reduced* hardware dedicated to maintaining parallelism and Figure 2.4(b) shows a processor with *increased* hardware dedicated to maintaining parallelism. This figure shows the power/run-time tradeoff between all circuitry related to improving parallelism. In addition, fixed-rate circuitry is shown which represents all components of the processor that dissipates static power at a fixed-rate regardless of the level of parallelism. Also shown in Figure 2.4(a) is the excess energy due to fixed-rate power dissipation caused by a longer execution time.

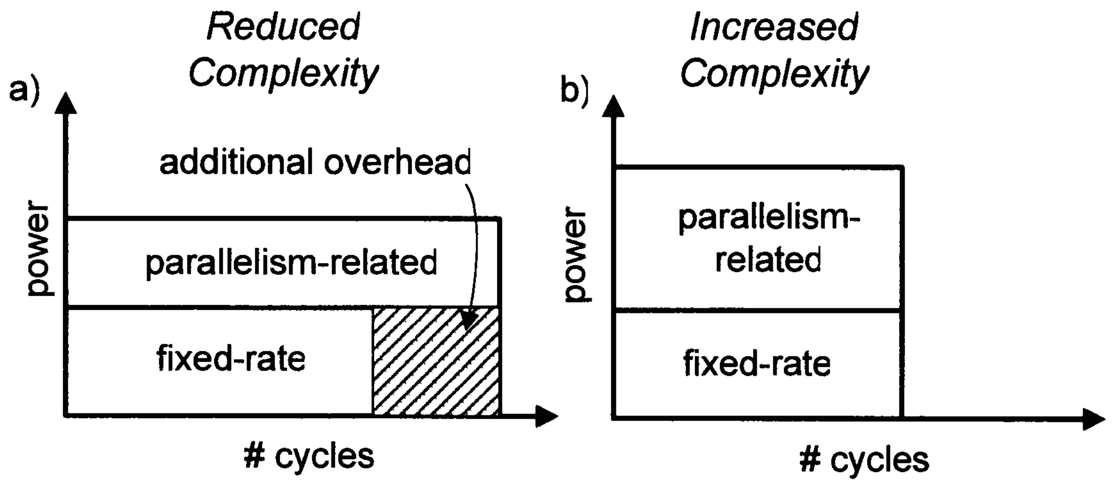


Figure 2.4: Parallelism-related versus Fixed-rate Power Dissipation.

This dissertation primarily focuses on the configuration of memory structures to reduce power. The memory hierarchy (and branch prediction) of a processor consumes a significant amount of power, often being the dominant source of power [57]. Memory systems have two sources of power loss. Dynamic power consumption is caused by frequent access of the memory and static power consumption is caused by leakage current. At the architectural level, power dissipation can be reduced in memory systems by correctly reducing the code size, organizing the code to reduce memory access, correctly sizing the memory, using a suitable cache organization, and using memory banking techniques.

2.4.3. ASIPs and Power

There is a significant body of research dealing with reduced power dissipation for general-purpose processors at the logic level [58,57,59,60,61,62], architectural level [63,64,65,66,67], and the system/OS level [68,69]. Previous work on power and configurability specific to branch prediction, the cache hierarchy, and instruction-set extensions and are provided in Sections 2.5.2, 2.5.3, and 2.6.2 respectively.

Research on power in ASIPs is still in its infancy. Work by Glökler et al. [70] uses gate-level simulations to jointly consider speed, area, and power while optimizing the number of pipeline stages, clock gating, logic netlist restructuring, data-path optimizations, instruction memory power reduction by optimized instruction encoding, and the implementation of coprocessors. They claim that these optimizations together improve power dissipation by 92%; however, they do not explicitly address the trade-offs between energy and runtime. Further, they only consider a limited set of course-grain parameters at the architectural level and do not automate the flow.

In this dissertation, ASIP power optimization is approached differently. The goal is to use a general-purpose approach that is highly automated and can be used to configure a wide variety of course- and fine-grain parameters at the architecture level. This makes automation difficult because of the large inclusive design space and because DSE takes place early in the flow thus making it difficult to evaluate the value of each candidate configuration.

2.5. Architectural Parameters

2.5.1. Motivation for Configuring the "Front-end"

As a rule of thumb, a processor uses an order of magnitude more power than an FPGA to compute the same task [3]. The most significant explanation for this problem is that processors fetch and decode instructions from the memory hierarchy, which is relatively slow, impedes *instruction level parallelism (ILP)*, and consumes significant area and power. Here, we refer to the combined fetch, decode, and memory hierarchy as the "front-end" of the processor.

The purpose of the front-end is to supply valid decoded instructions to the execution core with *low latency* and *high bandwidth* to maintain ILP. In the presence of highly control-based applications (which is typical for tasks assigned to processors), it is difficult to speculate which instructions should be delivered to the execution core after encountering a control operation such as a branch. On average, the CPU is in a steady state 50% of the time, stalled 20% of the time, and in transition between steady and stall 30% of the time [71]. A substantial amount of research has focused on improving ILP but with limited success.

Increases in ILP through front-end improvements typically come with the penalty of increased resources (cache, buffers, and history look-up tables) all of which add significant power consumption in addition to that of the main memory. The trend for memory hierarchy is towards increased size because of the growing gap between memory and processor performance [10]. It is for these reasons that this work focuses on the front-end and the memory hierarchy.

Configurability can be used to adjust the complexity of the front-end to control how instructions are speculatively fetched for a given application. If the front-end has more resources than necessary (over-speculation), then power is wasted through excess dynamic and static power. If there are not enough resources (under-speculation), then excess power is consumed because of reduced ILP, pipeline stalls and the execution of incorrectly fetched instructions.

2.5.2. Branch Prediction

An initial investigation of the role of branch predictor organization on power was described in [72]. This work concluded that it is better to spend extra power on a more complex branch predictor if it results in more accurate predictions and improves run-time. In spite of this conclusion, both methods proposed in [72] reduce power dissipation solely by reducing the capabilities of the branch prediction unit (BPU). They first suggest that power can be saved by banking the branch predictor in a similar way to what has been done in the past for caches. Second, they propose a prediction probe detector (PPD) that is used to switch off the BPU for non-branch instructions.

In [73], profiling is used to determine whether each branch instance is "biased" towards global or local predictability. Branch instructions are then encoded with a bit that specifies whether their direction should be predicted using a global type predictor such as GSelect [74] or a local type predictor such as Bimodal [75].

The approach taken in [76] is similar to [73] in that the application is profiled to determine which of the gated parts of the BPU should be switched off at run-time. In [76], the branch target buffer (BTB) is resized and parts of the hybrid predictor are disabled. In both cases, power dissipation is reduced in part by improving run-time. The two approaches differ in that resizing of the BTB in [76] requires extra hardware for run-time support. Further, [76] sizes the BTB *a priori* using a "brute-force" trial-and-error approach where they simulate all possible configurations in the search space. Even with just a few other dimensions in the search space, this approach would quickly become impractical.

Previous work has focused on run-time solutions by gating part of the BPU. The processor architecture is fixed but has extra hardware in the pipeline and BPU specifically to reduce power. Conversely, extra bits are encoded into the instructions to provide the processor with "hints" on how to save power. In either case, extra hardware must be added. The work discussed in Chapter 3 differs in that the architecture of the processor BPU is undecided until the synthesis tool/compiler profiles the application, compares all possible configurations using a cost function, and finds the correct trade-off between BPU complexity and execution time. Extra hardware is not needed to provide run-time support.

2.5.3. Cache Hierarchy

The cache hierarchy has a significant impact on the total power and performance of the processor. In some cases, the memory hierarchy of a microprocessor can consume as much as 50% of the system power [77]. One approach is to save energy in the cache [78] using schemes such as way prediction [79], selective cache way access [80], sub-banking [81], multi-banking [72], selective prediction [82], and confidence prediction and throttling [83].

Another approach for saving energy is to tune the cache based on the application which is the approach taken in this dissertation. Currently, many vendors offer tunable caches; however, the designer is left to manually choose the correct configuration for their application. Configurable caches are now offered by several vendors with most of them configurable at design time. This is true for processor cores such as Tensilica Xtensa [6], MIPS [8], ARM [42], ARC [7], and Altera Nios/Nios II [9]. Motorola offers a processor with reconfigurable cache hardware called M*Core [77] where individual ways in the second-level cache may be specified for use by data, instructions, or both. In addition, ways may be individually shut down.

Industrially available processors offer configurable caches; however, they do not provide mechanisms for automatically configuring them. This presents a significant problem to system designers who may not have in-depth knowledge of processor architecture and who most often have significant time-to-market pressures. This is especially true for cache due to the sheer number of possible configurations within its design space. Tensilica Xtensa currently has only one level of cache hierarchy but it can be configured in 6561 ways. Reconfigurable caches such as M*Core [147] have few parameters with few configurations; however, a second level unified cache increases the number of configurations to 17,640. Exploring these configurations has the potential to take many weeks thus slowing down the design cycle dramatically. To worsen the problem, the industry trend is towards increased cache complexity with an increase in the number of levels. Clearly, any advances in DSE would have a significant impact on cache configurability and performance.

Automatic cache configuration has been well-studied and includes work that configures the cache through simulation [84,85], trace reduction [86], and analytical approaches [87]. Several heuristics have been developed to help speed-up the time required to search the solution space while tuning the cache. The previously exhaustive searches used to tune the L1 cache of the Platune [88] framework was improved by Palesi et al. [89] using a genetic algorithm. Zhang et al. [90] propose a heuristic where each design parameter is searched in order of its impact on energy and performance. In doing so, a list of *Pareto-optimal* solutions is generated. Ghosh et al. [91] use an analytical model to explore cache size and associativity to directly find the configuration that meets the designer performance constraints.

Few methods exist for exploring multiple-level caches. Balasubramonian et al. [92] propose a method of redistributing the cache size between the L1 and L2 caches or between the L2 and L3 caches while maintaining a conventional L1 cache. They achieved a 43% energy improvement. Gordon-Ross et al. [93] designed a cache tuning heuristic that explores separate L1 data and instruction caches as well as separate L2 data and instruction caches. In that work, an examination of 7% of the design space yielded a 53% reduction in energy.

Current state-of-the-art heuristics by Gordon-Ross et al. [94] can achieve a search-space speedup of up to 500x over an exhaustive search resulting in a reported

62% energy savings and a 35% performance improvement. The downside to this heuristic is that it is specific to a particular architecture, the Motorola M*Core.

Configurable cache subsetting was proposed by Viana et al. [95] where a small subset of configurations can be identified for a specific set of benchmarks. The intention of this work is that a user with an application in the same domain as the original set of benchmarks need only consider a small subset of configurations rather than the entire solution space. Although their approach shows promise, the problem they address is different from this dissertation. Here, no assumptions are made that an application will fall into the domain of a prescribed set of benchmarks. To guarantee a high quality solution, *single-application* design exploration should be performed for the specific application in question.

2.6. Design Methodologies

2.6.1. Design Space Exploration (DSE)

System design space exploration typically follows a Y-chart approach [96] as shown in Figure 2.5. It illustrates a concept referred to as a *separation of concerns* [97, 17] where one or more descriptions of the application (workload, computation, and communication tasks) and the architecture specification are kept separate. A mapping phase binds application tasks to architecture building blocks. An evaluation of the mapping may include synthesis, compilation, transformation, and simulations, resulting in performance statistics. Constraints from the architecture, workload description, and application may influence the evaluation phase. Results from evaluation would determine whether or not successive iterations of mapping and evaluation will take place to iteratively improve the quality of the mapping. In this thesis, two aspects of the Y-chart are dealt with. First, we explore methods for improving DSE where intelligent coverage of the design space is achieved through systematic modification of the mapping phase. Second, we explore methods of improving the way by which mappings are evaluated.

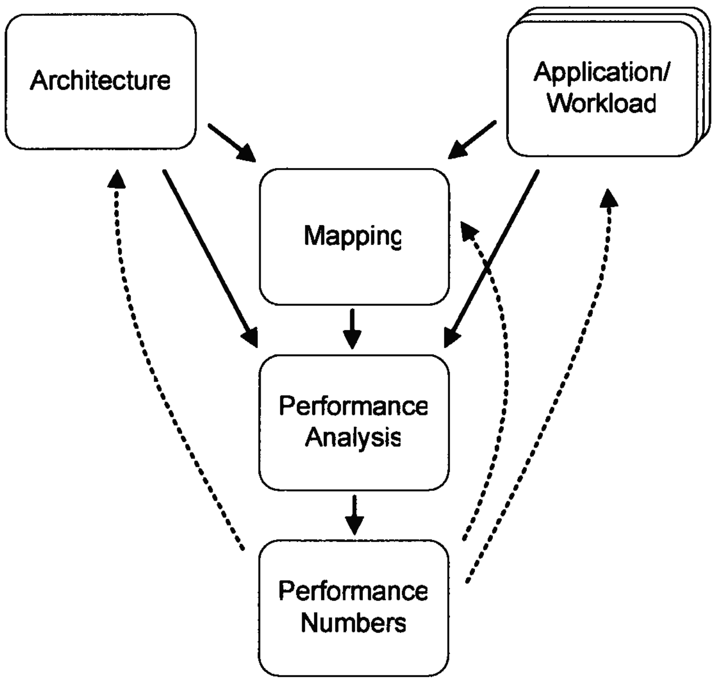


Figure 2.5: Y-chart representation of the separation of concerns for design space exploration.

A number of challenges exist relating to DSE. The dimensions of a design space are defined by the set of configurable parameters with each point representing a unique configuration. For a given set of N design parameters P_i, $0 \leq i < N$, with C_i possible configurations each, the total number of configurations in the design space can be expressed as,

$$C = \prod_{i=0}^{N-1} C_i \, , \tag{2.5}$$

Due to its sheer size, an exhaustive searching a design space becomes computationally prohibitive as the number of parameters grows. To make matters worse, complexity is further increased when optimizing multiple objectives; however, objective functions can be combined to form a *cost function* or used as a *Pareto Optimal* objective where all objectives are maximized to the extent that an improvement of one metric results in the degradation of another.

Aside from an exhaustive search, there are a number of more efficient strategies for exploring the design space. One approach is to randomly sample the design space using what is often referred to as a *Monte Carlo* based approach [98]. This approach lacks any form of intelligence; however, it has the advantage of an unbiased selection criterion which is less likely to overlook less than obvious optimal configurations. One approach based on Monte Carlo is *Simulated Annealing* [99] where the severity of changes made to the design that is allowed is gradually reduced as the algorithm progresses. Although Simulated Annealing is based on the random selection of candidate points, it has an intelligent filtering process in which it will only consider those "positive" candidates that drive exploration closer to an optimal solution. To avoid locally optimal solutions, Simulated Annealing will also consider a proportion of "negative" candidates based on the progress of the algorithm. In general, Monte Carlo based approaches are very effective for very large and complex design spaces. On the other hand, they are not very effective in taking advantage of well-defined structure in the design space.

Path-oriented approaches incorporate knowledge of the design space into DSE [100]. Hill-climbing approaches evaluate the neighborhood of the current configuration to determine which neighbors yield the greatest improvement to the design objective. Another path-oriented approach that can be used for DSE is based on genetic algorithms [101].

To help reduce exploration time, pruning techniques can be used that include hierarchical exploration, sub-sampling of the design space, and sub-division of the design space into independent parts. Sub-division decomposes the architecture according to natural spatial boundaries that are assumed to be independent of one another to a first degree. Given this assumption, the total size of the design space becomes:

$$C = \sum_{i=0}^{D} \prod_{j=0}^{n_i - 1} C_{i,j} \qquad (2.6)$$

where D is the number of independent parts, n_i is number of design parameters for the i^{th} part, and $C_{i,j}$ is the number of configurations for the j^{th} design parameter of the i^{th} part. Equation (2.6) relates to (2.5) in that:

$$N = \sum_{i=0}^{D} n_i \, . \qquad (2.7)$$

The total size of the design space is no longer the Cartesian product of design parameters as in Equation (2.5) but a sum of products. Thus, the total size of the design space is reduced significantly. An interesting form of the latter technique is used for ASIPs by Sanghavi et al. [102] of Tensilica and is presumably used to estimate power, area, and speed for their synthesis tools. It is an analytical approach to providing a first-degree decomposition of the design space in both space and time.

Sensitivity analysis [103,104] is another effective way of pruning the design space where all design parameters are treated as independent. In turn, the impact of each design parameter upon the design objective is measured while all other parameters are held constant. The results from this are used to determine the "sensitivity" of the design objective to each parameter. Parameters are then configured one at a time according to an ordering from most sensitive to least sensitive. As a consequence, the effective size of the design space becomes:

$$C = \sum_{i=0}^{N-1} C_i \tag{2.8}$$

Sensitivity prunes the design space significantly such that it is a summation of design parameter settings rather than a product. The disadvantage to sensitivity analysis-based approaches is that they do not adequately address correlations between parameters and so it is often inaccurate.

The Spacewalker algorithm used for PICO [36,105] exemplifies a complete algorithm that incorporates path-oriented DSE with several pruning techniques. More explicitly, Spacewalker uses a hill-climbing approach to find Pareto-optimal solutions. If the design space is too large, then it uses a manifold strategy to search only those points that are likely Pareto-optimal. Further, decomposition is used to separate parts of the architecture and then a path-oriented search is applied to each part. Results from each search are combined to form a system design space which is then searched to find a system solution.

Evaluation tools specific to ASIPs can also be sped up through hybrid simulation techniques such as [106] which combines instruction-level techniques and macro-modeling techniques for the base architecture and custom instructions. Trace-driven simulation provides a method for reducing the number of simulations needed during space exploration [107]. This approach collects memory access sequences from an initial simulation and uses them when subsequent configurations are evaluated.

2.6.2. Instruction Set Extensions

Instruction-Set Extensions (ISEs) are a specific instance of DSE where custom instructions are generated to improve processor performance. As an example, one custom instruction created through fusion might be the combination of two ADD operations followed by a multiply which is then fed into an inverter. Instruction-Set Extensions using fusion can be broken into three parts: *pattern enumeration*, *pattern selection (instruction selection)*, and *pattern matching*. During pattern enumeration, a list of candidate complex instructions is generated such that all candidates adhere to the microarchitectural constraints imposed by the processor architecture. A subset of these is chosen during the Pattern Selection phase based on a set of cost functions. Once a group of patterns has been identified, pattern matching then maps new instructions throughout the application.

Several approaches have been proposed to perform Instruction Enumeration and Instruction Selection [29,108,109,110,111,112]. All of these are significant contributions towards the ASIP flow; however, they do not address the effects of pipeline stalls due to data hazards between functional units (FUs) of the processor. This motivates some of the proposed work described in this thesis. Like many algorithms in ASIP and compiler research, the methods used in previous work operate on intermediate representations of the application by adding, removing, and clustering operations. All algorithms of this type would benefit from the addition of a "hazard-aware" performance predictor for architectures with pipeline data hazards to reduce overall run-time of the application.

The authors of [29] and [113] assume an ideal pipeline because predicting performance would otherwise be difficult. The authors of [29] state that "it is not trivial to compute the total latency" of the application with data hazards. The authors of [113] claims that data hazards need not be considered in the presence of functional unit to functional unit (FU-to-FU) forwarding[2]. As stated above, FU-to-FU forwarding is a possible solution; however, it would require new multiplexers to be added to all new FUs. This might be practical for a small number of custom FUs but it would likely add significant power and performance penalties with large numbers of FUs. With a trend towards increased number of FUs, it is important to evaluate the impact of data hazards on instruction fusion.

Thus far, there is little work that considers power when fusing instructions. Sun et al. [114] develop a cost function to reduce speed and then quantify the power saved as a biproduct of the reduced run-time. They claim to improve performance by an average of 3.4X, energy by an average of 3.2X, and energy-delay by an average of 12.6X. This work serves as an example of the significant gains possible with application-specific configurability; however, it uses an ad-hoc methodology that does not fully explore its design space.

This dissertation will build upon the ideas presented in this chapter to improve the speed and automation of processor customizability and extensibility. An ASIP flow will be developed that incorporates a novel DSE approach that will drastically reduce the number of simulations required for architecture optimization. This will be used to

[2] FU-to-FU forwarding is the forwarding of output from a functional unit to a previous stage of the pipeline in order to avoid a data hazard.

configure the BPU and the cache hierarchy of an embedded processor. Further, the capabilities of ISE configuration will be improved by speeding up the evaluation of candidate instructions using a novel performance prediction approach. All of the proposed approaches will be used to improve execution time (and thus ILP) and reduce power, thus saving energy.

Chapter 3: Optimization via Design Space Modeling

3.1. Introduction

The previous chapter introduced the design space exploration (DSE) problem and discussed the difficulties associated with its run-time complexity. It further discussed how DSE run-time can be reduced through better exploration and through faster point evaluations. In addition, some of the known approaches for DSE including both path-oriented and random-based methods were described. It was shown that DSE is a key bottleneck in the automatic configuration of ASIPs [115].

In this chapter, a new paradigm for DSE is introduced [116, 117]. This paradigm is based on a predictive model of the design space using statistical methods. It is much faster than simulation and it helps users to expose the tradeoffs between different parameters in different regions of the design space. Requiring only the evaluation of a small sample of the design points, all other points in the design space can be estimated through *DSE modeling*. The model must be accurate enough to predict changes in performance due to architectural changes.

To illustrate how *Design Space Modeling* could be used in a hypothetical problem, consider the mesh plot shown in Figure 3.1. The x-axis and y-axis represent two parameters, and the z-axis represents the cost function of a DSE problem. The domains or neighborhoods for hypothetical one-dimensional and two-dimensional problems are shown as a line and a rectangle on the contour map, respectively. Shown on the mesh plot are the corresponding predictive models of the design space. Each model was constructed from a sample of points known through simulation which are represented as "dots". From these models, all other points within their domain can be estimated.

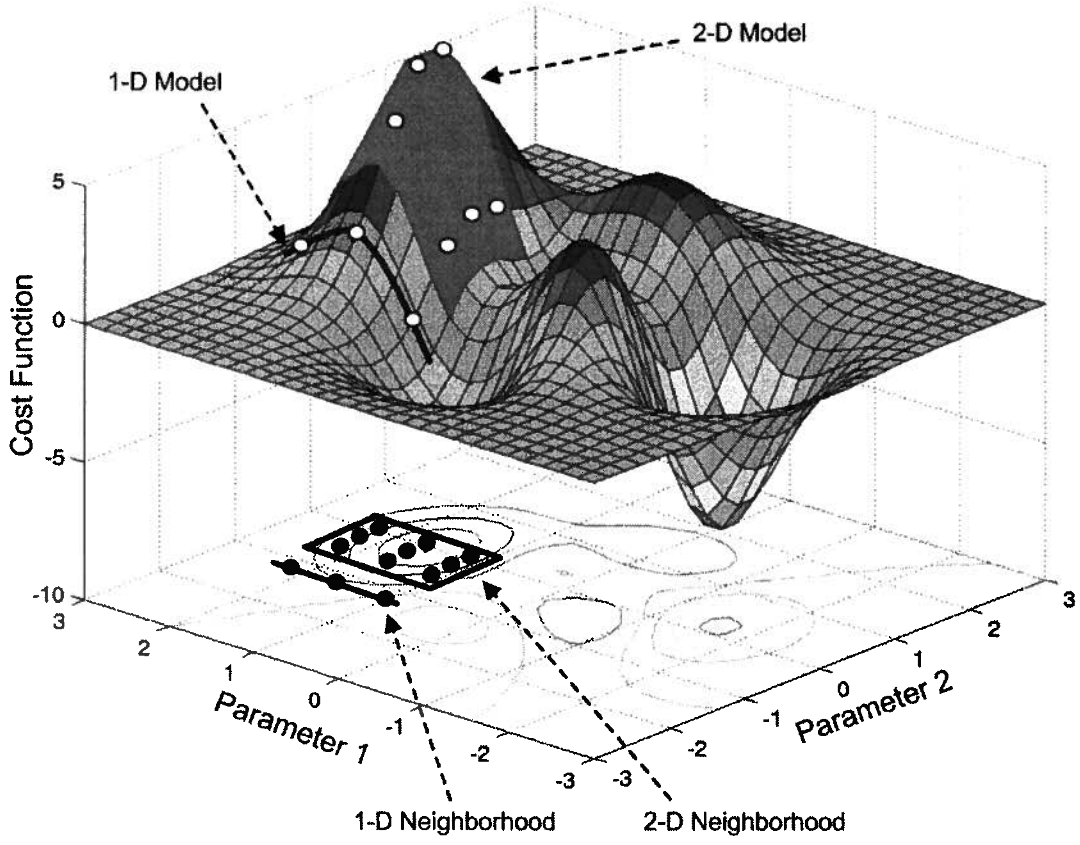


Figure 3.1: Hypothetical 1-D and 2-D design space models.

The speedup achievable by design space modeling over exhaustive methods is dependent on the desired level of accuracy and the number of dimensions. For example, the 1-dimensional neighborhood from Figure 3.1 has 3 known points. If this neighborhood were used to estimate 7 other points out of 10 total points, then there would be a $10/3 \cong 3X$ speedup. For the 2-dimensional problem, there would be 9 sampled points out of 100 total points thus the speedup would be $100/9 \cong 11X$. The speedup in four dimensions would be two orders of magnitude. Although the speedup expressed in this manner is somewhat unrealistic (since exhaustive exploration is assumed), the key idea is that the cost of each estimated point is extremely cheap compared to the cost of an evaluated point.

The rest of this chapter describes the details of Design Space Modeling for ASIP optimization. Section 3.2 introduces the relationship of design space modeling to existing approaches. Section 3.3 outlines the major steps involved in design space modeling. Section 3.4 discusses several approaches for constructing design space models which have been developed concurrently. Of these approaches, two are contributions of this dissertation (*Manual Decomposition Approach* and *Automatic Non-parametric Regression Approach*). Section 3.5 provides a simple example of the Manual Decomposition Approach applied to branch prediction.

3.2. Relationship to Previous Methods

3.2.1. Spatial versus Temporal Sampling

The two multiplicative sources for long DSE run-times are the relatively long simulations times needed to evaluate each point in the design space and the exponentially large number of points in the space. Prior efforts for improving DSE run-time have focused on *temporal sampling* which obtain samples of instruction traces in the time domain, reducing the costs per simulation by reducing the size of simulator inputs. Eeckhout et al. [118] and Nusbaum and Smith [119] study profiling techniques to simplify workloads during simulation. This is accomplished by constructing smaller, synthetic benchmarks with similar characteristics. Sherwood et al. [120] propose a method of reducing the simulator input by identifying representative simulation points within the instruction trace. In all cases, simulations are effectively compressed in the time domain.

Design Space Modeling tackles the DSE problem via *spatial sampling* which is orthogonal to temporal sampling. By sampling only a subset of the points within the design space, it helps mitigate the intractability and inefficiency of traditional techniques that sweep design parameter values and exhaustively simulates all points within a constrained space. Effectively, simulations are compressed in the spatial domain. Spatial and temporal sampling can be combined to further reduce DSE run-time as shown in Figure 3.2. Starting at the top of this figure, the instruction trace is first sampled temporally to form a compressed instruction trace. This trace is then used for simulations of architecture configuration points sampled spatially from the design space. These simulated values are then used to construct the design space model as shown at the bottom of the figure.

3.2.2. Relationship to Heuristic Approaches

Design space modeling differs from heuristic search algorithms in terms of scope and use. With respect to being a pure optimization algorithm, design space modeling serves as an alternative to heuristics such as hill-climbing. The comparative speed of these algorithms is highly dependent on the nature of the application and the design space. To date, a quantitative comparison between design space modeling and heuristics such as those in [121] have not been conducted. Unfortunately, it is not trivial to make a "fair" comparison against the proposed Design Space Exploration approach due to the unavailability of heuristic implementations suitable for comparison. Heuristic implementations require significant customization and tuning for a given problem and therefore it is unlikely to find one that matches the specific architecture, application and experimental conditions of the problem outlined in this dissertation. Alternatively, a heuristic implementation could be implemented and optimized from scratch; however, this would be a relatively difficult and lengthy undertaking. Such a comparison was not made in this dissertation.

Qualitatively, a modeling approach has the following capabilities beyond that of heuristic algorithms:

- It generates predictions for all points in the design space. Unlike heuristics, design space modeling can be queried to explore different tradeoffs amongst parameters in different regions of the design space.
- It can provide a tunable knob for controlling the tradeoffs between different objectives. Unlike heuristics, the modeling can be adapted immediately and without simulations for a varying cost function.
- It can quickly verify that a novel architecture feature is not a consequence of other parameters chosen.
- It is not as susceptible to noise as are heuristic approaches. A heuristic approach, such as hill-climbing, can be negatively impacted by noise differences between neighboring points. This is true because the algorithm makes decisions based on only a small sample of local points. Design Space Modeling, however, is primarily concerned with high level trends; model construction is generally based on sample points spread over a large proportion of the space. In doing so, the model filters out much of the fine-grain noise.

Each of these capabilities would be a significant addition to any automated tool used to configure an ASIP.

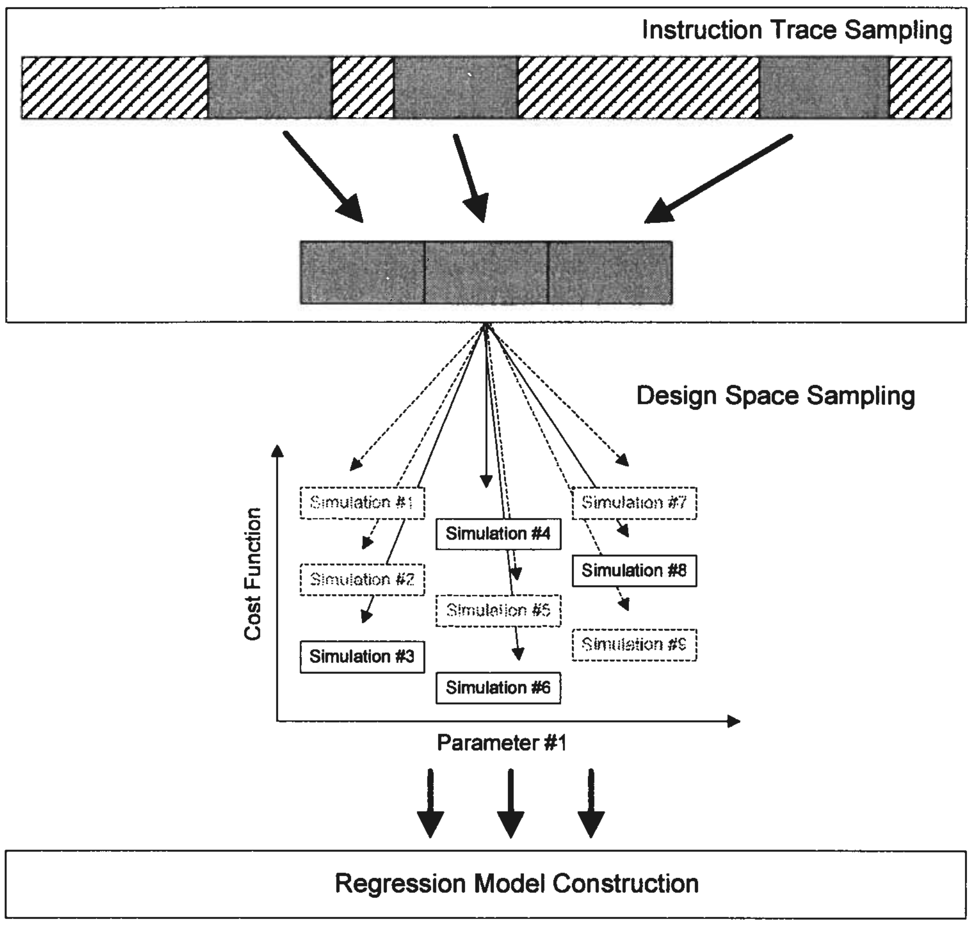


Figure 3.2: DSE using an approach with combined spatial and temporal sampling (adapted from [122]).

It is possible to combine modeling with heuristic methods. One approach would be to use a rough design space model to serve as a "map" to help guide a path-oriented heuristic. Another approach would be to use design space modeling as a method of evaluating an extended neighborhood during hill-climbing rather than just using the immediate neighbors. Implementing these approaches would not be trivial and would therefore make for interesting future research.

3.3. Modeling Paradigm

The proposed paradigm for design space modeling includes the following steps: *Design Space Definition*, *Spatial Sampling Specification*, *Model Construction*, and *Optimization*. Each of these steps is discussed in further detail in the following sections.

3.3.1. Design Space Definition

The design space must be inclusively defined as described by the Mescal Methodology in Section 2.2.3. Thus, a broad set of parameters should be included to increase the likelihood of discovering optimal configurations. This stage can include appropriate pruning of the design space based on micro-architectural and high-level constraints (i.e. area, power, and delay) so that simulations are not carried out on unnecessary or non-feasible solutions.

3.3.2. Methods of Sampling Points

As discussed earlier, simulations are effectively compressed in the spatial domain via spatial sampling. Several sampling policies are possible as follows:

1) Evenly Distributed: Perhaps the most natural sampling policy is to sample equally for all dimensions of the design space and to evenly space the points over the range of permissible values for each dimension. In doing so, this approach does not bias any given dimension or any part of the range within each dimension. The advantage of this approach is that requires little knowledge of the landscape *a priori*.

The disadvantage is that it may require course sampling granularity. If, for example, there are 3 samples per dimension for a 5-dimensional space then the total number of samples is $3^5=243$. Once the number of samples per dimension is specified, the required number of points is automatically established. If the sampling rate were reduced to 2 samples per dimension or 5 samples per dimension then the total number of samples would be 32 and 1024, respectively.

2) Random Selection: This sampling policy is based on a random selection with uniform probability over all dimensions and throughout the entire range of values in each dimension. In doing so, no bias is placed towards any part of the design space regardless of the application. Similar to the evenly distributed policy, random selection does not require knowledge of the design space *a priori*. The advantage of this approach is that any number of additional points can be added to the total number of sampled points.

3) Regional Sampling: This approach places increased emphasis on particular regions or dimensions of interest. Selection of which regions or dimensions to be emphasized is based on domain-specific knowledge or on knowledge gained through previous runs during an iterative approach. Regional sampling could be used in conjunction with an evenly distributed selection or a random selection policy.

4) Weighted Sampling: Similar to regional sampling, weighted sampling places emphasis on particular regions or dimensions of the design space during model training. However, it differs from regional sampling in that regions are emphasized by placing extra weight for points in that region as opposed to sampling more points. Of course, this policy is only applicable to modeling methods where the formulation of the model can use weights [122].

5) Adaptive: This approach adjusts the granularity based on model error variances for each sampled point. Samples with larger variances are likely to be poorly predicted; including such samples for model training will likely improve model accuracy. For this policy, samples are iteratively added to the training set based on their measured variance and how much they differ from previously chosen samples [123].

The latter three sampling policies can improve the quality of the design space model; however, they are more complex and computationally expensive. Regional sampling and weighted sampling require that the Euclidean distance between each candidate point and all other points be found. While evenly distributed selection and random selection are inherently parallel, adaptive selection introduces a feedback component that limits parallelism [122].

3.3.3. Model Construction

Once the sample set is created, the design space model is constructed from the sampling set and can be done using one of the following four approaches: *Manual Regression Approach*, *Artificial Neural Networks Approach, Manual Decomposition Approach*, and *Non-Parametric Regression Approach.* Each of these approaches will be discussed in further detail in Section 3.4.

An in-depth model built manually benefits from several steps of analysis [122]. One or more of these analysis steps can be incorporated into each approach depending on the desired level of model accuracy versus automation:

1) Hierarchical Clustering: Clustering examines the correlation between different parameters. Parameters that are highly correlated to others can be pruned from the DSE problem, thus reducing model size.

2) Association Analysis: Quantitatively, the trends between parameters and the cost function can be captured to reveal the degree of non-monotonicity and nonlinearity. This can be used to determine how the model will estimate the relationship between the cost function and each dimension. Further, parameters could be pruned if it is found that they are insignificant and do not greatly impact the cost function.

3) Correlation Analysis: Correlations can be used to dictate the choice of nonlinear transformation required for the relationship between the parameter and the cost function.

4) Model Specification: Domain-specific knowledge is used to dictate the relationships between parameters and the cost function and between parameters. Based on the results from correlation analysis, the relative flexibility of nonlinear functions can be specified.

5) Assessing Fit: A statistic such as R^2 can be used to determine the overall fit of a design space model.

3.3.4. Optimization

Once the design space model has been constructed, it can provide highly-accurate estimates of the cost function for all points throughout the design space. The prominent use for this model is to quickly find the optimum point within the design space. If the design space represents the set of all ASIP architecture configurations and the cost function is the overall power dissipation, then the minimum-valued point in the predictive model would specify the optimal power.

While model accuracy is heavily emphasized in this dissertation, it is important to recognize that high levels of accuracy may not be required in order to make good optimization choices. In fact, good configuration choices are possible as long as important characteristics of the curve of the design space and the model are the same. Thus, one model construction approach that provides a less accurate model in order to reduce construction time may make better optimization choices than one with an equal or more accurate model. This would be true if the less accurate model was able to better capture the important trends of the design space.

Other uses for design space models include Pareto frontier analysis, design space characterization, and parameter sensitivity analysis [124].

3.4. Construction Approaches

This section describes the following four methods for design space modeling: *Manual Regression Modeling*, *Automatic Learning Modeling*, *Manual Decomposition Approach,* and *Automatic Non-Parametric Regression Modeling.* The first two methods were developed by two other research groups concurrently with this research [125,126], while the latter two are key contributions of this dissertation.

3.4.1. Manual Regression Approach

Lee et al. [125] use in-depth statistical analysis to manually build a regression model of the design space and then uses statistical inference to estimate all other points. To construct the model, all aspects described in Section 3.3.3 are employed. Hierarchical clustering is an iterative approach that amalgamates parameters based on the correlation coefficient. Association analysis is performed using scatter plot analysis, and correlation analysis is based on a rank-based measure of correlation called Spearman ρ^2. All of these approaches require some level of manual analysis.

A general class of models is used to represent the design space where the response (cost function) is a weighted sum of functions $g_j(\cdot)$ plus random noise. Thus,

$$\begin{aligned} f(y_i) &= \beta g(x_i) + e_i \\ &= \beta_0 + \sum_{j=1}^{p} \beta_j g_j(x_{i,j}) + e_i \end{aligned} \tag{3.1}$$

where $x_i=x_{i,1},\ldots,x_{i,p}$ denote the p predictors (architecture parameters) for a response variable, y_i. $\beta=\beta_0,\ldots,\beta_p$ denotes the set of regression coefficients and e_i is the error and is an independent random variable with zero mean and constant variance. $f(\cdot)$ and $g(\cdot)=g_1(\cdot),\ldots,g_p(\cdot)$ are transformations that can be applied to the response and predictors, respectively, to stabilize non-constant error variance or to account for nonlinear correlations between the response and the predictors. Sometimes the interaction between the response, y, and a predictor, x_1, can be correlated by another predictor, x_2. In this case, a third predictor, $x_3=x_1x_2$, can be introduced such that,

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_1 x_2 + e \tag{3.2}$$

The coefficients of the regression are solved using the method of least-squares.

As determined by association analysis, not all predictor-response relationships can be adequately modeled using a linear function. More advanced techniques are implemented including the above-mentioned transformations and parametric spline curves. Spline curves are an alternative to polynomial regressions which can result in undesirable peaks and valleys. Further, a good polynomial fit in one region of the predictor's range can affect the fit in other parts. A manual analysis is required to determine whether or not each predictor-response relationship should be modeled using a linear function or a spline.

To construct a spline, the function is divided into intervals defining multiple different continuous polynomials with end-points called *knots*. The number of knots can vary and determines the quality of fit. Lee et al. [127] found restricted cubic splines more appropriate than linear splines.

Because this approach requires advanced knowledge of statistics and manual effort to construct the model, it runs counter to the motivation of this dissertation which is towards a completely automated solution for ASIP configuration [127]. On the positive side, the model is highly accurate and the relationships between the response and predictors are highly transparent. This approach is suitable for general-purpose processor development where accuracy is more important than the cost of human resources, but not for use in an automatic ASIP configuration tool.

3.4.2. Artificial Neural Networks Approach

İpek et al. [126] use artificial neural networks (ANN) that sample points in the design space to train the network to model the function describing predictor-response relationships. ANN is a machine-learning model capable of modeling complex nonlinear relationships and is relatively resilient to noise.

Figure 3.3 illustrates a simple *fully-connected*, *feed-forward* ANN with an input layer, hidden layer, and an output layer. Predictions are made at the output layer as a weighted contribution of the inputs through the hidden layer which implements an *activation function*. According to [126], the activation function must be nonlinear, monotonic, and differentiable using a sigmoid function. More explicitly, outputs are calculated from the sigmoid function (output layer), as follows:

$$o = \sigma(net) = \frac{1}{1+e^{-net}} \tag{3.3}$$

where,

$$net = \sum_{i=0}^{n} w_i x_i \tag{3.4}$$

such that w_i is the weight for the i^{th} input x_i of n inputs.

İpek et al. [126] update the weights in Equation (3.4) using back-propagation. A gradient descent approach in a weighted space is use to minimize the squared error between simulation results and model predictions. An iterative approach gradually improves the quality of the model with each training input and increases its nonlinearity. The weight update function also incorporates a momentum term to avoid local minima. The sampling policy used for by İpek et al. is adaptive.

The ANN-based approach addresses all steps of model construction as specified in Section 3.3.3. Hierarchical clustering was not originally used in [126]; however, it may be completed once before the automatic configuration tool based on ANN is deployed to users. Association analysis and correlation analysis are not necessary with the ANN-based approach since non-monotonicities and nonlinearities are addressed automatically through neural network learning. In the model specification step, relationships between the response and the predictors are automatically determined through learning. Predictor-predictor relationships can, to a limited extent, be automatically captured by ANN; however, this is not done transparently.

This approach is very effective and is fully automated. Similar to the manual regression approach, it models the response as a nonlinear weighted sum of predictors. Further, it is relatively resilient to noise and does not suffer from "over-fitting" as would a polynomial fit. On the downside, the ANN-based approach is not as computationally efficient as the manual regression-based approach [127]. Further, it does not scale well to problems with many variables that have complex interactions. To some extent, the ANN-based approach is able to capture these interactions; however, this is not done transparently. Thus, it is difficult to predict how well ANN would scale to increased predictor interactions.

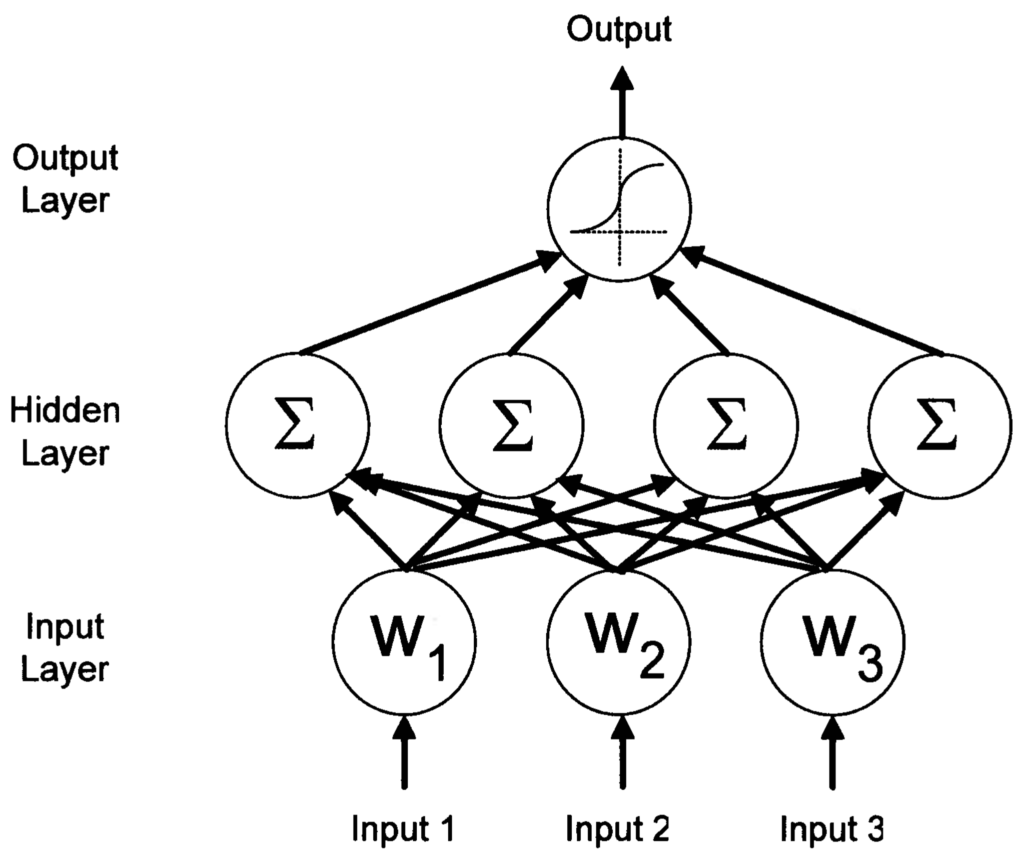


Figure 3.3: A typical artificial neural network.

3.4.3. Manual Decomposition Approach

Manual Decomposition is a first attempt at modeling the design space and represents the straightforward approach based on polynomial regressions [128]. The basic idea is to characterize the behavior of the response function using a linear, quadratic, or higher-order polynomial. Regression methods are used to determine the suitable order of the polynomial. Based on the order of the method, a corresponding number of points are evaluated in the solution space. The goal is to minimize the number of points evaluated so a low-order polynomial is preferred. Unfortunately, the accuracy of the predictive model suffers as a result.

To improve accuracy, a decomposition approach similar to that proposed by Sanghavi et al. [102] of Tensilica can be used. If low-order polynomials can be used on the decomposed functions, then only a few points need to be evaluated in the solution space, thus limiting the number of samples required. This approach is best described in the context of a simple example as presented in Section 3.5. The main advantage of this approach is its low run-time complexity and its simplicity. However, it suffers from being a time-consuming and error-prone manual modeling process.

3.4.4. Non-Parametric Regression Approach

A key contribution in this dissertation is the use of non-parametric regression schemes for model construction in design space exploration for ASIPs [116,117]. Rather than using parametric regressions to form a design space model as with the *Manual Decomposition* approach, it is proposed that a *local regression statistic (LOESS)* [129] be used. Standard regressions require that a class of functions be defined *a priori*; however, this is generally not possible for all many DSE problems. Non-parametric regression statistics solve this problem in that they do not need to be parameterized; one need not specify whether the overall fit will be linear, quadratic, cubic, etc. Instead, the estimated points indirectly define the function. Collectively, design space modeling combined with a LOESS-based construction is given the name *Local Regression Modeling-based Design Space Exploration (LRM-DSE).* A detailed description of LOESS and LRM-DSE can be found in Section 4.2.

To illustrate the approach, consider the two plots in Figure 3.4 where x is the predictor variable and $\mu(x)$ is the response variable. The large dots indicate the known points that were evaluated at great cost. A parametric linear regression would produce the line shown in Figure 3.4(a) whereas the actual solution is quadratic in nature. Figure 3.4(b) computes the desired intermediate points using the known points and the LOESS technique (depicted conceptually as 4 linear regressions). This approach is better able to capture the characteristics of the actual behavior thus improving our ability to predict optimal configurations. A more detailed discussion of this approach appears in Chapter 4.

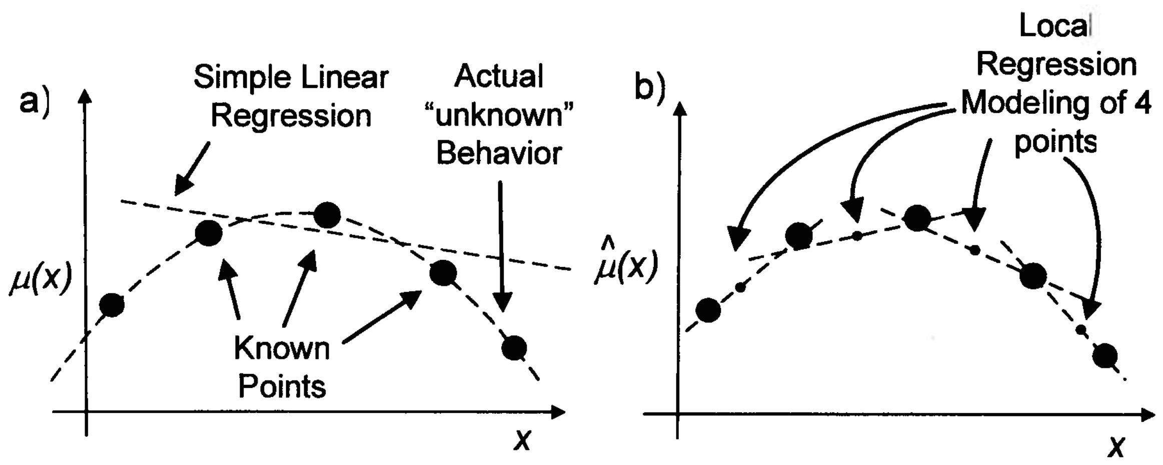


Figure 3.4: (a) Simple Parametric Linear Regression (b) LOESS-based approach.

There are other methods of non-parametric regression; however, the local regression method was found to be the best fit in terms of flexibility, computational complexity, and availability of tools. Although the results are not presented here, experimentations were conducted to assess the suitability of local regressions against local smoothing. Local smoothing is similar to local regressions with the caveat that smoothing results in increased bias at the end points of the neighborhood. This is especially problematic for our application where we have relatively few known points.

Table 3.1: Comparison of various model construction approaches.

	Manual Decomposition (Section 3.5)	Manual Regression [125]	ANN [126]	LRM-DSE (Chapter 4)
Automated	No	No	Yes	Yes
Algorithm Complexity	Fast	Fast	Slow	Fast
Model Accuracy	Good	Very Good	Very Good	Good
Response-Predictor Transparency	Very Good	Very Good	Poor	Good
Sampling Policy	Evenly Distributed	Random	Adaptive	Evenly Distributed / Random
Construction Approach	Regression with Decomposition	Regressions with Spline	Neural Networks	Non-parametric Regressions
Application	GPP Development	GPP Development	ASIP Configuration Tool	ASIP Configuration Tool

LRM-DSE addresses all steps of model construction as specified in Section 3.3.3. Like the ANN-based approach, hierarchical clustering can be completed once before deploying the automatic configuration tool to customers. Association analysis and correlation analysis are not necessary since non-monotonicities and nonlinearities are addressed automatically through the non-parametric regression. In the model specification step, relationships between the response and the predictors are automatically determined through a non-parametric fit to the sample data. Predictor-predictor relationships can, to a limited extent, be automatically captured by LOESS; however, improved results could be attained by adding predictor interaction terms to the regression model.

A comparison of the various methods is shown in Table 3.1. The non-parametric approach shares advantages with both the manual regression approach and the ANN-based approach. Similar to the ANN-based approach, LRM-DSE is fully automated, thus it is suitable for use in an automatic ASIP configuration tool. Similar to the manual regression approach, LRM-DSE is based on solving linear systems of equations to perform regression. The ANN-based approach differs in that the edge weights must be iteratively refined to reduce error with each iteration. The time of convergence is usually larger than that required to numerically solve least squares for regression, especially if the predictor count is small [127]. Response-predictor transparency is higher for the regression-based approaches than the ANN-based approach. The most transparent approach is the manual regression approach because of the in-depth analysis required.

3.5. Example: Manual Decomposition Applied to the BPU

3.5.1. Overview

We now apply the concepts described in the prior sections to a concrete example of optimizing the *branch prediction unit (BPU)* of a processor for power [128]. This example is small enough to illustrate many of the concepts but complex enough to gain insight into the key issues. The model construction approach used in this example is *Manual Decomposition* for which that the response-predictor function is manually decomposed and then assessed through a series of parametric regressions. Once the model is completed, only a few points must be evaluated to capture the behavior in a solution space. The main advantage of this approach is its low run-time complexity and simplicity. However, it is a manual process which must be repeated whenever the parameters or response functions are modified. This approach is also important in that it motivates the automated approach discussed in Chapter 4.

3.5.2. Branch Prediction

Branch prediction has long been an important area of study for high-performance processors because it has a significant impact on the attainable instruction level parallelism (ILP). As processors become wider and pipelines become deeper, the penalty of a misprediction grows. To offset this, branch predictors have become more complex with each generation of processor design. As a consequence, branch predictors may account for a significant portion of the total power dissipation (more than 10% of the total processor in some cases [130]).

There are many ways in which the branch prediction unit (BPU) can be configured to reduce power dissipation for a specific application. One approach is to automatically choose a low-power branch prediction policy. While larger, more complex, branch predictors dissipate more power, they have better prediction rates so the processor takes fewer cycles to execute the application thus resulting in less overall power dissipation. This approach is not taken here. Instead, the branch predictor is configured by sizing its *pattern history table* (PHT) based on the application. The PHT is a significant part of the BPU as it is a lookup table that stores history information of recent branches thus providing a way of predicting the outcome of future branches. Sizing the PHT has a significant impact on performance and is a less drastic departure from the base architecture.

An increase in PHT size improves the branch prediction rate thus reducing the number of cycles wasted due to branch prediction penalties. With fewer cycles needed to execute the application, the overall processor will consume less energy. On the other hand, an increase in the size of the PHT will increase the energy consumption of the BPU. This is because the PHT is typically implemented as a standard SRAM cell array so an increase in the number of entries will result in more cells, longer bit lines, and larger decoders. Larger bit line capacitances result in an increase in switching power, and an increase in the number of cells results in more leakage power. The goal here is to configure the PHT size with the correct trade-off between BPU complexity and execution time.

To solve the complete problem, there are many other variables that must be configured in an ASIP to minimize power dissipation. Together, these variables define the solution space. The most accurate way of solving for the optimal point of this solution space is a "brute force" search where every point is evaluated through simulation. However, if there are many variables all with many possible configurations, a "brute force" approach quickly becomes intractable.

Three types of branch predictors are shown in Figure 3.5. For the Bimodal branch predictor [75], PHT entries are indexed by the *m* least significant bits of the branch address. For the GSelect branch predictor [74], PHT entries are indexed by the *m* least significant bits of the branch address concatenated with the *n* least significant bits of the branch history register (BHR). The BHR is a shift register that tracks a history of outcomes for all recent branches. By combining bits from the branch address and the BHR, GSelect bases its prediction partially on the global history of all recent branches and partially on the local history of a specific branch instruction. The Hybrid predictor [131] has both a Bimodal predictor for local predictions and a GSelect predictor for global predictions. A second Bimodal predictor, called the meta-predictor, is used to predict which of these two sub-predictors will provide a more accurate result for the current branch instance.

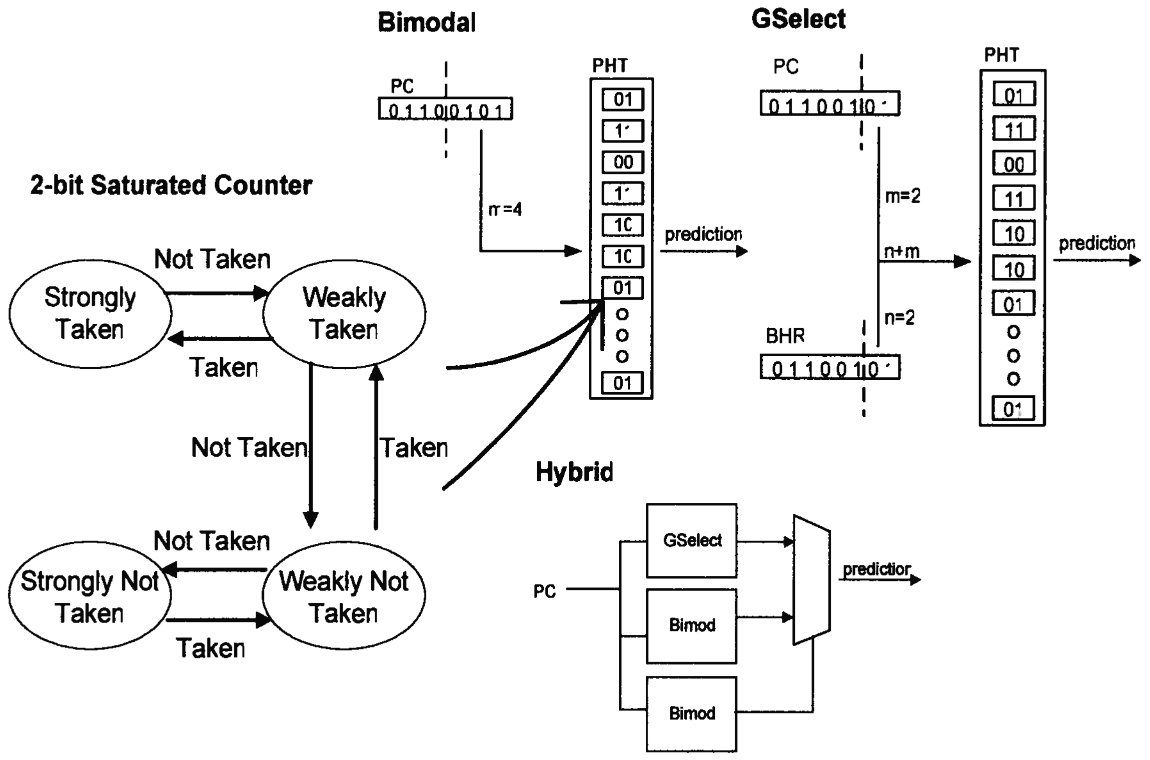


Figure 3.5: The Bimodal, GSelect, and Hybrid Branch Predictors.

Each entry of the PHT is a 2-bit saturated counter [45] that is incremented for branches that are taken and decremented for branches that are not taken. A branch is predicted as taken if this counter is "10" or "11 and as not-taken for "00" or "01". A state machine for the 2-bit counter is also shown in Figure 3.5.

3.5.3. Measuring Aliasing via Simulation

Experiments conducted in [132] found that two-level predictors such as GSelect are close to optimal when implemented with unlimited resources. Due to power, timing, and area constraints, PHTs must allow more than one branch and/or branch history to be mapped to each table entry. This is referred to as *aliasing* or *interference*. When *constructive aliasing* occurs, the BPU correctly predicts the branch direction by "coincidence" whereas *destructive aliasing* predicts the wrong direction [133]. Aliasing has been shown to be the dominating factor affecting branch prediction accuracy [134]. The larger the PHT, the smaller the chance that branches will interfere with one another.

An important assumption of the approach here is that aliasing can be used to predict how the overall power of the processor will be affected by the size of the PHT. First, this assumption is justified by showing that there is a linear relationship between the degree of aliasing measured for a given PHT size and the hit rate of the BPU. Next, a linear relationship is found between the BPU hit rate and the number of cycles of penalty needed to execute the application. Last, the number of cycles of penalty and the overall power dissipation of the processor are found to have a linear relationship. By combining these relationships, aliasing measurements can be used to estimate power and can therefore be used to predict which PHT size will minimize power.

For a given simulation, aliasing statistics can be collected for all possible PHT sizes in parallel. A convenient implementation for this is the binary tree data structure shown in Figure 3.6. The branch address is concatenated with the global branch history and then the *r* least significant bits are used to index the PHT (enclosed in a dashed box). Each row, *r*, of the figure represents a different size of the PHT, $S=2^r$, where $r=0...16$. The nodes of the tree at a given level *r* represent all of the possible table entries for a table of size *S*.

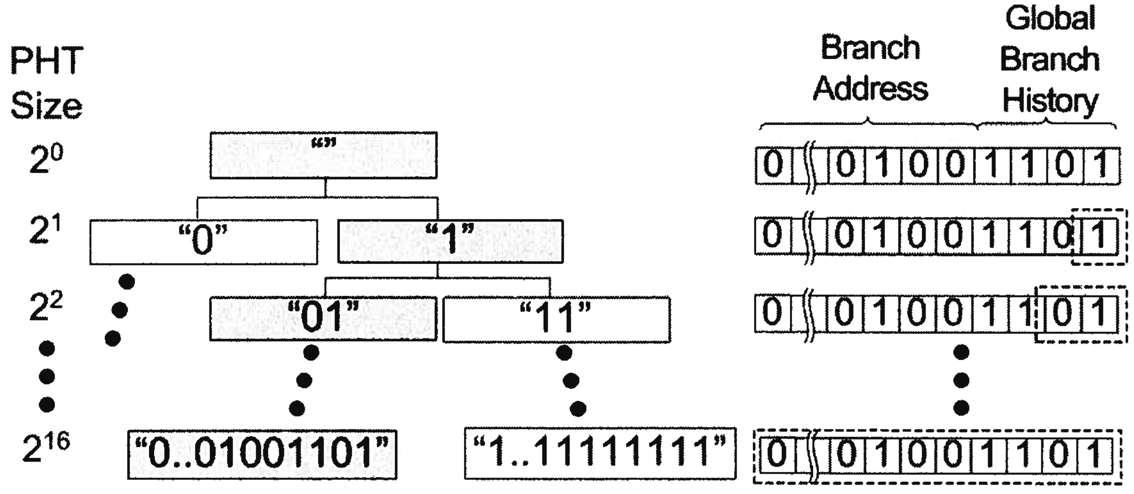


Figure 3.6: Data structure used to collect aliasing statistics for all PHT configurations in parallel.

When a branch is encountered during simulation, the branch address is concatenated with the global branch history to form the *full index*. The binary tree is then traversed from the root to a leaf based on the bits of the *full index* starting from the least significant bit. In this figure, the shaded nodes define the path taken for the index "01001101". As the tree is traversed, statistics are updated for exactly one node per level.

To measure aliasing, each node of the binary tree structure stores the last branch {address, history} pair from which it was mapped. When a new branch is mapped to a node, the current {address, history} pair is compared to that previously stored. If they differ then aliasing has occurred [44]. In addition to storing a {address, history} pair, each node keeps track of the number of times aliasing occurred. This approach for collecting statistics allows us to perform one simulation to simultaneously measure the aliasing rate for each PHT size in parallel by summing the total number of aliasing occurrences across each level (i.e., for each PHT size).

3.5.4. Model Construction

Modeling construction can be carried out using the claim that the aliasing per PHT size can be used to predict the total power dissipation of the processor. To reduce the number of simulations needed to configure the BPU, spatial sampling is used as dictated by the modeling paradigm described in Section 3.3. Although not shown here, experiments were conducted to determine the appropriate spacing of sample points within the design space.

To simplify regression, the response-predictor relationship was manually decomposed in a similar way to that proposed by Sanghavi et al. [102] of Tensilica. This procedure is as follows: First, the aliasing statistics are gathered for all BPU configurations in parallel during a single simulation. Second, the relationship between aliasing and power is decomposed into simpler relationships in order to apply regressions. Third, the combined set of regressions form the design space model which is then used to estimate how changes in the BPU configuration affect the overall power dissipation of the processor. From this model, the optimal PHT size is selected.

Many of the analysis steps described in Section 3.3.3 such as hierarchical clustering, association analysis, and correlation analysis were not necessary for the design space in this example because it had only one response-predictor relationship to model and it was simplified to a set of linear relationships by construction using decomposition. As will be discussed in Section 3.5.6.1, the quality of fit is assessed using the correlation coefficient, R^2.

To predict the success rate of the branch predictor, an expression that relates the hit rate probability to aliasing is derived. The probability of aliasing is ρ_a=*P(aliasing)*. Given that a branch prediction lookup is aliased, constructive aliasing occurs with a probability ρ_{ca}=*P(correct pred.| aliasing)* and destructive aliasing occurs with probability $1-\rho_{ca}$=*P(incorrect pred. | aliasing)*. When a PHT lookup is not aliased, the probability that the 2-bit saturated counter makes a correct prediction is ρ_{cna}=*P(correct pred. | non-aliasing)*. Accounting for the cases with and without aliasing, we use the following expression as the probability of a hit (i.e., hit rate):

$$HR = (1-\rho_a)\cdot\rho_{cna} + \rho_a\cdot\rho_{ca}. \tag{3.5}$$

For an ideal branch predictor with infinite PHT resources, the hit rate is $HR=\rho_{cna}$. When comparing two different processors with finite but different PHT sizes, the following is an expression for the change in hit rate:

$$\begin{aligned}\Delta HR = HR_2 - HR_1 &= (\rho_{ca}-\rho_{cna})\cdot(\rho_{a,2}-\rho_{a,1})\\ &= \beta\cdot(\rho_{a,2}-\rho_{a,1}) = \beta\cdot\Delta\rho_a\end{aligned} \tag{3.6}$$

As stated earlier, only one simulation is needed to collect aliasing statistics for all configurations; however, a minimum of two simulations are needed to find the slope, β, of the hit-ratio versus aliasing curve. Experimental data supported the approximation that β is roughly constant.

To predict the number of cycles, we assume the following relationship between the hit rate and the number of cycles of execution:

$$T = T_{ideal} + N_{CB} \cdot t_p \cdot (1 - HR) \tag{3.7}$$

where T_{ideal} is the number of cycles needed to execute the application when branch prediction is 100% accurate, N_{CB} is the number of conditional branches encountered, t_p is the penalty for a branch misprediction, and HR is the hit rate. Using Equation (3.7), the change in the number of cycles when comparing two different PHT sizes is:

$$\Delta T = T_2 - T_1 = t_p \cdot N_{CB} \cdot (HR_1 - HR_2) = \xi \cdot \Delta HR \tag{3.8}$$

Equation (3.8) relates the change in the number of cycles needed to execute the program (due to branch penalties) to the change in the hit rate. ξ is the slope of this relationship and is approximated to be constant based on observations made from the data.

The change in overall power dissipation of the processor can be divided into the change in power dissipated by the BPU (leakage and switching), ΔP_{BP}, and that dissipated by the rest of the processor, $\Delta P_{T\text{-}BP}$. As the size of the PHT changes, these two components are affected differently. An increase in the PHT size improves the branch prediction rate thus reducing the number of cycles devoted to branch penalties. With fewer cycles needed to execute the program, the entire processor will consume less energy. On the other hand, an increase in the size of the PHT will increase the energy consumption of the BPU even though there is a fixed number of lookups.

The change in the overall power dissipation of the processor can be approximated by:

$$\Delta P = \Delta P_{BP} + \Delta P_{T-BP} = \gamma \cdot \Delta N + \alpha \cdot \Delta T \tag{3.9}$$

where the second term is the change in power dissipation of the entire processor except for the BPU. It can be approximated as a linear function of the change in the number of clock cycles. The first term is the change in power dissipation of the BPU and can be approximated as a linear function of the PHT size, ΔN.

Figure 3.7 shows the four functions found through decomposition and the linearity of the decomposition. If these functions had not exhibited a linear relationship, than a quadratic or cubic could have been used.

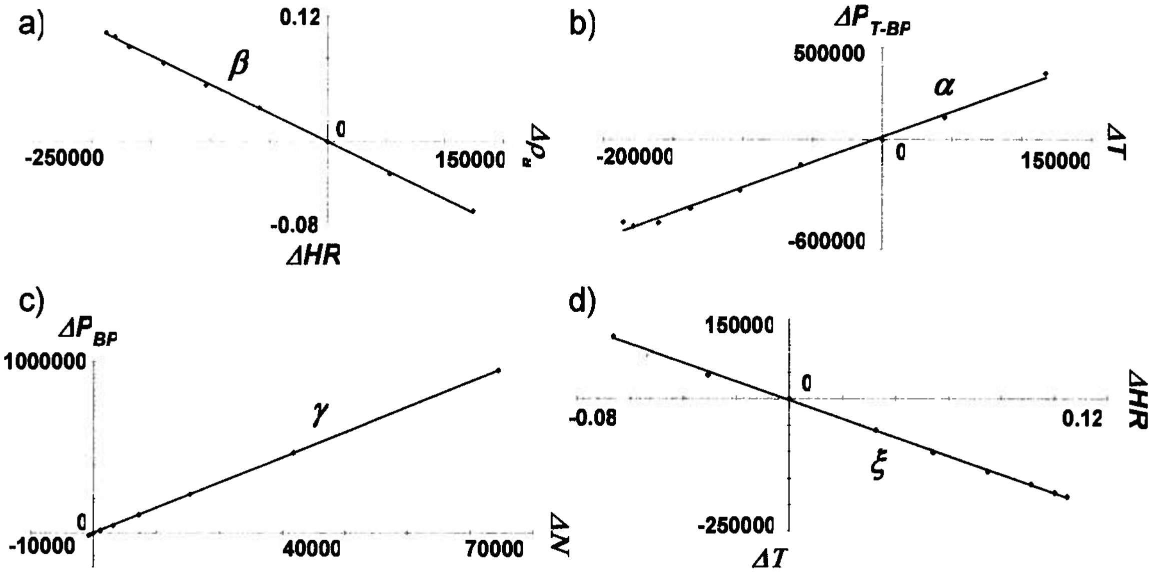


Figure 3.7: Linearity of regression coefficients, α, β, ξ, and γ.

Equations (3.6), (3.8), and (3.9) can be combined to form a complete expression for the change in the total processor power dissipation as a function of aliasing and the PHT size:

$$\Delta P = \Delta P_{BP} + \Delta P_{T-BP} = \gamma \cdot \Delta N + \alpha \cdot \xi \cdot \beta \cdot \Delta \rho_a \tag{3.10}$$

where α, ξ, β, and γ, are slope parameters that have to be measured by simulating the processor with two different configurations. On the other hand, only one simulation is needed to determine $\Delta\rho_a$ for all PHT sizes in parallel. This significantly reduced the number of evaluations needed in the design space. Together, these statistics are combined using Equation (3.6) to estimate the power dissipation of a processor for a particular PHT size.

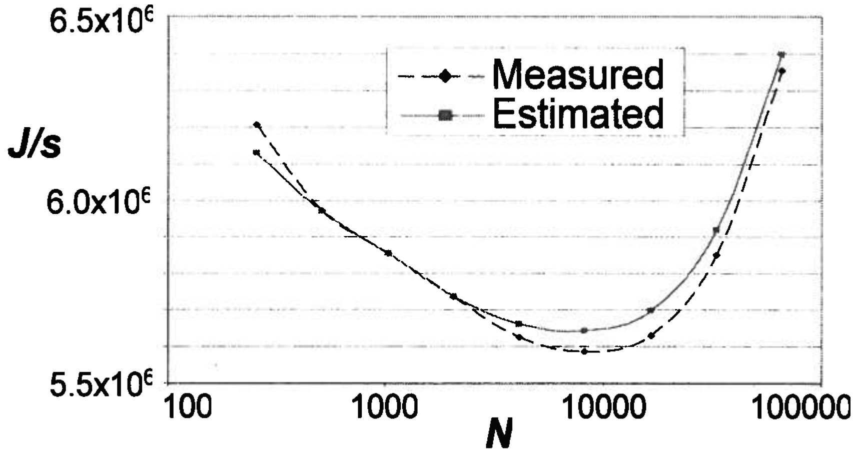


Figure 3.8: Estimated and measured total power dissipation for the *gcc*.

Two curves are shown for the application *gcc* in Figure 3.8. One curve is the estimated total power dissipation (using Equation (3.10)) as a function of the PHT size and the other curve is that measured using Power Analyzer [135] which is a detailed cycle-accurate power estimation tool that models internal switching power, I/O switching power, and leakage power. Power Analyzer adds power estimation functionality onto Simplescalar [136], an instruction-level simulator. This figure shows that the model given by Equation (3.10) is fairly accurate at estimating the total power dissipation of the processor. In this example, a parametric regression-based configuration tool would choose a PHT size of 8192 which coincided with the actual minimum point.

3.5.5. Experimental Platform

The experimental platform used in this chapter is based on the StrongARM architecture [137] as specified in Table 3.2. By default, a Bimodal branch predictor is assumed with a PHT size of 2048. The Bimodal predictor, a 1024-entry GSelect predictor and a Hybrid predictor are studied on the platform. The default Hybrid predictor has sub-predictor sizes of 1024, 2048, and 2048 for the GSelect, Bimodal, and meta-predictor, respectively (based on default values from Power Analyzer [135]).

Table 3.2: Base processor specifications of the StrongARM processor.

Processor Core	
Issue Width:	4 instructions per cycle
Pipeline Length:	5-stage
Functional Units:	
Memory Hierarchy	
L1 Data Cache Size:	128-set, 4-way, 32B blocks
L1 Instruction Cache Size:	512-set, 1-way, 64B blocks
L2:	Unified, 1024-set, 4-way, 64B blocks
Branch Predictor	
Combined Predictor:	Bimod: 2048-entry
	GAg: 1024-entry, 8-bit BHR
BTB:	512-entry, 4-way

Figure 3.9 illustrates the flow used to implement software compilation and hardware configuration. A set of benchmarks drives the overall system. A "C" language description of the application is parsed and optimized by a modified version of the Trimaran compiler infrastructure [138] which then emits ARM v.4 assembly code. This code is then assembled and linked using standard GNU tools [139]. Results from the

local regression tool are used to configure the compiler and the simulation tool through architecture description files. In accordance with Power Analyzer, energy is reported as the sum of individual power dissipation values sampled for all cycles. Units are reported as Joules*cycles/sec.

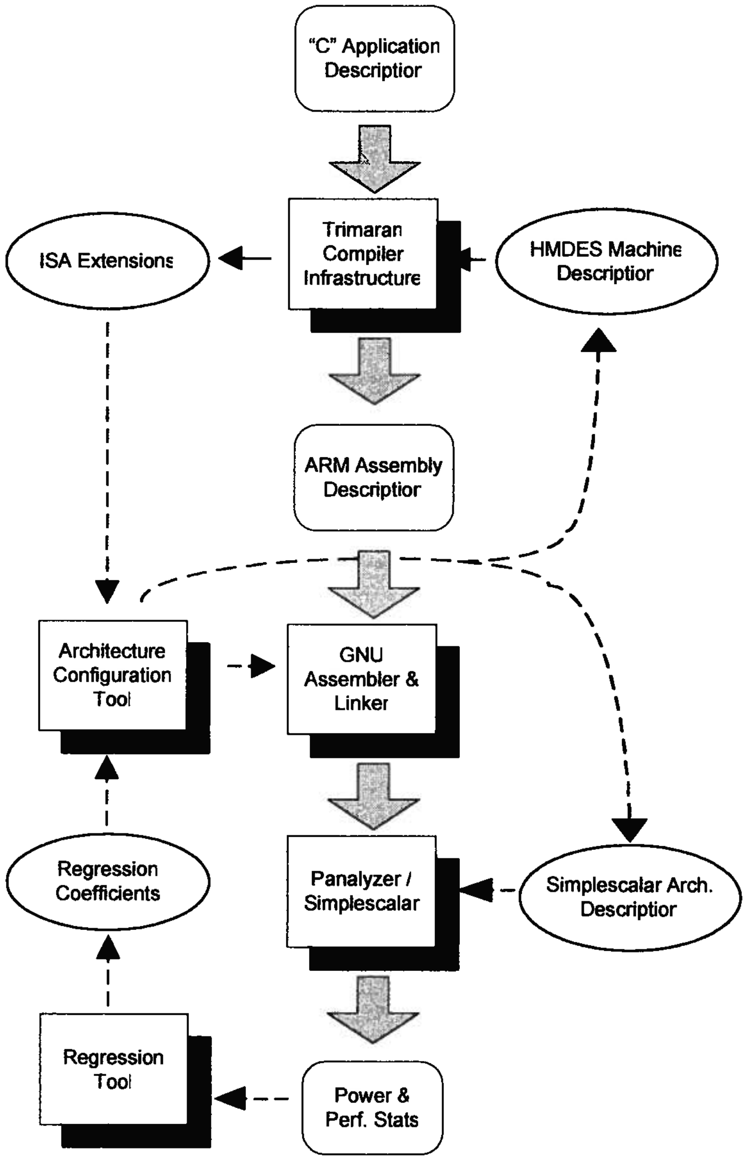

"C" Application Descriptior
ISA Extensions
Trimaran Compiler Infrastructure
HMDES Machine Descriptior
ARM Assembly Descriptior
Architecture Configuration Tool
GNU Assembler & Linker
Regression Coefficients
Panalyzer / Simplescalar
Simplescalar Arch. Descriptior
Regression Tool
Power & Perf. Stats

Figure 3.9: Flow used to run experiments on BPU configuration. Solid arrows represent the sequence of steps used in the standard flow. Dashed arrows represent the additional steps introduced to add configurability.

Benchmark applications are from the SPECcpu2000 Integer Suite [140] and the MediaBench Suite [141]. Both are commonly used benchmark suites where SPECcpu2000 has general-purpose applications and MediaBench has communications and multimedia applications. For SPECcpu2000, only integer applications were used because floating point benchmarks are generally easy to predict and have few dynamic branches. Each benchmark was simulated to a maximum of 50 million instructions which is long enough to saturate the PHT but short enough to make experimenting feasible.

3.5.6. Results

3.5.6.1. Quality of the Linear Relationships

After model construction, it is important to assess the quality of the result. In the case of the BPU model, we must analyze the quality of the assumed linear relationships. The results must validate the assumption or a higher-order method must be used. Using Power Analyzer, simulations were run on all benchmarks over all PHT sizes ranging from 16 to 65535 for the Bimodal and GSelect branch predictors. For each benchmark, "least-squares" linear regression slopes β, ξ, α, and γ were determined based on the statistics gathered for ΔHR versus $\Delta\rho_a$, ΔT versus ΔHR, $\Delta P_{T\text{-}BR}$ versus ΔT, and ΔP_{BR} versus ΔN, respectively. As an indication of the overall quality of fit for each regression, correlation coefficients (R^2) are given in Table 3.3. The closer a value to 1, the higher the quality of fit.

Most regressions in Table 3.3 exhibit a high quality of fit except for *bzip2* and *parser*. According to the first column, *bzip2* and *parser* have the lowest percentage of instructions that are branches and are therefore less likely to saturate the PHT. For these benchmarks, aliasing will likely be low regardless of the size of the PHT. Therefore, the "quality" of the linear relationship between ΔHR and $\Delta\rho_a$ will be dampened and overshadowed by "noise" generated by ρ_{ca} and ρ_{cna}. This inaccuracy has a negative impact on our ability to predict power dissipation. These results suggest that the accuracy of our power prediction methodology is less effective for applications with a low utilization of branch predictor resources.

Table 3.3: The correlation coefficient (R^2) for all least square regressions.

	% Instr are branches	R^2 for β		R^2 for ξ		R^2 for α		R^2 for γ		Avg R^2
		GSelect	Bimodal	GSelect	Bimodal	GSelect	Bimodal	GSelect	Bimodal	
bzip2	7.9	0.781	0.025	0.634	0.438	0.026	0.284	1.000	1.000	0.524
parser	9.6	0.996	0.624	0.985	0.971	0.008	0.728	1.000	1.000	0.789
gzip	9.9	0.984	0.747	0.996	1.000	0.999	0.993	1.000	1.000	0.965
gcc	15.7	0.999	0.998	0.999	0.999	0.994	0.998	1.000	1.000	0.998
string	16.0	0.992	0.983	0.999	0.978	0.986	0.977	1.000	1.000	0.989
twolf	16.3	0.993	0.978	0.996	1.000	0.995	0.993	1.000	1.000	0.994
CRC32	16.6	0.679	0.754	0.976	0.993	0.929	0.101	1.000	1.000	0.804
vortex	18.7	0.982	0.985	0.984	0.998	0.996	0.984	1.000	1.000	0.991
average	13.8	0.926	0.762	0.946	0.922	0.742	0.757	1.000	1.000	0.882

3.5.6.2. Configured PHT

The prediction method was used to configure a processor with Bimodal and GSelect predictors. For each predictor, only two evaluations using Power Analyzer were carried out to estimate the slope parameters β, ξ, α, and γ. At the same time, aliasing was measured for all PHT sizes in parallel. Using the cost function defined by Equation (3.10), the total power dissipation was predicted for all configurations. Each benchmark was then simulated using the configuration with the lowest estimated power as determined by the cost function. Power dissipation results were then compared against that of the default branch predictors. These results are shown in Table 3.4.

Results indicate an average reduction in total power dissipation of 5.4% (maximum of 12.7%) for a processor with GSelect predictor and 0.1% (maximum of 1.3%) for a processor with a Bimodal predictor. A processor with the optimal PHT size results in a 5.8% improvement and 0.7% improvement for the GSelect and Bimodal predictors, respectively. These results suggest that, by configuring the branch predictor using the parametric regression-based cost function, the power reduction is close to that achievable by the "optimal" PHT configuration.

Table 3.4: Power reduction after using the proposed configuration approach for GSelect and Bimodal.

	Percent Power Reduction			
	GSelect		Bimodal	
	Manual Decomposition	Optimal	Manual Decomposition	Optimal
bzip2	-1.8%	0.0%	-0.3%	0.8%
Parser	-0.3%	0.1%	1.3%	1.6%
Gzip	4.3%	5.2%	0.1%	0.3%
Gcc	4.6%	4.6%	-0.1%	0.0%
String	6.3%	6.4%	0.0%	0.4%
Twolf	12.7%	12.7%	0.4%	2.1%
CRC32	9.1%	9.1%	-0.9%	0.0%
Vortex	8.6%	8.6%	0.2%	0.2%
average	5.4%	5.8%	0.1%	0.7%

Table 3.4 indicates that there is power reduction for all benchmarks except for *bzip2* and *parser* which result in a small increase in the power dissipation. This is acceptable to some degree since the "optimal" configuration provided too little or no power reduction for these benchmarks. The effects of this were also evident in the previous section which showed that the miss rate of *bzip2* and *parser* did not "behave well" as a linear function of aliasing. The minimal power reduction of these two benchmarks does not show a deficiency in our approach because there was very little power reduction possible.

3.6. Summary

A novel algorithm was developed to improve DSE. This approach was based on the manual construction of a design space model from a small sample of points within the design space. This model can be used for optimization in an ASIP architecture configuration problem. The net result is a reduction in the time needed to perform DSE because only a subset of the design space needs to be simulated.

To help illustrate many of the concepts of design space modeling, an example design space model was constructed to configure the PHT of a branch prediction unit. The most difficult part of the process was to extract linear relationships between the response function and the measurable quantities so that the number of evaluations could be kept to a minimum. Fortunately, such relationships existed in the power function and worked well. If, on the other hand, a quadratic or cubic regression were required then the model would require more effort to construct. Further, any changes made to either the architecture parameters or the response function would require that the entire analysis be repeated. In any case, results showed that the proposed approach could produce near optimal results while only having to simulate a fraction of the design space.

Chapter 4: Application of Non-parametric Design Space Modeling to Architecture Exploration

4.1. Introduction

In the previous chapter, a composition of linear regressions was manually constructed by identifying basic relationships between an objective function and architecture parameters (PHT of the BPU). Using this approach, the number of design points requiring evaluation was reduced; however, the approach required extensive manual intervention thus making it infeasible for use with more complex design spaces with many dimensions.

In this chapter, a novel DSE approach is proposed that *can model the design space automatically using non-parametric statistics* [116,117]. With only a small subset of the solution space sampled, a model of the design space is constructed using LOESS with interpolation. From this model, statistical inference is used to estimate the values of all other points. This, in turn, is used to perform architecture optimization. This approach is given then name *Local Regression Modeling-based Design Space Exploration (LRM-DSE).* To demonstrate its features, LRM-DSE is used to perform a 5-dimensional DSE representing a two-level cache hierarchy tuning problem.

4.2. LRM-DSE

4.2.1. Overview

To help illustrate how LRM-DSE works in more detail, consider the hypothetical problem in Figure 4.1. In this figure, the x-axis serves as the predictor variable and the y-axis as the response variable for a LOESS regression. Known data points are denoted by large dots.

Figure 4.1(a) illustrates how non-parametric models such as LOESS are defined by the data itself and not by the user. The local regression behavior of the underlying trend is then estimated for each point thus providing both a value and slope estimate. In this figure, value estimates are represented with an "x" and slope estimates are represented by a dashed line.

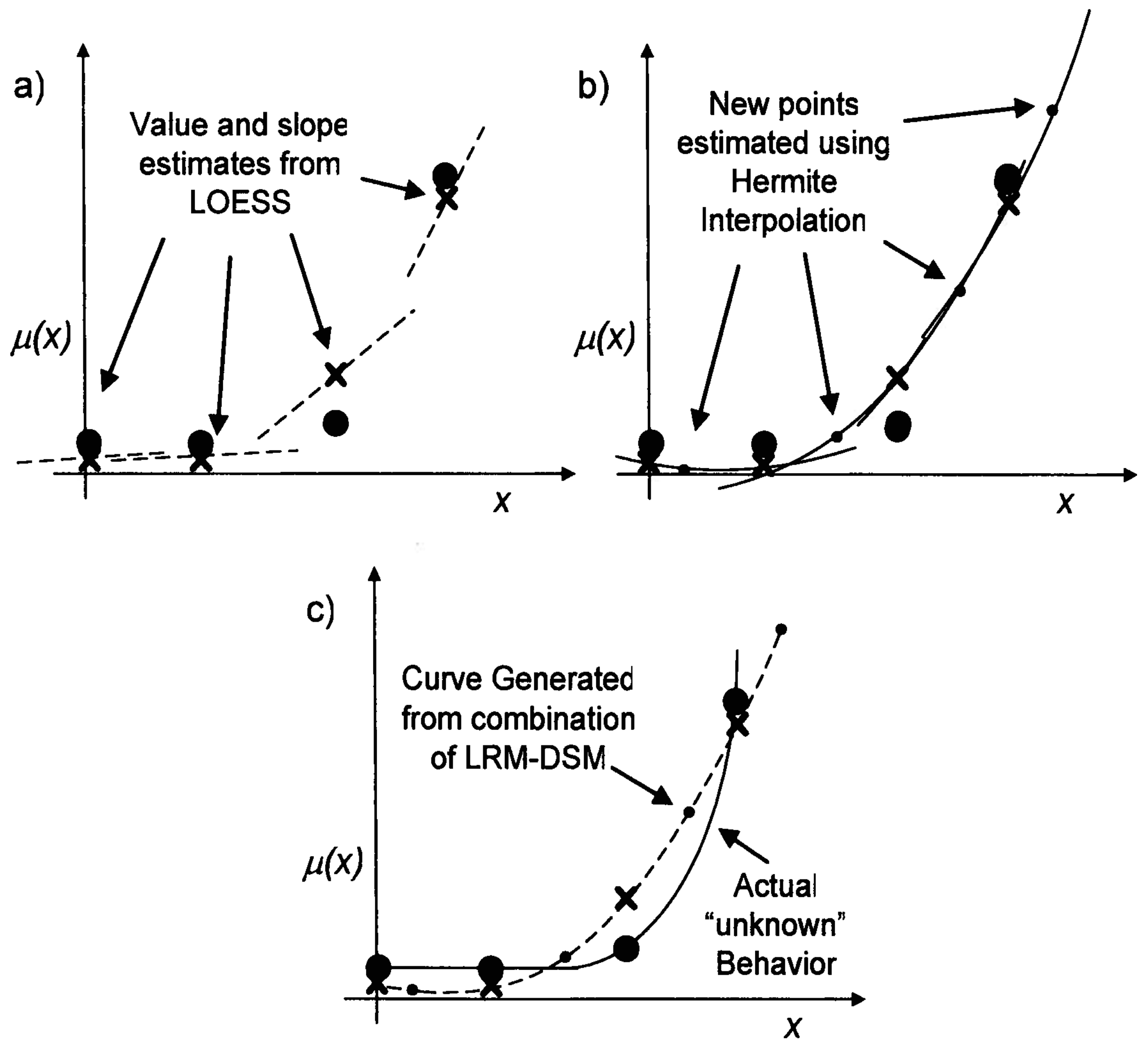

a)
Value and slope estimates from LOESS
μ(x)
x
b)
New points estimated using Hermite Interpolation
μ(x)
x
c)
Curve Generated from combination of LRM-DSM
μ(x)
Actual "unknown" Behavior
x

Figure 4.1: LRM-DSE uses both LOESS and Hermit Interpolation

Once the local behavior has been determined for all known points, a piece-wise cubic Hermite polynomial interpolation can be used to estimate the value of all unknown points (shown as small dots in Figure 4.1(b)). In doing so, the model can be used to predict all unknown points without the need for parameterization.

Now that estimates have been found for all predictor values, a curve can be defined for the entire domain. This curve represents the LRM-DSE model and can now be used to select the optimal point. In Figure 4.1(c), this curve is represented as a dashed line and the actual function is represented by a solid line.

To extend LOESS to multiple dimensions, LRM-DSE uses a *Generalized Additive Model* of the form,

$$Y = b_0 + b_1 * f_1(X_1) + b_2 * f_2(X_2) + \ldots + b_m * f_m(X_m) + e \tag{4.1}$$

where Y is the response variable, X_1 through X_m represent m values for the predictor variables, and b_0 through b_m represent regression coefficients estimated by multiple regression. f_1 through f_m represent unspecified non-parametric functions of the predictor variables [142]. e is the error and is an independent random variable with zero mean and constant variance.

4.2.2. Local Regression Basics

A regression models the dependency between the observations of a response variable, Y, and the values of a predictor variable, x. Normal regression models take the form,

$$Y_i = \mu(x_i) + \varepsilon_i \tag{4.2}$$

such that $\mu(x_i)$ is an unknown function and represents the mean of Y at x_i and ε_i represents the error which is an independent and identically distributed random variable with zero mean and variance $\sigma^2 < \infty$. For a parametric regression, $\mu(x_i)$ is specified to be a member of a parametric family such that its coefficients are the parameters.

LOESS differs from parametric regressions (of Section 3.5) in that $\mu(x_i)$ is determined from a local subset of the data points defined by a window centered at x_i [129]. For a given point x_i (and its corresponding window), we assume that $\mu(x_i)$ is well-behaved and can be locally approximated by a member of a simple parametric class such as a linear or quadratic polynomial. Using local regression, an estimate, $\hat{\mu}(x_i)$, is then calculated using weighted values of Y in its local region. Together, all such $\hat{\mu}(x_i)$ are combined to form the overall estimated function spanning the entire data set. Our ability to model local behavior using a low-order polynomial stems from Taylor's theorem that states that any differentiable function can be approximated locally by a straight line and a twice differentiable function can be approximated by a quadratic. Hence, there is a smoothness assumption in the functions being estimated using LOESS.

It is important to emphasize that although a polynomial is used for fitting locally, the overall fit covering the entire range of data is non-parametric. The polynomial is only used to find the value and slope of the estimation point (current center of the regression window). The accumulation of these estimates forms the overall LOESS fit which need not be parameterized *a priori.*

For most applications, it is sufficient for locally-fitted polynomials to be either linear or quadratic. For ASIP DSE, we are generally concerned with the coarse-grained behavior of a relatively-small neighborhood of the design space. As a consequence, we use linear least-squares regressions for the work presented here.

If the local regression is approximated by a linear function within a neighborhood of x_i, then the closed-form estimate of $\mu(x)$ is [143]:

$$\hat{\mu}(x) = b_0 + (x - \overline{x}_w)b_1 , \tag{4.3}$$

where

$$b_0 = \frac{\sum_{i=1}^{n} w_i(x)Y_i}{\sum_{i=1}^{n} w_i(x)} \qquad b_1 = \frac{\sum_{i=1}^{n} w_i(x)(x_i - \overline{x}_w)Y_i}{\sum_{i=1}^{n} w_i(x)(x_i - \overline{x}_w)^2}$$

$$\overline{x}_w = \frac{\sum_{i=1}^{n} w_i(x)x_i}{\sum_{i=1}^{n} w_i(x)} \tag{4.4}$$

and $w_i(x)$ is a suitable weighting function.

The closed-form estimate of Equation (4.4) is based on a least-square fitting criterion which satisfies the following:

$$\min \sum_{i=1}^{n} w_i(x)(Y_i - \hat{\mu}(x)) \tag{4.5}$$

To estimate the value of unknown data points using known data points, the local regression weight function from Cleveland [30] was modified so that unknown data points have their weights set to zero:

$$w_i(x) = \begin{cases} W\left(\dfrac{\|x_i - x\|}{h}\right), & x \in \text{known points} \\ 0, & x \in \text{unknown points} \end{cases}, \tag{4.6}$$

and known data points are weighted with the symmetric weight function. From [129,143], an often-used weight function is a tricube function:

$$W(x) = (1 - |x|^3)^3. \tag{4.7}$$

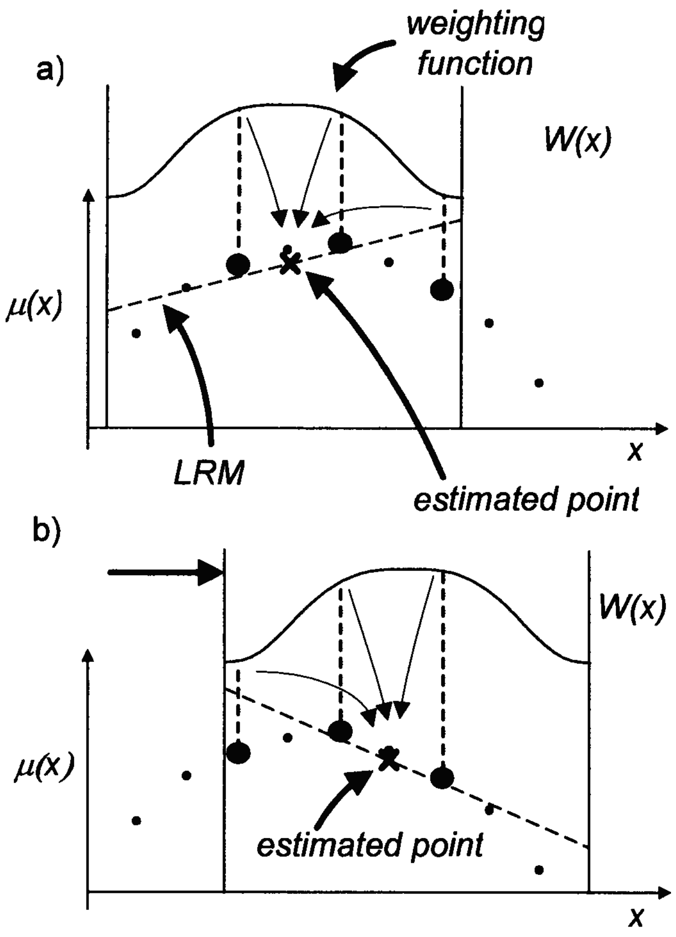


Figure 4.2: The regression window is shifted from left to right for different points being estimated (shown as an "x").

To illustrate the process, Figure 4.2 shows a set of data points with the x-axis as the predictor variable and the y-axis as the response variable. Both of these plots have the same three known data points (large dots) and six unknown data points (small dots). To perform a local regression, a regression window is adjusted with the point to be estimated at the center of the neighborhood as in Figure 4.2(a). The response values of all known data points are assigned weights based on their distance from the fitting point. For a given fitting point, x, Equation (4.7) is used to determine the i^{th} weight. These weighted points contribute according to Equation (4.4) to the calculation of the unknown point (estimated point). In Figure 4.2(b), the window is shifted to the right to center the next unknown point.

4.2.3. Point Selection

Point Selection determines the number of points and their location for evaluation. The number of points can be determined from the *bandwidth* and *smoothing parameters*. The bandwidth, h, of a local regression defines the number of data points used to estimate local regression at each fitting point. It has a significant impact on the fit of the local regression through its variance-bias trade-off. If the bandwidth is too small then the resulting fit will be too noisy (will have large variance). On the other hand, if the bandwidth is too large, then the polynomial will not provide a good quality fit. As a result, $\hat{\mu}(x_i)$ could be distorted and will therefore have increased bias.

An approach specified in [144] uses a smoothing parameter, α, defined as:

$$\frac{\lambda+1}{n} < \alpha < 1 \tag{4.8}$$

where λ is the degree of the local regression polynomial and n is the size of the data set. For a linear least-squares, $\lambda=1$. If we assume that $n=9$, then it follows from Equation (4.8) that $0.22<\alpha<1$. To balance bias and variance, the smoothing parameter should, by "rule-of-thumb", lie between 0.25 and 0.5 for most applications [144].

For LRM-DSE, a nearest neighbor specification for bandwidth is used. This ensures that the fitting neighborhood always has a fixed number of points. With this assumption, the bandwidth can be expressed as $h \geq \lceil n\alpha \rceil$. If $n=9$, then $h \geq 2$. When using the more practical rule-of-thumb, the bandwidth should be $h \geq 3$. This specification is suitable for a linear fit where the response is known for all data points in the regression window. Thus, the following minimum exists for the number of known points used for a local regression at each position of the local regression window:

$$\begin{array}{ll} h \geq 2 & \text{hard minimum} \\ h \geq 3 & \text{conservative mimimum} \end{array} \tag{4.9}$$

Beyond the minimum specified in Equations (4.9), the size of the sample set is important in that it dictates the degree to which we can model fine-grain trends in the design space. If we sample a large number of points (i.e. >1000 points), we will have a more accurate model of the design space but at the expense of increased simulation time. If we have a smaller sample set (i.e. <1000 points), simulation time will be saved but at the expense of a poorly-modeled design space.

4.2.4. Quality of Fit

To determine the quality of fit and to make comparisons between two or more local regression models, a simple criterion is needed. We propose a criterion that estimates unknown data points and compares them to their true simulated value. We call this criterion the *unknown response validation score (URV)* and it has an average sum of squares form:

$$URV(\hat{\mu}) = \frac{1}{m}\sum_{i=1}^{m}(Y_i - \hat{\mu}(x_i))^2 \tag{4.10}$$

where *m* is the number of unknown data points and the prediction mean squared error is arithmetically averaged over all data points estimated using our approach.

URV is similar to the commonly used *coefficient of determination*, R^2, as used in Section 3.5.6.1; however, URV only evaluates points in the solution space that are unknown. The quality of optimization stemming from the statistical model is dependent on the error of estimates for only those points that are unknown (small dots in Figure 4.1(b)) and not from points that are known ("x" points in Figure 4.1(a)).

4.2.5. LRM-DSE Configuration Methodology

The LRM-DSE methodology is composed of six major steps. First, a neighborhood of interest must be specified which dictates what part of the design space is modeled. Second, a subset of the points in the design space are selected and simulated. Third, LOESS is used on the sampled points in order to model the underlying trend. Fourth, this model is used to estimate the value of all remaining points within the neighborhood. Estimated points are then included in the model. Last, the model is used to identify which point or region in the neighborhood optimizes the objective function. A detailed description of these steps follows:

Step 1: *Specify the Neighborhood to be Modeled:*

The neighborhood of exploration defines the subset of points within the design space modeled for one instance of LRM-DSE. Irrespective of whether LRM-DSE is used as a stand-alone DSE algorithm or if it is used as part of another algorithm, the neighborhood of exploration must be specified. In many cases where the terrain of the design space is coarse-grain, the neighborhood can include the entire design space. In other cases where the terrain is more complex, it is more effective to use smaller neighborhoods. In such a case, LRM-DSE would be the inner loop of an algorithm where the outer loop is hill-climbing or hierarchical exploration. For such cases, each iteration of the outer loop would shift its neighborhood closer towards the direction where the global minimum seemingly resides.

For both examples presented in this thesis, we expected relatively smooth trends between each dimension of the design space and the objective function. As a consequence, we treat LRM-DSE on its own as a DSE algorithm and specify the neighborhood as the entire design space.

Step 2: *Determine Sampling Points:*

Once a neighborhood has been specified, we have the freedom to choose the data points that make up the sample of points simulated and those to be estimated. As discussed in Section 4.2.3, the size of the sample set defines a relative trade-off between model accuracy and simulation time. The two sampling policies investigated here are random-based selection and evenly-distributed selection. Random-based has the advantage that the sample set can be sized with finer granularity. The accuracy of a random-based selection versus evenly-distributed selection will be compared in Section 4.3.4.1.

Step 3: *Model the Underlying Trend by Applying LOESS to Sampled Points:*

For each known point within the neighborhood, local regression techniques are used to estimate its value such that an overall "smoothing" trend can be found (note that they may not be equal). These estimates do not replace known values found through simulation but are used in the next step to estimate the value of unknown points.

To perform LOESS, we incorporated a free statistics environment into our flow called R [145]. Multi-dimensional LOESS was performed using the R-compatible Locfit library offered by Bell Laboratories [146]. As discussed in Section 4.2.1, LOESS is a non-parametric statistic such that designers need not specify the precise characteristics of the model *a priori.*

Step 4: *Estimate all Unknown Points:*

The values of unknown points are computed based on the estimated values of known/simulated points. They are estimated in LRM-DSE using cubic Hermite polynomial interpolation for large data sets. With large numbers of data points, LOESS could become computationally expensive. This is especially true for multidimensional design spaces.

For small data sets, LRM-DSE estimates unknown points using another approach. This approach is a natural extension of the LOESS calculation in Equation (4.4) and can be used to estimate unknown points in the same way as it can for known points. The difference, however, is that unknown points are treated as "singularities" by setting their weight to zero.

Step 5: *Use Model to Select Point to Maximize Objective:*

This step simply chooses the optimal point or region in a neighborhood of the design space based on the model generated in the previous two steps. In the cache example to be discussed in the chapter, the optimization metric is power. The estimated optimal configuration within the design space is selected by simply choosing the lowest-valued point (whether known or estimated) in the model.

This 5-step process sets the parameters for a given ASIP for a target application. Practically, the use of regression modeling with DSE is well-suited for parameters associated with smooth functions. It is ineffective if the function has singularities or the parameter has a limited set of options. For example, there is little use in performing regression on a parameter with only two values that specify whether or not a multiply-accumulate unit should be included in the architecture, or whether an integer or floating-point architecture is suitable. The power or performance of a memory, on the other hand, can be expressed as a smooth function of its size and is therefore better-suited for our approach. This is also true for many of the parameters related to the cache, branch prediction, and register file.

4.3. Example: Tuning the Cache Hierarchy

4.3.1. Cache Hierarchy

The true capability of LRM-DSE is best illustrated on a cache memory example since this is an important application with many parameters. In addition, the memory cache of a processor has a large impact on its total power and performance. In some cases, the memory hierarchy can consume as much as 50% of the system power [147]. Currently, many vendors offer tunable caches; however, the designer is left to manually choose the correct configuration for their application.

Cache size is a function of the number of lines in the cache where each line is indexed by a memory address. The more lines in a cache, the fewer the cache misses. In fact, an infinitely large cache will have no cache misses. The number of additional cycles required to execute an application is significantly affected by cache misses. Power, on the other hand, is affected by cache size in two competing ways. A larger cache will dissipate more power per cycle; however, a larger cache will reduce the total number of cycles to complete a task, thus reducing overall power. Cache sizes are typically chosen in powers of 2 which causes large variations in performance and power.

Block size and the associativity both affect the size of the cache by increasing line size. As a consequence, they directly affect cache power dissipation. With respect to performance, the block size and associativity affect the number of execution cycles in a several ways. A larger block size improves spatial locality; however, it also incurs a larger penalty when a miss occurs. A higher associativity reduces the number of misses due to conflict; however, it also reduces the number of entries in the cache thus increasing the number of capacity misses. When comparing all parameters, it has been shown that line size and block size have a higher impact on performance than associativity [148]. For this reason, we focus on the configurability of the line size and block size.

Recent generations of processors tend to have larger caches and more levels of cache. The trend towards more levels of cache is a consequence of the growing trend towards increased memory size. As the size of main memory increases, the size gap between successive stages in the memory hierarchy increases. If the granularity of cache sizes is too coarse then there can be an inadequate match between the varying levels of locality within the application and the cache levels to serve them properly. The downside to more levels of cache is that additional levels of cache hierarchy require overhead circuitry. Also, more levels of hierarchy make DSE much more difficult because of the increased number of configurable parameters.

4.3.2. Tuning the Cache using LRM-DSE

The set of all possible cache configurations in the following example is defined by nine parameters for two levels of cache. Table 4.1 lists the range of values used for each parameter. To reduce the simulation time needed to provide comparisons to LRM-DSE, cache configurability is limited to dimensions P_1 through P_5 with the remaining dimensions set to a single fixed value.

Table 4.1: The dimensions of the design space resulting in 19,278 configurations.

	Description	Range
P_1	L1 instr. cache size	{ 1K, 2K, 4K, 8K, ... , 256K }
P_2	L1 data cache size	{ 1K, 2K, 4K, 8K, ... , 256K }
P_3	L2 unified cache size	{ 32K,64K,...,2M }
P_4	L1 instr. block size	{ 8, 16, 32, 64, 128, 256 }
P_5	L1 data block size	{ 8, 16, 32, 64, 128, 256 }
P_6	L1 unified block size	{ 64 } *(fixed)*
P_7	L1 instr. block size	{ 1-way } *(fixed)*
P_8	L1 data block size	{ 4-way } *(fixed)*
P_9	L1 unified block size	{ 4-way } *(fixed)*

Configurability was provided for both the cache size and the line size because these parameters have been shown to have a higher impact on performance than associativity [148]. After removing illegal combinations, the resulting design space has 19,278 valid configurations which is sufficient to demonstrate the approach.

4.3.3. Experimental Platform

Figure 4.3 illustrates the experimental platform used to test our approach. A set of benchmarks drives the overall system. A "C" description of each application is compiled using the GNU cross-compiler toolset [149]. The code is then executed with Power Analyzer [135]. Results from the local regression tool are used to configure the compiler and the simulation tool through architecture description files. In accordance with Power Analyzer, energy is reported as the sum of individual power dissipation values sampled for all cycles. Units are reported as Joules*cycles/sec.

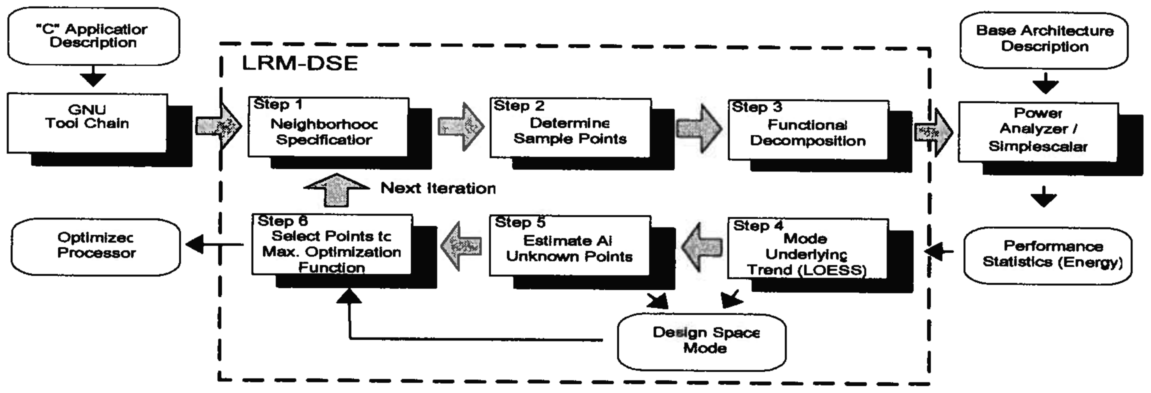

"C" Application Description
GNU Tool Chain
Optimized Processor
LRM-DSE
Step 1 Neighborhood Specification
Step 2 Determine Sample Points
Step 3 Functional Decomposition
Next Iteration
Step 6 Select Points to Max. Optimization Function
Step 5 Estimate All Unknown Points
Step 4 Model Underlying Trend (LOESS)
Design Space Model
Base Architecture Description
Power Analyzer / Simplescalar
Performance Statistics (Energy)

Figure 4.3: The LRM-DSE experimental flow incorporating LRM-DSE.

Experiments were conducted using the StrongARM architecture [150] as the base processor. The GSelect branch predictor and a 2-level cache have been added to the processor in order to perform DSE experiments in the following sections. As a baseline for comparisons, we use a typical default configuration where the level 1 data cache is

16K, has a line size of 32 and an associativity of 4. The level 1 instruction cache is 16K, has a lines size of 32 and associativity of 1. The unified level 2 cache is 256K, has a line size of 64, and an associativity of 4. The default size of the branch predictor pattern history table (PHT) is $k=1024$.

Ten benchmark applications were taken from the SPECcpu2000 Integer Suite [151] and the MediaBench Suite [152]. Each benchmark was limited to a maximum of 100 million instructions (with a 50 million instruction warm-up) to allow the exploration of larger design spaces.

It is important to note that our technique does not provide models for timing, power consumption, or area of the synthesized processor. Rather, it assumes an adequate level of accuracy in the simulation tool itself, and then predicts simulation results based on previous simulations. The main purpose of our approach is to drastically reduce the number of simulations that need be performed with the simulation tool.

4.3.4. Results

4.3.4.1. Quality of Estimates

The design space defined by the parameters in Table 4.1 was modeled using LRM-DSE. As mentioned earlier, a cubic Hermite polynomial interpolation is used to estimate unknown points. Initially, the distributed point selection policy used for cache was based on the conservative minimum in Equation (4.9) (3 samples per dimension). As a consequence, the number of points sampled and simulated was $3^5=243$ which corresponds to 1.3% of the design space. Initial experiments found that it is possible to further reduce the proportion of the design space simulated so experiments were conducted with random selection.

The following proportions of points randomly sampled were used for simulation: ρ = *{0.2%, 0.4%, 0.6%, 1%, 2%, 4%, 8%}*. With ρ = *0.2%*, the number of points sampled in each dimension is slightly above the hard minimum as defined in Equation (4.9).

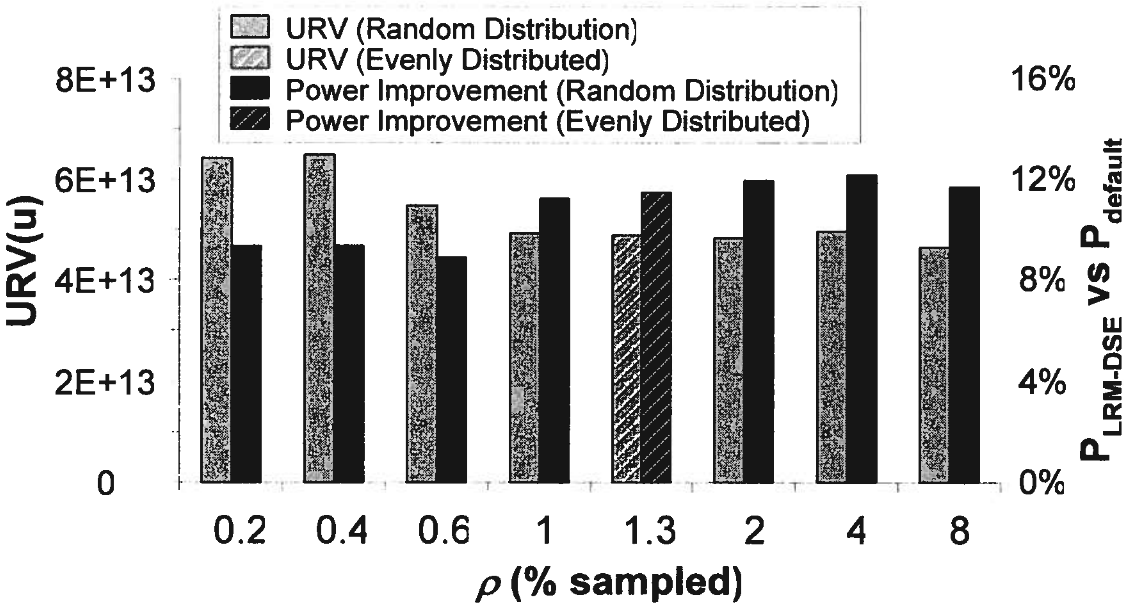


Figure 4.4: URV(μ) and percentage power improvement for different sampling densities.

Each point estimated using LRM-DSE in the design space was then compared against the simulated value and the value of URV (which is similar to the R^2 measure). URV results are shown in Figure 4.4 for all values of ρ. In addition, the average power improvement over the default configuration is shown for each value of ρ. Each bar is the average URV over all benchmarks. From this figure, it is evident that the quality of the LRM-DSE improves with higher sampling rates but levels off at around 1%. Also included in Figure 4.4 is the result for evenly-distributed point selection. Of the various random sampling rates, ρ = *1%* is a good tradeoff between quality and reduced number of simulations. This sampling rate corresponds to a 100x speed-up over an exhaustive search of the design space. Evidently, evenly-distributed point selection with $\rho = 1.3\%$ has roughly the same quality of fit as random selection with $\rho = 1\%$.

Table 4.2: The power reduction due to LRM-DSE for random and evenly distributed sampling.

	% Power Reduction ($P_{LRM\text{-}DSE}$ vs $P_{default}$)		% Power Reduction ($P_{optimal}$ vs $P_{default}$)
	Random (ρ=1%)	Evenly Distributed (ρ=1.3%)	
Qsort	11.6	12.0	13.9
CRC32	8.6	8.6	8.9
bitcount	8.0	8.2	10.0
bzip2	53.0	53.0	53.0
g721decode	9.3	9.3	10.9
gzip	9.0	10.3	12.6
mcf	14.2	14.2	14.2
Mpeg2dec	5.9	5.8	6.5
sha	9.1	9.1	9.2
Twolf	10.5	10.5	10.5
Average	**13.9**	**14.1**	**15.0**

4.3.4.2. Power Savings

In Table 4.2, energy results are provided for sampling rates of $\rho = 1\%$ for random selection (100x speed-up) and $\rho = 1.3\%$ for evenly-distributed selection (~77x speed-up). On average, LRM-DSE is able to configure the cache to *improve the overall energy dissipated by the processor* by 13.9% for random sampling and 14.1% for evenly distributed sampling. These results are very close to the 15% achievable with optimum configuration. Although not shown, these results correspond to a *reduction in energy* *consumed by the cache* of 32.8% for random selection and 31.9% for distributed selection. Note that, while evenly-distributed selection provides a more energy-efficient solution than random selection for the overall processor, this is not the case for cache energy dissipation. In fact, some benchmarks may achieve a reduction in overall processor energy by increasing the size of the cache (and therefore increasing the energy dissipated by the cache). Perhaps more importantly, these results indicate that random point selection is nearly as effective as evenly distributed point selection.

Based on these results and additional analysis of the benchmarks, we believe that random sampling is more a favorable point selection policy over the evenly distributed approach. Random sampling achieves nearly the same results as evenly-distributed selection but also allows a larger variety of sampling rates. The ability to fine-tune the sampling rate is desirable for those applications where increased accuracy is important or if there is a need for fewer simulations.

4.3.4.3. Offsets from Optimal Configurations

It is interesting to examine the cases where LRM-DSE was offset from the optimal. Table 4.3 shows the optimal and LRM-DSE configurations for a subset of benchmarks where they differ by 2-3%. In this table, any parameter values using LRM-DSE (random distribution) that differ from the optimal configuration are underlined. For these cases, LRM-DSE did not obtain the ideal L2 cache size and the L1 instruction cache size. These results suggest that some dimensions may require more sampling than others. In spite of these variations, the overall results are all close to the optimal as shown in Table 4.3.

Table 4.3: LRM-DSE and optimal configurations are compared for four benchmarks with parameters that differ underlined.

Optimal	L2 unified	L1 data	L1 inst
Qsort	{8K,64,4}	{2K,16,4}	{8K,16,1}
Bitcount	{8K,64,4}	{1K,16,4}	{32K,64,1}
Gzip	{64K,64,4)	{4K,64,4}	{8K,64,1}

LRM-DSE	L2 unified	L1 data	L1 inst
Qsort	{32K,64,4}	{2K,16,4}	{16K,16,1}
Bitcount	{16K,64,4}	{1K,32,4}	{8K,32,1}
Gzip	{16K,64,4)	{8K,64,4}	{4K,64,1}

4.4. Conclusions

In this chapter, a method of modeling the design space of an ASIP using non-parametric statistics was introduced. By configuring a 2-level cache hierarchy, it was shown that LRM-DSE can find near-optimal configurations while only having to simulate a relatively small proportion of the design space. This accomplished automatically thus avoiding the extensive manual fitting required when using the manual decomposition approach from Section 3.5.

While tuning the cache, only 1% of the design space was evaluated. This resulted in an overall energy reduction of 13.9% on average with one design as high as 53%. This improvement is only a few percent less than that of the optimal configuration. Random sampling was shown to be as effective as evenly-distributed sampling, but allows more flexibility. The effective speedup is 100x compared to exhaustive sampling.

LRM-DSE is very promising as a general DSE approach in that it drastically reduces the number of simulations needed to find the global optimum. Because it is effective and fully automated, it places ASIPs within the reach of more system designers who may not have the in-depth knowledge of processor architecture needed to choose an appropriate configuration. This allows more system designers to take advantage of the power and performance benefits possible with ASIPs over off-the-shelf processors.

Chapter 5: Fast Evaluation of Instruction-Set Extensions

5.1. Introduction

Another important aspect of configurability for ASIPs is *Instruction-Set Extensions* where the instruction set of an embedded processor is extended with custom instructions to best suit an application or an application domain. Current ASIP methodologies are able to specify custom instructions automatically [5, 108]. This involves the grouping of a number of instructions into a cluster that can be implemented in hardware as a single complex instruction. This problem can be broken into three parts: *Pattern Enumeration*, *Pattern Selection (Instruction Selection)*, and *Pattern Matching*.

Figure 5.1(a) shows Path Enumeration of a hypothetical basic block of an application modeled as a directed acyclic graph (DAG). Each vertex in the DAG is a member of the base instruction-set with its functionality implemented in a *Functional Unit (FU)* for the execution stage of the pipeline. Edges represent data dependencies between operations. Shown at the top and bottom of the block are the entry and exit operations of the block, respectively. Pattern Enumeration is used to explore the solution space, and consequently to generate a list of all possible custom instructions in the basic block. This is repeated for all blocks in the application in order to generate a master list of candidates. Each candidate in the master list must adhere to the microarchitectural constraints imposed by the processor architecture. In this figure, only a subset of all possible candidate instructions are shown and circled.

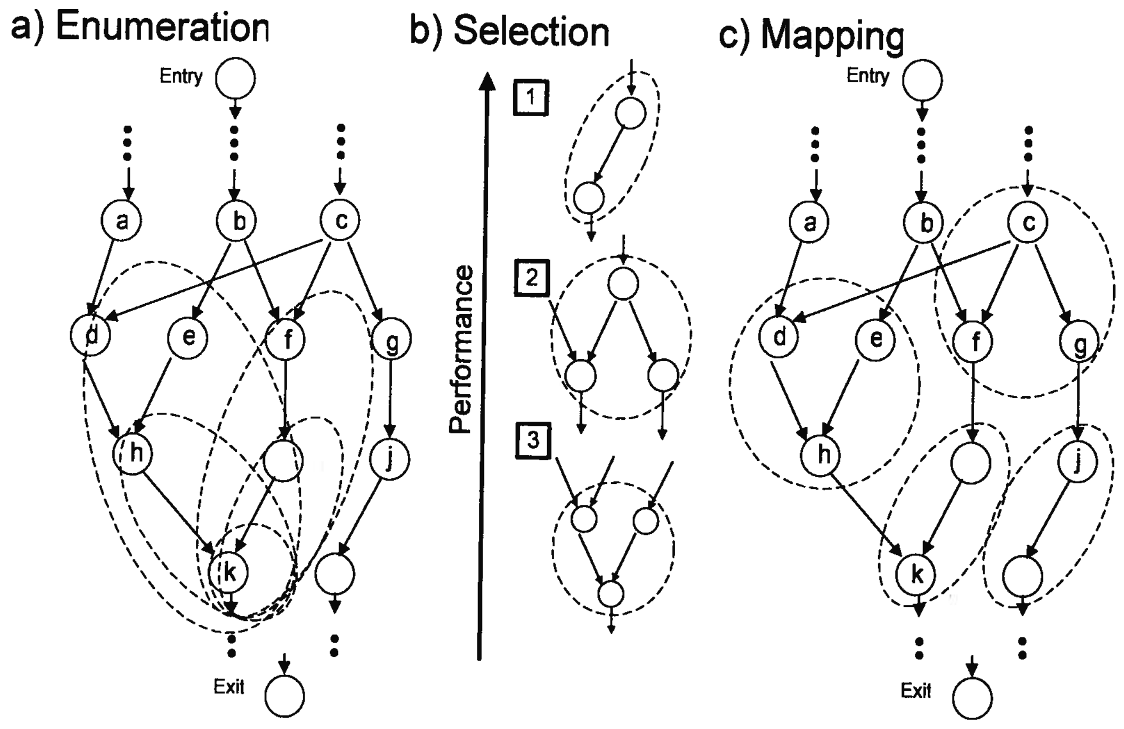


Figure 5.1: (a) Pattern Enumeration, (b) Instruction Selection, and (c) Instruction Mapping.

Once a master list of candidate instructions has been generated, the list is sorted according to the projected impact of each instruction on processor performance. A subset of the master list is chosen during the Instruction Selection phase of Figure 5.1(b) to implement as actual instructions in hardware. A high-level cost function is required to estimate the potential performance gain of the processor that would result by adding each candidate instruction. The cost function is typically based on a combination of the candidate's frequency of use, the projected impact on block latency, implementation area, and power when implemented as hardware. Figure 5.1(b) shows the top three custom instructions from a hypothetical master list which are tagged to be implemented as new custom instructions.

Figure 5.1(c) illustrates the mapping of how new instructions to a basic block. When a group of patterns has been identified, pattern matching is used to determine how to best map new instructions throughout the application. This pattern matching has been shown to be NP-hard and equivalent to a minimum-area technology mapping problem [29].

Several approaches have been proposed to perform Instruction Enumeration and Selection [110,111,112,114]. All of these are significant contributions towards the ASIP flow; however, they do not address the effects of pipeline stalls due to data hazards. Large errors in estimating the performance of applications with new instructions may result. This motivates the work in this chapter. Like many algorithms in ASIP and compiler research, the methods in [110,111,112,114] operate on intermediate representations of the application by adding, removing, and clustering operations. One way to improve all algorithms of this type would be the addition of a "hazard-aware" performance predictor for architectures with pipeline data-hazards.

Dependencies between instructions in the DAG have the potential to stall the pipeline whenever an instruction waits for a result from preceding instructions. This is illustrated in Figure 5.2 which shows a processor *arithmetic logic unit (ALU)* with custom functional units (FUs). To prevent stalls, many processor architectures implement a forwarding scheme where the output of an FU is forwarded from one stage to another so that the FU corresponding to the next instruction can immediately use the value (FU-to-FU forwarding). To accomplish this, multiplexers must be added to the inputs of all FUs. Further, for every custom instruction added to the architecture, a custom FU must be added to the hardware. Similar to the base architecture, custom FUs must contend with pipeline dependencies and stalls. As a result, a custom architecture can only be free of data hazards if multiplexers are added to all FUs so they can fully access results from all other FUs.

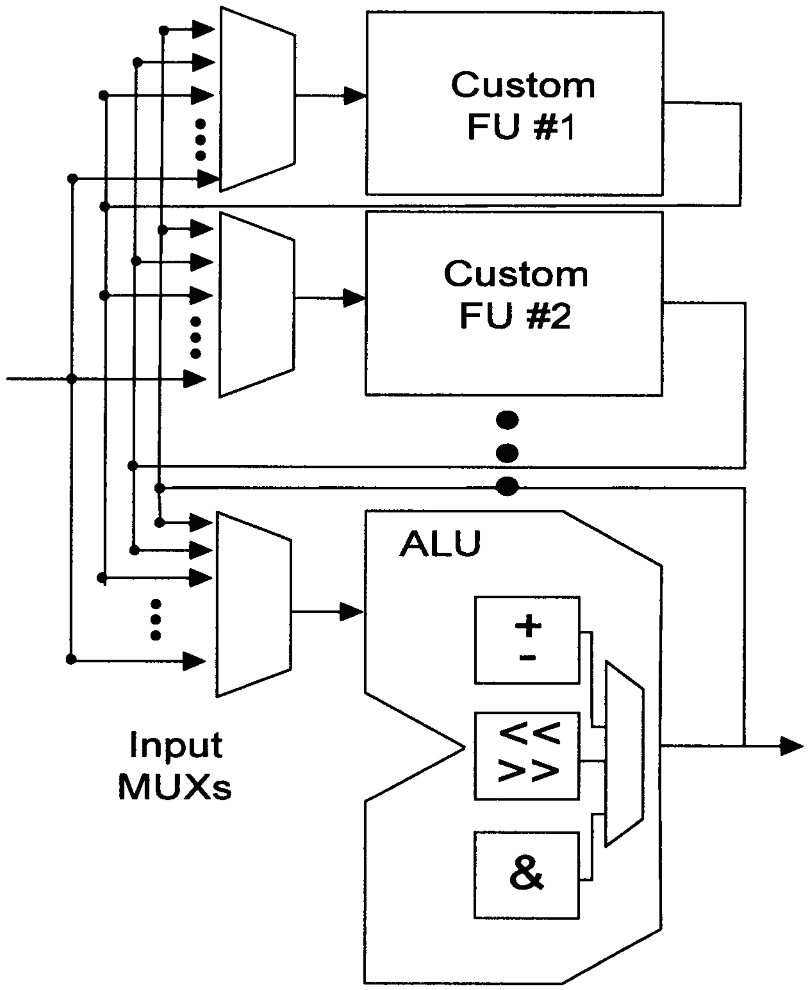


Figure 5.2: FU-to-FU forwarding (only one input is shown for simplicity).

To provide fast performance estimation, [29] and [113] assume an ideal pipeline. [29] states that "it is not trivial to compute the total latency" of the application with data hazards. [113] claims that data hazards need not be considered in the presence of FU-to-FU forwarding. As stated above, FU-to-FU forwarding is a possible solution; however, it would require new multiplexers to be added to all new FUs. This might be practical for a small number of custom FUs but it does not scale well thus it would likely add significant power and performance penalties with large numbers of FUs.

If FU-to-FU forwarding is not used, then it becomes difficult to compute the total latency of the application after substituting in custom instructions. Figure 5.3(a) shows a basic block that has been scheduled such that vertex labels indicate the order in which instructions enter the pipeline. In this figure, operations 4, 5 and 8 have been circled indicating a cluster of operations. Figure 5.3(b) shows the basic block after the cluster has been replaced by a custom instruction as represented by the black vertex. After making this transformation, it is difficult to assess the total run-time of the block without rescheduling operations. It is desirable to avoid rescheduling because it is expensive in terms of the run-time of the Instruction Selection algorithm. In fact, a typical scheduling algorithm such as List Scheduling has a run-time complexity of $O(N \cdot log(N) + E)$ where N is the number of vertices and E is the number of edges in the basic block [153]. Long run-time complexity is problematic because Pattern Selection can potentially iterate through hundreds of thousands of potential clusters and so it is imperative that fast techniques be developed for estimating run-time.

In this dissertation, estimation techniques were developed on the premise that FU-to-FU forwarding is too expensive and that data hazards must be considered. To accomplish this, a method is proposed for estimating how the schedule length (in cycles) of a block changes after substituting a cluster of instructions with a custom instruction [154]. Because this method is significantly faster than rescheduling, it can easily be incorporated into Instruction Selection, Enumeration and Matching heuristics to serve as a "hazard-aware" performance predictor. Thus, the following will be addressed in the following sub-sections: First, a new method will be proposed for estimating the performance improvement of automatic instruction-set processors with data hazards. Second, the impact of hazard-aware prediction on Instruction Enumeration and Selection will be quantified. This is accomplished by incorporating the proposed predictor into an Instruction Enumeration and Selection algorithm.

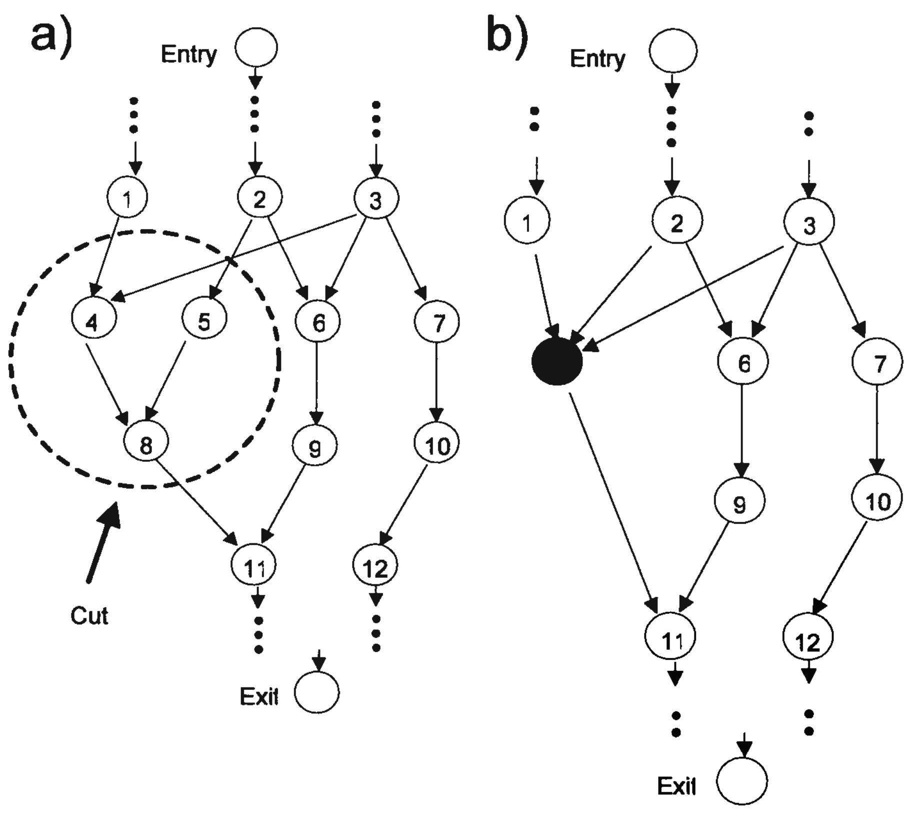


Figure 5.3: The schedule of a basic block (a) before and (b) after adding a custom instruction.

5.2. Enumeration and Selection

5.2.1. Problem Definitions

An application can be specified as a control data flow graph (CDFG) in which each node represents a basic block and each edge represents a control dependency. A basic block can be represented as a data flow graph and is modeled as a DAG $G=(V,E)$ such that V is the set of vertices representing basic instructions and E is the set of directed edges representing register dependencies.

A candidate instruction set extension (ISE) is represented by a cluster C which is defined as a subgraph of G, $C \subseteq G$. Because C will be implemented as a functional unit that will load/store values to/from the register file in a single clock cycle, it must adhere to *input and output constraints* such that the number of inputs and outputs must not exceed the number of input ports, N_{in}, and output ports, N_{out}, of the register file. In addition, C must adhere to a *convexity constraint* such that there must not exist a path P such that nodes $i \in C$ and $k \in C$ are connected through node $j \notin C$. Otherwise, the result provided by instruction i would not be available on time for operation k.

During selection, clusters are partially evaluated based on their impact on block latency when implemented as hardware. To determine this, the latency of the cluster in both hardware and software must be determined. The latency of the block is equivalent to its schedule length, which can be defined as follows: Every operation v_i takes $delay(v_i)$ time steps to execute. An operation v_i is said to be scheduled at time t_i if it starts execution at time step t_i. A dependency $<v_i, v_j> \subseteq E$ implies that an operation v_j can only be scheduled on or after $t_i+delay(v_i)$. A schedule of G is a function $S: V \rightarrow Z^+$ where each operation v_j is said to be scheduled at time $S(v_j)$. The execution time of a schedule S is the maximum completion time of all operations, $ET(S) = max\{S(v_j) + delay(v_i)\}$. The goal of the scheduling algorithm is to minimize $ET(S)$ so that the block is executed in the shortest number of cycles.

For the following sections of this thesis, List Scheduling [153] is used to perform instruction scheduling within the compiler. The goal behind performance prediction is to accurately and quickly estimate the schedule length that would be achievable through List Scheduling.

5.2.2. Performance Estimation (the Simple Predictor)

References [29], [110], and [113] perform Instruction Selection using a metric that estimates the speedup potential of each candidate instruction. For a given cluster of instructions, the number of cycles needed to execute all instructions as software is compared to the number of cycles needed to execute them in hardware. For the simple predictor, the execution time in hardware and software is,

$$T_{SW} = \sum_{all\_instrutions} \Lambda_{SW} \tag{5.1}$$

and,

$$T_{HW} = \left\lceil \sum_{CP\_instructions} \Lambda_{HW} \right\rceil \tag{5.2}$$

respectively, where Λ_{SW} and Λ_{HW} are the number of cycles needed to execute a specific instruction in software and hardware, respectively. *CP_instructions* is the set of all instructions along the critical path in hardware.

In hardware, a custom FU fully exploits all available parallelism to execute the cluster of instructions (spatial computing). As in Equation (5.1), an estimate of the execution time of the cluster is the sum of the latencies of all operations along the critical path.

With respect to software, [29], [110], and [113] assume that the processor pipeline is simplified such that there are no data hazards. With this simplification, the software latency can be easily calculated because instructions are executed sequentially (temporal computing). As in Equation (5.2), an estimate of the execution time is simply the sum of the latencies of all instructions in the cluster. For the remainder of this chapter, we call this approach the *Simple Predictor*. For this predictor, estimating performance is relatively straight-forward. If we refer back to the example in Figure 5.3, the estimated reduction in schedule length for operations 1 through 12 would be 2. In the next section, we will show how the problem becomes more complex when data hazards are considered. In fact, results presented later will show that the Simple Predictor results in a 50% error on average for architectures with data hazards.

5.3. Predicting the Impact of Hazards

In the previous sections, Instruction Enumeration and Selection algorithms as well as performance estimation techniques were discussed for ideal pipelines (i.e., no data hazards). Also, a formal definition of a schedule was provided. In this section, pipelines with data hazards are considered by first discussing how a schedule is affected by data hazards when custom instructions are introduced. Further, a detailed description of the proposed "hazard-aware" performance predictor is provided.

5.3.1. Considering Data Hazards

When data hazards are ignored, the *delay*(v_i) must only be long enough to avoid all structural hazards. Conversely, when data hazards are considered, the *delay*(v_i) must also allow for results computed by v_i to be written to registers. Because of this, it is no longer possible to estimate changes in the total execution time of the block by simply subtracting the sum of delays of all instructions in the cluster. Every instruction has the potential to be stalled by previous instructions or to stall subsequent instructions in order for dependencies to be resolved. When a cluster of instructions is removed from a block and replaced by a custom instruction, it can affect the optimal scheduling of the entire block. As a result, it is difficult to assess the impact of a cluster without performing a reschedule operation on the entire block. The method proposed in the next section partitions the schedule in order to obtain a reasonable "first-order" estimate of the impact of a custom instruction.

5.3.2. Proposed "Hazard-Aware" Predictor

The hazard-aware predictor estimates the impact of a custom instruction on performance in three stages. First, the block $G=(V,E)$ must be partitioned into levels based on its pre-schedule ordering and dependencies. Second, this partitioning is used to determine the best- and worst-case changes in delay for all levels directly affected by the transformation. Last, the best- and worst-case delays are combined using a summation weighted by the size of the cluster to generate an estimate.

After partitioning, there should be no dependencies between instructions that reside at the same level; however, instructions at a given level may be dependent on instructions at lower numbered levels. To perform this partitioning, the first node in the schedule (entry node) is added to the lowest level. The second node in the schedule is also added to the same level only if it does not depend on the first node. If it does depend on the first node, it will be added to a new level. More generally, as the predictor sequences through the schedule, each new instruction will be added to the current level only if it does not depend on any other instructions at that level. If it does, a new level must be created for that instruction. This partitioning step need only be completed once before using the predictor in an algorithm.

To illustrate this process, Figure 5.4(a) is an example DAG of a block divided into dependency levels. Each node represents an instruction with a label indicating its pre-scheduled ordering. Arrows represent data dependencies between instructions and dotted lines represent the divisions between dependency levels.

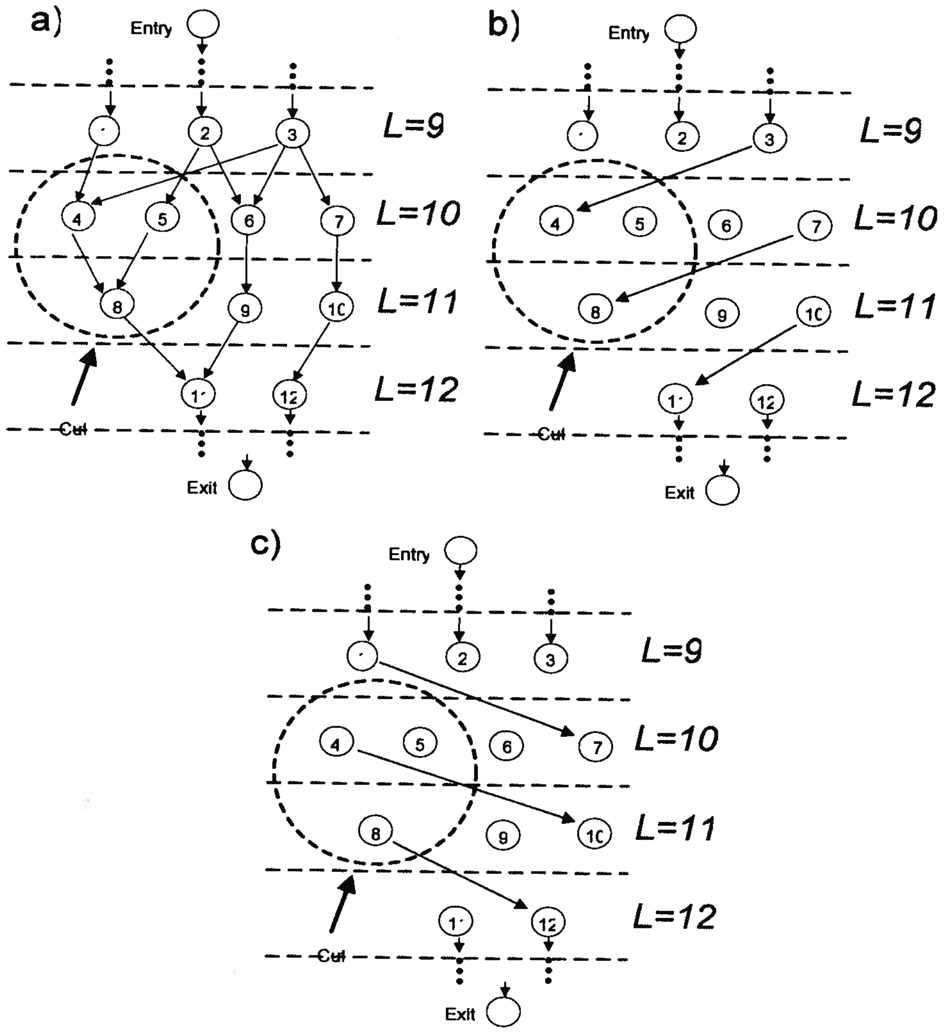


Figure 5.4: A DAG representation of a basic block (a) partitioned according to levels of dependency, (b) with worst case dependencies, and (c) with best case dependencies.

By organizing the block of instructions into levels as in Figure 5.4(a), the predictor isolates the impact of the custom instruction to only those levels with operations involved in the cluster. An approximation is made by assuming that the schedule for all other levels will not be significantly affected by the substitution of the cluster with a custom instruction. Suppose that a cluster has been identified as consisting of instructions 4, 5, and 8. After the transformation, these instructions will have been removed and replaced with a custom instruction at level *L=10*. To estimate the impact of this modification, the predictor will approximate the change in schedule length only for levels *L=10* and *L=11*.

To determine how the schedule length of levels *L=10* and *L=11* will be affected by a custom instruction, the predictor must calculate the worst-case and best-case possible schedule lengths for these levels. This must be performed without the knowledge of how the instructions at these levels would be ordered if they were to be fully rescheduled. What is certain is that all instructions at a given level can be swapped without violating dependencies. Because the four instructions at level *L=10* are not dependent on each other, they will be properly pipelined and will take only 4 cycles to execute no matter what order they are scheduled. Unfortunately, by not knowing their ordering, our predictor cannot determine whether or not the first instruction to be scheduled for a given level is dependent on the last instruction to be scheduled from the previous level. If it were as shown in Figure 5.4(b), this would result in the worst case delay and would add an additional *P-1* cycle stall penalty where *P* is the effective length of the pipeline. This stall would need to be added twice: once for the transition from level *L=9* to level *L=10* and once for the transition from level *L=10* to level *L=11*. If

P=10, the worst case delay for level *L=10* includes 4 cycles to execute instructions and two stalls,

$$D_{worst}(10) = 4+9+9=22 \tag{5.3}$$

To calculate the best-case delay as shown in Figure 5.4(c), the predictor assumes that the only dependency that exists is between the last instruction scheduled at the current level and the first level instruction scheduled at the previous level. Thus, the best case stall penalty would be *P* minus the number of instructions on both levels plus one. Two stalls for two level transitions would be added to the 4 cycles needed to execute the instructions at level 10. Thus,

$$D_{best}(10) = 4+(10-(3+4)+1)+(10-(4+3)+1)=12 \tag{5.4}$$

The above-mentioned approach only assesses the best- and worst-case delays for one level. To calculate the impact of a given custom instruction, delays for all levels directly affected by the transformation must be calculated. Using the same approach in the example, the best- and worst-case delays are summed for all levels. If the levels of interest are $L=\alpha$ to $L=\beta$, inclusive, then the worst and best case latencies are:

$$D_{worst}(\alpha,\beta) = (\beta-\alpha+2)(P-1)+\sum_{l=\alpha}^{\beta} N_l \tag{5.5}$$

and

$$D_{best}(\alpha,\beta) = \sum_{l=\alpha-1}^{\beta} \max(0, P-N_l-N_{l+1}+1)+\sum_{l=\alpha}^{\beta} N_l \tag{5.6}$$

respectively, where *P* is the length of the pipeline loop, N_l is the number of instructions at *level l*, and $\alpha \le \beta$. For levels *L=10* and *L=11* from the example,

$$D_{worst}(10,11) = (11-10+2)(10-1)+(4+3)=34 \tag{5.7}$$

and

$$D_{best}(10,11) = \max(0,10-3-4+1) + \max(0,10-4-3+1) + \max(0,10-3-2+1) + (4+3) = 21 \tag{5.8}$$

Once the worst and best case delays before the transformation have been solved using Equations (5.7) and (5.8), the predictor must perform the same calculation for the block after the transformation (i.e., replacement of cluster with the custom instruction). The maximum and minimum changes in delay can then be calculated as follows:

$$\Delta D_{\max}(\alpha,\beta) = D_{worst,before}(\alpha,\beta) - D_{best,after}(\alpha,\beta) \tag{5.9}$$

and

$$\Delta D_{\min}(\alpha,\beta) = D_{best.before}(\alpha,\beta) - D_{worst.after}(\alpha,\beta) \tag{5.10}$$

In the above, $\Delta D_{min}(\alpha,\beta)$ is a useful result because it serves as a lower-bound on the reduction in execution time. This is true because the calculation of $\Delta D_{min}(\alpha,\beta)$ provides a legal schedule that represents the worst-case reduction in cycles due to the transformation.

Once the maximum and minimum impact of a transformation has been determined, an estimate of the change in the overall schedule length due to the replacement of the cluster C with a custom instruction can be found. In our approach, a weighted summation of $\Delta D_{min}(\alpha,\beta)$ and $\Delta D_{max}(\alpha,\beta)$ is used,

$$\Delta D(\alpha,\beta) = (1-A)\cdot\Delta D_{\min}(\alpha,\beta) + A\cdot\Delta D_{\max}(\alpha,\beta) \tag{5.11}$$

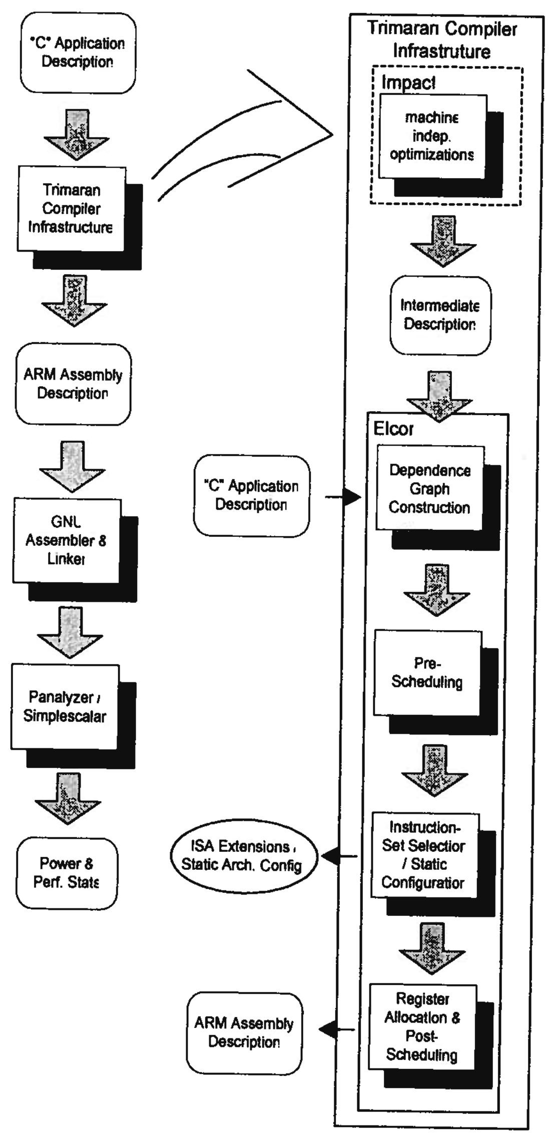
where $0 \le A \le 1$ is a weight coefficient. A is determined dynamically on a per block basis and can be a function of block characteristics such as the code length, the number of levels, or the number of levels to block size ratio. For simplicity, we restrict A to be a linear function of cluster size. This works well because the larger the cluster size, the larger the proportion of instructions at each level that will be consumed within the cluster. With a larger proportion of each level consumed within the cluster, it is more likely that a longer stall will arise between levels as a consequence of the transformation. As a result, the $\Delta D_{max}(\alpha,\beta)$ term will become the more prominent factor term in Equation (5.11).

As the enumeration algorithm adds or removes operations to the current candidate to specify a new candidate, it keeps track of the number operations at each dependency level and the maximum and minimum level involved. This information can be retrieved by the "hazard-aware" predictor in constant time to calculate the first term in Equation (5.5). The second terms in Equations (5.5) and (5.6) can be incremented or decremented in constant time as new operations are added to the cluster during instruction enumeration. Pre-calculated values for the first term in Equation (5.6) can be stored in a hash table and retrieved in constant time. Overall, the run-time complexity of the "hazard-aware" estimator is $O(1)$, or constant time complexity. Thus, it has the same run-time complexity as the simple predictor and significantly lower run time complexity than List Scheduling which is $O(N \cdot log(N) + E)$, where N is the number of vertices and E is the number of edges in the basic block.

5.4. Experimental Platform

A system was developed to conduct the configuration experiments described above. Figure 5.5 illustrates the flow that was implemented in this work to perform software compilation and hardware configuration. A "C" description of the application is parsed and optimized by a modified version of the Trimaran compiler infrastructure

[138] which then produced ARM v.4 assembly code and a file listing all instruction-set extensions added to the ISA. In a similar way to [155], the base processor was modified to be more "ARM-like". In particular, the architecture was scaled down to a single pipeline with three execution stages where only one instruction can occupy a given stage at a time. All instructions require three clock cycles to pass through the pipeline. This is a simplified architecture; however, it is sufficient to demonstrate the effects of data hazards.



"C" Application Description
Trimaran Compiler Infrastructure
ARM Assembly Description
GNU Assembler & Linker
Panalyzer / Simplescalar
Power & Perf. Stats
Trimaran Compiler Infrastruture
Impact
machine indep. optimizations
Intermediate Description
Elcor
Dependence Graph Construction
Pre-Scheduling
Instruction-Set Selection / Static Configuration
Register Allocation & Post-Scheduling
"C" Application Description
ISA Extensions, Static Arch. Config
ARM Assembly Description

Figure 5.5: The Trimaran flow with modifications for ISEs.

The code was then assembled and linked using standard GNU tools [149]. The binary code can be executed using an instruction-set simulator such as Simplescalar [156] or Panalyzer [157]. As illustrated in Figure 5.5, instruction enumeration and selection was added to the back-end of Trimaran. Instruction enumeration and selection was implemented using the branch-and-bound algorithm presented in [110] but with the "hazard-aware predictor" outlined in Section 5.3.2. To maximize the number of opportunities for identifying new instructions, instruction enumeration and selection were placed before register allocation to reduce false dependencies.

5.5. Results

5.5.1. Accuracy of the Performance Predictor

The "hazard-aware" predictor was incorporated into the flow described in Section 5.4. A set of custom instructions was generated using Instruction Enumeration and Selection. For each basic block, the cluster that resulted in the maximum reduction in execution cycles was chosen. The change in clock-cycles for each block was estimated using the "hazard-aware predictor and was then compared to the best value found through List-Scheduling. A similar comparison was made between the Simple Predictor and the List-Scheduler. The purpose of this experiment was to gauge the quality of the proposed predictor against existing methods.

Table 5.1 presents results for benchmark applications from the MediaBench Suite [141] and the Trimaran Distribution [138]. These benchmarks are representative of larger suites frequently used in the microarchitectural community. Results include the percent error in execution time per block as predicted by the hazard-aware predictor versus the simple predictor [29,113].

Results show that the hazard-aware predictor has a mean block error of 15.4% while the simple predictor has a mean block error of 50.8%. Further, the standard deviation of block error is 18.9% and 65.8% for the hazard-aware predictor and the simple predictor, respectively. These results show that the hazard-aware predictor is more accurate than the simple predictor. Further, the smaller standard deviation of the hazard-aware predictor indicates that it provides more consistent predictions.

Table 5.1: Error in execution time per block as predicted by the "hazard-aware" estimator.

	mean % error		std.dev. % error	
	Simple Pred.	Hazard Aware Pred.	Simple Pred.	Hazard Aware Pred.
bmm	35.6	14.3	46.8	8.9
g721decode	97.2	18.6	95.9	21.7
g721encode	93.5	18.1	96.1	21.3
mpeg2dec	35.9	15.1	51.1	18.4
mpeg2enc	20.8	14.3	54.6	30.8
parms_test	21.9	12.2	50.1	12.4
mean	50.8	15.4	65.8	18.9

The results provided above are promising; however, the "hazard-aware" predictor has its limitations. If the candidate cluster resides on dependency levels with many operations, the accuracy of the predictor becomes worse. In particular, the greater the number of operations per level and the fewer the number of levels occupied by the cluster, the greater the separation between the worst- and best-case bound given by Equations (5.5) and (5.6), respectively. With a large separation between these bounds, predictor accuracy drops. Although the Simple Predictor is not susceptible to this limitation, results show that the "hazard-aware" predictor provides more accurate estimates on average.

5.5.2. Effects of Data-Hazards on Enumeration & Selection

In the previous section, the hazard-aware predictor was shown to be significantly more accurate than the simple predictor. The hazard-aware predictor was then incorporated into an instruction enumeration and selection algorithm to serve as a figure of merit when comparing candidate clusters. The Trimaran compiler was used along with the branch-and-bound algorithm from [110] with the hazard-aware predictor as a cost function. Candidate clusters from all blocks were added to a master list of candidates. To approximate the effects of having an area constraint, only a predefined proportion λ of these candidates could be implemented as custom instructions. The best λ proportion of these cluster candidates were chosen based on their expected run-time improvement using the hazard-aware predictor. The performance improvement for each benchmark is reported in Figure 5.6.

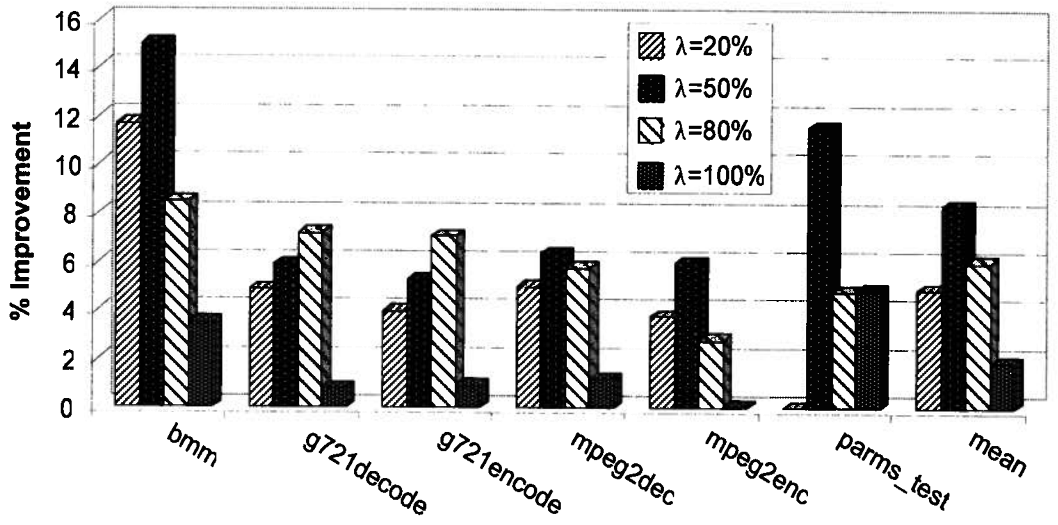


Figure 5.6: Improvement in ISEs due to the "hazard-aware" predictor.

Figure 5.6 compares overall performance when the hazard-aware performance predictor is used versus the simple predictor. In both cases, the architecture has a pipeline with data hazards. Averaged over all benchmarks (see right-most profile), the difference in performance improvement between the two predictors was found to be 4.9%, 8.4%, 6.0%, and 1.9% for λ values of 20%, 50%, 80%, and 100%, respectively. These results are significant because their benefit to overall processor speed is additive; on average, an improvement towards the overall speed can be expected for each custom instruction added to the ISA.

The improvement is best for the first three values of λ when the hazard-aware predictor plays a role in choosing candidates at the block level and at the application level. When λ=100%, all candidates at the application level are used as custom instructions. In this case, the improvement is less prominent because candidates are only compared at the block level where there is less granularity in the quality of candidates that adhere to all microarchitectural constraints.

5.6. Conclusions

In this chapter, a new evaluation method was proposed for evaluating points in the custom instruction design space. In particular, a "hazard-aware" performance predictor was developed for pipelined architectures. In addition to providing an estimate to the change in execution cycles of an application due to custom instructions, the "hazard-aware" predictor also provides a strict lower-bound. The estimate calculated by the tool was shown to be more accurate than the existing predictor which ignores hazards and is faster than List-Scheduling. By adding this predictor to an Instruction Enumeration and Selection flow, performance was improved by as much as 8.4%.

This is just one example of an algorithm and flow that benefits from the increased prediction accuracy, but other algorithms used for Instruction-Set Extensions would also benefit. The proposed predictor serves as an alternative for predicting performance when data hazards are present and improved schedule length accuracy is important.

Chapter 6: Conclusions

The embedded systems market is continuing to grow as the market demand for communications and portability grows. ASIPs are continuing to expand their application in embedded systems because of the significant performance and power gains that are achievable over off-the-shelf processors. Evidence of this lies with the growing interest in ASIPs in both academia and industry. The growing trend towards heterogeneous multiprocessor systems-on-chip (MPSoCs) will further increase the demands for ASIPs.

ASIPs are still in their infancy and have plenty of room for improvement. This is, in part, because the set of configurable parameters is limited and because most of the configuration must be done manually. From an industry perspective, the key problems associated with ASIPs are lack of automation breadth and depth. In terms of breadth, the ASIP flow should be thoroughly automated, thus making it useable for the average system designer, and should cover a wide range of configurable parameters and base processors. With respect to depth, ASIP tools should be able to deliver more performance than currently available. From the perspective of academia, the key bottleneck to the automatic configuration of ASIPs is design space exploration (DSE). As better DSE methodologies are developed, the benefits will propagate to the user providing faster, more accurate, and more useable tools, thus serving the needs of the industry.

A typical characteristic of most DSE problems for ASIPs is that the cost of evaluating each point in the design space is very expensive. Each configuration of the processor affects the outcome of the optimization goal in a complex manner that is not easily estimated. Typically, evaluation of a point requires configuration of the processor and compiler flow, and simulation of the processor using the target application. Because each evaluation is so expensive, Monte Carlo-based optimization algorithms are typically not appropriate.

Current research in ASIPs has focused automatic configuration problems for specific parts of the architecture such as the instruction-set. All of these problems can be categorized into one of the following two key topics: how can DSE exploration be improved and how can evaluation time be reduced.

6.1.1. Research Summary

In this dissertation, an approach for DSE exploration is proposed to solve a variety of ASIP configuration problems called *Design Space Modeling.* Design Space Modeling is suitable for optimization problems and for exploring trade-offs between parameters and objectives. With only a small sample of points simulated, this approach uses statistical inference to construct a model of the design space. Using this model, system designers can find an optimal processor to suite their application while avoiding the time consuming process of evaluating all possible architecture configurations manually. Although the focus of this dissertation has been on ASIPs, the proposed approach can be applied to many other optimization problems in other disciplines.

To the best of the author's knowledge, the approach to DSE modeling for ASIPs outlined in this dissertation is a novel approach that is computationally efficient and requires only a limited number of simulations to construct the model. Comparable works were published concurrently by two other research groups ([125] and [126]); however, they focused on general microprocessor modeling rather than ASIPs. Also unique to the approach outlined in this dissertation is that the design space model is constructed using non-parametric statistics (LRM-DSE). This approach can construct the design space model with the speed and transparency of the *Manual Regression Approach* [125] and the automation of the *Artificial Intelligence Approach* [126]. The *Non-parametric Approach* suffers from reduced model accuracy relative to competing approaches; however, this may not affect its ability to be used for optimization because of its effectiveness in identifying overall trends in the design space.

The combination of fast and transparent model construction with a high level of automation makes the *Non-parametric Approach* the most appropriate approach for DSE in high level ASIP architecture design tools. Full automation improves the accessibility of ASIP tools for system designers and allows them to explore a larger variety of designs earlier in the design flow. Faster model construction allows designers to explore a larger variety of architectures earlier. Because architecture DSE is typically performed early in the design flow when many design decisions involve course-grain changes, small reductions in model accuracy are tolerable if automation and modeling speed are improved.

In addition to fast exploration, design space exploration for instruction-set extensions requires fast evaluation of candidate instructions because they must often evaluate a very large number of candidates. Previous methods were too simple in that they did not consider data dependencies within the pipeline. To the best of the author's knowledge, the approach proposed in this dissertation is the first published work to consider data hazards within the pipeline while evaluating candidate custom instructions. This approach improves the quality of instruction-set selection over previous methods while maintaining constant time run-time complexity. In spite of this improvement over previous approaches, the proposed approach continues to suffer from significant error. As a consequence, future work should be directed at further improvements in accuracy.

In summary, the main contributions of this thesis are as follows:

- A fast design space exploration methodology for ASIPs that models the design space through statistical inference by dramatically reducing simulation time (100x reduction for the 2-level cache problem). This approach is called *Design Space Modeling.*
- A variant of the Design Space Modeling approach that constructs the model using non-parametric statistics. This approach is called LRM-DSE. A qualitative comparison is made between LRM-DSE and two other approaches developed concurrently by other research groups.
- LRM-DSE is applied to cache tuning with results comparable to that of state-of-the-art methods developed specifically for cache. In fact, architectures configured using LRM-DSE improved power dissipation results to within 2-3% of the optimal.
- A fast performance estimation approach for evaluating configuration of instruction-set extensions. This approach improves the effectiveness of Instruction-set Selection and Enumeration by as much as 8.4% per custom instruction by considering data hazards. Even with the additional processing necessary to consider data hazards, the proposed estimator maintains constant run-time complexity.

The modeling approach proposed in this dissertation is a promising approach. The interest and effort made by other research groups in developing similar work helps to validate the importance of the research direction. Further, the robustness of the approach is supported by the success of other research groups that used a variety of implementations and experimental platforms. It is expected that the quality of the modeling will improve with ongoing research and it will be applied to a wider variety of architectural features. Not only does this methodology serve as a noise-resistant alternative to traditional heuristics for optimization but it can also be combined with other heuristics to improve run-time and quality. One approach, for example, could use a coarse-grain design space model to serve as a "map" to guide the path-oriented heuristic. Another approach could use design space modeling as a method for evaluating an extended neighborhood during hill-climbing approaches rather than just immediate neighbors.

Acceptance of this approach by the industry will require a more thorough understanding of when the approach works well in comparison to traditional heuristics approaches. Further, it will require a deeper understanding of when each variant of the approach is most effective. Once this has been determined by the research community, this approach will likely have an impact on the industry at some level.

The experimental approach used in this dissertation was in line with approaches taken in industry. All experiments used state-of-the art simulation tools and compilers. All experiments targeted the most relevant questions for each topic and were conducted on commonly used benchmark suites. In spite of this, however, several important questions were not addressed in the dissertation. For example, a quantitative comparison between the various other model construction approaches would have been beneficial. Because the DSE approaches developed in this dissertation were done so concurrently with the other research groups, a description of these approaches was not available in the literature until recently. Thus, it was not possible to perform a quantitative comparison.

Another major limitation of the experimental approach was an incomplete quantitative comparison of the various sampling approaches. In the context of LRM-DSE, both evenly-distributed sampling and random sampling were compared; however, there may be benefit to regional, weighted, and adaptive sampling policies.

6.1.2. Limitations

It has been suggested that design space modeling is computationally less expensive than path-oriented heuristics. Evidence for this is based on qualitative cost measures but a thorough comparison is still necessary. Research may show that, in some circumstances, the number of points to be evaluated in a sample set is greater than the number of points evaluated during a path-oriented search.

In LRM-DSE, the specific variation of design space modeling proposed here, there are a few known limitations. Because LRM-DSE is based on LOESS, it is most effective on smooth functions. With respect to ASIP configuration, LRM-DSE works best with memory-related parameters. In this dissertation, the cache hierarchy was configured; its parameter set is likely a very good candidate for LRM-DSE. However, many other parameters such as those with discrete values are incompatible with LRM-DSE.

6.1.3. Future Work

6.1.3.1. Short Term Direction

The immediate direction for future work is to improve the quality of design space modeling through gradual improvements of the LRM-DSE statistics. This will require a careful quantitative comparison between the different approaches developed to date so that the best features can be drawn from each.

In addition, a study should be conducted to determine when design space modeling is suitable and when heuristics are suitable. From this, hybrid solutions can be developed that combine both approaches.

6.1.3.2. Long Term Direction

Due to the size of integrated circuits possible with sub-100 nanometer processes, Multiprocessor Systems-on-Chip (MP-SoCs) have become a more prevalent design style where the system has multiple processors. Such systems provide a mechanism for real task-level concurrency thus improving the performance necessary for many of the real-time multimedia and communications applications prevalent today. In addition, parallelism allows for voltage and frequency scaling that saves energy when compared to a single processor solution with the same computational burden. MP-SoCs often have a variety of processor types (heterogeneous MP-SoCs) with a variety of different memory types throughout, thus creating a platform with components specifically tailored to the tasks assigned to them.

Using ASIPs (and configurability in general) in MP-SoCs is a natural next step for improved performance over the use of general-purpose processors and for faster time-to-market over the use of custom blocks. Configurability for MP-SoC does come with a price, however; to add such capabilities introduces additional levels of complexity.

The shift towards an increased number of processors on a system will require improved concurrency models, more user-friendly development and debugging environments, improved system modeling, faster simulation, higher-level programming abstractions to aid in platform-independent software development, and more efficient methodologies for finding optimal configurations of a highly-configurable system. These design problems are collectively part of an emerging methodology referred to as Electronic System-Level (ESL) design. The need for sophisticated ESL tools goes far beyond the current offerings of the electronic design automation (EDA) industry. Once developed, ESL design would allow companies to produce better products faster.

A good direction for future work could be to develop ESL methodologies that incorporate configurability and extensibility in MP-SoCs. The primary focus of this should be on ASIPs; however, configurable memory and field programmable gate arrays (FPGAs) must also be addressed. If successful, this goal would allow designers to develop high performance MP-SoCs without the need for the costly development of fixed-logic or ASICs. As a consequence, more companies would be able to produce more competitive products while reducing their time-to-market window.

The contributions of this dissertation help improve the automation and usability of design space exploration for ASIP configurability. In doing so, it helps to place future ASIPs within the reach of more system designers. This will allow system designers to create products capable of meeting the demands of future communications, medical, entertainment, and security applications. It will also contribute to the exceptional design automation challenges that lay ahead as we tackle heterogeneous multiprocessor computing.

References

[1] G. E. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, April 19, 1965.

[2] Semiconductor Industry Association, International Technology Roadmap for Semiconductors - ITRS 2005 [Online], Available: http://www.itrs.net/Links/2005ITRS/Home2005.htm.

[3] K. Keutzer, S. Malik and A. R. Newton, "From ASIC to ASIP: The Next Design Discontinuity", *Proc. of ICCD*, 2002.

[4] I. Kuon, J. Rose, "Measuring the Gap Between FPGAs and ASICs," in *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, Vol.26, No.2, February 2007.

[5] D. Goodwin, D. Petkov, "Automatic Generation of Application Specific Processors", *CASES*, 2003.

[6] R. E. Gonzalez, "Xtensa: A Configurable and Extensible Processor," IEEE Micro, vol. 20(2), pp. 60-70, Mar. 2000.

[7] ARC International Corp., http://www.arc.com.

[8] MIPS Technologies, http://www.mips.com.

[9] Altera Corp., http://www.altera.com.

[10] M. V. Wilkes, "The Memory Gap and the Future of High Performance Memories," in *ACM SIGARCH Computer Architecture News*, 2001.

[11] Actel Semiconductor Ltd. [Online], Available: http://www.actel.com.

[12] Xilinx Corp. [Online], Available: http://www.xilinx.com.

[13] J.R. Hauser, J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," proc. of *IEEE Synposium on FPGAs for Custom Computing Machines*, 1997.

[14] S. Hauck, T.W. Fry, M.M. Hosler, J.P. Kao, "The chimaera reconfigurable functional unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 12 Issue 2, pages 206-217.

[15] K. Keutzer, "Overview of Configurable Architecture", lecture notes [Online]. Available: http://www.cs.berkeley.edu/~culler/cs252-s02/slides/kk-reconfig-over-2-02.pdf

[16] I. Verbauwhede, "Lecture 9: Domain Specific Processors," lecture notes [Online], Available: http:// www.esat.kuleuven.ac.be/~iverbauw/Courses/HJ94/lectures06/Les9_2slidespp_2006_Viterbi.pdf

[17] M. Gries, K. Keutzer (editors). "Building ASIPs: The Mescal Methodology," Springer, 2005.

[18] J. Sato, M. Imai, T. Hakata, A. Alomary, and N. Hikichi, "An Integrated Design Environment for Application Specific Integrated Processors." In Int. Conference on Computer Design (ICCD), pages 414-417, Oct. 1991.

[19] Improv Systems [Online], Available: http://www.improvsys.com

[20] T. V. K. Gupta, R. E. Ko, and R. Barua, "Compiler-directed Customization of ASIP Cores," *proc. of CODES*, 2002.

[21] A. De Gloria and P. Faraboshci, "An Evaluation System for Application Specific Architectures," *proc. of the 23th Annual International Workshop on Microarchitecture and Microprogramming*, 1990. IEEE Computer Society TC-MICRO and ACM SIGMICRO.

[22] O. Wahlen, T. Glokler, A. Nohl, A. Hoffmann, R. Leupers, and H. Meyr, "Application Specific Compiler/Architecture Codesign: A Case Study," *proc. of LCTES'02-SCOPES'02*, 2002.

[23] P. Mishra, N. Dutt, A. Nicolau, "Functional Abstraction driven Design Space Exploration of Heterogeneous Programmable Architectures," *proc. of ISSS*, 2001.

[24] M. Arnold and H. Corporsal, "Designing Domain-Specific Processors," *proc. of CODES*, 2001.

[25] D. Fischer, J. Teich, R. Weper, U. Kastens, and M. Theis, "Design Space Characterization of Architecture/Compiler Co-Exploration," *proc. of CASES*, 2001.

[26] V. Kathail, shail Aditya, R. Schreiber, B. R. Rau, D. C. Cronquist, and M. Sivaraman. Pico: Automatically designing custom computers. Computer, 35(9):39–47, Sept 2002.

[27] M. K. Jain, M. Balakrishnan, A. Kumar, "ASIP Design Methodologies: Survey and Issues," *proc. of vlsid, p. 76, The 14th International Conference on VLSI Design (VLSID '01)*, 2001.

[28] M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *proc. of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, 1988.

[29] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures", *proc. of 12th FPGA*, 183-189, 2004.

[30] CoWare Inc. [Online], Available: http://www.coware.com.
[31] Stretch Inc. [Online], Available: http://www.stretchinc.com.
[32] Synfora Inc. [Online]. Available: http://www.synfora.com.
[33] Mimosys [Online], Available: http://www.mimosys.com.
[34] A. Hoffmann and T. Kogel and A. Nohl and G. Braun and O. Schliebusch and O. Wahlen and A. Wieferink and H. Meyr. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. *IEEE Transactions on Computer-Aided Design*, 20(11):1338–1354, Nov. 2001.
[35] ACE [Online], Available: http://www.ace.nl/
[36] V. Kathail, S. Aditya, R. Schreiber, B.R. Rau, D.C. Cronquist, and M. Sivaraman, "Pico: automatically designing custom computers," in *Computer*, 35, 9 (Sept. 2002), 39–47.
[37] Integrated System Design Lab, Osaka University [Online], Available: http://vlsilab.ics.es.osaka-u.ac.jp/.
[38] Express Users Manual [Online],
Available: http://www.ics.uci.edu/~express/pubs/EXPRESSION_manual.pdf
[39] M. Schlett. "Trends in embedded microprocessor design," *IEEE Computer*, 31(8):44-49, August 1998.
[40] J. Turley, "The Two Percent Solution," *Embedded Systems Design* [Online],
Available: http:// www.embedded.com/story/OEG20021217S0039.
[41] ARM7TDMI Technical Reference Manual, ARM Ltd. [Online],
Available: http://infocenter.arm.com.
[42] ARM Ltd. [Online], Available: http://www.arm.com.
[43] AMBA 3 Specification, ARM Ltd [Online],
Available: http://www.arm.com/products/solutions/AMBAHomePage.html
[44] Intel StrongARM SA-1110 Data Sheet. 2000.
[45] D. A. Patterson, J. L. Hennessy, "Computer Architecture: A Quantitative Approach," Morgan Kaufmann Publishers, Inc., 2nd addition, 1996.
[46] D. Sima, "The Design Space of Register Renaming Techniques," IEEE Micro, 2000. Micro, Vol. 20 No. 5, pp. 70-83. Sep/Oct 2000.
[47] M. Hind, "Pointer Analysis: Haven't We Solved This Problem Yet?," *Proc. of PASTE'01*, pages 54 – 61, Jun 2001.
[48] M. Evers and T.-Y. Yeh, "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors," in *proc. of the IEEE*, vol. 89, No. 11, Nov. 2001.
[49] J. Davidson and S. Jinturkar, "Improving Instruction-level Parallelism by Loop Unrolling and Dynamic Memory Disambiguation," in *proc. of the 28th Annual International Symposium on Microarchitecture*, 1995.
[50] J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", in *IEEE Trans. on Computers*, C-30, 7 (July 1981). 478-490.
[51] D. W. Wall, "Limits of Instruction-Level Parallelism," Proceedings *of the fourth international conference on Architectural support for programming languages and operating systems*, 1991.
[52] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," *Proceedings of the 19th Annual International Symposium on computer Architecture*, 1992.
[53] D.A. Hodges, H.G. Jackson, and R.A. Saleh, "Analysis and Design of Digital Integrated Circuits: In Deep Submicron Technology," *McGraw-Hill*, 3rd edition (July 2003).
[54] A. P. Chandrakasan, R. Broderson, "Low Power Digital CMOS Design," *Kluwer Academic Publishers*, 1995, p. 473-484.
[55] N.S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan, "Leakage Current: Moore's Laws Meets Static Power," *Computer*, 36(12):68-75, Dec. 2003.
[56] T. Claasen, "High Speed: Not the Only Way to Exploit the Intrinsic Computational Power of Silicon," *proc. of Int. Solid State Circ. Conf. 97*, Digest of Tech. Papers, 1997.
[57] T. Mudge, "Power: A First-Class Architectural Design Constraint," *Computer*, vol. 34, no. 4, April 2001, pp. 52-57.

[58] D.A. Hodges, H.G. Jackson, and R.A. Saleh, "Analysis and Design of Digital Integrated Circuits: In Deep Submicron Technology," McGraw-Hill, 3rd edition (July 2003).

[59] R. Puri, L. Stok, S. Bhattacharya, "Keeping Hot Chips Cool," *Proc. of Design Automation Conference*, 2005.

[60] D. Marculescu, "On the Use of Microarchitecture-Driven Voltage Scaling," *Proc. of ISCA*, 2000.

[61] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark, "Formal Online Methods for Voltage/Frequency Control in Multiple Clock Domain Microprocessors," *proc. of ASPLOS*, 2004.

[62] S. K. Srinivasan, J. C. Park, and V. J. Mooney III, "Combining Data Remapping and Voltage/Frequency Scaling of Second Level Memory for Energy Reduction in Embedded Systems," *in Embedded Systems Microelectronics Journal*, 2003.

[63] P. Shaumont, B.-C. C. Lai, W. Qin, and I. Verbauwhede, "Cooperative Multithreading on Embedded Multiprocessor Architectures Enables Energy-scalable Design," *proc. of DAC*, 2005.

[64] E. Talpes and D. Marculescu, "Increased Scalability and Power Efficiency by Using Multiple Speed Pipelines," *in the proceedings of ISCA*, 2005.

[65] R. I. Bahar and S. Manne, "Power and Energy Reduction ia Pipeline Balancing," *Proc. of the 28th Annual Symposium on Computer Architecture*, 2001.

[66] Y.-T. Hung, T.-Y. Lin, R.-G. Chang, "Power-Aware Compilation with Architectural Support and Instruction Scheduling," *proc. of 11th WHPC*, 2005.

[67] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose Loops Sink Chips," *Proc. of HPCA*, 2002.

[68] S. Ghiasi, J. Casmira, and Grunwald, "Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption," *proc. of Workshop on Complexity Effective Design*, 2000.

[69] L. Benini, A. Bogliolo, and G. De Micheli, "A Survey of Design Techniques for System-Level Dynamic Power Management," *IEEE Transactions on VLSI Systems*, Vol. 8, No. 3, June 2000.

[70] T. Glökler, A. Hoffman, and H. Meyr, "Methodical Low-Power ASIP Design Space Exploration," *Journal of VLSI Signal Processing 33*, 229-246, 2003.

[71] P. Michaud, A. Seznec, and S. Jourdan, "Exploring Instruction-Fetch Bandwidth Requirements in Wide-Issue Superscalar Processors," *proc. of PACT*, 1999.

[72] D. Parkikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power Issues Related to Branch Prediction," *proc. of HPCA*, 2002.

[73] M. Ekpanyapong, P. Korkmaz, and H. S. Lee, "Choice Predictor for Free," *Proc. of the Ninth Asia-Pacific Computer Systems Architecture Conference*, 2004.

[74] S.-T. Pan, K. So, J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", proc. of ASPLOS V, 1992.

[75] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proc. of the 8th Annual International Symposium on Computer Architecture*, 1981.

[76] D. Chaver, L. Pinuel, M. Prieto, and F. Tirado, "Branch Prediction on Demand: an Energy-Efficient Solution," *proc. of the International Symposium on Low Power Electronics and Design (ISLPED)*, 2003.

[77] A. Malik, W. Moyer, and D. Cermak,"A low power unified cache architecture providing power and performance flexibility," in *Int. Symposium on Low Power Electronics and Design (ISLPED),* 2000.

[78] C.-L. Su and A. M. Despain, "Cache Design Trade-offs for Power and Performance Optimization: A Case Study," *proc. of ISLPED*, 1995.

[79] M. Powell al. "Reducing Set-Associative Cache Energy via Way Prediction and Selective Direct-Mapping," *proc. of International Symposium on Microarchitecture*, 2001.

[80] D. H. Albonesi, "Selective Cache Ways, On-Demand Cache Resource Allocation," *Proc. of International Symp. on Microarchitecture*, 1999.

[81] K. Ghose and M. B. Kamble, "Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation," *proc. of ISLPED*, 1999.

[82] A. Baniasadi, A. Moshovos, "Branch Predictor Prediction: A Power-Aware Branch Predictor for High-Performance Processors," *Proc. of the 20th International Conference on Computer Design*, 2002.

[83] J. Aragan, J. Gonzalez, and A. Gonzalez, "Power-Aware Control Speculation Through Selective Throttling," *proceedings of Intl. Symposium on High-Performance Computer Architecture*, 2003.

[84] K. Vivekanandarajah, T. Srikanthan, and C. T. Clarke, "Profile Directed Instruction Cache Tuning for Embedded Systems," *proc of ISVLSI*, 2006.

[85] C. Chakrabarti, "Cache Design and Exploration for Low Power Embedded Systems," *proc. Int. Conference of Performance, Computomg, and Commun.ications*, 2001.

[86] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Trans. Comput.*, 38(12):1612-1630, 1989.

[87] A. Ghosh and T. Givargis, "Analytical Design Space Exploration of Caches for Embedded Systems," *proc. of DATE*, 2003.

[92] Givargis, T. and Vahid, F. Platune: a tuning framework for system-on-a-chip platforms. In IEEE Trans. Comp. Aided Design of Integrated Circuits and Systems (TCAD), 21, 11 (Nov. 2002).

[93] M. Palesi and T. Givargis, "Multi-objective design space exploration using genetic algorithms," in *Intern. Workshop on Hardware/Software Co-design (CODES)*, 2002.

[94] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache for low energy embedded systems," in *Trans. on Embedded Comp. Sys*, 4, 2 (2005), 363-387.

[95] A. Gnosh and T. Givargis, "Cache optimization for embedded processor cores: an analytical approach," in *Intern. Conf. on Computer-aided Design (ICCAD)*, 2003.

[96] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architecture," in *IEEE/ACM International Symposium on Microarchitecture (Micro-33)*, 2000.

[97] A. Gordon-Ross, F. Vahid, and N. Dutt, "Automatic tuning of two-level caches to embedded applications.," in *Design, Automation, and Test in Europe (DATE)*, 2004.

[98] A. Gordon-Ross and F. Vahid, "Fast configurable-cache tuning with a unified second-level cache," in *Symposium on Low Power Electronics and Design (ISLPED)*, 2005.

[99] P. Viana, A. Gordon-Ross, E. Keogh, E. Barros, and F. Vahid, "Configurable cache subsetting for fast cache tuning," in *Design Automation Conference (DAC)*, 2006.

[96] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *11th Int. Conf. Application-Specific Systems*, 1997.

[97] K. Keutzer, A. Newton, J. Rabaey, and A. Sangiovanni-Vincetelli, "System-level design: orthogonalization of concerns and platform-based design," in *IEEE Transactions on Computer-Aided Deisgn of Integrated Circuits and Systems*, vol. 19, No. 12, December 2000.

[98] D. Bruni and A.B.L.Benini, "Statistical design space exploration for application-specific unit synthesis," In *38th Design Automation Conference (DAC*, pages 641-646, 2001.

[99] S Kirkpatrick, CD Gelati Jr, MP Vecchi, "Optimization by Simulated Annealing," in *Science*. Vol. 220, No. 4598, pp. 671-680, 1983.

[100] M. Gries, "Methods for evaluating and covering the design space during early design development," in *Technical Report UCB/ERL M03/32*, Electronics Research Lab, Univ. of California Berkeley (2003)

[101] M. Palesi and T. Givargis, "Multi-objective design space exploration using genetic algorithms," in *10th International Symposium on Hardware/Software Codesign(CODES)*, pages 67-72, May 2002.

[102] J. Sanghavi, A. Wang, "Estimation of Speed, Area, and Power of Parameterizable, Soft IP", in *proc. of DAC*, 2001.

[103] W. Fornaciari, D. Sciuto, C. Silvano, V. Zaccaria, "A sensitivity-based design space exploration methodology for embedded systems, Design Automation for Embedded Systems," Kluwer Academic Publishers 7 (1-2) (2002) 7-33.

[104] T. Givargis, J. Henkel, F. Vahid, "Interface and cache power exploration for core-based embedded system design," *proc. of International Conference on Computer-Aided Design (ICCAD)*, 1999, pp. 270-273.

[105] G. Snider, "Spacewalker: automated design space exploration for embedded computer systems," tech. report HPL-2001-220, Hewlett-Packard Laboratories, Palo Alto, California, 2001.

[106] Y. Fei, S. Ravi, A. Raghunathan, N. K. Jha, "A Hybrid Energy-Estimation Technique for Extensible Processors," in *IEEE TCAD*, Vol. 23, No. 5, May 2004.

[107] R. A. Uhlig and T. N. Mudge, "Trace-Driven Memory Simulation: A Survey," in *ACM Computing Surveys*, Vol,. 29, No. 2, June 1997.

[108] A. Peymandoust, L. Pozzi, P. Ienne, and G. De Micheli, "Automatic Instruction Set Extension and Utilization for Embedded Processors", in *proc. of ASAP*, 2003.

[109] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "Custom-Instruction Synthesis for Extensible-Processor Platforms", *IEEE TCAD*, VOL. 23, NO. 2, February 2004.

[110] K. Atasu, L. Pozzi, and P. Ienne, "Automatic Application-Specific Instruction-Set Extensions under Microarchitectural Constraints", in *Inter. Journal of Parallel Prog.*, December 2003.

[111] M. Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, and A. Sangiovanni-Vincentelli, "HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform," in *proc. of Int. Symp. Hardware/Software Codesign*, Estes Park, CO, May 2002.

[112] N. Clark, H. Zhong and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization", *proc. of the 36th international Symp. on Microarchitecture* (MICRO-36'03), 2003.

[113] P. Ienne, L. Pozzi, and M. Vuletic, "On the Limits of Processor Specialisation by Mapping Dataflow Sections on Ad-hoc Functional Units", *Technical Report 01/376*, Swiss Federal Institute of Technology Lausanne (EPFL), Computer Science Department (DI), Lausanne, December 2001.

[114] F. Sun, S. Ravi, A. Rahgunathan, and N. K. Jha, "Synthesis of Custom Processors based on Extensible Platforms," *proc. of ICCAD*, 2002.

[115] N. Dutt, K. Choi, "Configurable processors for embedded computing," in *IEEE Computer*, 36(1):120–123.

[116] Hallschmid, P., Saleh, R., "Fast Design Space Exploration using Local Regression Modeling with Application to ASIPs," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (IEEE-TCAD)*, Vol. 27, No. 3, March 2008, pp. 508-515.

[117] P. Hallschmid, R. Saleh, "Automatic Cache Tuning for Energy-Efficiency using Local Regression Modeling," in *Design Automation Conference (DAC)*, June 2007.

[118] Eeckhout et al., "Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox," *IEEE Micro*, vol. 23, no. 5, Sept.-Oct. 2003, pp 26-38.

[119] S. Nussbaum and J. Smith, "Modeling Superscalar Processors via Statistical Simulation," in *Proc. Int'l Conf. Parallel Architectures and Compilation Techniques (PACT 01)*, 2001.

[120] T. Sherwood et al, "Automatically Characterizing Large Scale Program Behavior," in *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 02)*, 2002.

[121] S. Eyerman, L. Eeckhout, and K.D. Bosschere, "The Shape of the Processor Design Space and its Implications for Early Stage Explorations," in *Proc. of 7th WSEAS International Conference on Automatic Control, Modeling, and Simulation*, 2005.

[122] L. C. Lee, D. M. Brooks, "Spatial Sampling and Regression Strategies," in *IEEE Micro Special Issue: Hot Tutorials*, May 2007.

[123] E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, R. Caruana, "Efficiently exploring architectural design spaces via predictive modeling," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[124] B. Lee, B., and D. Brooks, "Illustrative Design Space Studies with Microarchitectural Regression Models," in *Proc. Intl'l Symp. High Performance Computer Architecture (HPCA'07)*, 2007.

[125] B.C Lee and D.M. Brooks. "Accurate and efficient regression modeling for microarchitectural performance and power prediction." in *ASPLOS'06*, October 21-25, 2006.

[126] P. İpek, S.A. Mckee, B.R. de Supinski, M. Shulz, and R. Caruana, "Efficiently Exploring Architectural Design Spaces via Predictive Modeling," in *ASPLOS'06*, October 21-25, 2006.

[127] B.C. Lee, D.M. Brooks, B.R. de Supinski, M. Schulz, K. Singh, and S. A. McKee, "Methods of Inference and Learning for Performance Modeling of Parallel Applications," in *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[128] P. Hallschmid, R. Saleh, "Fast Configuration of an Energy-Efficient Branch Predictor," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Germany, March 2006.

[129] W.S. Cleveland, "Robust locally weighted regression and smoothing scatterplots," in *Journal of the American Statistical Association*, 829-835, 1974.

[130] D. Parkikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan, "Power Issues Related to Branch Prediction," in *Proc. of HPCA*, 2002.
[131] S. McFarling, "Combining Branch Predictors," Digital Equipment Corp., WRL Tech.Note TN-36, 1993.
[132] I.-C.K. Chen, J.T. Coffey, and T.N. Mudge. "Analysis of Branch Prediction via Data Compression," *Proc. of ASPLOS*, 1996.
[133] P. Michaud, A. Seznec, and R. Uhlig, "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," in *Proc. of ISCA*, 1997.
[134] S.Sechrest, C. Lee, and T. Mudge, "Correlation and Aliasing in Dynamic Branch Predictors," *Proc. of ISCA*, 1996.
[135] N. S. Kim, T. Austin, T. Mudge, and D. Grunwald, "Challenges of Architectural Level Power Modeling," Book Chapter from Power Aware Computing, 2001, ed. R. Melhem and R. Graybill.
[136] D. Burger and T.M. Austin, The SimpleScalar Tool Set, Version 2.0. *Computer Architecture News*, June 1997.
[137] Intel StrongARM SA-1110 Data Sheet. 2000.
[138] A. Nene, S. Talla, B. Goldberg, H. Kim, and R. Rabbah, "Trimaran – An Infrastructure for Compiler Research in Instruction Level Parallelism." NYU, 1998.
[139] GNU GCC [Online]. Available: http://gcc.gnu.org
[140] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.specbench.org.
[141] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Proc. of Micro, 1997.
[142] Schimek, M. G. (ed.) (2000) Smoothing and Regression: Approaches, Computation and Application. John Wiley.
[143] C. Loader, *Local regression and likelihood*, Springer, 1999.
[144] NIST Processes Modeling Handbook [Online], Available: http://www.itl.nist.gov/div898/handbook/pmd/pmd.htm
[145] R [Online]. Available: http://www.r-project.org/.
[146] Locfit [Online]. Available: http://cm.bell-labs.com/cm/ms/departments/ sia/project/locfit/
[147] A. Malik, W. Moyer, and D. Cermak, "A Low Power Unified Cache Architecture Providing Power and Performance Flexibility," in *Int. Symposium on Low Power Electronics and Design*, 2000.
[148] C. Zhang, F. Vahid, "Cache Configuration Exploration on Prototyping Platforms," in *IEEE Intern. Workshop on Rapid System Proto.*, 2003.
[149] GNU GCC [Online]. Available: http://gcc.gnu.org
[150] Intel StrongARM SA-1110 Data Sheet. 2000.
[151] Standard Performance Evaluation Corporation. SPEC CPU2000 Benchmarks. http://www.specbench.org.
[152] C. Lee, M. Potkonjak, and W.H. Mangione-Smith, "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proc. of Micro*, 1997.
[153] K. Cooper and L. Torczon, "Engineering a Compiler," *Morgan-Kaufmann*, 2004.
[154] P. Hallschmid and R. Saleh, "Hazard-Aware Performance Prediction for Automatic Instruction-Set Selection," in *IEEE International Symposium on VLSI Design, Automation & Test (VLSI-DAT)*, 2006.
[155] D. Jain, A. Kumar, L. Pozzi, and P. Ienne, "Automatically customising VLIW architectures with coarse grained application-specific functional units", in *Proc.of SCOPES*, 2004.
[156] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, Version 2.0." *Computer Architecture News*, pages 13-25, June 1997.
[157] N. S Kim, T. Kgil, V. Bertacco, T. Austin, and T. Mudge, "Microarchitectural Power Modeling Techniques for Deep Sub-Micron Microprocessors," in *proc. of ISLPED'04*, 2004.