UNIVERSITY OF CALIFORNIA, SAN DIEGO

Application-Tuned Processor Architectures

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
from the Department of Computer Science and Engineering

by

Timothy Sherwood

Committee in charge:

Brad Calder, Chairperson
Fred Chong
Pamela Cosman
Norman Jouppi
Andrew Kahng
Dean Tullsen

2003

Copyright
Timothy Sherwood, 2003
All rights reserved.

The dissertation of Timothy Sherwood is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

2003

Dedications

This dissertation is dedicated to the many friends, family, labmates, and mentors whose hard work, understanding, patience, and support made it possible.

To Professor Brad Calder, who has been a constant source of professional and research guidance, and a near infinite reservoir of enthusiasm. I owe much of my success (both academic and foosball) to his pitbull-like tenacity, his passion for new ideas, and his trust. I will strive to be as good as an advisor to my students as he has been to me.

To the others who have mentored me, including the members of my thesis committee. To Dr. Fred Chong who first introduced me to architecture research and gave me a healthy taste for wacky ideas, Dr. Pamela Cosman, Dr. Andrew Kahng, Dr. Norman Jouppi and Dr. Dean Tullsen for there expert guidance and helpful questions on my committee, Dr. Geoff Voelker for all the words of encouragement and for teaching me how to give a great presentation, and to Dr. George Varghese for his contagious energy and know how.

To my many labmates, co-workers, and friends. To Leeann Bent, Jeff Brown, Lori Carter, Wei Chuang, Jamison Collins, Holly Dail, Greg Hamerly, Yuanfang Hu, Chandra Krintz, Rakesh Kumar, Jeremy Lau, Satish Narayanasamy, Mark Oskin, Glenn Reinman, Mike Semanko, John Seng, Beth Simon, Alan Su, Nathan Tuck, and Eric Tune for all their freindship, foosball, and assistance. To Barbara Kreaseck for her friendship, her laughter, and for both Hillary and the Magic Desk. To Erez Perelman for his tireless work, sense of humor, and habenaros. To Suleyman Sair for his many many hours of hard work, his friendship, his help, his suctioning goalposts to his head, and for staying up late at deadlines even though he wakes up at 4am. To Anissa Elayadi for always being there when I need to let down. To Kip Nalley for always being there if I need to double down. To Kevin Hauck for always being there to lay the smack down. To Laura and Kristina, for always being my little sisters, and to all my hockey friends for pretending not to care about bad goals;

To my parents, who always supported me when I was down, empathized with me when I was upset, taught me when I was slow, fought beside me when I was wronged, believed me when I was sick, and love me always. I can't thank them enough.

While all those people have played a great part in my finishing this dissertation, one person has been there for me every day. From the first day of graduate school to the last day of commencement, she has been my constant companion. Without her unflagging support, her caring, her gentle smile, her cute little ratties (Jub-Jub and Üter), her devotion, her peanut butter cookies, her warmth, and her tenderness, I cannot imagine any of this. For these reasons, and for so many others, I dedicate this dissertation to Mei Liao.

I felt a clearing in my mind
As if my brain had split;
I tried to match it, seam by seam,
But could not make them fit.

The thought behind I strove to join
Unto the thought before,
But sequence ravelled out of reach
Like balls upon the floor.
– Emily Dickinson

TABLE OF CONTENTS

Signature Page . . . iii

Dedication Page . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . x

List of Tables . . . xiii

Acknowledgments . . . xiv

Vita and Publications . . . xvi

Abstract . . . xix

I Introduction . . . 1
A. Embedded Processors . . . 2
1. Customized Processors . . . 3
2. Design Constraints . . . 5
B. Overview of Dissertation . . . 7
1. Program Phase Behavior . . . 8
2. Component Specialization . . . 9
3. Inter-component Tradeoffs . . . 10
4. Architecture-Algorithm Co-exploration . . . 10
5. Processor Adaptation . . . 11

II Analysis of Program Behavior . . . 13
A. Introduction . . . 13
B. Methodology . . . 16
C. Using Basic Block Vectors . . . 17
1. Basic Block Vector . . . 18
2. Basic Block Vector Difference . . . 18
3. Basic Block Difference Graph . . . 20
4. Basic Block Similarity Matrix . . . 22
5. Vector Similarity and IPC . . . 25
D. Clustering . . . 27
1. Clustering Overview . . . 29

2. Phase Finding Algorithm 30
3. Clusters and Phase Behavior 37
E. Finding Simulation Points 39
1. Single Simulation Points 40
2. Multiple Simulation Points 42
F. Related Work 48
G. Summary 52

III Component Specialization 55
A. Introduction 55
B. Uses of Finite State Machine Predictors 58
1. Value, Address, and Load Speculation 58
2. Branch Prediction 58
3. Speculative Threaded Execution 59
4. Cache Management 59
5. Power Control 59
C. Prior Work 59
1. FSM Predictor Implementations 60
2. Automatically Generating Predictors 61
D. Automated Design of a FSM Predictor 62
1. Overview 62
2. Modeling 64
3. Pattern Definition 65
4. Pattern Compression 66
5. Regular Expression Building 67
6. FSM Creation 68
7. Start State Reduction 68
8. Synthesis 70
E. Methodology 71
F. FSM Predictors in General Purpose Processors 72
1. Value Prediction 72
2. Value Prediction Confidence 73
3. Automatically Designing Confidence Predictors 73
4. Results 74
G. Branch Predictors for Customized Processors 76
1. Customized Processors 76
2. Customized Branch Prediction Architecture 79
3. Generating Traces to Train FSM Branch Predictors 81
4. Estimating Area of FSM Predictors 83
5. Branch Prediction Results 84
6. Custom Finite State Machine Examples 87

H. Summary 90

IV Inter-Component Tradeoff 92
A. Introduction 92
B. Design Navigation 96
1. Process Flow for Creating an Application Specific Processor . . 96
2. Using Sherpa to Guide Design 98
C. Implementation 99
1. Overall Approach 100
2. Simple Linear Constraint System 101
3. Data Linearization 104
4. Final Constraint System 106
D. Sub-problem Characterization 108
1. Data and Instruction Cache 109
2. Branch Predictors 114
3. Multiplier Unit 114
4. Core Area 115
E. Putting It All Together 115
1. Running the Overall Solution 116
2. Results 117
F. Related Work 125
1. Configurable Cores 125
2. Design Exploration for Application Specific Processors 126
3. Performance Estimation 127
G. Summary 128

V Algorithmic Level Tradeoff 130
A. Introduction 130
B. Routing in the Backbone 132
1. Backbone Router Model 133
2. Routing Requirements 135
3. Our Approach 136
C. Network Processors 136
1. Independent Processors with Shared Memory 137
2. Pipeline of Processors 139
3. Our Solution: Pipelined Wide Word Memory 140
D. Memory Architecture 141
1. Motivation for Wide Word Memory Architectures 141
2. A High Throughput Pipelined Memory 142
3. Characterization of Architecture 145
4. Programmer's View of Memory 147

5. Mapping an Algorithm to Wide Words . . . 148
6. Related Architectures . . . 149
E. Co-Exploring Network Algorithms and Memory Design . . . 150
1. Algorithm and Memory Design Co-Exploration . . . 151
2. Network Algorithms . . . 153
3. Combined Analysis . . . 166
F. Summary . . . 169

VI Adaptive Architectures . . . 171
A. Introduction . . . 171
B. Related Work . . . 174
C. Methodology . . . 176
D. Phase Capture . . . 177
1. Phase-Based Behavior . . . 177
2. Tracking Phases by Executed Code . . . 179
3. Capturing the Code Profile . . . 182
4. Forming a Footprint . . . 183
5. Classifying a Footprint to a Phase ID . . . 186
6. Per-Phase Performance Metric Homogeneity . . . 190
E. Phase Prediction . . . 192
1. Markov Predictor . . . 193
2. Run Length Encoding Markov Predictor . . . 195
3. Predictor Comparison . . . 195
F. Applications . . . 198
1. Frequent Value Locality . . . 198
2. Dynamic Data Cache Size Adaptation . . . 200
3. Dynamic Processor Width Adaptation . . . 203
G. Summary . . . 206

VII Summary and Future Work . . . 209
A. In This Disseration . . . 209
1. Phase Behavior in Programs . . . 210
2. Automatic FSM predictor design . . . 212
3. The tradeoff of different components . . . 214
4. Architecture-Algorithm Co-exploration . . . 215
5. Processor Adaptation . . . 217
B. Algorithms and Computer Architecture . . . 220

Bibliography . . . 222

LIST OF FIGURES

II.1 Time Varying behavior of the program `wave` 15

II.2 Example of capturing the Basic Block Vectors 19

II.3 The basic block difference graphs. 21

II.4 Basic block similarity matrix for the programs `gzip` and `bzip`. 25

II.5 Time varying graph for `gzip-graphic`. 26

II.6 Cluster graph for `gzip-graphic`. 26

II.7 Comparison of similarity with random projection 27

II.8 Time varying graph for `gcc-166`. 28

II.9 Cluster graph for `gcc-166`. 28

II.10 Correlating the vector difference with the difference in IPC seen for two intervals. 29

II.11 Motivation for random projection down to 15 dimensions (D=15). 33

II.12 Projection and Clusters for gzip, 1037 total points 34

II.13 Plot of average IPC variance and max IPC variance versus the BIC. 38

II.14 Single Simpoint Comparison Barchart 40

II.15 Multiple simulation point results. 45

II.16 Average error results for SPEC 2000 48

III.1 State machine predictor optimization example 69

III.2 Comparison of SUD versus custom confidence predictors 77

III.3 Customized branch prediction architecture. 80

III.4 Area versus number of states for custom predictors 82

III.5 Branch predictor accuracy–area tradeoffs 85

III.6 Simple state machine predictor example 88

III.7 A more complex state machine predictor 88
IV.1 High level view of Sherpa design flow 97
IV.2 Overview of the Sherpa design guide implementation. 97
IV.3 Simple linear function creation for tradeoffs 102
IV.4 Peicewise Linear Model for tradeoffs 102
IV.5 Translated Peicewise Linear Model 103
IV.6 Constraint system for a piece-wise linear sub-subproblem model. 107
IV.7 Area-delay tradeoffs for a single cache configuration. 111
IV.8 Instruction cache tradeoffs 112
IV.9 Data cache tradeoffs 113
IV.10 Brach prediction tradeoffs 113
IV.11 Verification of Performance Estimator. 116
IV.12 Overall area/perfomance tradeoffs 118
IV.13 Tradeoffs made for the application `jpeg`. 121
IV.14 Overall area reduction for `gs` with custom FSM 122
IV.15 Diagram of how performance and area interact 123
IV.16 Performance versus area tradeoffs as analyzed by Sherpa 125
V.1 Several ways of building network processors 138
V.2 Pipelined memory diagram 143
V.3 Programming model for pipelined memory 143
V.4 Quantifying the overhead and benefit of pipeline memory stages. 146
V.5 Memory bandwidth versus size of word accessed 146
V.6 Hardware and Algorithm tradeoff interaction 152
V.7 IP lookup table before and after controlled prefix expansion. . . 154

V.8 Area-Performance tradeoffs for each algorithm 158
V.9 Example state machine generated by the aho-corasick algorithm. 161
V.10 Word width versus network bandwidth 162
V.11 Examination of the best configurations found under different assumptions. 167
VI.1 Application behavior changing over time 178
VI.2 Phase classification architecture. 180
VI.3 Accumulator table occupancy over time 184
VI.4 Determining an adequate number of counters 186
VI.5 Phase coverage 187
VI.6 Sensitivty versus Accuracy in the phase tracker 189
VI.7 Phase transition activity 194
VI.8 Phase prediction architecture 196
VI.9 Phase ID Prediction Accuracy. 197
VI.10 Phase aware value locality study 200
VI.11 Data cache reconfiguration 202
VI.12 The IPC of `gcc` as split up by the phase detector. 204
VI.13 Processor Width Adaptation. 205

LIST OF TABLES

II.1 Baseline Simulation Model. 17

II.2 Single simulation points for SPEC 2000 benchmarks. 43

II.3 Multiple simulation points for SPEC 2000 benchmarks. 44

IV.1 Area versus Delay optimization for cache 111

IV.2 Quantifying the independence of subproblems. 119

VI.1 Baseline Simulation Model for Phase Tracker. 177

VI.2 Phase Homogeneity for `gzip` and `gcc` 191

Acknowledgments

The text of Chapter II is in part a reprint of the material as it appears in the proceedings of the Tenth Annual International Conference on Architectural Suport for Programming Languages and Operating Systems. The dissertation author was the primary researcher and author and the co-authors listed on this publication [122] directed, supervised, or assisted in the research which forms the basis for Chapter II.

The text of Chapter III is in part a reprint of the material as it appears in the proceedings of the 28th Annual International Symposium on Computer Architecture. The dissertation author was the primary researcher and author and the co-author listed on this publication [118] directed and supervised the research which forms the basis for Chapter III.

The text of Chapter IV is in part a reprint of the material as submitted to the proceedings of the 30th Annual International Symposium on Computer Architecture. The dissertation author was the primary researcher and author and the co-authors involved in the submission directed and supervised the research which forms the basis for Chapter IV.

The text of Chapter V is in part a reprint of the material as it appears in the proceedings of the 30th Annual International Symposium on Computer Architecture. The dissertation author was the primary researcher and author and the co-author listed on this publication [127] directed and supervised the research which forms the basis for Chapter V.

The text of Chapter VI is in part a reprint of the material as it appears in the proceedings of the 30th Annual International Symposium on Computer Architecture. The dissertation author was the primary researcher and author and the co-author listed on this publication [126] directed, supervised, and assisted in the research which forms the basis for Chapter VI.

VITA

August 3, 1976	Born, Long Beach, California
1998	B.S. in Computer Science University of California, Davis
1997–1998	Student Researcher, University of California, Davis
1999	Internship, HP Labs, Palo Alto
2000	M.S. in Computer Science University of California, San Diego
2000	Internship, Compaq Western Research Labs, Palo Alto
2003	Doctor of Philosophy University of California, San Diego

PUBLICATIONS

Timothy Sherwood, George Varghese, and Brad Calder. A Pipelined Memory Architecture for High Throughput Network Processors, *In the proceedings of the 30th Annual Intl. Symposium on Computer Architecture* (ISCA 2003), June 2003. San Diego, California

Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase Tracking and Prediction, *In the proceedings of the 30th Annual Intl. Symposium on Computer Architecture* (ISCA 2003), June 2003. San Diego, California

Suleyman Sair, Timothy Sherwood, and Brad Calder. A Decoupled Predictor-Directed Stream Prefetching Architecture, *IEEE Transactions on Computers, Vol 52 No 5,* May 2003

Satish Narayanasamy, Timothy Sherwood, Suleyman Sair, Brad Calder, and George Varghese. Catching Accurate Profiles in Hardware , In the proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA-9), February 2003. Anaheim, California.

Timothy Sherwood, Erez Perelman, Greg Hamerly and Brad Calder. Automatically Characterizing Large Scale Program Behavior, *In the proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS 2002), October 2002. San Jose, California

Suleyman Sair, Timothy Sherwood, and Brad Calder. Quantifying Load Stream Behavior. In the proceedings of the *8th International Symposium on High-Performance Computer Architecture* (HPCA-8) February 2002. Cambridge, Massachusetts

Timothy Sherwood and Brad Calder. Patchable Instruction ROM Architecture. In the proceedings of *the International Conference on Compiler, Architecture, and Synthesis for Embedded Systems* (CASES 2001) November 2001. Atlanta, Georgia

S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. Volume: 20, Issue: 11, Page(s): 1355 -1371, Nov 2001

Timothy Sherwood, Erez Perelman and Brad Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In the proceedings of *The Intl. Conference on Parallel Architectures and Compilation Techniques* (PACT 2001) Sept, 2001. Barcelona, Spain

Timothy Sherwood and Brad Calder. Automated Design of Finite State Machine Predictors for Customized Processors. In *The proceedings of the 28th Annual Intl. Symposium on Computer Architecture* (ISCA 2001) June, 2001. Goteborg, Sweden

S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth Sensitive Code Generation in a Custom Embedded Accelerator Design System. In the proceedings of *The 5th International Workshop on Software and Compilers for Embedded Systems* (SCOPES 2001) March, 2001. St. Goar, Germany.

Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-Directed Streaming Buffers. In the proceedings of *The 33rd Annual International Symposium on Microarchitecture* (Micro-33), December 2000. Monterey, California.

Timothy Sherwood and Brad Calder. *Loop Termination Prediction*. In the proceedings of *The Third International Symposium on High Performance Computing* (ISHPC2K), October 2000. Toyko, Japan.

Timothy Sherwood and Brad Calder. Toolblocks: An Infrastructure for the Construction of Memory Hierarchy Analysis Tools. In the proceedings of *The European Conference on Parallel Computing* (Euro-Par), Muinch, Germany. August 2000.

Bryan Talbot, Timothy Sherwood, and Bill Lin. Cached IP Lookup for Terabit Speed Routers. In *The proceedings of the IEEE Global Communications Conference* (GlobeCom), December 1999.

Mark Oskin, Frederic T. Chong, and Timothy Sherwood. ActiveOS: Virtualizing Intelligent Memory. *In the Proceedings of the International Conference on Computer Design* (ICCD), Austin, Texas. October 1999.

Timothy Sherwood, Brad Calder, and Joel Emer. Reducing Cache Misses Using Hardware and Software Page Placement. In *The Proceedings of the International Conference on Supercomputing* (ICS), June 1999.

Mark Oskin, Frederic T. Chong, and Timothy Sherwood. Active Pages: A Model of Computation for Intelligent Memory. In The 1998 International Symposium on Computer Architecture (ISCA), Barcelona, Spain.

Mark Oskin, Frederic T. Chong, Aamir Farooqui, Timothy Sherwood, and Justin Hensley. Low Power Design of Page-Based Intelligent Memory. In the *Workshop on Power-Driven Microarchitecture* held with the 1998 International Symposium on Computer Architecture (ISCA). Barcelona, Spain. June 1998

Mark Oskin, Timothy Sherwood, Justin Hensley, Sinclair Yeh, and Frederic T. Chong. Sharing Data in Page-Based Intelligent Memory. In *The Seventh Workshop on Scalable Shared-memory Multiprocessors* held with the 1998 International Symposium on Computer Architecture (ISCA). Barcelona, Spain. June 1998

ABSTRACT OF THE DISSERTATION

Application-Tuned Processor Architectures

by

Timothy Sherwood
Doctor of Philosophy in Computer Science and Engineering
University of California, San Diego, 2003
Professor Brad Calder, Chair

The performance and resource requirements of every application are unique. Yet most processors are designed to do well, not on a particular application, but instead on the average case over a range of applications. Processors are typically designed around a broad suite of applications so that one piece of hardware can be used in many different contexts. In contrast to this, the continuing expansion of the embedded market has created a significant demand for low-cost high-performance computing solutions that can be buried deep inside products. These embedded processors are typically responsible for the execution of only a small number of applications, which means that it is possible to tune the design of the processor to the needs of the application for which it will be used.

Embedded processors not only typically execute a small number of applications, they are also often integrated into the same physical chip as many other components. This level of integration means that the processors are now being sold not as physical silicon, but rather as descriptions which can be combined with other design elements before fabrication. This change in the way that processors are sold provides us with an opportunity to take into consideration specific application behaviors during the physical design of the processor. By customizing a processor to the needs of an application, we can provide faster

designs that take less area. In order to make this possible, the customization needs to be realized with a minimum of additional engineering design effort. To this end, this dissertation focuses on novel automated techniques for analyzing program behavior, customizing portions of the processor to an application, making intelligent design tradeoffs, and examining the effect of algorithmic choices on the circuit level realities of their implementation.

I

Introduction

Every program that runs on a processor has different performance and resource needs, and by tailoring an embedded processor architecture to the task at hand we can increase performance, reduce power consumption, and improve area efficiency. The common methodology governing the design of processors today is to tune the design of a processor to work well over a very broad set of applications, for example the entire SPEC benchmark suite [28]. As an end user or a system designer hoping to make use of a general purpose commodity processor, one needs to hope that their application has behavior that is well represented in the suite of benchmarks that were used to design the processor.

For example, if my program has very difficult to predict branches, but the suite of applications that the processor has been designed around have very easy to predict branches, then the default branch prediction mechanisms will probably be inadequate. However, if the engineer of the branch predictor was able to hardwire in the behavior of the branches in a targeted application then it has the potential to outperform any generic scheme. While modifying the physical design of a processor to take advantage of some program behavior may not be realistic in the general purpose computing market, it may be possible for those processors which are embedded into products.

I.A Embedded Processors

Embedded computing has long been the meat and potatoes of the computing world. For many years embedded processors have been used in phones, printers, routers, automobiles, practically everything we depend on. With the proliferation of system level integration and the popularity of smart devices, network appliances, and portable electronics we have seen even further demand for cheap, low power processing ability. In 1998, 2.5 billion embedded processors were installed, compared to 100 million general purpose machines [113]. By 2002, the market for network enabled devices will outgrow the general purpose computing platform market [113].

Systems designers are increasingly asked to deliver more complex, more powerful, smaller and cheaper systems in constantly shrinking design cycles. As Moore's law marches on, the transistor budget available for a given design continues to increase by an incredible 58% per year, but the amount of design area that a single designer can use is only growing at 21% per year, creating a design gap [61].

The number of transistors on a single chip is increasing to the point that in many systems, processors now share the die with other components or even other processors. For example, a cell phone system typically has two or more processors. One may be used for processing user interfaces and one may be used for the underlying embedded control. In addition to multiple processors, a single chip may even be integrated with a digital signal processor, memory, or other custom components. To support this model it is now common for vendors to sell descriptions of processor cores rather than actual silicon itself.

A major part of the design cycle is the development of processing elements that can provide the necessary computational power within constraints on power, size and design time. Up until very recently there have been very few

choices for the embedded system designer. The designer can choose to design a custom ASIC for the design, which has the advantages of high performance at low per-unit price when used in high volume markets but has the drawback of long design and verification times. The other choice is an off the shelf embedded processor. Embedded processors allow rapid development times and low development cost in terms of both time and money, but with performance lagging behind ASICs. The gap between the these two methods for dealing with such systems, ASICs and off-the-shelf processors, leave many unsatisfied with the tradeoffs between long design cycles and lower performance. However, this need not be the case. To address this problem there is an emerging technique, which will add a new and important point in the spectrum of solutions, customized processors.

I.A.1 Customized Processors

The increased level of system integration has created a unique opportunity for customization. As we fold different components onto the same chip, vendors are now selling descriptions of processors, rather than physical silicon instantiations. If it is not physical silicon that is begin sold we can potentially modify the processor core descriptions to be customized for a given application. The idea is to be able to automatically change the description of the components to better suit their intended use. For example, if a processor is only going to be used for compression, why not change the description of the processor to be especially good at a compression application? This is the central idea behind customized processors.

Customized Processors use compiler analysis, directed profiling, and design automation techniques to take a generalized architectural model and create a specific instance of it, which is optimized to a given application or set of applications. In this way the processor is "tuned" to it's environment, allowing it

to take advantage of application specific behavior such as regularities in memory accesses or branch patterns. This new processor, running the specific code it was developed for, should be able to outperform the general purpose equivalent while still providing rapid design times. Customizing a processor can range from the synthesis of a completely new processor to having a general processor core where a few specific architecture components are customized. Typically the design relies upon optimizing and combining well understood lower level primitives.

Currently there is a resurgence of interest in customized processors with many research projects and companies working in this domain. The majority of interest comes from companies interested in increased system level integration. There are two major approaches to customized processors. The tailoring of a processor can be performed either at run-time (through adaptation) or design-time (through customization).

Design-time customization is a way of taking into consideration application specific behavior at the design (pre-silicon) stage. This means that application behaviors can be exploited by specialized structures that are custom built for just one application or a small suite of applications. As will be discussed in Section I.B, the majority of this dissertation concentrates on this type of tailoring as it provides the most flexibility from which to come up with new engineering solutions. There are several efforts concurrently working on the design of such systems, each starting with their own architectural template [49, 114]. While the approach they use is different, they are all built by combining well understood lower level primitives. The primitives with which these systems are built include cache or other form of local memory, prefetcher, branch prediction, and the computation core itself. Each of these components could be adapted to a given application to increase performance, reduce power, and decrease cost. In this dissertation we develop and examine a set of novel approaches to problems of

architectural level automation of these primitives for the purposes of embedded computing.

The second type of tailoring is run-time (post-silicon) adaptation. This approach attempts to take advantage of post-silicon customization through reconfigurability. Examples of systems that support reconfigurability are Altera's NIOS [131] processor core, the CHIMEARA chip [154], and reconfigurable caches [151]. In order to do run-time adaptation we need specialized configurable hardware structures or tunable parameters that are adjusted to executing applications. While these two types of tailoring seem quite different, much of the difficulty in either sort of tailoring is in the creation of sufficiently powerful automatic analysis techniques. And while the architectures used for the two types of tailoring are quite different, much of the analysis can be shared as will be discussed in chapter VI.

I.A.2 Design Constraints

Now that we have discussed customized processors, we now need to know what we are customizing . What are the goals of customizing a processor? In this section, we lay out the four major metrics of interest in the embedded space: power, performance, design time, and area.

Performance: The performance of embedded systems is still of great concern. As is stated in work by Fisher at Hewlett Packard Laboratories [43, 42, 41], there are many applications that require high performance, from digital signal processing and voice recognition in cell phones to embedded processors in printers to automotive applications. In these applications processor performance is still the primary limiting factor. Many of these systems have secondary constraints that make high performance difficult to achieve such as small form factor, battery life, or reliability, but all would benefit from highly efficient high performance

embedded cores.

Design Effort: As was mentioned earlier, the growing amount of design effort required for each generation of chip is of great concern. Even though we constantly have more and more chip real estate to work with, a designer still has to make effective use of it. This can either mean inflated design times, with slower time to market, or enlarged design teams, making design more costly and harder to manage.

Area: The area taken up by an embedded processor is very important to the cost of overall system. Area is related to the cost of fabrication: the larger the chip is, the more the mask will cost, and the less chips you can pack onto a single die. If an embedded processor is too large it will dominate the overall cost of the chip.

Power: Thermal and battery life concerns are two of the major factors in the design of modern embedded systems. In terms of battery life, low power systems enable the same system to be used for a longer amount of time before a recharge or battery replacement is needed, which effects the user end experience. The other option is to add a battery with more storage at the cost of heavier, larger and more expensive devices. A system that generates too much heat can require a larger form factor and higher packaging cost for more elaborate cooling mechanisms.

While power, performance and design effort are of utmost concern to an embedded system designer, the bottom line always comes down to cost. This is a very different mentality from the high performance community. A high performance system seeks to get the highest performance out of a given amount of chip area, on the other hand the embedded designer seeks to minimize the area needed to perform a given task in a given amount of time. For low end systems this can usually be accomplished through the use of off-the-shelf microprocessors, but for

systems with medium to high performance requirements ASICs must be designed and tested which is a costly and time consuming endeavor. In this dissertation, we concentrate quantitatively on performance and area, and qualitatively address design time. We leave power reduction techniques as the subject of future work.

I.B Overview of Dissertation

As was mentioned above in section I.A.1, there are several companies and research groups that are concentrating on platforms for building customized architectures. While these companies are focused on customization, they still rely on the designer to control every aspect of the customization. If our goal is to free up engineers to concentrate on the important aspects of the design process, we need to *automate* the time consuming aspects of customization. Introducing automation further enables new optimizations that would not be feasible if they had to be performed for every application that was targeted by hand.

The architectures and techniques presented in this dissertation use various forms of profiling, analysis, configuration, and design to automatically change the processor to best fit the applications running on it. Each of the chapters in this dissertation dives into a specific way in which automated analysis techniques can be combined with customization or adaptation to help with either performance or efficiency. Additionally, each of the techniques can be performed in a completely automatic fashion and in this dissertation shows how this can be done. The rest of this chapter provides a high level overview of the techniques presented in the rest of the chapters and describes how they fit together.

We have developed customization and adaptation techniques that are based on five main steps: program analysis, component specialization, inter-component tradeoff, algorithm level tradeoff, and run-time phase detection. Each of these steps brings us closer to the goal of processors tuned to applications, and

each step is described in detail in the chapters that follow.

I.B.1 Program Phase Behavior

Understanding program behavior is the foundation of computer architecture and program optimization. Many programs have wildly different behavior on even the largest of scales (the complete execution of the program), which greatly complicates the steps of profiling and simulation that are integral to any customization scheme. To do detailed simulation of a single minute of execution requires on the order of 50 days of simulation time. The challenge is to be able to understand and quantify the behavior of the whole program without having to do detailed simulation of full program runs. This will enable us to rapidly evaluate different design decisions and properly model the reaction of a system's behavior to various architectural changes.

Chapter II presents automatic techniques that are capable of finding and exploiting large-scale program behavior. The first step towards this goal is the development of a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. Basic Block Vectors, a hardware independent metric based solely on the percentage of execution that each basic block in the program accounts for, efficiently summarize phase behavior in programs. Once we have an efficient way of capturing general program behavior, the next step is to characterize the long term repeating behaviors of the program as a set of phases to enable an accurate representation of the program. In Chapter II we show that this problem of classifying sections of execution into phases is related to the problem of clustering from machine learning, and then develop an algorithm to quickly, effectively, and automatically break up the full execution of the program into several phases, where the elements of each phase are very similar.

Once this classification is completed, analysis and optimization can be performed on a per-phase basis which is significantly faster. These techniques have many benefits, including enabling fast and accurate cycle-level performance estimation and providing an analytical starting point from which whole new classes of optimizations based on program phases can be explored.

I.B.2 Component Specialization

Once we have an efficient way of modeling the interaction between an application and any architectural level customization, we can then begin customizing parts of the processor. Customized processors are built from a set of parts, such as the cache, branch predictor, and data path, each of which can be tuned to the application to increase performance, reduce power, or decrease area. Finite state machines (FSM) are a fundamental building block in computer architecture, and are used to control and optimize all types of prediction and speculation, even in the embedded space. They can be used for branch prediction, cache replacement policies, confidence estimation, and accuracy counters for a variety of optimizations.

Chapter III is a presentation of a framework for the automated design of small FSM predictors, which can be used to create application tuned branch predictors and confidence estimators. Using these tools, a hardware design for a customized branch predictor is produced directly from application profiles without the need for user interaction. The predictors are generated from algorithms known in combinatorial optimization, statistical modeling, and regular expressions, in addition to my own optimization algorithms.

In customized processors these finite state machine representations can be built into hardware directly, but there are many additional applications of this technology to branch profiling and branch correlation modeling. Additionally, in

Chapter III we describe ways in which customized state machines can be used to instantiate very small and very effective value prediction confidence counters.

I.B.3 Inter-component Tradeoffs

Once we have the ability to customize portions of a chip we can begin to combine them in ways that are the most beneficial to the end user or designer. It is impossible to gauge the benefit of any optimization done on a specific subset of the chip unless its effect on the system as a whole is examined.

For example, if I design a new cache that is ten percent faster but takes fourteen percent more area, is this a good design tradeoff? The answer to this question relies on the constraints of the system (area, performance, and power) and the relative benefit from the other components. For example, it may be the case that with fourteen percent more area you could instead get twenty percent performance improvement from a combination of larger branch prediction tables and instruction cache. To provide this context and provide early feedback to help the designer navigate their way through the architecture design space, the Sherpa design framework is described in Chapter IV.

The approach used is to decompose the overall problem into a set of mostly linear sub-problems that can be traded-off against one another with integer-linear programming. The power of this system is that by calculating good tradeoffs, the system blurs the line between performance and area optimizations allowing designers to rapidly see the effect of decisions and optimizations on the design space.

I.B.4 Architecture-Algorithm Co-exploration

Inter-component tradeoff is a powerful way of exploring the interaction between separate parts of the chip and viewing the way in which they are interact-

ing. Taking this analysis to the next level requires that we step past the analysis of components with one another, and consider the interaction of the algorithms and components as one unified system. In Chapter V this idea is explored in the context of network processors.

The current generation of backbone routers (OC-192) can deliver 10Gb/s of bandwidth *per link* even in the worst case, and the next generation will be moving 40 Gb/s. Thus each line card must process up to 125 million packets per second. Designing custom chip solutions that are capable of forwarding packets at this speed is becoming increasingly difficult as bandwidth requirements grow, network protocols become more complicated, and higher frequencies complicate physical design. A scalable and programmable custom architecture for executing network algorithms is needed to reduce the amount of engineering cost currently expended in the development of backbone routers.

In Chapter V we show that a new class of programmable architectures for backbone routers, based on the manipulation of wide memory words, are feasible design alternatives to custom ASICs for many interesting network algorithms. We begin with an examination of a pipelined memory design that emphasizes throughput over latency, and co-explore its architectural tradeoffs with the design of several important network algorithms. Through this co-exploration, we show that a programmable architecture can efficiently exploit behavior inherent to most common network algorithms.

I.B.5 Processor Adaptation

Design time customization works very well at using application specific information for embedded applications, but many times we would like to get these advantages into high performance general-purpose (desktop/workstation/mainframe) computers where customization is not a realistic design choice. Luckily many of

the techniques that are developed in this dissertation for customization have analogs in adaptive systems.

Both require a set of techniques that take advantage of program behavior in processors, first by quantifying the program behavior and then by tailoring to its needs. The difference between the two is that in the adaptive model we need to rely on run-time data gathering techniques that use hardware profiling rather than static analysis, and then configure and tune the hardware without user or compiler interaction.

Presented in Chapter VI are a set of techniques for gathering program phase information at run-time. This is the first and most crucial step in developing adaptive versions of any customization optimizations as this sets the level of granularity on which optimization can be effectively applied. The application of this primitive is examined in several different contexts to demonstrate its universal usefulness. For example it can be used to guide cache resizing, value profiling, and instruction front-end throttling.

II

Analysis of Program Behavior

II.A Introduction

In order to customize an architecture to an application we must begin by understanding the behavior of the application. The problem is that programs can have wildly different behavior over their run-time, and these behaviors can be seen even on the largest of scales. This greatly complicates the process of customization as it makes it very difficult to quantify what the effect of any optimization is on the program behavior as a whole. In addition to this, understanding these large scale program behaviors can unlock many new optimizations and potential for customization.

To enable these optimizations and allow for the accurate modeling of program behavior, we must first develop the analytical tools necessary to automatically and efficiently analyze program behavior over large sections of execution.

In this chapter we show how to perform such an analysis based on the development a hardware independent metric that can concisely summarize the behavior of an arbitrary section of execution in a program. In [121], we present the use of *Basic Block Vectors* (BBV), which uses the structure of the program that

is exercised during execution to determine where to simulate. A BBV represents the code blocks executed during a given interval of execution. Our goal is to find a set of windows of executed instructions that match the whole program's execution, so that these smaller windows of execution can be used for simulation instead of executing the program to completion. Using the BBVs provides us with a hardware independent way of finding this small representative window.

In this chapter, we examine the use of BBVs for analyzing large scale program behavior. We use BBVs to explore the large scale behavior of several programs and discover the ways in which common patterns, and code, repeat themselves over the course of execution. We quantify the effectiveness of basic block vectors in capturing this program behavior across several different architectural metrics (such as IPC, branch, and cache miss rates).

In addition to this, there is a need for a way of classifying these repeating patterns so that this information can be used for optimization. We show that this problem of classifying sections of execution is related to the problem of *clustering* from machine learning, and we develop an algorithm to quickly and effectively find these sections based on clustering. Our techniques automatically break the full execution of the program up into several sets, where the elements of each set are very similar. Once this classification is completed, analysis and optimization can be performed on a per-set basis.

We demonstrate an application of this cluster-based behavior analysis to simulation methodology for computer architecture research. By making use of clustering information we are able to accurately capture the behavior of a whole program by taking simulation results from representatives of each cluster and weighing them appropriately. This results in finding a set of simulation points that when combined accurately represents the target application and input, which in turn allows the behavior of even very complicated programs, such as `gcc`, to

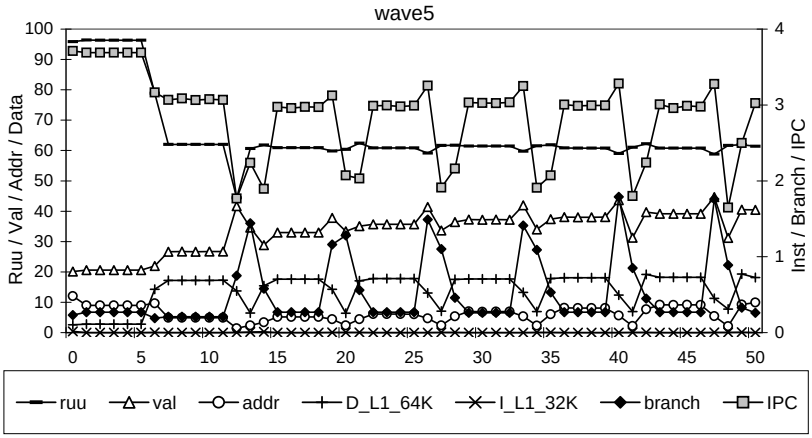

Figure II.1: Time Varying behavior of the program `wave`

be captured with a small amount of simulation time. We provide simulation points (points in the program to start execution at) for Alpha binaries of all of the SPEC 2000 programs. In addition, we validate these simulation points with the IPC, branch, and cache miss rates found for complete execution of the SPEC 2000 programs.

For architecture evaluation, the section of the program's execution simulated isof great importance to the relevance and correctness of the study. An example of this is the SPEC95 program `wave` which has three distinct types of execution. This can be seen if Figure II.1. The first 5 billion instructions have an IPC of around 4. After this initialization phase, `wave` begins the code which accounts for most of its execution time. During this time the program alternates between two different sections of execution, one with an IPC of 2 and one with an IPC of 3. Depending on where the program is simulated, very different results will be found.

The rest of the chapter is laid out as follows. First, a summary of the methodology used in this research is described in Section II.B. Section II.C presents a brief review of basic block vectors and an in depth look into the

proposed techniques and algorithms for identifying large scale program behaviors, and an analysis of their use on several programs. Section II.D describes how clustering can be used to analyze program behavior, and describes the clustering methods used in detail. Section II.E examines the use of the techniques presented in Sections II.C and II.D on an example problem: finding where to simulate in a program to achieve results representative of full program behavior. Related work is discussed in Section II.F, and the techniques presented are summarized in Section II.G.

II.B Methodology

In this chapter we used both ATOM [135] and SimpleScalar 3.0c [18] to perform our analysis and gather our results for the Alpha AXP ISA. ATOM is used to quickly gather profiling information about the code executed for a program. SimpleScalar is used to validate the phase behavior we found when clustering our basic block profiles showing that it corresponds to the phase behavior in the program's performance and architecture metrics. The baseline microarchitecture model we simulated is detailed in Table II.1. We simulate an aggressive 8-way dynamically scheduled microprocessor with a two level cache design. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch mis-prediction.

We analyze and simulated all of the SPEC 2000 benchmarks compiled for the Alpha ISA. The binaries we used in this study and how they were compiled can be found at: `http://www.simplescalar.com/`.

Instruction Cache	8k 2-way set-associative, 32 byte blocks, 1 cycle latency
Data Cache	16k 4-way set-associative, 32 byte blocks, 2 cycle latency
Unified L2 Cache	1Meg 4-way set-associative, 32 byte blocks, 20 cycle latency
Memory	150 cycle round trip access
Branch Predictor	hybrid - 8-bit gshare w/ 8k 2-bit predictors + a 8k bimodal predictor
Out-of-Order Issue	out-of-order issue of up to 8 operations per cycle, 128 entry re-order buffer
Mechanism	load/store queue, loads may execute when all prior store addresses are known
Architecture Registers	32 integer, 32 floating point
Functional Units	8-integer ALU, 4-load/store units, 2-FP adders, 2-integer MULT/DIV, 2-FP MULT/DIV
Virtual Memory	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table II.1: Baseline Simulation Model.

II.C Using Basic Block Vectors

A basic block is a section of code that is executed from start to finish with one entry and one exit. We use the frequencies with which basic blocks are executed as the metric to compare different sections of the application's execution to one another. The intuition behind this is that the behavior of the program at a given time is directly related to the code it is executing during that interval, and basic block distributions provide us with this information.

A program, when run for any interval of time, will execute each basic block a certain number of times. Knowing this information provides us with a fingerprint for that interval of execution, and tells us where in the code the application is spending its time. The basic idea is that knowing the basic block distribution for two different intervals gives us two separate fingerprints, which we can then compare to find out how similar the intervals are to one another. If the fingerprints are similar, then the two intervals spend about the same amount

of time in the same code, and the performance of those two intervals should be similar.

II.C.1 Basic Block Vector

A *Basic Block Vector* (BBV) is a single dimensional array, where there is a single element in the array for each static basic block in the program. For the results in this chapter, the basic block vectors are collected in intervals of 100 million instructions throughout the execution of a program. At the end of each interval, the number of times each basic block is entered during the interval is recorded and a new count for each basic block begins for the next interval of 100 million instructions. Therefore, each element in the array is the count of how many times the corresponding basic block has been entered during an interval of execution, multiplied by the number of instructions in that basic block. An example of this can be seen in Figure II.2. By multiplying in the number of instructions in each basic block we insure that we weigh instructions the same regardless of whether they reside in a large or small basic block. We say that a Basic Block Vector which was gathered by counting basic block executions over an interval of $N \times 100$ million instructions, is a Basic Block Vector of duration N.

Because we are not interested in the actual count of basic block executions for a given interval, but rather the proportions between time spent in basic blocks, a BBV is normalized by having each element divided by the sum of all the elements in the vector.

II.C.2 Basic Block Vector Difference

In order to find patterns in the program we must first have some way of comparing two Basic Block Vectors. The operation we desire takes as input

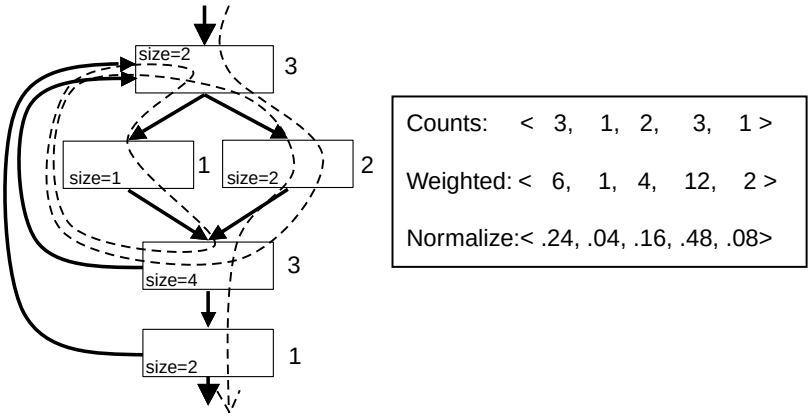

Figure II.2: Example of capturing the Basic Block Vectors

two Basic Block Vectors, and outputs a single number that tells us how close they are to each other. There are several ways of comparing two vectors to one another, such as taking the dot product or finding the Euclidean or Manhattan distance. In this chapter, we use both the Euclidean and Manhattan distances for comparing vectors.

The Euclidean distance can be found by treating each vector as a single point in D-dimensional space. The distance between two points is simply the square root of the sum of squares just as in $c^2 = a^2 + b^2$. The formula for computing the Euclidean distance of two vectors a and b in D-dimensional space is given by:

$$EuclideanDist(a, b) = \sqrt{\sum_{i=1}^{D} (a_i - b_i)^2}$$

The Manhattan distance on the other hand is the distance between two points if the only paths you can take are parallel to the axes. In two dimensions this is analogous to the distance traveled if you were to go by car through city blocks. This has the advantage that it weighs more heavily differences in each dimension (being closer in the x-dimension does not get you any closer in the y-dimension). The Manhattan distance is computed by summing the absolute value of the element-

wise subtraction of two vectors. For vectors a and b in D-dimensional space, the distance can be computed as:

$$ManhattanDist(a, b) = \sum_{i=1}^{D} |a_i - b_i|$$

Because we have normalized all of the vectors, the Manhattan distance will always be a single number between 0 and 2 (because we normalize each BBV to sum to 1). This number can then be used to compare how closely related two intervals of execution are to one another. For the rest of this section we will be discussing distances in terms of Manhattan distance, because we found that it more accurately represented differences in our high-dimensional data. We present the Euclidean distance as it pertains to the clustering algorithms presented in Section II.D, since it provides a more accurate representation for data with lower dimensions.

II.C.3 Basic Block Difference Graph

We now concentrate on understanding how the execution of a program changes over time. We did this by creating a Basic Block Difference Graph [121]. The Basic Block Difference Graph is a plot of how well each individual sample in the program compares to the target Basic Block Vector created for the entire run to completion. In this case, target vector is the full execution of the program to completion.

Figure II.3 shows the plot of all of these BBV differences across the entire execution creating a *Basic Block Difference Graph*. For each consecutive interval of 100 million instructions, we create a BBV and calculate its difference from the target BBV (the full execution). The x-axis is the number of instructions in 100 millions, and the y-axis is our measure of comparison[1], the BB Difference, between the interval in question and the target vector. A basic block difference

[1]

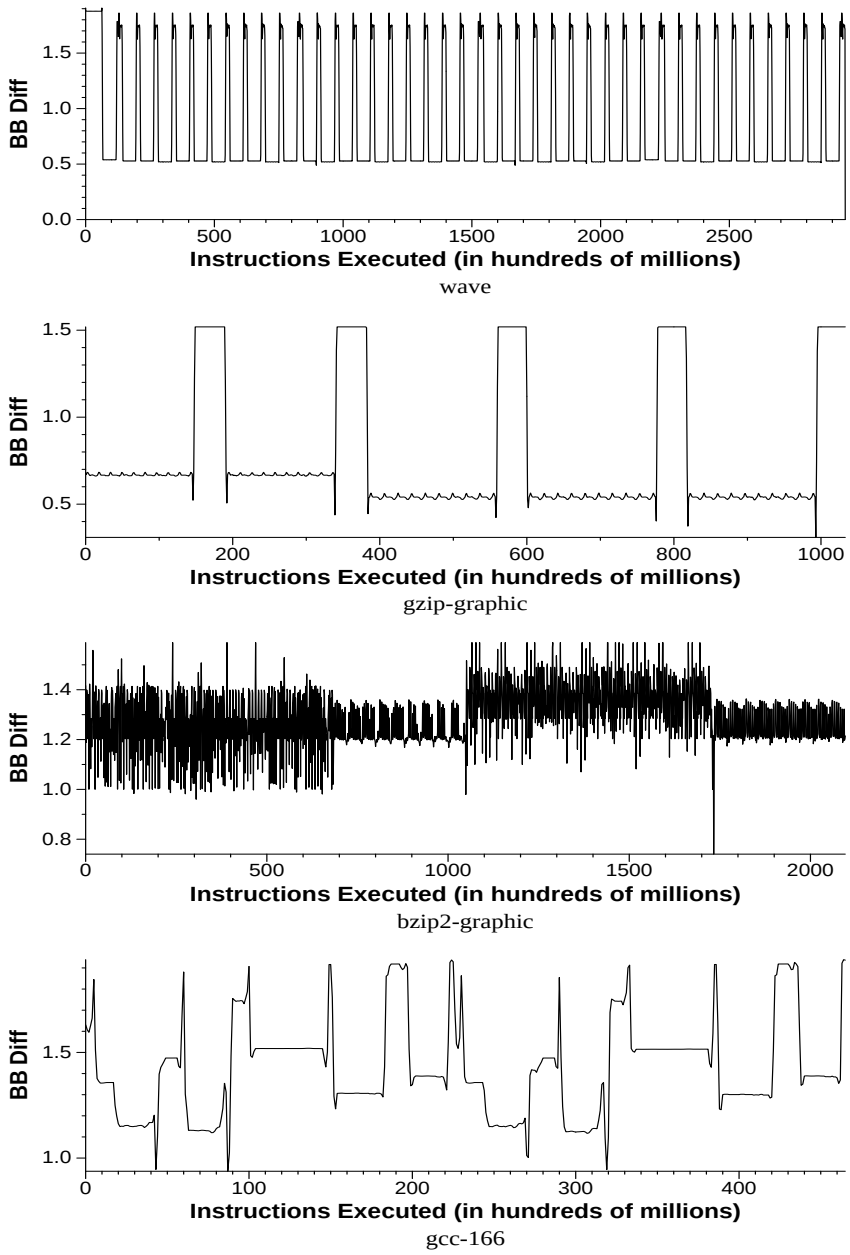

Figure II.3: The basic block difference graphs.
Each x-axis unit represents 100 million executed instructions. The graphs show the Basic Block Vector difference on the y-axis, which is calculated by comparing the target BBV (the full execution of the program) with the BBV generated for each 100 million interval of executed instructions.

of 2 means that the two vectors are completely orthogonal and share no basic blocks traversals in common, while a difference of 0 is the result of a perfect match between a BBV and the target vector. So for example, if the graph goes all the way to 0, then the basic blocks in the interval at that point are executing in exactly the same *porportion* as the complete execution of the program.

As shown in II.3, most of these programs start with a section of code which is very different from the full execution of the program (i.e. close to 2). This section is usually setting up and initializing structures for the compute intensive part of execution. Then a repetitive pattern is seen through the rest of the execution of the program. But not all programs follow this trend as can be seen by `gcc` and `bzip`. We used the basic block difference graph to try to find a single simulation point that was representative of the whole program's execution, which works well for programs like `wave`. But this does not work well for program's like `gcc` and `bzip`. This motivated us to explore new techniques for representing the execution of the program over time.

II.C.4 Basic Block Similarity Matrix

Now that we have a method of comparing intervals of program execution to one another, we can now concentrate on finding phase-based behavior. A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being a contiguous interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase to include all similar sections of execution regardless of temporal adjacency.

A key observation from this chapter is that the phase behavior seen in any program metric is directly a function of the code being executed. Because of this we can use the comparison between the Basic Block Vectors as an approximate bound on how closely related any other metrics will be between those two

intervals.

To find how intervals of execution relate to one another we create a *Basic Block Similarity Matrix*. The similarity matrix is an upper triangular $N \times N$ matrix, where N is the number of intervals in the program's execution. An entry at (x, y) in the matrix represents the Manhattan distance between the basic block vector at interval x and the basic block vector at interval y.

Figures II.4(left and right) and II.7(left) shows the similarity matrices for `gzip`, `bzip`, and `gcc` using the Manhattan distance. The diagonal of the matrix represents the program's execution over time from start to completion. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter they are the more different they are (the Manhattan distance is closer to 2).

The top left corner of each graph is the start of program execution and is the origin of the graph, $(0, 0)$, and the bottom right of the graph is the point $(N-1, N-1)$ where N is the number of intervals that the full program execution was divided up into. The way to interpret the graph is to start considering points along the diagonal axis drawn. Each point is perfectly similar to itself, so the points directly on the axis all are drawn dark. Starting from a given point on the diagonal axis of the graph, you can begin to compare how that point relates to it's neighbors forward and backward in execution by tracing horizontally or vertically. If you wish to compare a given interval x with the interval at $x + n$, you simply start at the point (x, x) on the graph and trace horizontally to the right until you reach $(x, x + n)$.

To examine the phase behavior of programs, let us first examine `gzip` because it has behavior on such a large scale that it is easy to see. If we examine an interval taken from 70 billion instructions into execution, in Figure II.4 (left), this is directly in the middle of a large phase shown by the triangle block of dark

color that surrounds this point. This means that this interval is very similar to it's neighbors both forward and backward in time. We can also see that the execution at 50 billion and 90 billion instructions is also very similar to the program behavior at 70 billion. We also note, while it is very easy to see that the phase interval at 70 billion instructions is similar to the phases at interval 10 and 30 billion, but they are not as similar as to those around 50 and 90 billion. Compare this with the IPC and data cache miss rates for `gzip` shown in Figure II.5. Overall, Figure II.4(left) shows that the phase behavior seen in the similarity matrix lines up quite closely with the behavior of the program, with 5 large phases (the first 2 being different from the last 3) each divided by a small phase, where all of the small phases are very similar to each other.

The similarity matrix for `bzip` (shown on the right of Figure II.4) is very interesting. `Bzip` has complicated behavior, with two large parts to it's execution, compression and decompression. This can readily be seen in the figure as the large dark triangular and square patches. The interesting thing about `bzip` is that even within each of these sections of execution there is complex behavior. This, as will be shown later, makes the behavior of `bzip` impossible to capture using a small contiguous section of execution.

A more complex case for finding phase behavior is `gcc`, which is shown on the left of Figure II.7. This similarity matrix shows the results for `gcc` using the Manhattan distance. The similarity matrix on the right will be explained in more detail in Section II.D.2. This figure shows that `gcc` does have some regular behavior. It shows that, even here, there is common code shared between sections of execution, such as the intervals around 13 billion and 36 billion. In fact the strong dark diagonal line cutting through the matrix indicates that there is a good amount of repetition between offset segments of execution. By analyzing the graph we can see that interval x is very similar to interval $(x + 23.6B)$ for all

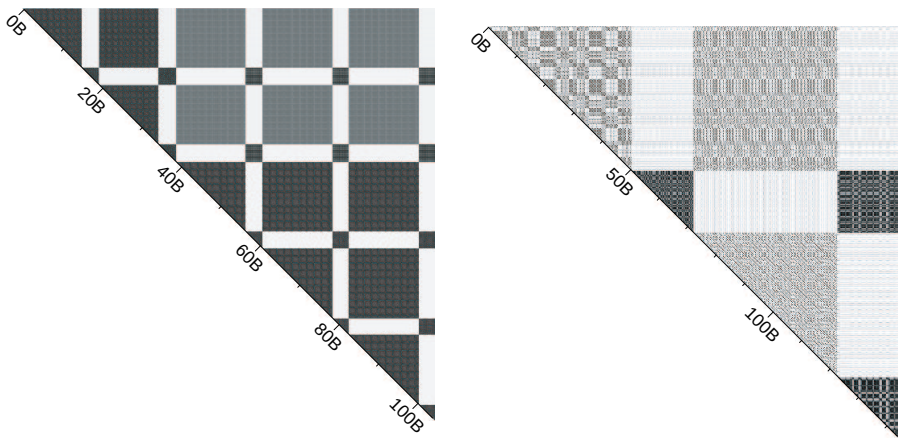

Figure II.4: Basic block similarity matrix for the programs `gzip` and `bzip`. The program `gzip-graphic` is shown to the left, and `bzip-graphic` is on the right. The diagonal of the matrix represents the program's execution to completion with units in billions of instructions. The darker the points, the more similar the intervals are (the Manhattan distance is closer to 0), and the lighter the points the more different they are (the Manhattan distance is closer to 2).

x.

II.C.5 Vector Similarity and IPC

Figures II.5 and II.8 show the time varying behavior of `gzip` and `gcc`. The average IPC and data cache miss rate is shown for each 100 million interval of execution over the complete execution of the program. The time varying results graphically show the same phase behavior seen by looking at only the code executed. For example, the two phases for `gcc` at 13 billion and 36 billion, shown to be very similar in Figure II.7, are shown to have the same IPC and data cache miss rate in Figure II.8.

Figure II.10 shows how smaller distances can translate into more similar behaviors. The x-axis is the Manhattan vector distance. For a given x-axis point, the y-axis represents the percentage of intervals that have that vector distance,**and** had less than an IPC difference specified in the legend. Four lines

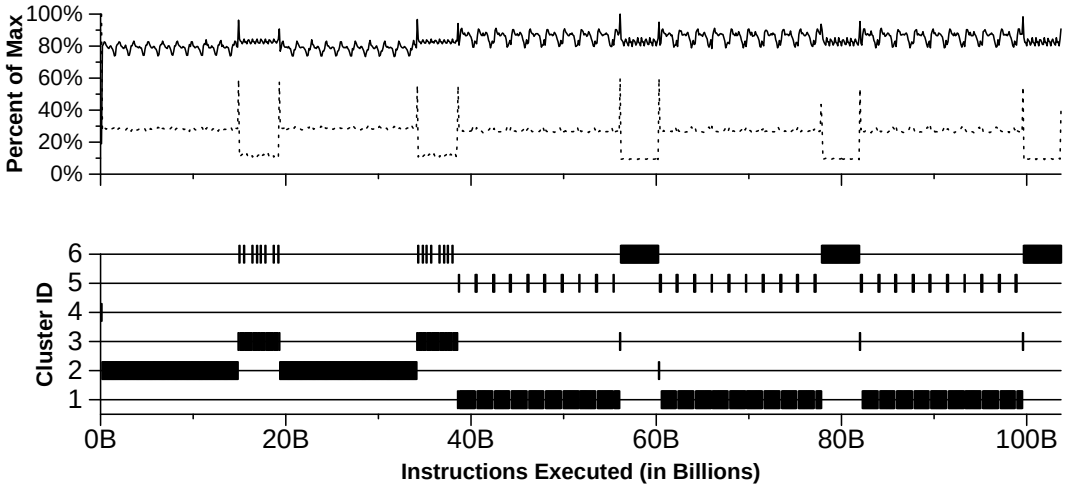

Figure II.5: Time varying graph for `gzip-graphic`.
The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure II.6: Cluster graph for `gzip-graphic`.
The full run of the execution is partitioned into a set of 6 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

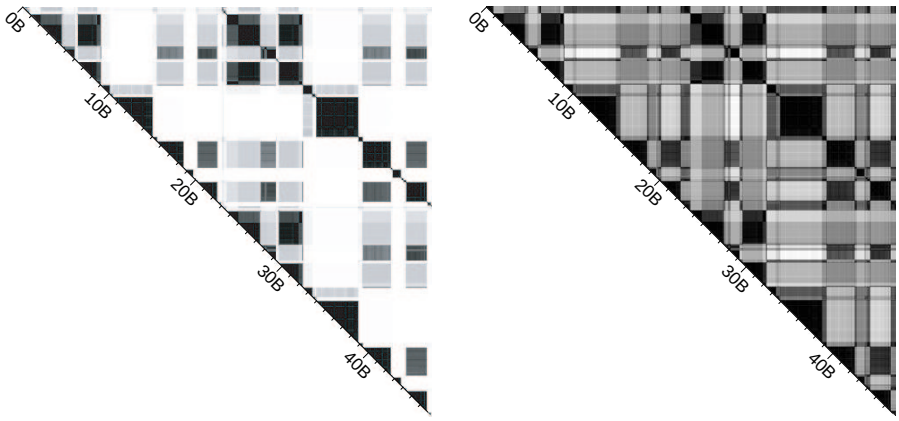

Figure II.7: Comparison of similarity with random projection
The original basic block similarity matrix for the program `gcc` (shown left), and the similarity matrix for `gcc-166` drawn from projected data (on right). The figure on the left use the original basic block vectors (each of which has over 100,000 dimensions) and uses the Manhattan distance as a method of difference taking. The figure on the right uses projected data (down to 15 dimensions) and uses the Euclidean distance for difference taking.

are shown, each one representing a different IPC error threshold. For example, the results show that 99% of the intervals that had a vector difference of 0.1 or less, also had an IPC difference of 0.25 or less. 92% of the intervals had an IPC difference of less than 0.1 or less also had a vector difference of 0.1 or less. These results show that grouping together vectors that have a small distance, will also group together vectors that have similar performance. This is the reason why our clustering algorithm, presented in the next section, performs so well.

II.D Clustering

The basic block vectors provide a compact and representative summary of the program's behavior for intervals of execution. By examining the similarity between them, it is clear that there exists a high level pattern to each program's execution. In order to make use of this behavior we need to start by delineating a method of finding and representing the information. Because there are so many

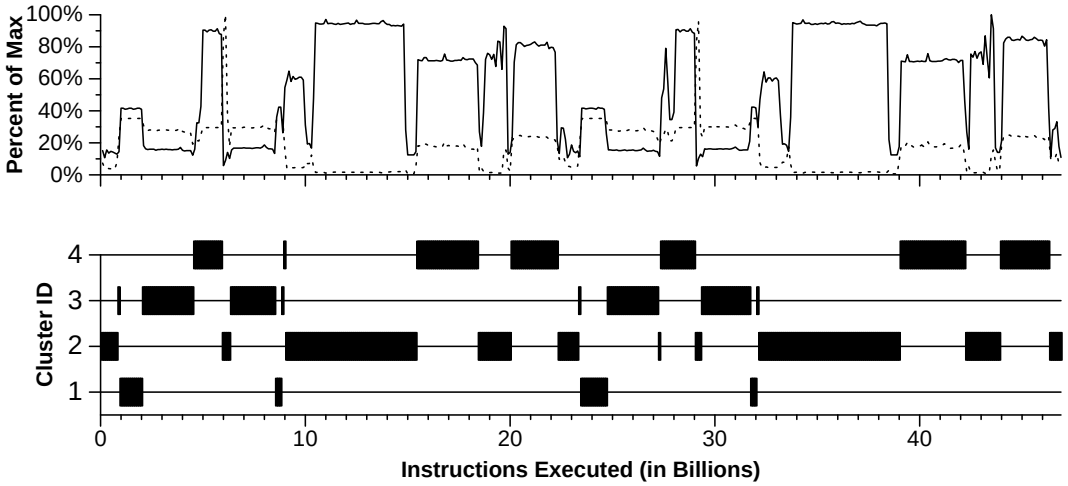

Figure II.8: Time varying graph for `gcc-166`.
(top graph) The average IPC (drawn with solid line) and L1 data cache miss rate (drawn with dotted line) are plotted for every interval (100 million instructions of execution) showing how these metrics vary over the program's execution. The x-axis represents the execution of the program over time, and the y-axis the percent of max value the metric had during execution. The results are non-accumulative.

Figure II.9: Cluster graph for `gcc-166`.
(top graph) The full run of the execution is partitioned into a set of 4 clusters. The x-axis is in instructions executed, and the graph shows for each interval of execution (every 100 million instructions), which cluster the interval was placed into.

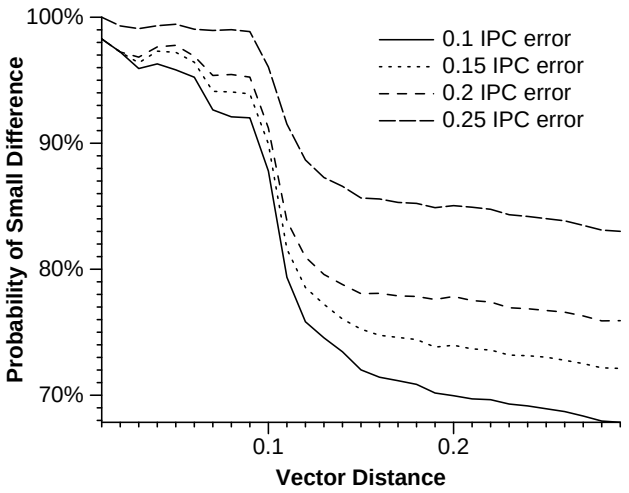

Figure II.10: Correlating the vector difference with the difference in IPC seen for two intervals.

intervals of execution that are similar to one another, one efficient representation is to group the intervals together that have similar behavior. This problem is analogous to a *clustering* problem. Later, in Section II.E, we demonstrate how we use the clusters we discover to find multiple simulation points for irregular programs or inputs like `gcc`. By simulating only a single representative from each cluster, we can accurately represent the whole program's execution.

II.D.1 Clustering Overview

The goal of clustering is to divide a set of points into groups such that points within each group are similar to one another (by some metric, often distance), and points in different groups are different from one another. This problem arises in other fields such as computer vision [67], document classification [150], and genomics [11], and as such it is an area of much active research. There are many clustering algorithms and many approaches to clustering. Classically, the two primary clustering approaches are Partitioning and Hierarchical:

Partitioning algorithms choose an initial solution and then use iterative updates to find a better solution. Popular algorithms such as k-means [80] and Gaussian Expectation-Maximization [15, pages 59–73] are in this family. These algorithms tend to have complexity that is linear in the size of the dataset.

Hierarchical algorithms [66] either combine together similar points (called agglomerative clustering, and conceptually similar to Huffman encoding), or recursively divides the dataset into more groups (called divisive clustering). These algorithms tend to have complexity that is quadratic in the size of the dataset.

II.D.2 Phase Finding Algorithm

For our algorithm, we use random linear projection followed by k-means. We choose to use the k-means clustering algorithm, since it is a very fast and simple algorithm that yields good results. To choose the value of k, we use the Bayesian Information Criterion (BIC) score [69, 97]. The following steps summarize our algorithm, and then several of the steps are explained in more detail:

1. Profile the basic blocks executed in each program to generate the basic block vectors for every 100 million instructions of execution.

2. Reduce the dimension of the BBV data to 15 dimensions using random linear projection.

3. Try the k-means clustering algorithm on the low-dimensional data for k values 1 to 10. Each run of k-means produces a clustering, which is a partition of the data into k different clusters.

4. For each clustering ($k = 1 \ldots 10$), score the fit of the clustering using the BIC. Choose the clustering with the smallest k, such that it's score is at least 90% as good as the best score.

Random Projection

For this clustering problem, we have to address the problem of dimensionality. All clustering algorithms suffer from the so-called "curse of dimensionality", which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases. For the basic block vectors, the number of dimensions is the number of executed basic blocks in the program, which ranges from 2,756 to 102,038 for our experimental data, and could grow into the millions for very large programs. Another practical problem is that the running time of our clustering algorithm depends on the dimension of the data, making it slow if the dimension grows too large.

Two ways of reducing the dimension of data are dimension selection and dimension reduction. Dimension selection simply removes all but a small number of the dimensions of the data, based on a measure of goodness of each dimension for describing the data. However, this throws away a lot of data in the dimensions which are ignored. Dimension reduction reduces the number of dimensions by creating a new lower-dimensional space and then projecting each data point into the new space (where the new space's dimensions are not directly related to the old space's dimensions). This is analogous to taking a picture of 3 dimensional data at a random angle and projecting it onto a screen of 2 dimensions.

For this work we choose to use *random linear projection* [31] to create a new low-dimensional space into which we project the data. This is a simple and fast technique that is very effective at reducing the number of dimensions while retaining the properties of the data. There are two steps to reducing a dataset X (which is a matrix of basic block vectors and is of size $N_{intervals} \times D_{numbb}$, where D_{numbb} is the number of basic blocks in the program) down to D_{new} dimensions using random linear projection:

- Create a $D_{numbb} \times D_{new}$ projection matrix M by choosing a random value

for each matrix entry between -1 and 1.

- Multiply X times M to obtain the new lower-dimensional dataset X' which will be of size $N_{intervals} \times D_{new}$.

For clustering programs, we found that using $D_{new} = 15$ dimensions is sufficient to still differentiate the different phases of execution. Figure II.11 shows why we chose to project the data down to 15 dimensions. The graph shows the number of dimensions on the x-axis. The y-axis represents the k value found to be best on average, when the programs were projected down to the number of dimensions indicated by the x-axis. The best k is determined by the k with the highest BIC score, which is discussed in Section II.D.2. The y-axis is shown as a percent of the maximum k seen for each program so that the curve can be examined independent of the actual number of clusters found for each program. The results show that for 15 dimensions the number of clusters found begins to stabilize and only climbs slightly. Similar results were also found using a different method of finding k in [55].

The advantages of using linear projections are twofold. First, creating new vectors with a low dimension of 15 is extremely fast and can even be done at profile time. Secondly, using only 15 dimensions speeds up the k-means algorithm significantly, and reduces the memory requirements by several orders of magnitude over using the original basic block vectors.

Figure II.12 is a illustration of the idea of random projection. In this picture the basic block data for the program `gzip` has been reduced down to only two dimensions. Even at this very low dimentionality we can still see the clusters present.

Figure II.7 shows the similarity matrix for `gcc` on the left using original BBVs, whereas the similarity matrix on the right shows the same matrix but on the data that has been projected down to 15 dimensions. For the reduced

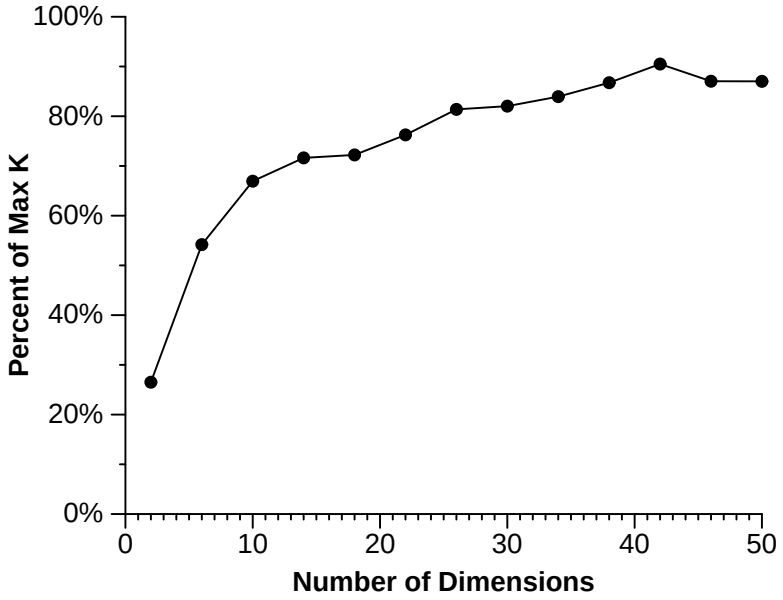

Figure II.11: Motivation for random projection down to 15 dimensions (D=15). The x-axis is the number of dimensions of the projection, and the y-axis is the percent of the max number of clusters found for each program averaged over all spec programs. The results show that as you decrease the number of dimensions too far (the lowest point is two dimensions) the true clusters become collapsed on one another, and the algorithm cannot find as many clusters. By D=15 most of this effect has gone.

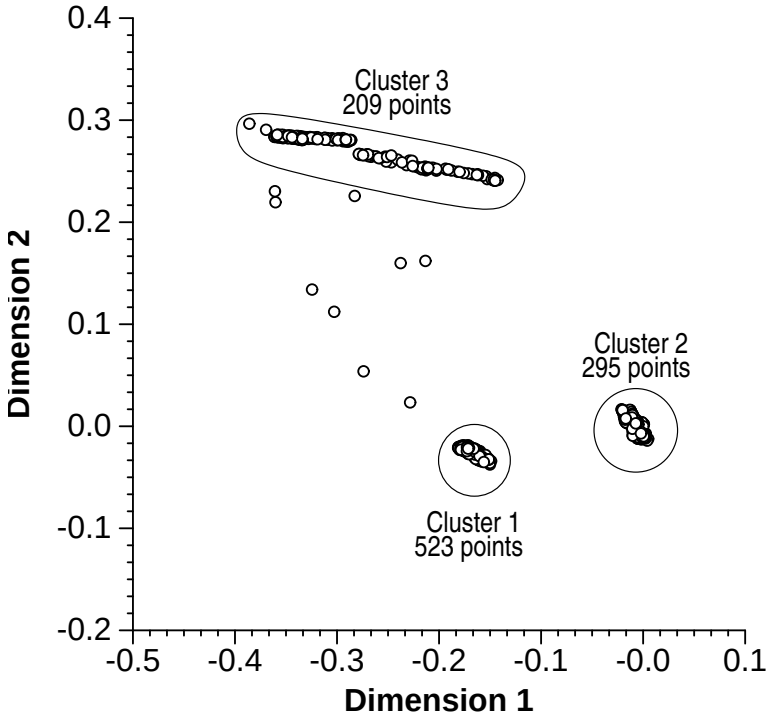

Figure II.12: Projection and Clusters for gzip, 1037 total points

dimension data we use the Euclidean distance to measure differences, rather than the Manhattan distance used on the full data. After the projection, some information will be blurred, but overall the phases of execution that are very similar with full dimensions can still be seen to have a strong similarity with only 15 dimensions.

K-means

The k-means algorithm is an iterative optimization algorithm, which executes as two phases, which are repeated to convergence. The algorithm begins with a random assignment of k different centers, and begins its iterative process. The iterations are required because of the recursive nature of the algorithm; the cluster centers define the cluster membership for each data point, but the data point memberships define the cluster centers. Each point in the data belongs to, and can be considered a member of, a single cluster.

We initialize the k cluster centers by choosing k random points from the data to be clustered. After initialization, the k-means algorithm proceeds in two phases which are repeated until convergence:

- For each data point being clustered, compare its distance to each of the k cluster centers and assign it to (make it a member of) the cluster to which it is the closest.
- For each cluster center, change its position to the centroid of all of the points in its cluster (from the memberships just computed). The centroid is computed as the average of all the data points in the cluster.

This process is iterated until membership (and hence cluster centers) cease to change between iterations. At this point the algorithm terminates, and the output is a set of final cluster centers and a mapping of each point to the cluster that it belongs to. Since we have projected the data down to 15 dimensions,

we can quickly generate the clusters for k-means with k from 1 to 10. In doing this, there are efficient algorithms for comparing the clusters that are formed for these different values of k, and choosing one that is good but still uses a small value for k is the next problem.

Bayesian Information Criterion

To compare and evaluate the different clusters formed for different k, we use the *Bayesian Information Criterion* (BIC) as a measure of the "goodness of fit" of a clustering to a dataset. More formally, the BIC is an approximation to the probability of the clustering given the data that has been clustered. Thus, the larger the BIC score, the higher the probability that the clustering being scored is a "good fit" to the data being clustered. We use the BIC formulation given in [97] for clustering with k-means, however other formulations of the BIC could also be used.

More formally, the BIC score is a penalized likelihood. There are two terms in the BIC: the likelihood and the penalty. The likelihood is a measure of how well the clustering models the data. To get the likelihood, each cluster is considered to be produced by a spherical Gaussian distribution, and the likelihood of the data in a cluster is the product of the probabilities of each point in the cluster given by the Gaussian. The likelihood for the whole dataset is just the product of the likelihoods for all clusters. However, the likelihood tends to increase without bound as more clusters are added. Therefore the second term is a penalty that offsets the likelihood growth based on the number of clusters. The BIC is formulated as

$$BIC(D,k) = l(D|k) - \frac{p_j}{2}\log(R)$$

where $l(D|k)$ is the likelihood, R is the number of points in the data, and p_j is the number of parameters to estimate, which is $(k-1)+dk+1$ for $(k-1)$ cluster

probabilities, k cluster center estimates which each require d dimensions, and 1 variance estimate. To compute $l(D|k)$ we use

$$\begin{aligned} l(D|k) &= \sum_{i=1}^{k} -\frac{R_i}{2}\log(2\pi) - \frac{R_i d}{2}\log(\sigma^2) - \frac{R_i - 1}{2} \\ &\quad + R_i \log(R_i/R) \end{aligned}$$

where R_i is the number of points in the ith cluster, and σ^2 is the average variance of the Euclidean distance from each point to its cluster center.

For a given program and inputs, the BIC score is calculated for each k-means clustering, for k from 1 to N. We then choose the clustering that achieves a BIC score that is at least 90% of the spread between the largest and smallest BIC score that the algorithm has seen. Figure II.13 shows the benefit of choosing a BIC with a high value and its relationship with the variance in IPC seen for that cluster. The y-axis shows the percent of IPC variance seen for a given clustering, and the corresponding BIC score the clustering received. Each point on the graph represents the average or max IPC variance for all points in the range of ±5% of the BIC score shown. The results show that picking clusterings that represent greater than 80% of the BIC score resulted in an IPC variance of less than 20% on average. The IPC variance was computed as the weighted sum of the IPC variance for each cluster, where the weight for a cluster is the number of points in that cluster. The IPC variance for each cluster is simply the variance of the IPC for all the points in that cluster.

II.D.3 Clusters and Phase Behavior

Figures II.6 and II.9 show the 6 clusters formed for `gzip` and the 4 clusters formed for `gcc`. The X-axis corresponds to the execution of the program in billions of instructions, and each interval (each of 100 million instructions) is tagged to be in one of the N clusters (labeled on the Y-axis). These figures, just as for Figures II.4 and II.7, show the execution of the programs to completion.

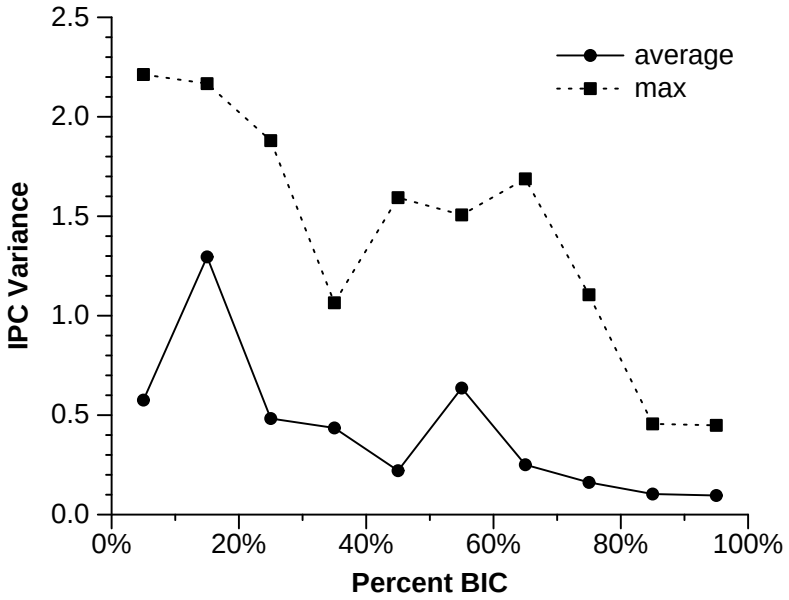

Figure II.13: Plot of average IPC variance and max IPC variance versus the BIC. These results indicate that for our data, a clustering found to have a BIC score greater than 80% will have, on average, and IPC variance of less than 0.2.

For `gzip`, the full run of the execution is partitioned into a set of 6 clusters. Looking to Figure II.4(left) for comparison, we see that the cluster behavior captured by our tool lines up quite closely with the behavior of the program. The majority of the points are contained by clusters 1,2,3 and 6. Clusters 1 and 2 represent the large sections of execution which are similar to one another. Clusters 3 and 6 capture the smaller phases which lie in between these large phases, while cluster 5 contains a small subset of the larger phases, and cluster 4 represents the initialization phase.

In the cluster graph for `gcc`, shown in Figure II.9, the run is now partitioned into 4 different clusters. Looking to Figure II.7 for comparison, we see that even the more complicated behavior of `gcc` is captured correctly by our tool. Clusters 2 and 4 correspond to the dark boxes shown parallel to the diagonal axis. It should also be noted that the projection does introduce some degree of error

into the clustering. For example, the first group of points in cluster 2 are not really that similar to the other points in the cluster. Comparing the two similarity matrices in Figure II.7, shows the introduction of a dark band at (0,30) on the graph which was not in the original (un-projected) data. Despite these small errors, the clustering is still very good, and the impact of any such errors will be minimized in the next section.

II.E Finding Simulation Points

Modern computer architecture research relies heavily on cycle accurate simulation to help evaluate new architectural features. While the performance of processors continues to grow exponentially, the amount of complexity within a processor continues to grow at an even a faster rate. With each generation of processor more transistors are added, and more things are done in parallel on chip in a given cycle while at the same time cycle times continue to decrease. This growing gap between speed and complexity means that the time to simulate a constant amount of processor time is growing. It is already to the point that executing programs fully to completion in a detailed simulator is no longer feasible for architectural studies. Since detailed simulation takes a great deal of processing power, only a small subset of a whole program can be simulated.

SimpleScalar [18], one of the faster cycle-level simulators, can simulate around 400 million instructions per hour. Unfortunately many of the new SPEC 2000 programs execute for 300 billion instructions or more. At 400 million instructions per hour this will take approximately 1 month of CPU time.

Because it is only feasible to execute a small portion of the program, it is very important that the section simulated is an accurate representation of the program's behavior as a whole. The basic block vector and cluster analysis presented in Sections II.C and II.D will allow us to make sure that this is the

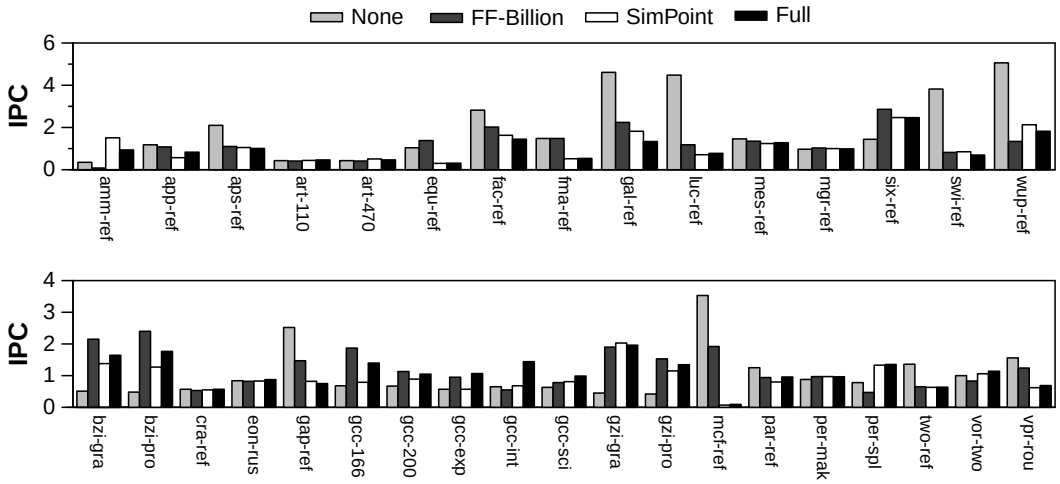

Figure II.14: Single Simpoint Comparison Barchart
The shows the simulation results starting simulation at the start of the program (none), blindly fast forwarding 1 billion instructions, using a single simulation point, and the IPC of the full execution of the program.

case.

II.E.1 Single Simulation Points

In [121], we used basic block vectors to automatically find a single simulation point to potentially represent the complete execution of a program. A *Simulation Point* is a starting simulation place (in number of instructions executed from the start of execution) in a program's execution derived from our analysis. That algorithm creates a target basic block vector, which is a BBV that represents the complete execution of the program. The Manhattan distance between each interval BBV and the target BBV is computed. The BBV with the lowest Manhattan distance represents the single simulation point that executes the code closest to the complete execution of the program. This approach is used to calculate the long single simulation points (LongSP) described below.

In comparison, the single simulation point results in this chapter are

calculated by choosing the BBV that has the smallest Euclidean distance from the centroid of the whole dataset in the 15-dimensional space, a method which we find superior to the original method. The 15-dimensional centroid is formed by taking the average of each dimension over all intervals in the cluster.

Figure II.14 shows the IPC estimated by executing only a single interval, all 100 million instructions long but chosen by different methods, for all SPEC 2000 programs. This is shown in comparison to the IPC found by executing the program to completion. The results are from SimpleScalar using the architecture model described in Section II.B, and all fast forwarding is done so that all of the architecture structures are completely warmed up when starting simulation (no cold-start effect).

The first bar, labeled `none`, is the IPC found when executing only the first 100 million instructions from the start of execution (without any fast forwarding). The second bar, `FF-Billion` shows the results after blindly fast forwarding 1 billion instructions before starting simulation. The third bar, `SimPoint` shows the IPC using our single simulation point analysis described above, and the last bar shows the IPC of simulating the program to completion (labeled `Full`). Because these are actual IPC values, values which are closer to the `Full` bar are better.

The results in Figure II.14 shows that the single simulation points are very close to the actual full execution of the program, especially when compared against the ad-hoc techniques. Starting simulation at the start of the program results in an average error of 210%, whereas blindly fast forwarding results in an average 80% IPC error. Using our single simulation point analysis we reduce the average IPC error to 18%. These results show that it is possible to reasonably capture the behavior of most programs using a very small slice of execution.

Table II.2 shows the actual simulation points chosen along with the pro-

gram counter (PC) and procedure name corresponding to the start of the interval. If an input is not attached to the program name, then the default ref input was used. Columns 2 through 4 are in terms of the number of intervals (each 100 million instruction long). The first column is the number of instructions executed by the program, on the specific input, when run to completion. The second column shows the end of initialization phase calculated as described in [121]. The third column shows the single simulation point automatically chosen as described above. This simulation point is used to fast forward to the point of desired execution. Some simulators, debuggers, or tracing environments (e.g., gdb) provide the ability to fast forward based upon a program PC, and the number of times that PC was executed. We therefore, provide the instruction PC for the start of the simulation point, the procedure that PC occurred in, and the number of times that PC has to be executed in order to arrive at the desired simulation point.

These results show that a single simulation point can be accurate for many programs, but there is still a significant amount of error for programs like `bzip`, `gzip` and `gcc`. This occurs because there are many different phases of execution in these programs, and a single simulation point will not accurately represent all of the different phases. To address this, we used our clustering analysis to find multiple simulation points to accurately capture program behavior, which we describe next.

II.E.2 Multiple Simulation Points

To support multiple simulation points, the simulator can be run from start to stop, only performing detailed simulation on the selected intervals. Or the simulation can be broken down into N simulations, where N is the number of clusters found via analysis, and each simulation is run separately. This has

Benchmark	Single Simulation Points				
	Length	Init	SP	PC	Proc Name
ammp	3265	24	109	120026834	mm_fv_update_nonbon
applu	2238	3	2180	120018520	buts_
apsi	3479	3	3409	1200380ac	dctdxf_
art-110	417	75	341	12000fbb0	match
art-470	450	83	366	12000f5d0	match
bzip2-graphic	1435	4	719	120012a5c	spec_putc
bzip2-program	1249	4	459	12000ddd0	sortIt
bzip2-source	1088	4	978	12000d774	qSort3
crafty	1918	462	775	120021730	SwapXray
eon-rushmeier	578	140	404	12004e1b4	viewingHit
equake	1315	35	813	120012410	phi0
facerec	2682	356	376	12002d1f4	$graphroutines$localmove_
fma3d	2683	192	2542	1200e3140	scatter_element_nodal_forces_
galgel	4093	3	2492	12002db00	syshtn_
gap	2695	639	675	120050750	CollectGarb
gcc-00-166	469	61	390	1200d157c	gen_rtx
gcc-00-200	1086	151	737	1200ceb04	refers_to_regno_p
gcc-00-expr	120	27	37	120191fd0	validate_change
gcc-00-integrate	131	14	5	120119800	find_single_use_in_loop
gcc-00-scilab	620	139	208	120100d54	insert
gzip-graphic	1037	158	654	120009c00	fill_window
gzip-log	395	91	266	12000d280	inflate_codes
gzip-program	1688	112	1190	120009660	longest_match
gzip-random	821	152	624	12000a14c	deflate
gzip-source	843	68	335	12000a224	deflate
lucas	1423	11	546	120021ef0	fft_square_
mcf	618	15	554	12000911c	price_out_impl
mesa	2816	6	1136	1200a30f0	general_textured_triangle
mgrid	4191	21	3293	1200160f0	resid_
parser	5467	388	1147	12001edfc	region_valid
perlbmk-diffmail	399	56	142	12007f974	regmatch
perlbmk-makerand	20	3	12	12008268c	Perl_runops_standard
perlbmk-perfect	290	69	6	12008268c	Perl_runops_standard
perlbmk-splitmail	1108	162	451	12007fc98	regmatch
sixtrack	4709	250	3044	120167894	thin6d_
swim	2258	3	2080	120019130	calc1_
twolf	3464	7	1067	120041094	ucxx1
vortex-one	1189	36	272	12006289c	Mem_GetWord
vortex-three	1330	177	565	1200336a8	Part_Delete
vortex-two	1386	206	1025	12005e6fc	Mem_NewRegion
vpr-place	1122	4	593	1200224ec	get_non_updateable_bb
vpr-route	840	12	477	120025c80	get_heap_head
wupwise	3496	11	3238	12001d680	zgemm_

Table II.2: Single simulation points for SPEC 2000 benchmarks. Columns 2 through 4 are in terms of 100 million instruction executed. The length of full execution is shown, as well as the end of initialization. SP is the single simulation point using the approach in this chapter. The procedure in which the simulation point occurred and its PC are also shown. The last 6 digits of PC of each SimPoint is given in hex, so the address is formed from 120xxxxxx.

Benchmark	Multiple Simulation Points									
	point 1	point 2	point 3	point 4	point 5	point 6	point 7	point 8	point 9	point 10
ammp	3026(13.8)	1774(31)	595(15.3)	1068(1.3)	2128(7.4)	1607(12.6)	2437(4.9)	3112(11.5)	2480(2.2)	
applu	624(22.1)	1625(22.5)	1956(18.8)	2234(6.6)	1380(15.5)	1507(14.5)				
apsi	2107(5.6)	2863(14)	1007(70.7)	896(7.7)	1618(2)					
art-110	82(42.9)	255(41.2)	50(15.8)							
art-470	300(36.2)	46(14.7)	236(49.1)							
bzip2-graphic	168(11.7)	1042(3.7)	430(7.5)	762(16.2)	106(15.3)	519(11.6)	872(8.2)	195(5.6)	148(2)	1435(18.2)
bzip2-program	140(11)	468(12.3)	78(6.2)	990(16)	445(7.4)	1005(7)	94(6.9)	606(14)	859(14.6)	341(4.7)
bzip2-source	395(16)	511(4.3)	64(29.1)	488(7.3)	530(8.6)	177(34.7)				
crafty	123(25)	510(19.7)	664(22.7)	1123(32.5)						
eon-rushmeier	260(6.6)	238(23.7)	337(20.9)	435(35.6)	216(13.1)					
equake	874(12.2)	1292(36.7)	463(12.2)	336(24.1)	3(3.2)	62(11.6)				
facerec	1976(60.1)	1528(2.5)	1935(3.9)	1398(29.2)	348(4.3)					
fma3d	112(7)	209(0.6)	842(68.4)	1600(11)	47(0.1)	509(13)				
galgel	3511(5.5)	2081(11)	3466(11.2)	516(31.6)	2141(2.7)	2181(29)	2161(3.3)	1017(5.5)		
gap	1114(8.2)	1196(58.1)	88(12.7)	2189(14)	2609(7.1)					
gcc-00-166	238(6.4)	149(42.2)	30(21.3)	404(30.1)						
gcc-00-200	8(45.8)	587(17.9)	921(10.9)	575(14.5)	1011(11)					
gcc-00-expr	63(12.5)	81(15.8)	42(16.7)	25(4.2)	9(45.8)	88(5)				
gcc-00-integrate	118(9.2)	41(27.5)	102(21.4)	9(20.6)	57(3.8)	73(17.6)				
gcc-00-scilab	255(54.2)	39(9.5)	231(13.2)	379(15.8)	170(7.3)					
gzip-graphic	961(45.4)	87(28.5)	373(7.3)	1(0.1)	461(5.2)	566(13.4)				
gzip-log	207(24.1)	171(16.5)	157(16.7)	330(23.5)	71(19.2)					
gzip-program	228(22.7)	779(21.4)	472(9.1)	1410(20.4)	594(26.4)					
gzip-random	484(0.9)	625(0.2)	580(51)	811(16.8)	200(30.9)	1(0.1)				
gzip-source	248(14.5)	327(13.2)	167(17.7)	656(27.8)	373(24.4)	720(2.5)				
lucas	982(21.4)	602(10.7)	1370(21.4)	458(28)	524(18.6)					
mcf	268(39.6)	425(11)	205(30.1)	468(4.5)	316(10.8)	143(3.9)				
mesa	1846(35.3)	2806(0.7)	398(35.3)	977(28.8)						
mgrid	43(24.2)	3459(22.8)	807(20.1)	3110(16.3)	2476(16.6)					
parser	3342(25.1)	1771(29.8)	5102(19.7)	2008(19.4)	4772(6)					
perlbmk-diffmail	6(1)	355(62.7)	11(0.5)	397(0.8)	12(3.3)	239(31.8)				
perlbmk-makerand	1(5)	20(20)	6(75)							
perlbmk-perfect	39(59.3)	207(40.7)								
perlbmk-splitmail	704(44.9)	596(9.1)	232(21.7)	461(21.8)	501(2.6)					
sixtrack	6(1.7)	1719(98.3)								
swim	1951(29.8)	38(14)	777(24.7)	710(13.8)	2101(17.8)					
twolf	312(17)	2888(11.3)	3268(11.7)	961(20.4)	2054(39.5)					
vortex-one	536(17.1)	366(23.3)	115(8.2)	1068(17.2)	878(34.2)					
vortex-three	934(25.4)	1129(11.4)	96(8.9)	47(11.1)	586(17.8)	485(25.4)				
vortex-two	635(7.6)	752(24.5)	554(21.9)	930(7.4)	360(15.3)	397(23.2)				
vpr-place	166(25.5)	857(21.6)	1(0.2)	362(12.8)	1057(12)	547(27.9)				
vpr-route	559(29.9)	89(28)	353(23.8)	3(2.6)	490(15.7)					
wupwise	1811(43.3)	91(8)	3055(43.2)	1524(5.4)						

Table II.3: Multiple simulation points for SPEC 2000 benchmarks.
The columns list the multiple simulation points found (in 100s of millions of instructions). The first number is the starting place of the simulation point relative to the *start* of execution. The second number shows the weight given to the cluster that simulation point was taken from, and is used when weighing the final results of the simulation (weights are shown as percentage).

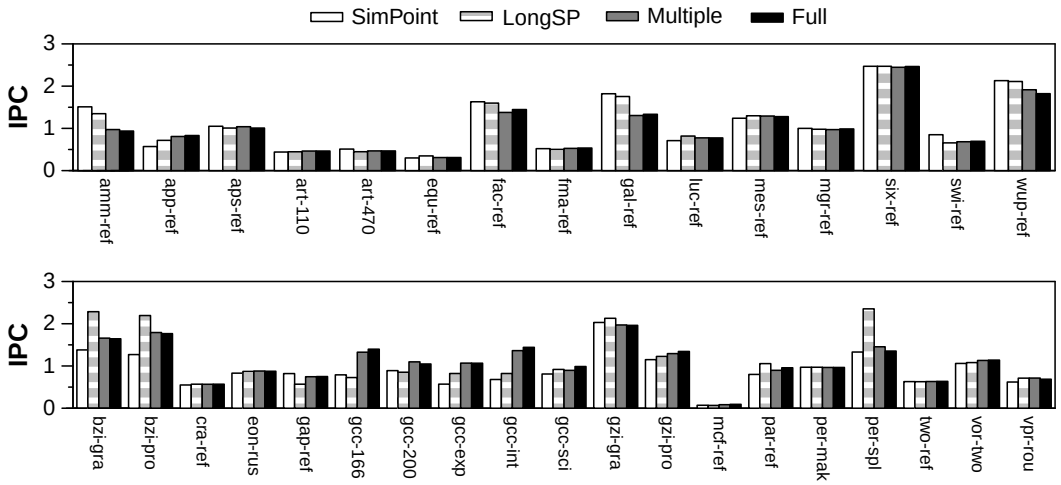

Figure II.15: Multiple simulation point results.
Simulation results are shown for using a single simulation point simulating for 100 million instructions, LongSP chooses a single simulation point simulating for the same length of execution as the multiple point simulation, simulation using multiple simulation points, and the full execution of the program.

the further benefit of breaking the simulation down into parallel components that can be distributed across many processors. This is the methodology we use in our simulator. For both cases results from the separate simulation points need to be weighed and combined to arrive at overall performance for the program [27]. Care must be taken to combine statistics correctly (simply averaging will give incorrect results for statistics such as rates).

Knowing the clustering alone is not sufficient to enable multiple point simulation because the cluster centers do not correspond to actual intervals of execution. Instead, we must first pick a representative for each cluster that will be used to approximate the behavior of the full cluster. In order to pick this representative, we choose for each cluster the actual interval that is closest to the center (centroid) of the cluster. In addition to this, we weigh any use of this representative by the size of the cluster it is representing. If a cluster has only one point, it's representative will only have a small impact on the overall outcome

of the program.

We pick a BBV inside a cluster the same way we choose single simulation points above, but considering only the points inside a cluster. For each cluster, we find the point in the cluster that is closest to the target vector (cluster center) in the 15-dimensional data. This allows us to choose the most representative point within the already tightly formed cluster, which in turn allows us to choose the most representative BBV interval within the different phases of execution.

The last part of the algorithm is to assign a weight to each simulation point. This weight is used to weigh the metrics collected in each simulation point to form an overall result. The weight is the size (number of BB Vectors) of the clusters. Each simulation point from a cluster is assigned a weight equal to the number of BBVs in that cluster divided by the number of BBVs in the complete run of the program.

Table II.3 shows the multiple simulation points found for all of the SPEC 2000 benchmarks. For these results we limited the number of clusters to be at most six for all but the most complex programs. This was done, in order to limit the number of simulation points, which also limits the amount of warmup time needed to perform the overall simulation. The cluster formation algorithm in Section II.D takes as an input parameter the max number of clusters to be allowed. Each simulation point contains two numbers. The first number is the location of the simulation point in 100s of millions of instructions. The second number in parentheses is the weight for that simulation point, which is used to create an overall combined metric. Each simulation point corresponds to 100 million instructions.

Figure II.15 shows the IPC results for multiple simulation points. The first bar shows our single simulation points simulating for 100 million instructions. The second bar `LongSP` chooses a single simulation point, but the length

of simulation is identical to the length used for multiple simulation points (which may go up to 1 billion instructions). This is to provide a fair comparison between the single simulation points and multiple. The `Multiple` bar shows results using the multiple simulation points, and the final bar is IPC for full simulation. As in Figure II.14, the closer the bar is to `Full`, the better.

The results show that the average IPC error rate is reduced to 3% using multiple simulation points, which is down from 17% using the long single simulation point. This is significantly lower than the average 80% error seen for blindly fast forwarding. The benefits can be most clearly seen in the programs `bzip`, `gcc`, `ammp`, and `galgel`. The reason that the long contiguous simulation points do not do much better is that they are constrained to only sample at one place in the program. For many programs this is sufficient, but for those with interesting long term behavior, such as `bzip`, it is impossible to approximate the full behavior.

Figure II.16 is the average over all of the floating point programs (top graph) and integer programs (bottom graph). Errors for IPC, branch miss rate, instruction and data cache miss rates, and the unified L2 cache miss rate for the architecture presented in Section II.B are shown. The errors are with respect to these metrics for the full length of simulation using SimpleScalar. Results are shown for starting simulation at the start of the program `None`, blindly fast forwarding a billion instructions `FF-Billion`, single simulation points of duration 1 (`SimPoint`) and k (`LongSP`), and multiple simulation points (`Multiple`).

The first thing to note is that using the just a single small simulation point performs quite well on average across all of the metrics when compared to blindly fast-forwarding. Even though a single SimPoint does well, it is clearly beaten by using the clustering based scheme presented in this chapter across all of the metrics examined. One thing that stands out on the graphs is that the

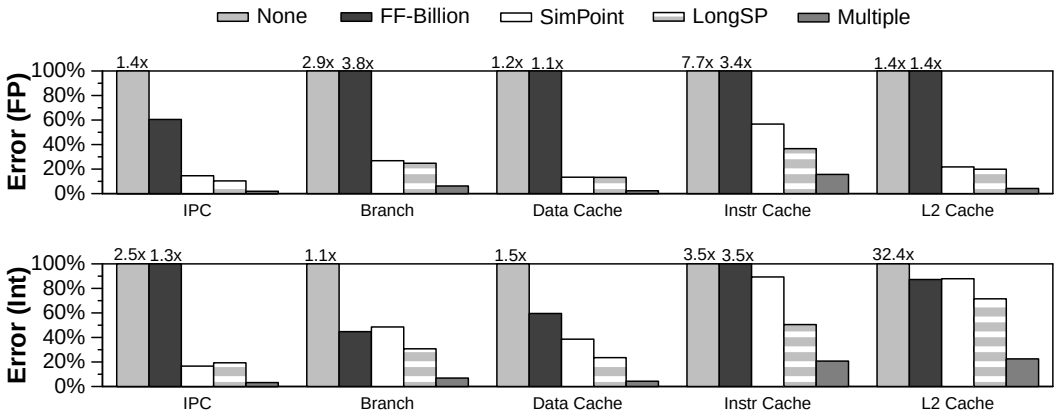

Figure II.16: Average error results for SPEC 2000
Average error results shown for the SPEC 2000 floating point (top) and integer (bottom) benchmarks for the metrics of IPC, branch misprediction, instruction, data and unified L2 cache miss rates.

error rate of the instruction cache and L2 cache appear to be high (especially for the integer programs) despite the fact that our technique is doing quite well in terms of overall performance. This is due to the fact that we present here an arithmetic mean of the errors, and there are several programs that have high error rates due to the very small number of cache misses. If there are 10 misses in the whole program, and we estimate there to be 100, that will result in a error of 10X. We point to the overall IPC as the most important metric for evaluation as it implicitly weighs each of the metrics by it's relative importance.

II.F Related Work

In this section we describe work related to finding simulation points, techniques for using sampling for simulation, and statistical simulation.

Time Varying Behavior of Programs: In [116], we provided a first attempt at showing the periodic patterns for all of the SPEC 95 programs, and how these vary over time for cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy. Skadron et. al [128] also exam-

ined creating similar time varying graphs for branch miss rates. They then used these graphs to manually choose where to fast-forward to pick out the simulation periods, similar as to what was proposed in [116].

Training Inputs and Finding Smaller Representative Inputs: One approach for reducing the simulation time is to use the training or test inputs from the SPEC benchmark suite. For many of the benchmarks, these inputs are either (1) still too long to fully simulate, or (2) too short and place too much emphasis on the startup and shutdown parts of the program's execution, or (3) inaccurately estimate behavior such as cache accesses do to decreased working set size.

KleinOsowski et. al [71], have developed a technique where they manually reduce the input sets of programs. The input sets were developed using a range of approaches from truncating of the input files to modification of source code to reduce the number of times frequent loops were traversed. For these input sets they develop, they make sure that they have similar results in terms of IPC, cache, and instruction mix.

Fast Forwarding and Check-pointing: Historically researchers have simulated from the start of the application, but this usually does not represent the majority of the program's behavior because it is still in the initialization phase. Recently researchers have started to *fast-forward* to a given point in execution, and then start their simulation from there, ideally skipping over the initialization code to an area of code representative of the whole. During fast-forward the simulator simply needs to act as a functional simulator, and may take full advantage of optimizations like direct execution. After the fast-forward point has been reached, the simulator switches to full cycle level simulation.

After fast-forwarding, the architecture state to be simulated is still cold, and a warmup time is needed in order to start collecting representative results. Efficiently warming up execution only requires references immediately proceeding

the start of simulation. Haskins and Skadron [56] examined probabilistically determining the minimum set of fast-forward transactions that must be executed for warm up to accurately produce state as it would have appeared had the entire fast-forward interval been used for warm up [56]. They recently examined using reuse analysis to determine how far before full simulation warmup needs to occur [57].

An alternative to fast forwarding is to use check-pointing to start the simulation of a program at a specific point. With check-pointing, code is executed to a given point in the program and the state is saved, or checkpointed, so that other simulation runs can start there. In this way the initialization section can be run just one time, and there is no need to fast forward past it each time. The architectural state (e.g., caches, register file, branch prediction, etc) can either be stored in the trace (if they are not going to change across simulation runs) or can be warmed up in a manner similar to described above.

Automatically Finding Where to Simulate: Our simulation work is based upon our basic block distribution analysis as described in prior sections. Recent work on finding simulation points for data cache simulations is presented by Lafage and Seznec [73]. They proposed a technique to gather statistics over the complete execution of the program and use them to choose a representative slice of the program. They evaluate two metrics, one which captures memory spatial locality and one which captures memory temporal locality. They further propose to create specialized metrics such as instruction mix, control transfer, instruction characterization, and distribution of data dependency distances to further quantify the behavior of both the program's full execution and the execution of samples.

Statistical Sampling: Several different techniques have been proposed for sampling to estimate the behavior of the program as a whole. These tech-

niques take a number of contiguous execution samples, referred to as clusters in [27], across the whole execution of the program. These clusters are spread out throughout the execution of the program in an attempt to provide a representative section of the application being simulated. Conte et. al [27] formed multiple simulation points by randomly picking intervals of execution, and then examining how these fit to the overall execution of the program for several architecture metrics (IPC and branch and data cache statistics). Our work is complementary to this, where we provide a fast and metric independent approach for picking multiple simulation points based just on basic block vector similarity. When an architect gets a new binary to examine they can use our approach to quickly find the simulation points, and then validate these with detailed simulation in parallel with using the binary.

To use sampling one has to address the issue of how to deal with the state of the machine when switching from one simulation point to starting the simulation of another simulation point. One option for providing meaningful results is to first sample a large number of sequential instructions in order to provide results due to the time it takes to warm up the architecture structures (.e.g, caches) as well as taking a large number of samples to be sure to capture the large scale behavior of the program. Conte et.al. [27] proposed another option for the reconciliation of such disjoint sample points, whereby the structures holding state are not reset between simulation points. For example, the cache would not be flushed and the BTB would not be reset when switching simulation from one point to the next. The hope is that the state of the machine from the end of one simulation point is similar to the start of another disjoint point.

Statistical Simulation: Another technique to improve simulation time is to use statistical simulation [96]. Using statistical simulation, the application is run once and a synthetic trace is generated that attempts to capture the whole

program behavior. The trace captures such characteristics as basic block size, typical register dependencies and cache misses. This trace is then run for sometimes as little as 50-100,000 cycles on a much faster simulator. Nussbaum and Smith [95] also examined generating synthetic traces and using these for simulation and was proposed for fast design space exploration. We believe the techniques presented in this chapter are complementary to the techniques of Oskin et al. and Nussbaum and Smith in that more accurate profiles can be determined using our techniques, and instead of attempting to characterize the program as a whole it can be characterized on a per-phase basis.

II.G Summary

A good understanding of the underlying behavior of programs is critical to the process of customization and adaptation. As we have shown in this chapter, many programs have wildly different behavior on even the very largest of scales (over the full lifetime of the program). While these changes in behavior are drastic, they are not without order, even in very complex applications such as `gcc`. In order to help future compiler and architecture researchers in exploiting this large scale behavior, we have developed a set of analytical tools that are capable of automatically and efficiently analyzing program behavior over large sections of execution.

The development of the analysis is founded on a hardware independent metric, *Basic Block Vectors*, that can concisely summarize the behavior of an arbitrary section of execution in a program. We showed that by using Basic Block Vectors one can capture the behavior of programs as defined by several architectural metrics (such as IPC, and branch and cache miss rates).

Using this framework, we examine the large scale behavior of several complex programs like `gzip`, `bzip`, and `gcc`, and find interesting patterns in

their execution over time. The behavior that we find shows that code and program behavior repeat over time. For example, in the input we examined in detail for `gcc` we see that program behavior repeats itself every 23.6 billion instructions. Developing techniques that automatically capture behavior on this scale is useful for architectural, system level, and run-time optimizations. We present an algorithm based on the identification of clusters of basic block vectors that can find these repeating program behaviors and group them into sets for further analysis. For two of the programs `gzip` and `gcc` we show how the clustering algorithm results line up nicely with the similarity matrix and correlate with the time varying IPC and data cache miss rates.

It is increasingly common for computer architects and compiler designers to use a small section of a benchmark to represent the whole program during the design and evaluation of a system. This sort of approximation is critical to doing any sort of design exploration needed when doing customization. This leads to the problem of finding sections of the program's execution that will accurately represent the behavior of the full program. We show how our clustering analysis can be used to automatically find multiple simulation points to reduce simulation time and to accurately model full program behavior. We call this clustering tool to find single and multiple simulation points *SimPoint*. SimPoint along with additional simulation point data can be found at: `http://www.cs.ucsd.edu/~calder/simpoint/`. For the SPEC 2000 programs, we found that starting simulation at the start of the program results in an average error of 210% when compared to the full simulation of the program, whereas blindly fast forwarding resulted in an average 80% IPC error. Using a single simulation point found, using our basic block vector analysis, resulted in an average 17% IPC error. When using the clustering algorithm to create multiple simulation points we saw an average IPC error of 3%.

Automatically identifying the phase behavior using clustering is beneficial for architecture, compiler, and operating system optimizations in general as well. To this end, we have used the notion of basic block vectors and a random projection to create an efficient technique for identifying phases on-the-fly [126], which can be efficiently implemented in hardware or software. Besides identifying phases, this approach can predict not only when a phase change is about to occur, but to which phase it is about to transition. The details of this scheme can be found in Chapter VI. We believe that using phase information can lead to new compiler optimizations with code tailored to different phases of execution, multi-threaded architecture scheduling, power management, and other resource distribution problems controlled by software, hardware or the operating system.

III

Component Specialization

III.A Introduction

As was mentioned in earlier chapters, customized embedded processors use compiler analysis and design automation techniques to take a generalized architectural model and create a specific instance of it optimized to a given application or very small set of applications. This process can range from whole processor synthesis to having a general processor core where a few specific architecture components are customized. Typically the design relies upon optimizing and combining well understood lower level primitives.

A common structure in both general purpose and custom processors is the finite state machine predictor, which is most commonly used as a branch prediction counter or for confidence estimation. The more established uses of these finite state machine predictors (e.g., branch prediction) are now finding their way into embedded processors and DSPs [76].

A *Finite State Machine* (FSM) consists of a set of states, a start state, an input alphabet, and a transition function that maps an input symbol and current state to next state. A Moore machine extends this with an output on each state. If one considers the case where the alphabet and output symbols

are constrained to be only $\{0, 1\}$, a finite state machine can be used to generate yes/no predictions. The output at a given state is its prediction of the next input. Either intentionally or accidentally, many FSMs in computer architecture are of this form, with the most famous of these being the two-bit saturating counter. In a branch predictor, these simple state machines generate predictions of taken after a series of takens, or not-taken after seeing a series of not-takens.

Branch prediction is the most common form of speculation in current processors [37, 149, 84, 129]. During the instruction fetch stage, a FSM is used to predict if a branch should be taken or not-taken, and then in the next cycle instructions from the predicted direction are fetched. Proposed branch prediction architectures can achieve predictions accuracies in the 95 percentile range. Even so, studies have shown that using perfect prediction can increase the IPC from 25% to over twice the IPC in comparison to these highly accurate predictors, when examining the SPEC 95 INT programs [99]. Most of the research in branch prediction concentrates on improving the hashing functions into the table of FSM predictors, how to combine multiple predictors, and eliminating alias effects. In this chapter we evaluate the benefit of using our automated approach to creating new FSM predictors used in the branch prediction tables.

Confidence estimation is used to guide speculation for many of the recently proposed speculative architectures. A confidence estimator is a FSM associated with a given predictor, which provides a measure of how accurate the corresponding prediction is. Value prediction is an example of a speculative approach that uses confidence estimation. Value prediction breaks true data dependencies in a program by predicting the outcome of a stalled instruction, and allowing dependent instruction to speculatively execute using the predicted value. Confidence estimation is critical to obtaining processor performance. Value prediction research has shown 40% speedups on average if perfect confidence estimation is

used, and only 10% speedup using traditional up/down saturating counters, when using a value miss-recovery architecture that re-fetches all the instructions after a value misprediction [22]. Since the cost of a misprediction is extremely high, predictions are only to be used when the confidence is high. Therefore, improving the prediction confidence can provide significant gains.

In this chapter we present an automated approach for creating finite state machine predictors. We use profiles to generate a language of predictions of a certain type, which is expressed as a regular expression. The regular expression is then converted into a FSM. The VHDL for synthesis is then generated from the FSM. Using this approach we can automatically generate FSM predictors that perform well over a suite of applications, tailored to a specific application, and even a specific instruction. We demonstrate the ability to automatically generate accurate FSM predictors tailored to specific branches for branch prediction. We then examine using our approach to generate accurate confidence estimation for value prediction to be used for a general purpose processor. The FSM is customized to achieve the best average performance over the whole suite of applications examined.

Section III.B describes other potential uses for FSM predictor customization and initial results for using our approach for automatic generation of confidence estimators for value prediction. Section III.C describes current FSM predictors used in computer architecture, and prior work into automatically generating predictors. Section III.D details our approach to automatically generate FSM predictors. Section III.E lays out the methods used for gathering the results in this chapter, while Section III.F applys these techniques to problems in confidence estimation. Section III.G uses our automated approach to generate a custom branch prediction architecture, presents miss rates for our custom architecture, and examines in detail some of the FSM predictors generated by our

approach. Finally, section III.H provides a summary of our research.

III.B Uses of Finite State Machine Predictors

In this section we describe how FSM predictors are used in a few areas of computer architecture, to motivate the use of our approach.

III.B.1 Value, Address, and Load Speculation

Confidence counters have been used to guide many speculative execution architectures. These include value prediction, address prediction, instruction reuse, memoization, load speculation, and memory renaming [21]. In all of these architectures, speculation occurs to hide latency caused by a real or potential dependencies. When a misprediction occurs, the penalty can be costly. Therefore, confidence estimation counters are used to guide when to use the speculative optimization. In Section III.F, we examine automatically generating FSM confidence counters to guide value prediction for a general purpose processor.

III.B.2 Branch Prediction

A branch predictor uses FSM counters to provide conditional branch prediction, with the 2-bit counter being the most wildly known FSM predictor [129, 149]. In designing a branch predictor for a processor, our approach can be used to generated FSM branch prediction counters customized to a whole workload for a general purpose processor, or for a specific application or individual branches for a customized processor. In Section III.G, we examine the performance of generating FSM predictors for specific branches in the context of building a customized processor.

III.B.3 Speculative Threaded Execution

There have been several recent studies into speculative threaded execution in order to obtain more parallelism. These range from executing both paths of a branch [58, 140, 141, 70] to speculatively generating threads to run ahead of the main thread of execution [132, 138]. These architecture designs use FSM predictors to predict when to spawn speculative threads or when to execute down additional paths.

III.B.4 Cache Management

Cache management schemes have been proposed that perform intelligent replacement [82], cache exclusion [139], and they use a small FSM counter to determine when the optimization should be applied. In addition, prefetching architectures have used FSM predictors to determine when to initiate prefetching for a load and to guide stream buffer allocation [124].

III.B.5 Power Control

Manne et. al. [81] examined using confidence estimation to find branches that had a high miss rate, and then for those branches, stall the fetch unit until the branch direction is resolved. This can save a significant amount of power for branches that have a high miss rate, and is beneficial for low power processors. Musoll [92] proposed using hardware predictors to predict whether an access to certain hardware blocks can be avoided in order to save power.

III.C Prior Work

In this section we summarize current work on FSM predictors and prior work into automatically finding or generating predictors.

III.C.1 FSM Predictor Implementations

The majority of FSM predictors used in prior research are *Saturating Up and Down* (SUD) counters. Four values define a SUD counter – (saturation threshold, correct increment, wrong decrement, and a prediction threshold). A SUD counter can have a value between 0 and the saturation threshold. If the prediction is correct, the counter is incremented by the correct increment value. If it is incorrect, the counter is decreased by the wrong decrement value.

The majority of current processors use a branch predictor that indexes into a table of 2-bit SUD counters [129, 149]. The counter is incremented when the branch is taken, and decremented when not-taken, with a saturating threshold of 3. When the counter has a value less than or equal to 1, the branch is predicted as not-taken. If the counter has a value greater than 1 then it is predicted as taken. Some of these branch prediction architectures have several prediction tables, and a *Meta* table of SUD counters are used to pick which predictor to use.

Jacobsen et. al. [65] proposed the idea of confidence estimation and examined its relationship to branch prediction. In their study, they used saturating up and down (SUD) counters, and *Resetting Counters* to provide the confidence estimation. A resetting counter resets the counter back to 0 when there is a misprediction. Lick et. al. [140] used profiling to search for branch history patterns that provided correct predictions and that also had a high degree of confidence. They then examined a confidence estimator that predicted high confidence when these patterns were seen, and low confidence when the other patterns occurred. Grunwald et. al. [50] presented several new metrics for evaluating confidence estimators and provided a detail comparison of using SUD counters, resetting counters, and static confidence estimation.

Our focus in this research [118] is an automated approach for generating small FSMs. The outcome of each state of our FSM counter is a binary decision

of yes or no. This captures most of the implementations of FSMs, but not all. Confidence estimators can return a number representing a probability instead of a binary decision, and several different actions could be performed based upon the degree of confidence returned. Even so, most confidence estimation implementations have only one prediction threshold for the SUD counter used, and therefore fits into the space of FSM counters our approach addresses.

III.C.2 Automatically Generating Predictors

Burtscher and Zorn [19] examined using profiles to find N-bit value prediction histories that were highly confident. For each possible history, their profile gave what the prediction probability would be if values were predicted using that history. They then used this profile, along with a desired accuracy they wanted to achieve, to select which histories would be used to generate value predictions, and which histories would predict low confidence. This was then used to guide confidence estimation for their value prediction architecture.

Chen et. al. [25] examined using techniques from data compression to improve the performance of branch prediction. They looked at using *Prediction by Partial Matching* (PPM), where there are M tables from size 2 to 2^M. Each PPM entry contains a frequency for the number of times the next bit was 0 (not-taken) and the number of times it was (1) taken. All of the PPM tables are then searched in parallel for each history length. The PPM table entry that had the highest probability was then used for the prediction.

Emer and Gloy [39] have the closest prior work to ours where they examine using genetic programming with feedback to search the predictor design space. They developed a language to describe valid branch prediction architectures, which consists of a variety of predictor primitives (e.g., counters, tables, values), their sizes, and functions to combine these primitives. Using genetic pro-

gramming techniques, they search for new predictors by performing crossovers and mutating recent candidates, and they evaluate each predictors potential by examining how well it performs for a given set of benchmarks. In contrast, our approach automatically builds FSM predictors from behavioral traces, without searching. Our approach can generate FSM predictors that cannot be represented easily or naturally by the branch prediction language defined by Emer and Gloy. Conversely, our approach does not generate or examine the design space of table based prediction architectures. Therefore, our approach is better for finding efficient predictors for small design areas, whereas their approach can potentially find better solutions for designs with larger areas, and it may be beneficial to combine the two approaches.

III.D Automated Design of a FSM Predictor

To design a FSM predictor we go through a fairly complex design flow starting with profile information and finishing with synthesizable VHDL code. While the techniques described here can be used for any sort of predictor, we describe building FSM predictors for branch prediction throughout this section to help explain our approach. For clarity, we use a more general notation of 1 and 0 instead of taken and not taken respectively. We start off with a high level overview of the design chain we have developed and then discuss each step in detail with an example.

III.D.1 Overview

Regular expressions provide us with a way to specify patterns and then have them converted into finite state machines because they are both related by formal language. A formal language is a set of strings, either finite or infinite, made up of a sequence of elements from an alphabet, in our case the set of zeros

and ones.

A regular expression provides a recursive way of testing an input string to see if it belongs to a given language. The alternative is to define an iterative way of testing a string, which is what a finite state machine does. The two are related because they both perform the same function, recognizing strings, and a mapping from one to the other can always be found.

We now show a way of exploiting this relationship for the purpose of creating a predictor. Let us say that $s_i = \{b_1, b_2, ..., b_i\}$, which means s_i is the string formed by the concatenation of all the input sequences from the start until i. Now let $s = \{s_1, s_2, ..., s_i\}$. This is the set of all possible inputs the predictor could see over time. If we pick a subset L, of s, where L is the set of input strings that satisfy some metric we have chosen, we can say that L is the language of predict 1.

In this way, the problem of creating a FSM predictor reduces to that of finding a regular expression that describes L in a compact manner. Once we have the regular expression that recognizes L we can use established methods to convert it into a finite state machine that recognizes L. If we do this translation correctly, the finite state machine will indicate that it recognizes a member from L when it sees an input string in L. Because we picked L to be only the subset of s that we want to predict 1 on, we know that when we recognize this input string sequence that we should predict 1.

To find the language L we use profile information from the application. From the profile a model of the data is built, and this model can then be analyzed to find histories that are biased toward predicting 1. This set of histories is then compressed to a usable form and converted into a regular expression.

III.D.2 Modeling

To design a FSM predictor we start by tracing the target application suite to create a representative sequence of predictions for the applications.

An issue is determining what to trace, which depends on the intended use of the predictor. For branch prediction, we have two states 0 and 1, and the trace consists of a series of branch outcomes. We will use the following trace for the rest of the examples in this section:

$$t = 0000\ \ 1000\ \ 1011\ \ 1101\ \ 1110\ \ 1111^{1}$$

The next step after trace generation, is to build up a statistical model for the data. For this we use an Nth order Markov Model. An Nth order Markov Model is a table of size 2^N which contains $P[1|last\ N\ inputs]$ for each of the possible 2^N last N inputs in the trace. N serves as a limit to the amount of history that we may use in making our predictions.

For our example trace, we build up the following probabilities for a second order Markov table (N=2):

$$P[1|00] = 2/5 \quad P[1|01] = 3/5$$
$$P[1|10] = 3/4 \quad P[1|11] = 6/8$$

$P[1|00]$ is generated by finding all times that 00 is followed by a 1, in this case there are 5 cases of the pattern 00, and 2 of them are followed by a 1.

The corresponding predict 0 probabilities are calculated by subtracting the predict 1 probability from 1 because we only have 2 symbols in our alphabet. Note that while this scales exponentially with the size of N, it is still very reasonable for even the largest values of N we have examined. Having more knowledge of history after a certain point does not improve accuracy and we found that

[1] the trace is divided up into groups of four only for readability, it has nothing to do with the trace itself

point was well within the reach of the techniques described. For the predictors we are generating we did not see the need to go beyond $N = 10$.

III.D.3 Pattern Definition

Now that we have the probabilities that we need, we can continue with the next step which is picking the histories that we will eventually build the language L from. In the case of a branch predictor, where we simply wish to minimize the number of mispredictions, the task is straight forward. We simply pick all the histories that have a probability of preceding a 1 which is greater than or equal to 1/2 to form the language "predict 1". In our example above, that would be the set of histories $\{01, 10, 11\}$. We would like to predict a 1 whenever one of these histories appears in the input based upon our profile, since they had a probability greater than 1/2. Of course histories with probability equal to 1/2 can go either way.

We can make further design trade offs at this point in the form of don't care patterns. Some patterns only occur very rarely, and their inclusion into the set of histories will have almost no effect on the performance of the predictor but will make the job of minimizing more complicated. These history patterns can be placed into a third set, separate from the "predict 1" and "predict 0" sets, called the "don't care" set. We have found that by placing only the 1% least seen histories in the "don't care" set can reduce the size of the predictor by a factor of two with negligible impact on prediction accuracy.

Once we have passed this stage of the design flow, the function of the predictor is set and will not change significantly. The finite state machine that will be designed will return a 1 when histories in the "predict 1" set are seen, and 0 when histories in the "predict 0" set are seen. The output of the "don't care" set is still undecided at this point, and the next stage will take advantage of this

to compress the description of the sets.

III.D.4 Pattern Compression

Now that we have our three sets, "predict 1", "predict 0" and "don't care", the next step is to compress our description of the sets into a usable form. This is done by a standard logic minimization tool, where the input is a truth table. The input side of the truth table is the history patterns captured by our Markov Model, and the output side is the set that the history pattern belongs to. If we continue with our example, we have the out set partitioned up as follows:

$$\begin{aligned} \text{“predict 1”} &= \{01, 10, 11\} \\ \text{“predict 0”} &= \{00\} \\ \text{“don't care”} &= \emptyset \end{aligned}$$

From this we generate a truth table, where all the histories that are contained in the "predict 1" set have their output defined as one, and likewise for the "predict 0" set.

$$\{00 \rightarrow 0,\ 01 \rightarrow 1,\ 10 \rightarrow 1,\ 11 \rightarrow 1\}$$

We then use the logic minimization tool Espresso [109], to minimize this truth table down to a set of logic functions that describe the set of inputs that produce a 1. The logic function that is output is our compact description of the "predict 1" set, and it recognizes those histories that we wish to predict 1 or 0 on. This step also merges the "don't care" set into the "predict 1" and "predict 0" sets in such a way as to minimize the number of unique terms. The logic function that is generated is a sum of products representation. The "predict 1" set will satisfy this function and the "predict 0" set will not.

$$((x\ 1) \vee (1\ x))$$

Here we see the sum of products representation of our truth table, where an x represents an input that is unimportant to the outcome of the function. From this description we can now build our regular expression that will capture the language L.

III.D.5 Regular Expression Building

Once we have the minimized representation of the set of histories we need to build a regular expression which will match these history patterns whenever they come up in the sample input. More specifically, whenever we see an input string, whose trailing N bits match the minimized patterns for the "predict 1" set we return true.

We can build a regular expression as follows. Each term is a clause, each 0 is a 0, each 1 is a 1, and each don't care, denoted as x, matches either a 0 or a 1. Let use start with a single term, from above, $(1\ x)$. This translates to the regular expression $1\{0|1\}$ because it is a 1 followed by either a 0 or 1. Similarly $(x\ 1)$ translates to $\{0|1\}1$. Since a match can be caused by either one, we write the whole expression as $\{1\{0|1\}\} \mid \{\{0|1\}1\}$. However this is not quite complete.

Remember that the language L must describe all possible inputs for which it needs to return 1, not just the last two bits of the input. We overcome this problem by specifying the language to be any string that ends in the desired histories, which can be done by concatenating any number of 1s or 0s in the front of the histories. Thus, our final regular expression for the above history is:

$$\{0|1\}^{\star}\{\ 1\{0|1\} \mid \{0|1\}1\ \}$$

Once we have the desired regular expression there is still the matter of converting it to an efficient FSM.

III.D.6 FSM Creation

The first step in building a FSM from a regular expression is the construction of a non-deterministic finite state machine, which is a state machine that can be in more than one state at a time. Building a non-deterministic FSM is a fairly straight forward process of enumerating paths. Once the non-deterministic FSM is completed it is converted to a deterministic state machine using subset construction [59]. Subset construction can sometimes lead to an exponential blow up in the number of states, and there are better algorithms known, which take advantage of reduced alphabets, but we have found subset construction to be more than sufficient for the predictors we examine in this chapter.

At this point we now have a fully implementable finite state machine, however there are two more steps we take to reduce the number of states used by the state machine. We start by applying Hopcroft's partitioning algorithm [59]. This algorithm removes both unreachable and redundant states, although there are still unneeded states in the state machine.

III.D.7 Start State Reduction

Since the state machine must recognize all strings in the language, there are unnecessary start up states that are only used at the beginning of the execution where history bits are undefined. Since we are only interested in the steady state operation of the state machine, i.e. those strings with a length greater than N, we can cut out those nodes that are only used in parsing strings less than N in length. This goes against what was said earlier about not changing the semantics of the state machine, but this optimization only effects the behavior of the state

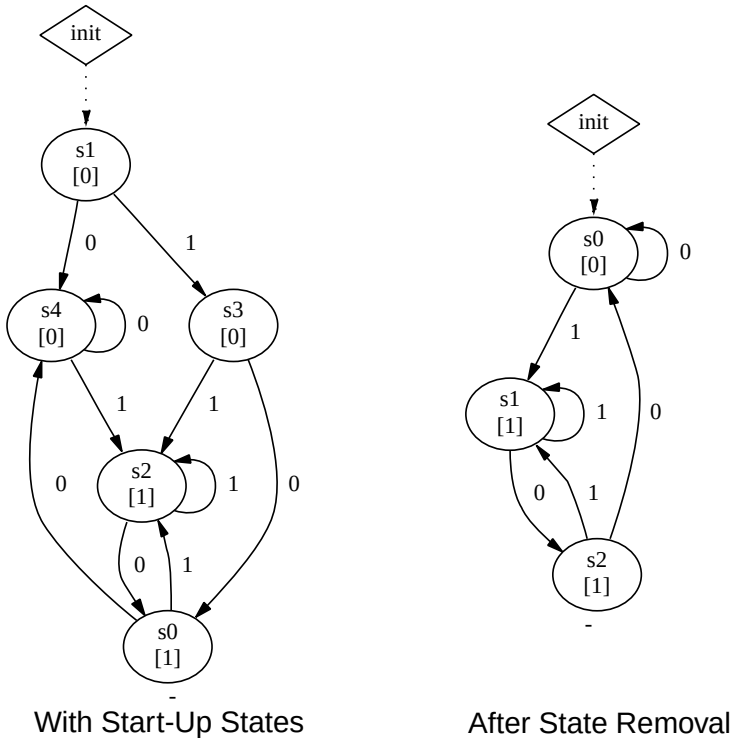

Figure III.1: State machine predictor optimization example

The state machine on the left was generated from the sequence t shown in section III.D.2 and optimized with Hopcroft's partitioning algorithm. The state machine on the right is what results from removing the start-up states and renumbering. This is the final state machine for t. Note that the patterns ending in 01, 10, and 11 are still captured correctly. The number in brackets shows the prediction produced by each state in the state machine.

machine on a small constant number of strings. There can be up to 2^N start-up states, and they typically account for around one half of all states in the machine.

If we look to Figure III.D.7 the problem can be clearly seen. The figure shows the state machines generated from our original sequence t. In the state machine on the left, the states S1 and S3 are not need needed after just the first couple of accesses to the state machine. We would like to remove these to save both state and transition logic complexity. This can be done by removing all nodes unreachable from the steady state operation of the state machine. All start-up states are unreachable from the steady state of operation because you can never have undefined history past the start-up phase. We exploit this fact by simply removing all nodes unreachable from steady state. Figure III.D.7 shows that the start-up states S1 and S3 have been removed and the remaining states have been renumbered. Note that the behavior of the finite state machine is still unchanged past the removed start-up states.

III.D.8 Synthesis

At this point we have almost reached our final destination. We have a finite state machine that produces a prediction based on the input. The predictions are governed by the last N bits of the input string, and that information is efficiently encoded into a finite state machine, as can be seen in Figure III.D.7. Starting from any node, and traversing the edges in the FSM following patterns (01, 10, 11) in our "predict 1" set will end at a node with a prediction of 1. Similarly, we will predict 0 for the pattern (00) in our "predict 0" set, starting from any node.

The final step is to actually do synthesis. Since the finite state machine is a well understood and commonly used primitive, every synthesis tool has some form of finite state machine input format. The job of synthesis is to find an

efficient hardware implementation for the state machine. This includes finding a good encoding for the states and their transitions. We translate our description of the finite state machine to VHDL, which is then read and analyzed by the Synposys design tool.

III.E Methodology

We used two sets of benchmarks to gather the results for this chapter. The first set of benchmarks were chosen because of their interesting confidence estimation behavior for value prediction as shown in [21]. These programs were `groff`, `gcc`, `li`, `go`, and `perl`.

The other set of benchmarks were chosen based on their applicability to an embedded environment and because they had interesting branch behavior. Three of the applications `compress`, `ijpeg`, and `vortex` are from the SPEC95 benchmark suite. We also include three programs from the MediaBench application suite. `Gsm decode` does full-rate speech transcoding, `g721 decode` does voice decompression, and `gs` is an implementation of a postscript interpreter.

We used ATOM [135] to profile the applications and generate the Markov Models over the full execution of the benchmarks. All applications were compiled on a DEC Alpha AXP-21264 architecture using the DEC C compiler under OSF/1 V4 operating system using full optimizations (`-O4`). Traces were gathered for 300 million instructions from the SimPoints recommended in Chapter II. After the Markov models were generated, generating all of the FSM predictors for each program using our automated approach took from 20 seconds to 2 minutes on a 500 MHZ Alpha 21264.

III.F FSM Predictors in General Purpose Processors

Structures in general purpose processors are customized to a suite of common applications used to evaluate the architecture during its design. This can include the SPEC2000 and other benchmark suites. Any customization applied to a general purpose processor needs to be general purpose, so specializing to individual programs is not practical. Therefore, we use an aggregate trace of several benchmarks together to represent the common behavior of the suite of programs. We then build a customized predictor to this aggregate trace, which is used to represent the general purpose behavior of the workload the architecture is designed to.

III.F.1 Value Prediction

Several architectures have been proposed for value prediction including last value prediction [78, 77], stride prediction [45, 48], context predictors [112], and hybrid approaches [142, 105]. In this research we focus on using a stride-based value predictor, since it provides the most performance for a reasonable amount of area devoted to value prediction [20].

A *stride* value predictor [26, 38, 112] keeps track of not only the last value brought in by an instruction, but also the difference between that value and the previous value. This difference is called the stride. The predictor speculates that the new value seen by the instruction will be the sum of the last value seen and the stride. We chose to use the two-delta stride predictor [38, 112], which only replaces the predicted stride with a new stride if that new stride has been seen twice in a row. Each entry contains a tag, the predicted value, the predicted stride, the last stride seen, and a saturating up and down confidence counter. We use a table size of 2K entries for the stride value predictor, which means there are also 2K confidence counters used (one for each table entry). We performed

value prediction for only load instructions.

III.F.2 Value Prediction Confidence

There have been several chapters that point out the benefit of applying value prediction to various troublesome instructions. It has further been shown that actually predicting the values of these instructions is error prone. It was noted by both Calder et. al. [22, 21] and by Burtscher and Zorn [19] that in order for value prediction to be a viable option, an improved confidence estimation technique is needed. Confidence estimation is a way for the processor to more prudently allocate resources based on whether or not it believes the values predicted are correct or not. In effect, two predictions are made, a value prediction and a meta-prediction as to whether or not the value prediction was correct.

In [22, 21], Calder et al. examined *Saturating Up and Down* (SUD) counters with several different saturation thresholds and wrong decrement values to vary the accuracy and coverage, and this was done by trial and error. They showed that no one SUD counter worked best for all programs. A very accurate SUD counter was needed for mispredicted values when using squash recovery to obtain increases in performance, but this resulted in low coverage of potential value predictions. In contrast, when value prediction used re-execution recovery, it did not have to be as accurate, since the miss penalty is small, and the SUD counter could instead concentrate on achieving a high coverage.

III.F.3 Automatically Designing Confidence Predictors

Instead of using a Saturating Up/Down counter for estimating the confidence of the value predictions we propose using our automatically designed FSM predictors to generate the confidence estimation hardware.

Each time a load was executed, we put into the trace, which was used to

create the Markov model, whether the load was correctly value predicted (1) or not (0). We then applied the techniques in Section III.D to automatically generate an FSM predictor. This automatically generated FSM provides a confidence prediction for each executed load predicting if it will going to be correctly value predicted or not. Because we merge all of the traces into a single Markov model, the resulting state machine will work well on any of the traces which were combinined into the model. This means that to make a confidence estimator using this technique, you instantiate many of these state machines into a table just as you would with an up/down counter. The difference is that the automatically designed state machines better capture the set of events that lead to a correct prediction.

We use the same design techniques as described in Section III.D, except that instead of targeting a single application we target a suite of general purpose applications. In order to make sure that the predictors generated are not specialized to a *specific* application, but rather to a suite of applications, we provide cross-trained results. For each application in our suite, we combine the traces from all of the other programs *excluding* the application to be used for reporting results. So for example, when we show results for `gcc`, the predictor has been trained on all of the other programs in the suite, but not on `gcc`.

III.F.4 Results

Figure III.2 shows value prediction confidence for saturating up/down counters versus state machine predictors that were designed automatically with cross-training across all benchmarks. The x-axis is the Accuracy of the value predictor when using the various confidence schemes. The accuracy is the percent of value predictions that were marked as confident, that were in fact correct predictions. An accuracy of 100% means that every-time the confidence predictor

marked a prediction as confident, it was a correct prediction. The y-axis is the coverage which is defined as the percent of correct value predictions that were allowed through by the confidence predictor. A coverage of 100% means that all of the times that the value predictor predicted a value correctly, it was marked as confident by the confidence predictor.

Accuracy and coverage trade off with each other. It is possible to get higher accuracy by being conservative and only marking those value predictions that you are very sure are correct as being confident. On the other hand, higher coverage can be had by being more trusting of the value predictor. This tradeoff can be clearly seen in figure III.2. The points for the up/down counters are the accuracy and tradeoff points for a variety of counter configurations. The points shown are for counters with a maximum value (number of states) of 5, 10, 20, and 40, miss penalties of 1, 2, 5, 10, and full, and for thresholds of 50% 80% and 90%.

This is to be compared to the lines in Figure III.2 that show the coverage and accuracy tradeoffs that can be made by using automatically designed state machines with history lengths of size 2 to 10. These results indicate that there is a significant amount of coverage to be gained for any given accuracy. Take `gcc` for example, at a target accuracy of 80%, the best configuration of saturating up-down counter gets a coverage of less than 10%. In comparison, our automatically designed state machine (derived from cross-training) is capable of covering 20% of correct predictions, which would result in a more than twice as many correct value predictions at the same accuracy.

It is interesting to note that our automatically generated FSM predictors converge with the saturating up-down counter results for extremely high accuracy requirements. This is because, in order to achieve the highest accuracies possible, the best solution is to simply wait for *very* long sequences of correct predictions

before becoming confident. Being this conservative comes at price though, which is very low coverage.

III.G Branch Predictors for Customized Processors

In this section we examine applying our automated approach to designing FSM predictors to the customization of branch predictors for customized processors. We begin with a discussion of our customized branch prediction architecture, and then the training approach we use for building the customized FSM branch predictors. We then evaluate the performance and area of the customized predictor and compare it to a range of prior branch predictors.

III.G.1 Customized Processors

As was mentioned in the introduction, the rapidly growing embedded electronics industry demands high performance, low cost systems with increased pressure on design time. The designer could choose to design a custom ASIC, which has the advantages of high performance at the cost of long design and verification times. An alternative choice is an off the shelf embedded processor. Embedded processors allow rapid development times and low development cost in terms of both time and money, but with performance typically lagging behind ASICs.

To address this problem there is an emerging technique which will add a new and important point in the spectrum of solutions. Automatically generated customized processors have the promise of delivering the needed performance with only slightly inflated design time.

A customized processor is a processor tailored to an individual application or a set of applications. The idea became a common research subject in the late eighties and early nineties with projects such as SCARCE [90] and The Ar-

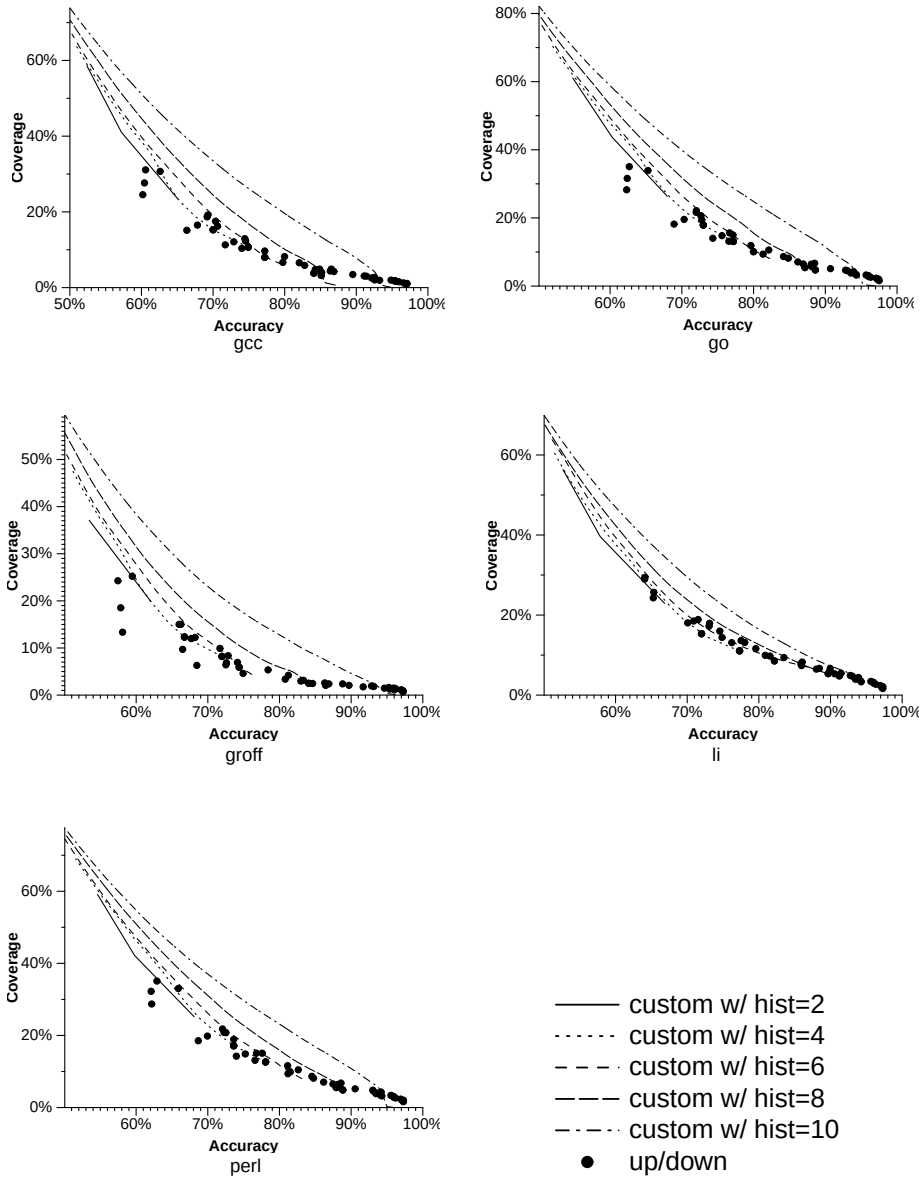

Figure III.2: Comparison of SUD versus custom confidence predictors
Value Prediction Confidence for saturating up/down counters versus state machine predictors that were designed automatically from cross-training across benchmarks. The x-axis is the Accuracy of the value predictor when using the various confidence schemes. The y-axis is the percent of correct value predictions that were allowed through by the confidence predictor.

chitect's Workbench (AWB) [44]. SCARCE was a flexible VLSI core for building application specific processors. AWB was a system built to help the designers evaluate design tradeoffs for building embedded processors.

Currently there is a resurgence of interest customized processors with many research projects and companies working in this domain. A great deal of the interest comes from increased system level integration, where it is increasingly common to place different parts of a system all onto one chip. To support this it is now common for vendors to sell descriptions of processor cores rather than actual silicon itself. These processor core descriptions can then be customized for a given application.

There are two major approaches to customized processors, one approach working towards pre-silicon customization, and the other approach pressing for post-silicon reconfiguration or adaptability. The first approach, which is being adopted by such systems as Hewlett Packard's PICO project [103, 3, 114] and Tensilica's Xtensa [49, 40], is intended to produce low-cost high-speed fixed hardware for embedded systems. The other approach attempts to take advantage of post-silicon customization through reconfigurability. Examples of systems that support reconfigurability are Altera's NIOS [131] processor core and the CHIMEARA chip [154].

There is also another range of freedom for these chips, which is the design level. Some systems, such as CHIMEARA [154], and PRISC [104], concentrate their application tailoring at the level internal to a functional unit. These systems work by tailoring the processor's functional units to the application running on it. For example, they might merge commonly executed expressions into a single instruction. Other options are to perform the customization at the architectural level of registers and number of functional units such as Xtensa [49], SCARCE [90] and Lx [43]. This allows the system designer to make high level

architectural tradeoffs for the application such as relieving register pressure or removing unused functional units. Still other designs have co-processors for a given application or type of application. Xtensa [40] supports the tight integration of co-processors into an architecture, while PICO [103, 3, 114] automatically generates co-processors in the form of a custom designed systolic array.

Even though there is a long history and many different projects, all of the projects have a similar high level overview. Begin with a configurable or parameterizable architectural template and customize it to fit the application at hand. Because all of these systems target a very specific application or suite of applications, and these applications are under the designers control, any one of these approaches could benefit from the techniques we present.

III.G.2 Customized Branch Prediction Architecture

It is increasingly common for current embedded processors to have branch prediction. For example, Intel's XScale (StrongARM-2) processor [76] has a 128 entry Branch Target Buffer (BTB), and each entry in the BTB has a 2-bit saturating counter which is used for branch prediction. Our goal is to build a branch predictor for a given application that will have the performance of a large general purpose predictor, with about the same area that is already being used by embedded branch predictors.

Our approach is to take a baseline predictor such as the local 2-bit counters from the XScale architecture, and extend this with custom designed FSM predictors for the hard to predict branches. We use the standard two bit counters for most branches and use the custom FSM predictors for branches that do not work well with the default predictor. In doing this, we limit both the amount of additional area we have to add to get good performance and the amount of code we have to hard-wire into the processor.

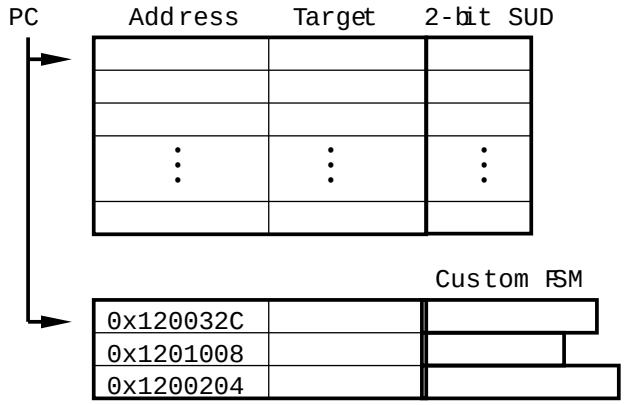

Figure III.3: Customized branch prediction architecture.

The architecture contains a traditional coupled BTB with 2-bit saturating up-down counters for conditional branch prediction. In addition, customized branch predictors are added for individual branches. A customized branch entry contains a tag (address), target and a custom FSM predictor for predicting the direction of the branch. The tag is associated with the branch that the FSM predictor was built for and is locked down by the system software.

The custom branch architecture we propose is seen in Figure III.3. We extend XScale's coupled BTB branch prediction architecture with a set of custom predictors that are hard-wired to particular branches. These custom predictors have the addresses that are associated with them locked down by the system software. While the state machines are fixed in hardware and are targeting specific branches, we wish to allow some software configurability should a re-compile of the software be needed after fabrication. This will allow the branches to move around in the address space but still use their custom state machines as long as the branch prediction patterns do not change.

The address of the branch is used to index into the BTB as well as the custom predictors. The custom branch entries perform a fully associative tag lookup to search for a match. If there is a match in the standard table then the two-bit saturating up-down counter is used to predict the bias of the branch. If instead there is a match in the custom table then the output on the current state

of the corresponding state machine is used for prediction. In the next section we describe how we generated the FSM predictors hard-wired into this architecture.

III.G.3 Generating Traces to Train FSM Branch Predictors

There are many different types of branch traces one could concentrate on to generate FSM predictors, and we only focus on one in this chapter. For the custom branch prediction architecture in Figure III.3, traces are generated on a global basis, and then used to generate a FSM predictor for individual branches. To generate a FSM for a specific branch, the traces used to train the FSM generator could include the local history of the branch, global history, or a combination of the two. We examined all of these and found that it is better to concentrate on capturing global correlation rather than a local history pattern because of the tendency of the global correlation to be more repeatable across different inputs, and our approach is efficient at finding and taking advantage of global correlations between branches.

The first step we perform in building our custom branch prediction architecture is to profile the application with our baseline predictor, in this case the Xscale predictor, which is a bimodal table of 2-bit counters. This identifies those branches that are causing the greatest amount of mispredictions. For each of these branches we generate traces and use them to create a Markov Model as discussed in Section III.D. To generate the Markov Models that we need for the branches we are concentrating on, we keep track of a single global history register of length N. When a branch is encountered in the trace, we update that branch's Markov Model with the outcome of the branch, given the history in the global history register. The Markov table is then fed into the FSM generator, and a customize FSM predictor is generated for that branch. Since the number of global histories that a given branch might see before it is being predicted is small

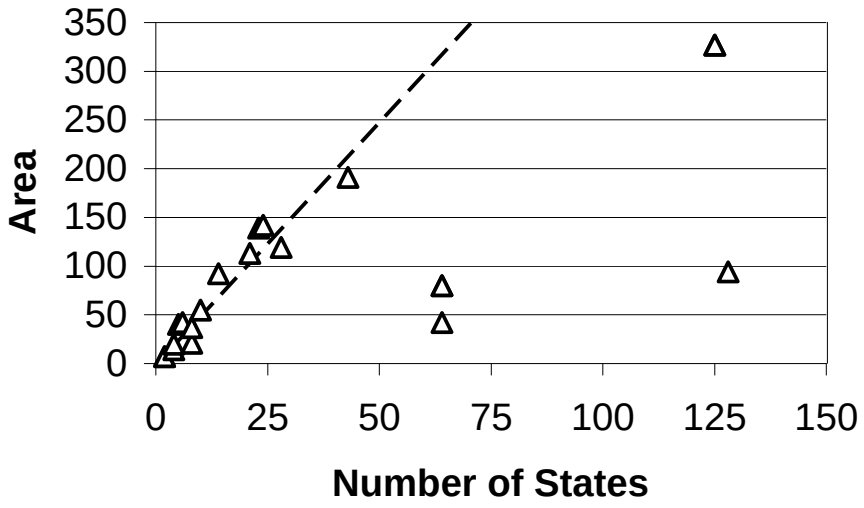

Figure III.4: Area versus number of states for custom predictors

The area of a random sample of the custom FSM predictors taken from all of the benchmarks we examined is shown. The area was determined by Synopsys and plotted against the number of states in that machine. The strong linear bound allows us to estimate area for design tradeoffs and evaluation

compared to the 2^N possible histories, the Markov Models can be compressed down significantly by only storing non-zero entries. For all the custom branch prediction results in this chapter we use a history of length 9.

For this design only the branch that the custom predictor is generated for uses the FSM for prediction, based on a tag match as described earlier. We update the custom finite state machines in a different manner than the standard local 2-bit counters. We update all of the custom predictors in parallel on every branch, rather than only matching branches because the prediction is based on gloabl history. On an update, the branch predictor moves each FSM predictor from one state to the next state based upon the prediction of the branch.

III.G.4 Estimating Area of FSM Predictors

In order to enable us to make high level design choices before synthesis, it is important to be able to estimate the area that our automatically generated state machines will take up. To establish a relationship between the state machine descriptions and their area, we took a random sample of custom FSM predictors generated across all of the benchmarks we examined and we then synthesized these state machines with Synopsys. Our samples account for 10% of all of the FSM predictors generated.

Figure III.4 shows the number of states in the state machine versus the area of the implementation. Each triangle represents one FSM predictor. The x-axis is the number of states in a given state machine, while the y-axis is the area as reported by Synopsys.

The results show, for most state machines, that the area is linearly proportional to the number of states in the machine. This trend line is drawn in with a dashed line. However this is not the case for all of the FSM predictors. For the state machines with a large number of states, the area is much less than would be predicted by this approximation. For these FSM predictors, there is a large number of states, but the machine is highly regular. Because of this, the state machine can be optimized down to a size much smaller than even some of the more chaotic state machines with less state.

We use the line in Figure III.4 to estimate the area for the rest of the FSM predictors in the results to follow. Even though the approximation does not hold for all of the predictors, it does bound the area of the predictors by the number of states in the state machine. We can use this approximation to make conservative estimates of area. For the rest of this chapter, we use this approximation to quantify area rather than performing synthesis on each state we wish to examine.

III.G.5 Branch Prediction Results

We compare out customized predictor against three other branch predictors. The first predictor we compare against is the the gshare predictor of McFarling [83]. The second predictor is a meta chooser predictor that contains a two-level local history branch prediction table, a global history table, and a meta chooser table that determines whether to use the local or global prediction for predicting the current branch. We call this the *Local Global Chooser* (LGC) predictor, and it is similar to the predictor found in the Alpha 21264. The final predictor we compare against is a set of per-branch two bit counters, as is found in the XScale processor. The XScale processor has a 2-bit counter associated with every branch target buffer entry, and not-taken is predicted on a BTB miss.

Figure III.5 contains the results for the six programs comparing our custom FSM predictors to the gshare, LGC, and XScale predictors. The x-axis is the total area of the predictor, including the BTB structure, while the y-axis is the misprediction rate. The results for the LGC and gshare were gathered over a range of sizes. We present results for custom predictors trained on inputs, which are different than the ones used for measuring performance. These results are denoted custom-diff. The custom-same results are when we use the same input for training and comparison, and it provides a limit to how well the FSM predictors may perform for that input.

The curve for the custom FSM predictors is generated by increasing the number of branches to be customized. The top-left most point in the curve is the XScale architecture with one custom branch predictor. The points on the curve immediately to the right are for using 2 custom branch predictors, then three, and so on. As we add more FSM predictors, the number of mispredictions is reduced and at the same time the area to implement the predictor grows. The results show that for all programs the misprediction rate decreases as we devote

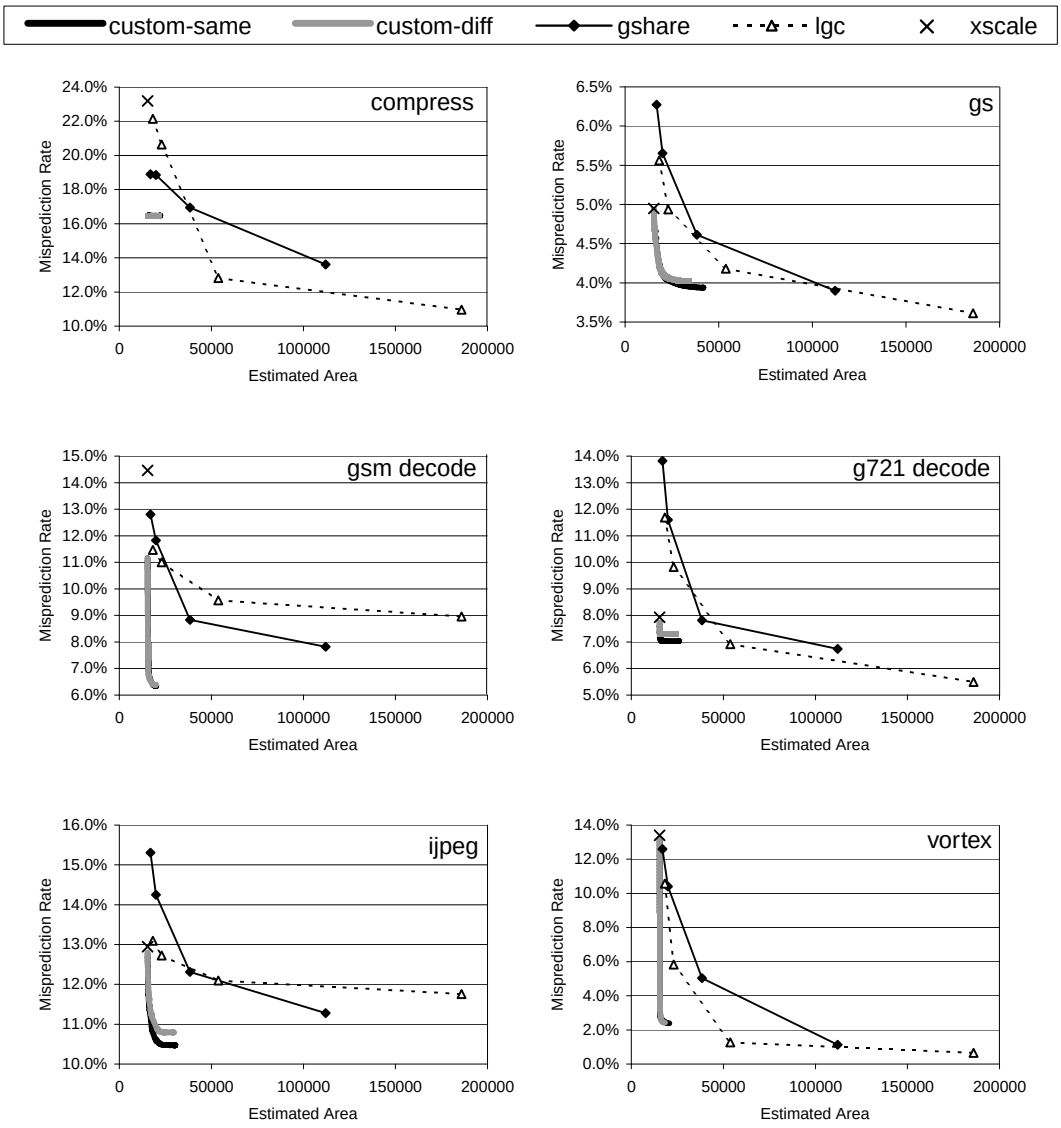

Figure III.5: Branch predictor accuracy–area tradeoffs
The misprediction rate versus estimated area for the six benchmarks examined is shown. Results are shown for the baseline XScale predictor, gshare, a meta predictor with a chooser between local and global history (LGC), and the customized branch predictor. The customized branch predictor results are for training on the same input used as gathering results testing, custom-same, and training on a different input from gathering results, custom-diff.

more and more chip area to the prediction of branches.

The first thing that is very noticeable is that there is little to no difference between custom-diff and custom-same. This implies that our training output has done a good job capturing the behavior that is inherent to the program. One could certainly use more than one input for training, and indeed this would be a good idea for verifying that the models generated correctly capture the behavior of the program.

For all of the programs, the custom predictors achieve the lowest misprediction rate of any predictor for their size. To beat the performance of these small predictors you would need a general purpose predictor that is 2 to 5 times larger. For some of the programs, even these large table sizes cannot perform better than our custom predictors.

For the program `compress` all of the benefit comes from the state machine for one branch. The misprediction rate is reduced from 23% to 16.5% by adding one custom FSM predictor. Adding more FSM predictors simply increases the area with little to no improvement in misprediction rate. Moderate table sizes of a LGC can outperform our customized predictors because there the branch causing the most mispredictions benefits from having local history. This branch would benefit from having a loop count instruction in a embedded processor, or could easily be captured via customizing the branch predictor to perform loop termination prediction to predict the branch [117].

For `g721`, we can see that the XScale does a very good job of capturing the behavior of most of the branches in the program. However with little extra area we can reduce the misprediction rate from 8% to just over 7%. For `vortex` and `gs` the misprediction rate is reduced significantly from XScale's default local 2-bit counters. For `gs` it is reduced from just under 5% to just over 4%, and for vortex the improvement is a reduction in miss rate from 13% to 3%. For these

two programs, a local-global chooser of over two times the size of the custom predictor is needed to get a lower misprediction rate.

The best results are seen for `ijpeg` and `gsm`. For both of these programs the misprediction rate is far below that of even the largest table we examined. These programs do not benefit from local history. This can be seen by comparing how LGC performs against gshare for these programs. Since our scheme captures the global correlation so efficiently we can outperform the largest gshare by half a percent for `ijpeg` and over a full percent for `gsm` while only using a fifth of the area.

III.G.6 Custom Finite State Machine Examples

Figures III.6 and III.7 are two examples of simple finite state machines that were generated using the techniques presented. Figure III.6 serves as a good starting point for understanding how the state machines are used to generate predictions.

The state machine in figure III.6 was automatically generated to target a particular branch in `ijpeg`. Each state is annotated with a prediction, shown in brackets. The state machine is updated by traversing an edge. An update with taken will traverse the edge labeled 1, while not-taken will traverse the edge labeled 0. This particular state machine was generated to capture a branch that is highly correlated with the branch that is two branches back in the history. For example, we would like to predict a 1 if the history pattern is "10" or "11" and predict 0 in all other cases. As can be seen in the figure, this is successfully captured by the finite state machine.

If you start in any state of the machine and you follow two transitions, first a 1 and then either a 0 or a 1, you will end up in a state that is labeled a 1. The converse is also true. If you follow an edge labeled 0, followed by either a 1

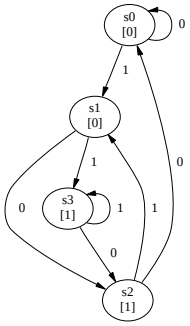

Figure III.6: Simple state machine predictor example
This simplefinite state machine is generated for a branch in `ijpeg` and captures the history pattern $1x$.

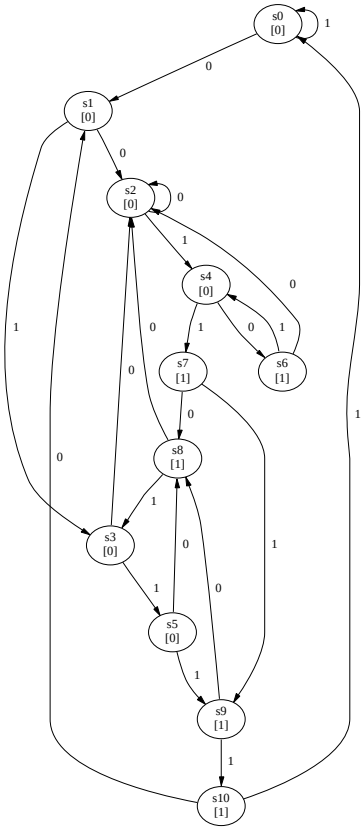

Figure III.7: A more complex state machine predictor

A slightly more complex state machine generated from `gs` that captures two patterns. Patterns that match $0x1x$ or $0xx1x$ predict taken while all others predict not-taken.

or a 0, the state you end up in will predict 0. This property is is maintained into even the most complicated predictor.

Figure III.7 is a slightly more complex state machine, this time generated for one of the branches in `gs`. This state machine captures two different patterns, $0x1x$ and $0xx1x$ where x is a "don't care". If, as above, you traverse any set of edges that match either of these patterns you will end up at a state predicting a 1. If the edges you traverse do not match these patterns you will end up in a state predicting 0.

For this branch there are four global history patterns that are seen the majority of the time, 001001010 which is biased taken, 010011010 which is biased not-taken, 010101010 which is biased taken, and 110010010 which is biased taken. There are other patterns which contribute but these represent over 90% of the patterns seen by this state machine. If you trace through these patterns, or just the last five digits of them, starting from any state you will end up in a state that predicts correctly. Because of this fact, it does not matter that we are updating the branch predictors for branches that may have never been trained on. No matter what state the machine has gotten itself into, as long as the final global sequence of branches leading up to the branch is captured by one of the patterns the state machine will make the correct prediction.

These results show that our automated steps can be used to accurately identify branches that are highly correlated, and the nature of the correlation. This information could then be used to potentially guide additional compiler and hardware optimizations.

Note that for a given custom FSM predictor, only a Markov table with histories of length H leading up to the branch to be predicted are used to generate the FSM predictor. Therefore, using our update on every branch policy, we could be updating a custom FSM predictor with branches that were never seen in the

histories used to generate the custom FSM predictor. This is fine because the FSM predictor we generated guarantees that the predictor will be in the desired prediction state when we fetch the candidate branch. The FSM predictor will be updated with state transitions for the H branches that come right before the branch that is assigned to the FSM predictor. Therefore, no matter what state the FSM predictor was in before performing the H branch updates, after the updates it will be in the desired prediction state.

III.H Summary

Predictive finite state machines are a fundamental building block in computer architecture and are used to control and optimize all types of prediction and speculation. In this chapter we present an automated approach to generating FSM predictors using profile information. This approach can be used to develop FSM predictors that perform well over a suite of applications, tailored to a specific application, or even a specific instruction.

The algorithm we present uses profiling to build a Markov model of the correlated behavior present in a target application. From this model we define a set of patterns to capture. These patterns are compressed into a usable form and then translated into a regular expression. From the regular expression we use subset construction to build a finite state machine. This state machine is then optimized for our uses using Hopcroft's Partitioning Algorithm and Start State Reduction.

We first examine using our automated approach to automatically generate a confidence estimation FSM to use for value prediction in a general purpose processor. The results show that we are able to automatically generate confidence estimation FSMs the can achieve higher accuracy for greater coverage than previously proposed saturating up/down counters. More importantly, our framework

automatically finds a Pareto optimal curve of custom FSMs solutions for trading off accuracy versus coverage.

We also present a customized branch prediction architecture that makes use of these custom built finite state machine predictors. The FSM predictors are generated for branches that are not easily captured by local two-bit counters. These custom state machines take up little area, and can efficiently and accurately capture the global correlation behavior of the target application. The global correlation is shown to be captured across input sets and results are presented for a variety of predictor sizes. For all of the programs examined, our custom predictors achieve a misprediction rate less than a general purpose predictor of twice it's size or more. For two of the programs, the custom branch misprediction rates are lower than general purpose predictors of five times their size.

The next question that we must ask from the results is, how will the affect the system as a whole? This question is not as easily answered as it might first seem. For example, could the area that is being used by the branch predictor be better utilized somewhere else in the system, or it it just wasted space? To answer these and other questions, we must dive into design space tradeoffs, and in Chapter IV we show how this can be done.

IV

Inter-Component Tradeoff

IV.A Introduction

As was mentioned in previos chapters, the continuing expansion of the embedded market has created a significant demand for low-cost high-performance computing solutions that can be realized with a minimum of engineering effort. While die area and power consumption is a concern when developing a desktop-processor, in the embedded design world these factors have a significant impact on the cost-point and ultimate functionality of a product. Application specific processors allow a designer to meet these goals by specializing the features of the processor to a particular embedded application. Architectural choices such as cache sizes, issue width, and functional unit mix can be traded off in the cost-energy-performance space on a per-application basis. Customization may also include extensions in the form of custom accelerators, for instance the inclusion of encryption instructions [147]. Building embedded processors tuned to a specific application enables a design that minimizes area and power while still meeting performance requirements. At the same time, because the design is based on a carefully tested template, the design time is reduced and reliability is increased significantly over a pure ASIC based approach.

In the desktop processing realm one might take an "everything and the kitchen sink" approach, focusing only on bottom-line performance. But in the fiercely competitive embedded domain system cost (related to area) is the dominating factor. Smaller processors means more integration resulting in less packaging cost, and less area means more chips per die and less defects per chip.

In this space, it is vital to determine how to effectively divide your area between the instruction cache, data cache, branch predictor, additional functional units, and other components. This is because cost is the single most important attribute for embedded applications. When designing a custom processor the designer needs to find the lowest cost implementation that meets the target performance constraints, and this problem by it's very nature requires that all optimizations be viewed in concert with all other parts of the processor. Faced with this extremely important goal of minimizing area and power, and a combinatorial number of design options, how should an embedded processor engineer proceed?

The goal of this chapter is to introduce a technique that assists this processor engineer (or automated compiler) in the navigation of the vast application specific processor design space. It is not practical to evaluate every possible alternative through detailed simulation. But by building a set of estimators and solvers a very good characterization of the design space can be built and explored. We present the *Sherpa* framework as a guide for this application specific processor exploration for in-order processors. Sherpa efficiently explores the in-order custom-processor design space because, while the space is non-linear and full of local minima, it is, to first order, decomposable. Custom processors are currently built from in-order processing cores [49, 130, 7, 2], and the design features of these, as we will show in Section IV.E, can be broken apart and modeled as independent optimization problems.

Sherpa is a framework for exploring these sub-problems and recombining

them to find ideal parameters for the usual processor features such as caches, and branch predictors. It is fully general, however, and can also explore whether or not to use custom instruction accelerators, or variations on functional unit type, for instance whether to use a slow or fast multiplier, or no hardware multiplier at all. More significantly, it optimizes these selections against all of the other design possibilities.

The Sherpa framework begins with an application and carves up the entire processor design space into a set of loosely independent regions. Each region represents a logical architectural component, such as the data-cache hierarchy, branch predictor, custom accelerator, etc. These regions are then treated as separate sub-problems and are explored using data-driven analytical models or high-level simulation. The result of these independent explorations is a set of characterizing functions that are combined to form a model of the entire design space suitable for constrained minimization. This global model is then optimized through standard integer-linear programming techniques to arrive at desirable parameter selections for each architectural component.

While integer-linear programming has been widely used in the ASIC design community to automate dataflow graph partitioning, to our knowledge we are the first to apply this powerful optimization technique to total-processor design space exploration. The significant research advance we contribute in this chapter is the characterization of the processor design space into component models, the constraint formalism to combine them, and the validation of these techniques in this environment. The research in this chapter bridges the gap between optimizing small ASIC dataflow graphs, and complete processor designs; it lays the foundation for optimization of more complex architectures. Additional contributions of this chapter are:

- An automated design space surveyor for embedded in-order custom-processors.

- A methodology to approximate various design options for processor features with piece-wise linear functions relating area to performance.
- An integer-linear programming formulation for combining these sub-problems using these piece-wise linear functions. This provides an approach for rapidly finding ideal parameters for the processor architecture.
- An in-depth analysis of trading off performance and area against each other. This leads to the result that when holistically designing a processor, instead of focusing only on a single component, area and performance optimizations can be used to the same effect.
- A demonstration of using Sherpa to decide whether to use a custom instruction accelerator. For our purposes we illustrate this with the design choice of whether or not to use a hardware multiplier. However, the technique generalizes to any custom instruction one is trying to decide whether or not to include in a design, and several different alternatives can be evaluated at the same time.
- A demonstration of how to use Sherpa to quickly evaluate the potential benefit of an architectural optimization. As a case study we examine the use of application specific finite state machine predictors for branch prediction III.

The rest of the chapter is organized as follows. In Section IV.B we give an overview of the design problem we are addressing. A detailed discussion of our constructed constraint system can be found in Section IV.C. The constraint system is built from the sub-problems that are then discussed in Section IV.D. Section IV.E presents the use of the Sherpa framework along with results. Section IV.F describes related work, and Section IV.G summarizes our findings.

IV.B Design Navigation

There are currently several commercial ventures that offer customizable RISC processors [49, 130, 7] such as the XTensa processor from Tensilica [49]. Tensilica provides support for XTensa in terms of compilation, synthesis, and verification [100]. Embedded designers use these tools by providing a high level specification for a processor core. Produced from them is a component ready for integration onto a systems-on-a-chip (SOC) product. While this is a significant advancement for customized processors, the designer is still for the most part unassisted in determining the optimal processor specification for their embedded application. This is where our research is focused with the Sherpa framework.

In this section we discuss the typical design flow used when generating a customized processor. We then illustrate where Sherpa integrates into this workflow to provide rapid feedback and design optimization.

IV.B.1 Process Flow for Creating an Application Specific Processor

We will discuss the workflow used to create an application specific processor, and then how Sherpa enhances it. For concreteness, our discussion will focus on the process used for systems similar to the XTensa processor from Tensilica, however, Sherpa can also be used by more automated tool-chains such as those proposed for the PICO [2, 3] custom processor (discussed further in Section IV.F). The current design process, as follows, involves several iterated steps to find an optimized processor configuration:

1. The embedded application is provided in a high-level source language, such as C or Java. The design team determines performance targets, e.g. page renders per minute, or frames per second.

2. The application is compiled to a binary, or in some cases several different

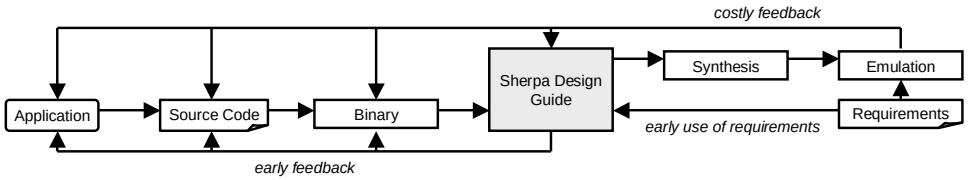

Figure IV.1: High level view of Sherpa design flow

The Application Specific Processor design flow with the Sherpa design guide. Sherpa minimizes the number of times the design must be iterated on. This allows the designer to spend their time optimizing the algorithms used or the way they are coded instead of having to iterate over the design space.

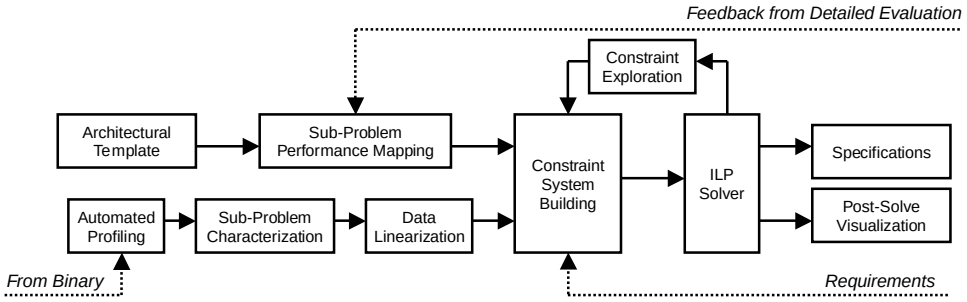

Figure IV.2: Overview of the Sherpa design guide implementation.

The system starts with the input binary and the set of requirements. The binary is profiled and the different sub-problems are analyzed. In the data linearization stage a set of piecewise linear functions are built for each sub-problem. These, along with an automatically constructed performance model, are converted into a constraint system which is solved using integer-linear programming. A set of different constraints can then be iterated over. The output of the solver can then be visualized for human examination and converted into a set of processor specifications for use by synthesis.

binaries depending upon the architectural options (available registers, instructions, etc).

3. Profiles of the application on the target platform are generated from simulation. These profiles focus the design teams effort on application and architectural bottlenecks. Architectural changes involve a lengthy, manual enumeration of the potential options such as cache and branch table sizes in an attempt to find an ideal configuration.

4. The design team then creates a new binary compiled to the potential architectural choices from the previous step. The processor specification is passed through the backend toolset, which generates an RTL description of the custom processor.

5. The binary is then re-simulated on the new custom processor and the entire process is iterated to further optimize the configuration.

The above process, when done manually, can be tedious and time consuming. It leaves to the designer the choice of what architectural options to modify by guessing the possible outcome prior to executing a lengthy tool-chain and simulation step. Unfortunately, the entire space of design options for a configurable core is vast in both performance and cost. Furthermore, this search space is filled with local optimas requiring something more clever than a straightforward greedy approach.

IV.B.2 Using Sherpa to Guide Design

While engineers can perform the above process by hand, they need to first spend significant time becoming familiar with the target application, and then must quantify the behavior of the program through exhaustive experimentation and analysis. We instead provide Sherpa as an automated system to exam-

ine program behavior and suggest near-optimal (which are often optimal) design decisions, allowing the designer to make informed decisions early and experiment with different optimizations. In addition, Sherpa also allows the designer to more easily deal with processor design tradeoffs resulting from late changes to the application.

Figure IV.1 shows the application specific design flow when using the Sherpa design guide. The goal of the Sherpa design navigator is to allow its user to make design tradeoffs early in the exploration phase and quickly narrow in on the bottlenecks of a system. Sherpa analyzes the application code from the existing custom processor design workflow and quickly searches the entire architectural design space for global optima. The embedded product design team then uses this information to narrow in on the final custom processor architecture.

Figure IV.2 shows the internal process of the Sherpa Design Guide (the gray box) shown in Figure IV.1. Sherpa takes the input application, and profiles it to generate a set of representative traces. Next, Sherpa decomposes the vast custom processor design space into a set of independent sub-problems (e.g., data cache, branch predictor, etc). The application trace is then used to initiate these models. Finally, Sherpa combines the models using an integer-linear program (ILP) solver to locate a globally ideal custom processor architecture configuration. This configuration, along with the design tradeoffs Sherpa made, is then passed back to the designer for examination. The designer can then focus attention on the significant architectural components and application sections.

IV.C Implementation

We now describe the implementation of a Design Navigator, Sherpa, beginning with a description of our overall approach and a description of the target architecture, and finishing with a detailed evaluation of the algorithms

used. We use the design navigator to find ideal sizings for instruction and data cache, branch prediction logic, and multiplier settings.

IV.C.1 Overall Approach

Our initial implementation is based on a constraint system that expresses the entire range of architectural options. The constraint system is structured so that an integer-linear solver will find an optimal solution from the possible design options while maintaining hard constraints, such as performance needs, but at the same time optimizing for design goals (such as area).

Our approach initially treats each customizable component separately. For each component, an area-performance model is developed by simulating a variety of component configurations with a representative instruction trace. This collection of area-performance models is then combined with integer-linear programming and solved to yield an ideal overall processor configuration. Before discussing the details of this process we wish to describe the customizable processor we are targeting in more detail.

This chapter optimizes a customizable architecture similar to the in-order XTensa Processor [49]. The reason that we have chosen to use this processor model is that the work done by Tensilica is mature enough that they have a workable design path from processor specification all the way down to silicon. They are able to do this because they work with the CAD tool designs further down the design chain to insure that the parameterized processors they build will be able to be synthesized into an efficient form. The Sherpa guide can be used as an automated front end to the XTensa design process to guide the selection of ideal parameters of the different components.

The parameterized processor we model is a single issue in-order 32-bit RISC core with parameterizable data cache, instruction cache, branch predic-

tion, an optional fast integer multiplier and core. There is a basic core size which contains the primary functionality of the chip, such as the execution box, instruction handling, bus controller, and memory management unit. We assume that the area of this core is unchanging and leave its optimization to future work. The parameterizable parts of the processor, are more auxiliary components used to reduce delay.

IV.C.2 Simple Linear Constraint System

Suppose we wish to customize only a single architecture parameter; for example the instruction cache size. Then we could simply model the performance of the processor with various cache sizes and pick the smallest one that still met our performance requirement. However, suppose instead that we were asked to choose the ideal instruction and data cache sizes that together made a system that met the performance requirement. In our case we define "ideal" as the configuration with the lowest area. A naive approach is to enumerate through and simulate all possible instruction and data cache size combinations and choose the optimal one. This exhaustive search will find the optimal configuration but is neither fast nor scalable.

Our approach is to divide this design space into two sub-problems (instruction cache area versus delay and data cache area versus delay), and to explore each sub-problem independently. The model presented in this chapter approximates the performance penalty from a sub-problem as a fixed length processor stall, where this stall affects the entire pipeline. For example, a cache miss can be represented as a fixed 50 cycle processor stall, and a branch misprediction as a 6 cycle stall. Even though all of the sub-problems we examine are not completely independent (e.g., branch prediction and instruction cache delays) the probability of these delays overlapping for an in-order processor is small for the

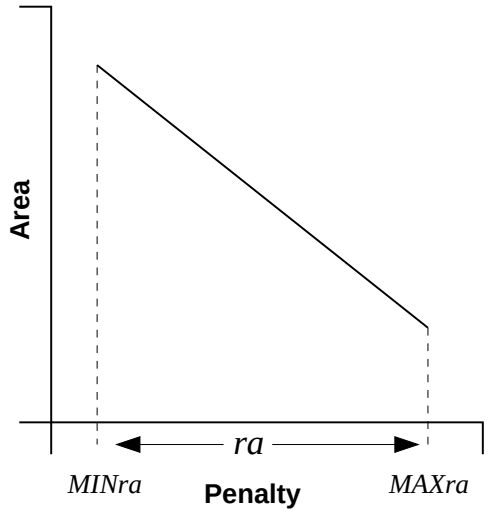

Figure IV.3: Simple linear function creation for tradeoffs

This graph shows two design points MINra and MAXra, and the creation of a simple linear function between these two points relating performance to area.

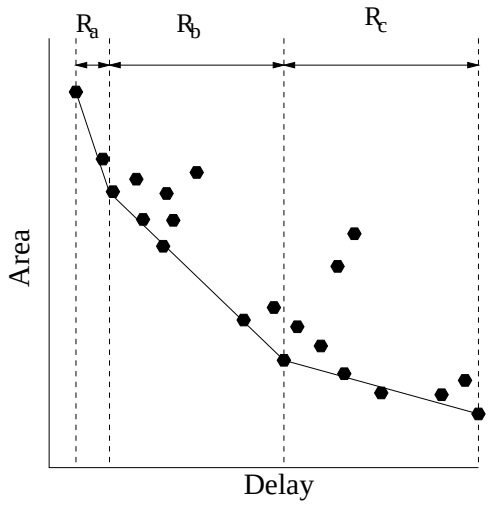

Figure IV.4: Peicewise Linear Model for tradeoffs

A graphical representation of the Piecewise Linear Model which is used to quantify the area-performance tradeoffs for each sub-problem. The Pareto optimal design points are estimated by intersecting line segments for which the slope and intercept are known.

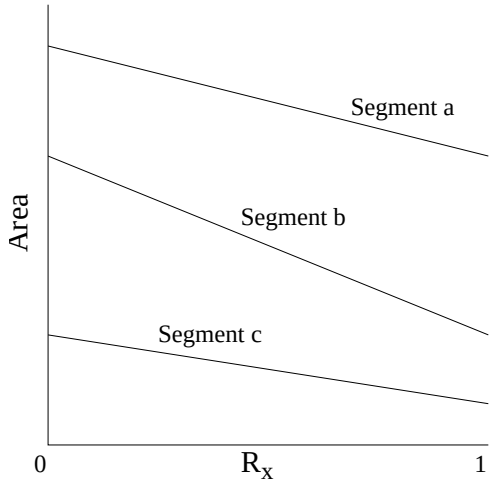

Figure IV.5: Translated Peicewise Linear Model

Translated piece-wise linear model. Note that for each line segment we see the same height on the y-axis but now the lines are stretched to be over the range of $[0, 1]$.

programs we examined. We measured the amount of dependence between the different subproblems using detailed cycle accurate simulation and found it to be negligible. This issue is discussed further in Section IV.E.

By allowing ourselves the simplifying assumption that these are independent sub-problems we can express the overall performance of our example two parameter processor by Equation IV.1:

$$T_{Total} = T_{Base} + T_{DataStall} + T_{InstrStall} \tag{IV.1}$$

where T_{Total} is the total execution time of the program, T_{Base} is the time to execute the program assuming a perfect memory, and the stalls for both the data cache and instruction cache are added for the corresponding configurations being considered.

In order to compare the different configurations, we need to create a function that relates area to performance. Figure IV.3 shows the performance

versus area for two cache configurations chosen for the data cache. For purposes of illustration, we assume this is a simple linear function; the next section introduces more realistic models. Given the two points in Figure IV.3, the contribution to processor stall from the cache can be expressed in terms of its structure via the equation $T_{Stall} = k \times Area$, where k is a constant. Performing this operation on both sub-problems creates two independent linear functions that relate performance to area that must be combined to find the ideal balance between the two. Rewriting Equation IV.1 we get:

$$T_{Total} = T_{Base} + (k_{Data} \times Area_{Data}) + (k_{Instr} \times Area_{Instr}) \tag{IV.2}$$

Using this equation we set a performance constraint $T_{Total} \leq T_{max}$, and solve for the two unknowns: $Area_{Data}$ and $Area_{Instr}$. The goal is to maintain this performance constraint while minimizing the area, which is expressed as:

$$Minimize : Area_{Total} = Area_{Data} + Area_{Instr} \tag{IV.3}$$

The result of this formulation is a system of linear equations and an optimization criteria. This can be effectively solved using linear programming tools. Commercial solvers from CPLEX are available, but we found the freely available *lpsolve* tool to be sufficient [13].

IV.C.3 Data Linearization

The previous example of finding the ideal balance between a set of instruction and data cache configurations relied upon the assumption that there is a strict linear equation to represent cache area to performance. Clearly this

assumption is simplistic and we must support a more complex area-performance model.

For this we turn to a piece-wise linear approximation of the actual function. To form this approximation we begin by identifying Pareto-optimal design points. A point p is said to be Pareto-optimal (on the axis in Figure IV.4) if there is no point p' such that p' requires less area and incurs less stall time. Intuitively, if a point is not Pareto-optimal then there is another design point that will achieve as good or better performance with less area. We use these Pareto-optimal points to help form a piece-wise linear approximation of the area-performance function.

Starting from the Pareto-optimal design points we greedily construct straight line segments. A point is added to a line segment if the correlation between that line segment and all the points that approximate it does not drop below a threshold. The correlation coefficient, r, provides the extent to which the bounding Pareto-optimal points being considered lie on a straight line. For the results we present in this chapter we insure that r is always greater than 0.98. When the correlation coefficient drops below this threshold, a new line segment is started. To insure that the line segments always intersect, the last data point included in a given segment is also included in the next segment.

The result of piecewise linearization can be seen in Figure IV.4. The points that are Pareto-optimal are connected by line segments that capture the overall area-performance trend. In this example three line segments R_a, R_b and R_c make up the entire piece-wise linear approximation. Our goal now is going to be to modify the constraint system to use these far more precise sub-problem models, which we describe next.

IV.C.4 Final Constraint System

A mixed integer-linear program is similar to a linear program except that some variables are optionally constrained to be whole numbers. Solving a mixed integer-linear program as opposed to a linear program is a straightforward process. We refer the interested reader to [75] for details on the internals of this algorithm.

To create our mixed integer-linear constraint system, the first step is to include each piece-wise linear function for each sub-problem into the constraint system. This is performed by breaking up each line-segment of the piece-wise linear approximations into separate components. These line segments are then translated to have a uniform length and starting point along the horizontal axis. Figure IV.5 depicts the same piece-wise linear function from Figure IV.4 after translation is performed. For each piece, we generate the functions for delay and area where the function for area and delay has been normalized between zero and one.

The linear function for area is calculated as
$Area_x = BaseArea_x + SlopeArea_x * R_x$, and the function for delay is calculated as $Time_x = BaseTime_x + SlopeTime_x * R_x$. In both of these equations, R_x is a design point with a value between zero and one. $BaseArea_x$ and $BaseTime_x$ are the start area and time for each linear piece in Figure IV.4.

Once distilled into separate line-segments we construct a mixed integer-linear program where only a single line-segment (from the piece-wise linear function) is selected for each sub-problem. The single line-segment is chosen using integer programming. To illustrate, Figure IV.6 depicts a new constraint system with these constraints. This constraint system works by choosing a tuple (iR_x, R_x) with Equation IV.4. iR_x is an integer value of either zero or one. R_x is a linear value between zero and one. This tuple selects which line segment to use

$$(iR_x, R_x) = ChooseOne\left(\{(iR_1, R_1), (iR_2, R_2), ..., (iR_n, R_n)\}\right) \tag{IV.4}$$

$$iR_1 + iR_2 + ... + iR_n = 1 \tag{IV.5}$$

$$iR_1 \geq 0, \quad iR_2 \geq 0, \quad ... \quad , \quad iR_n \geq 0 \tag{IV.6}$$

$$0 \leq \quad R_1 \quad \leq iR_1 \tag{IV.7}$$

$$0 \leq \quad R_2 \quad \leq iR_2 \tag{IV.8}$$

$$...$$

$$0 \leq \quad R_n \quad \leq iR_n \tag{IV.9}$$

$$\begin{aligned} 0 \leq \quad & Time_{required} \quad - \quad Time_{perfect} \\ - \quad & (BaseTime_1 * iR_1 + SlopeTime_1 * R_1) \\ - \quad & (BaseTime_2 * iR_2 + SlopeTime_2 * R_2) \\ - \quad & ... \\ - \quad & (BaseTime_n * iR_n + SlopeTime_n * R_n) \end{aligned} \tag{IV.10}$$

$$\begin{aligned} Minimize: \quad & (BaseArea_1 * iR_1 + SlopeArea_1 * R_1) \\ + \quad & (BaseArea_2 * iR_2 + SlopeArea_2 * R_2) \\ + \quad & ... \\ + \quad & (BaseArea_n * iR_n + SlopeArea_n * R_n) \end{aligned} \tag{IV.11}$$

Figure IV.6: Constraint system for a piece-wise linear sub-subproblem model.

with the integer variable iR_x, and the design point within that line-segment to examine using the variable R_x. Equation IV.4 cannot be implemented directly, and we describe next how to construct it using integer and linear constraints.

Equations IV.5 and IV.6 perform a line segment selection for the sub-problem by ensuring that only one line segment is selected, since each iR_x can only have a value of 0 or 1. Equation IV.5 insures that only one iR_x will be equal to one, and all the rest will be equal to zero. Since we normalized the linear functions to represent the area and delay using a variable between 0 and 1, we are able to reuse the variable iR_x to subtly constrain the selection of the design point R_x. This is shown in Equations IV.7 - IV.9, where the design points along the chosen line segment are examined.

Once a given tuple is selected, the translated $Base_{area,time}$ and $Slope_{area,time}$ values are used to determine performance impact and area cost of the choice. The modified performance bound is illustrated in Equation IV.10 and the new minimization goal is shown in Equation IV.11.

Figure IV.6 shows only the constraint system for one sub-problem. To generate the overall constraint system to examine the trade-offs between multiple sub-problems, each sub-problem has its own constraints to choose an iR_x and R_x particular to that sub-problem using equations similar to those shown in Equations IV.5 - IV.9. The time delay for each sub-problem is subtracted from Equation IV.10, and the area delay for each sub-problem is added into Equation IV.11. Then the overall system is run through the constraint solver to minimize the area, while meeting the specified $Time_{required}$.

IV.D Sub-problem Characterization

In the previous sections we presented our general methodology of partitioning the complete application specific processor design space into a set of

loosely independent sub-problems. In this section we discuss the sub-problems we explored and how we modeled them. The sub-problems of the design space that we chose to explore in depth are the instruction and data caches, the branch predictor configuration, and whether or not to include a fast hardware multiplier. The general methodology for constructing each of these models is to explore a sampling of points through direct simulation and then construct piece-wise linear models from the Pareto optimal design points.

For a given subproblem (e.g., branch predictor) we can trade off multiple designs (gshare, bimodal, 2-level) by plotting them all on the same Pareto graph. The piece-wise linear model built from that graph represents the ideal design to use for a given performance/area point. The same approach would be taken if we were to have the option of different types of accelerators for a given instruction or sets of instructions.

In this section we describe how we estimate the area for each design point considered as well as how to calculate the performance penalty for that configuration for the trace being examined.

IV.D.1 Data and Instruction Cache

We start with an examination of the caches in our architecture. As will be seen later in Section IV.E.2, the caches are the dominant configurable area of an application specific processor. It is thereforimperative that the models correctly capture the area-performance tradeoffs, since they are the first order determinant of the overall accuracy of our technique.

Cache design is a well-studied problem in computer architecture, and we wish to build upon past work in this area where appropriate. Reinman and Jouppi [106] present a very detailed cache design space walker based on the work presented in [144]. CACTI, when given a set of cache parameters, such as

associativity and cache size, will find the delay optimal cache array partitioning scheme (i.e. the fastest physical device layout for the given cache parameters). Mulder [91] presents a validated area model for caches and register files. For our research we use the CACTI design tool to aggressively optimize the partition of the cache and then use that configuration to calculate the area using Mulder's area model.

One concern with using CACTI to optimize for area-optimal caches is that its internal optimization goal is programmed to find delay-optimal designs. To verify this would not skew our results we modified the CACTI tool to produce Figure IV.7, which depicts the delay versus area tradeoff of over 9,000 cache partitions that CACTI examined for a single cache parameter set. Out of all of these configurations examined by CACTI, only three Pareto optimal ones were generated. This shows that the most performance-efficient partition is extremely close to the most area-efficient. Figure IV.1 depicts this area variation for a range of different cache configurations. We found at most a 4% difference in area from the most area-efficient to the most performance-efficient design. Therefore, we chose to use the delay optimized cache partitioning from CACTI for our results. Fortunately the cache partitionings are not program dependent hence they can be calculated once and stored in a database for later use by any optimization process.

To capture the program-dependent effect of the different cache options, we need to estimate the number of cache hits and misses the target applications will have for each cache configuration. This can be done through simulation or analytical modeling [4]. Efficient cache simulators have been proposed to simulate many different configurations at once [136]. For our work, we found that we could quickly simulate all of the reasonable power of two cache sizes (256 bytes to 64K), associativity (direct mapped to 4-way and fully associative) and line size (8 bytes

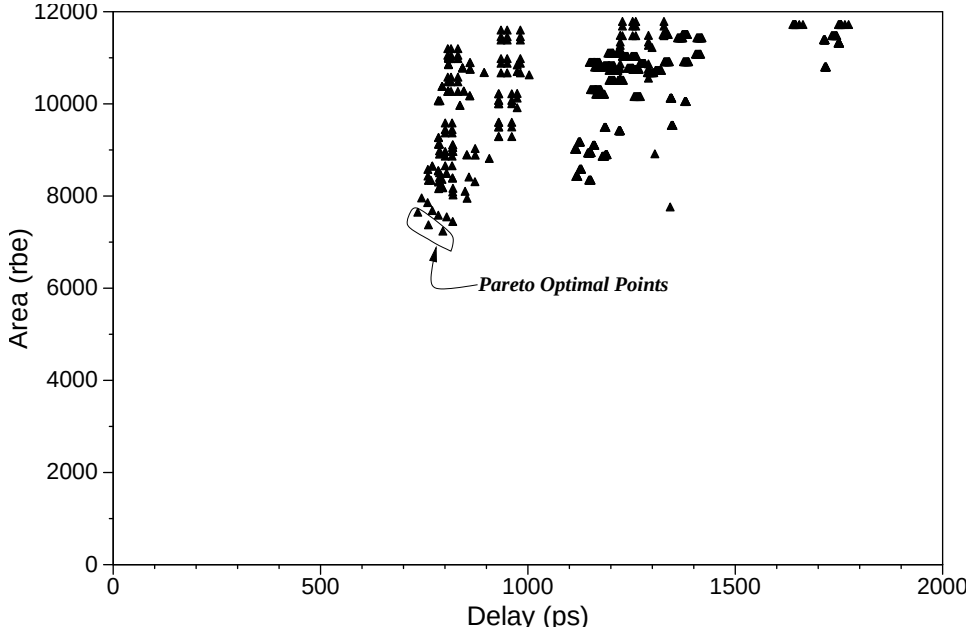

Figure IV.7: Area-delay tradeoffs for a single cache configuration.

This graph was generated by modifying the CACTI tool to output info on all configurations considered. For the example cache parameter set shown, over 9000 different configurations were considered but only three Pareto optimal configurations were found.

Size/Assoc	1	2	4	8
1K	±2.68%	±3.71%	±0.23%	±0.00%
4K	±3.30%	±0.49%	±0.25%	±0.12%
16K	±3.56%	±1.32%	±0.79%	±0.13%
64K	±2.14%	±0.65%	±0.66%	±0.40%

Table IV.1: Area versus Delay optimization for cache

Percent area variation between the most area efficient design examined by the CACTI design tool, and most delay efficient design. Results are shown for a 1K to 64K cache from direct mapped to 8-way associative. In CACTI, the cache partitioning scheme with the minimum delay is returned. A variation of less than 4% is seen in area when optimizing a cache configuration for area versus performance.

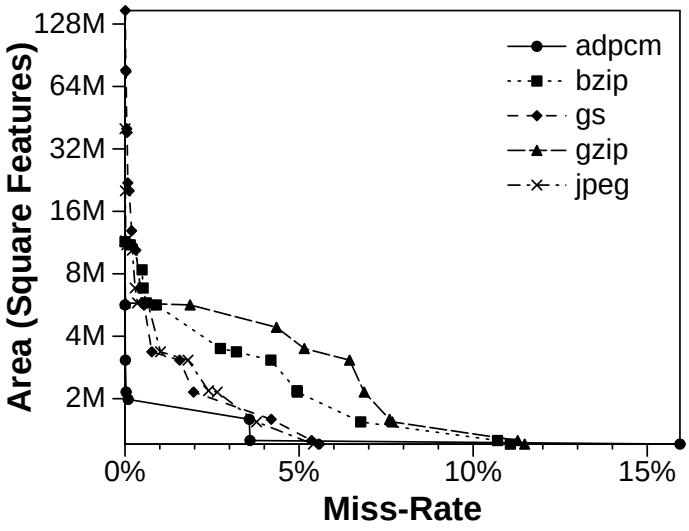

Figure IV.8: Instruction cache tradeoffs

Instruction cache miss rate versus area tradeoff for different cache sizes, associativities, and line sizes. Only the Pareto Optimal points are shown.

to 64 bytes) options. We then find the area of each of these configurations using the previously mentioned area model and CACTI partitioner.

Figure IV.8 shows the area versus miss rate for the instruction cache for the applications we explored, and Figure IV.9 shows the area versus miss rate for the data cache. From these points, we created the piece-wise linear models as described in the prior section.

Since our custom processor configuration is for an in-order core, the cache miss penalty can be easily modeled in our constraint system by assuming that each cache miss results in a fixed penalty of 50 cycles, during which time the processor is stalled. We investigated the impact of assuming hit under miss capability for data caches, but that optimization provided only marginal performance gains. Therefore, in this dissertation we only present results for caches that stall on misses.

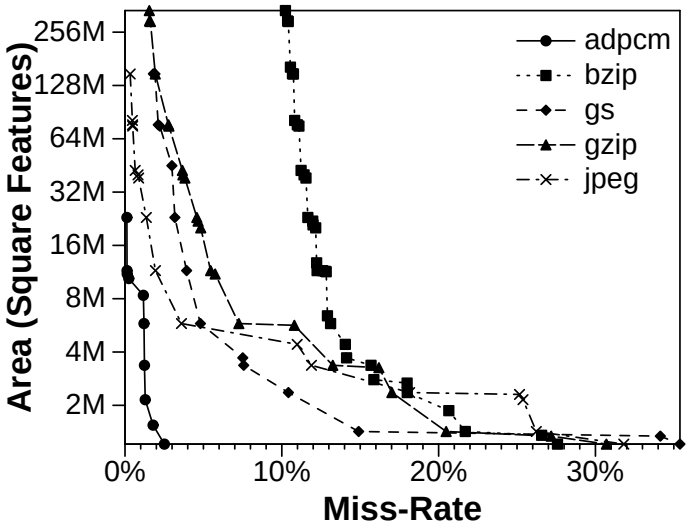

Figure IV.9: Data cache tradeoffs

Data cache miss rate versus area tradeoff for different cache sizes, associativities, and line sizes. Only the Pareto Optimal points are shown.

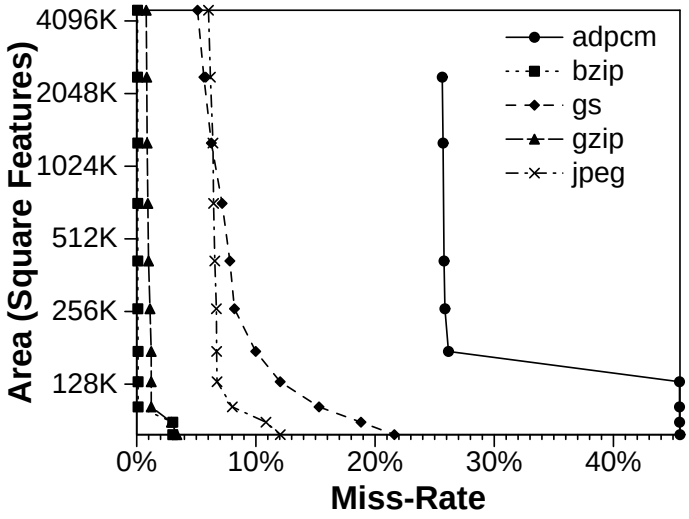

Figure IV.10: Brach prediction tradeoffs

Branch misprediction rate versus area tradeoff. Several branch prediction architectures were examined and the Pareto optimal area/miss rate points are plotted for each program.

IV.D.2 Branch Predictors

To examine the custom design space for branch prediction we examine selecting from the following well know branch predictors: (1) a table of 2-bit counters, (2) a global correlation predictor, and (3) a meta predictor that uses both local and global correlation information.

We examine the branch prediction miss rate for these different predictors for a variety of table sizes. Figure IV.10 shows the area versus miss rate for the Pareto optimal points in the different programs examined. For all of the programs examined the most area efficient branch predictor design was the table of 2-bit predictors similar to the one used in the XScale embedded processor [76]. To achieve better performance, significant resources are needed by the other predictor methods. For example, the Pareto curve for `adpcm` chooses a per-branch 2-bit for the low area points, and a meta predictor for higher area. It shows that the per-branch 2-bit counters give the lowest miss rate for an area less than 128K square features, and a meta predictor gave the best miss rate for an area greater than 128K. In crossing this design boundary the area is increased above 128K, but the misprediction rate is reduced from 45% down to 26%.

The misprediction rate measured here is used to calculate the performance penalty for our in-order processor model. By assuming a 6 cycle stall for each branch misprediction, and knowing how many branches we mispredict, we can estimate the total number of stall cycles incurred by the branch predictor.

IV.D.3 Multiplier Unit

The last configurable option we examined was a processor with and without a large fast hardware multiplier instead of the much smaller and slower iterative multiplier. This is modeled by counting the number of multiply instructions executed in each program. The reason that we have chosen this configurable

option is to show that our design optimization technique can also cleanly handle binary decisions such as whether or not to include a specialized functional unit. While deciding whether or not to use a fast multiplier is a binary decision, in an overall context the decision is complicated by the fact that its usefulness must be traded off against other, non-binary decisions. Thus, the actual decision is extremely difficult to answer by conventional techniques.

We model a hardware multiplier in the following way: if the fast hardware multiplier is used, then a multiply takes 2 cycles, with the area cost of 3 million square features. The area of the multiplier is derived from [111]. If the fast hardware multiplier is not used, the multiply is performed with a software routine, estimated to take 250 cycles to execute [131].

IV.D.4 Core Area

Finally we are left with the area of the actual execution core. Inside the core is the data path of the machine along with all of the control. While the control and data paths could be further optimized, we leave these subproblems for future work. These include the instruction fetch control, the instruction memory management unit, the data memory management unit, the bus controller and the basic functional units. The area for the remaining non-configurable set of functionality we used for this research is derived from [111] and is estimated to be 21 million square features.

IV.E Putting It All Together

Now that we have seen how the constraint system is built and solved in Section IV.C and how the sub-problems are formulated and characterized in Section IV.D, we are ready to examine how they work together to explore the design space.

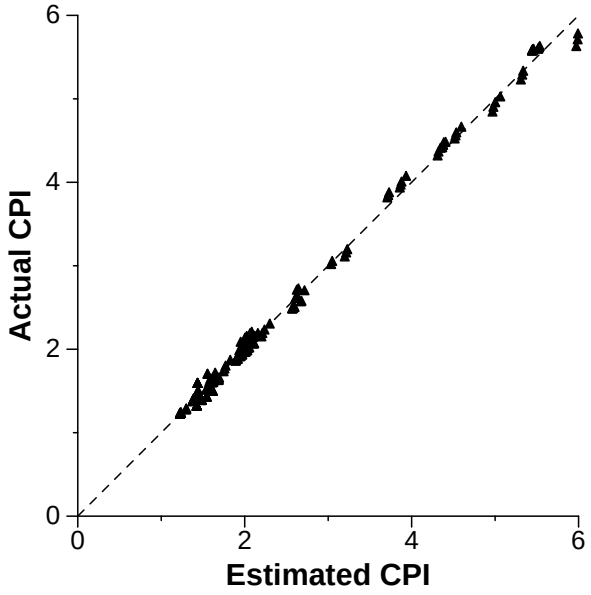

Figure IV.11: Verification of Performance Estimator.

The estimated CPI time as calculated by our formulation is plotted as a function of the performance determined through detailed cycle level simulation.

IV.E.1 Running the Overall Solution

When running the system on a given program we must go through a few steps. The first step is sub-problem characterization as presented in the previous section. During this step we perform simulation of the different design options for each sub-problem. From this we generate the Pareto-optimal points and piece-wise linear approximation to be used for the next stage. The simulation time here varies both on what is being investigated and the program itself. We use fairly brute force methods for examining each space, and this process is the most time consuming of all the operations.

We built our simulation infrastructure on top of ATOM [135] because it provides both high performance and ease of use for a RISC architecture similar to those supported by configurable cores. The design point enumerations for each of the subproblems are calculated at run-time. For the programs we examined

it takes anywhere from a couple of seconds to 10 minutes per simulation. However this process can be easily speed up though the use of smarter simulation algorithms [136], or analytical modeling as described in Section IV.F.3.

In addition to generating the estimated performances of the different sub-problems, we also need to estimate the total area consumed by each design point. This is done once for each given subproblem, and the results can be used for each of the different programs. This step takes less than 1 minute to run.

The final step of the optimization process is the actual construction of the constraint system. This step is very fast since many different design parameters can be examined quickly. To create charts of the pareto-optimal solutions, shown in Figures IV.12 and IV.E.2, we generated almost 50 design solutions in less than 8 seconds. Each one of these design solutions is the ideal combination of the different sub-problems and represents the best design point of many hundreds of millions. Because of this fast turn around many new design choices can be evaluated in real-time. For example, one could answer the question: what would the area impact be of reducing the cache miss penalty by 30%. This could be answered without re-running any simulations.

IV.E.2 Results

We used ATOM [135] to profile the applications, generate traces and simulate cache and branch models. All applications were compiled on a DEC Alpha AXP-21264 architecture using the DEC C compiler under OSF/1 V4 operating system using full optimizations (`-O4`). We chose a set of five benchmarks which could have applications in an embedded environment The application `ijpeg` is from the SPEC95 benchmark suite, and `gzip` and `bzip` are from the SPEC2000 benchmark suite. We also include two programs from the MediaBench application suite – `adpcm` is a speech compression and decompression

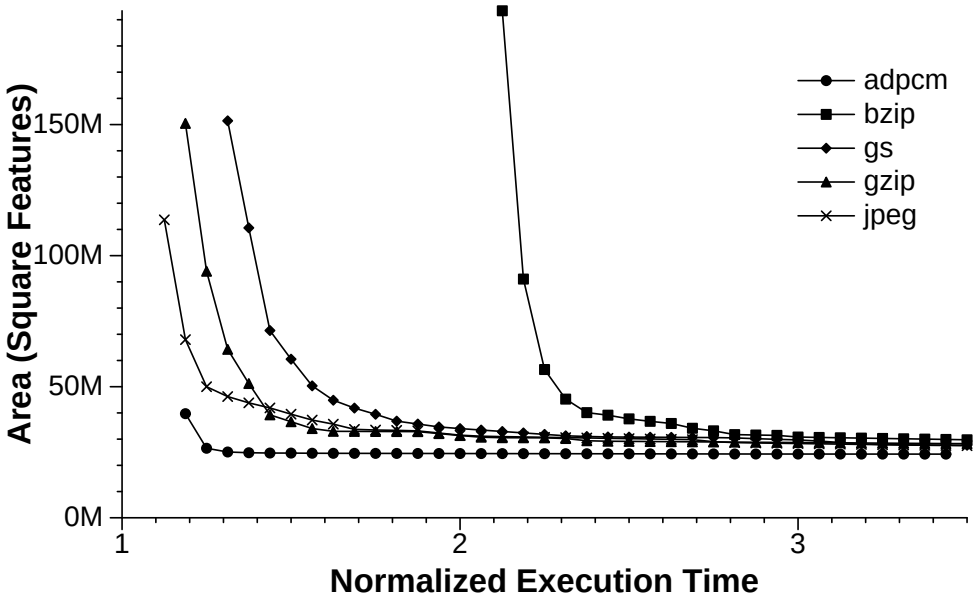

Figure IV.12: Overall area/perfomance tradeoffs

Total Area (in millions of square features) as a function of Normalized Execution Time for the different programs. The areas shown represent the minimum total size of the optimized core that can achieve the performance shown. The execution time is normalized to the performance of the base executable executed with no stalls of any kind.

program, and `gs` is an implementation of a postscript interpreter.

The first step in testing our system is to verify that the performance we estimate using our combination of piecewise linear subproblems accurately matches with a detailed pipeline simulation of the hardware. We compare the estimated CPI gathered for several design points chosen at random, with the CPI of a detailed cycle-level simulation using SimpleScalar 3.0a [18]. We assumed in our constraint model and the simulator that all cache and branch structures could be accessed in one cycle. The processor was single issue, and used the same latencies for the different sub-problems as described in section IV.D.

Figure IV.11 shows the results of this verification procedure. In this graph we have plotted the estimated performance of the processor from our con-

Fixed	Vary	Avg StdDev	Max StdDev
D-cache	I-cache	0.000000%	0.000000%
D-cache	Branch	0.000000%	0.000000%
I-cache	D-cache	0.004775%	0.046030%
I-cache	Branch	0.001985%	0.038971%
Branch	D-cache	0.007916%	0.080000%
Branch	I-cache	0.006044%	0.095263%

Table IV.2: Quantifying the independence of subproblems.
The first column in the table, labeled *Fixed*, shows the subproblem under examination, and the second column, labeled *Vary*, is the subproblem to which independence is being evaluated. For example, the first row in the table compares the independence of the data cache miss rate from the instruction cache miss rate. For the first row this is determined by holding the data cache size constant, varying the size of the instruction cache, and analyzing the change in data cache miss rate. This is then repeated for several sizes of data cache. The numbers reported are the average standard deviation and maximum standard deviation in miss rates across all fixed sizes evaluated.

straint system against that found through detailed cycle-level simulation. The results show a strong correlation between the estimates and actual values, with a correlation coefficient of $r = 0.99829$ for over 200 configurations chosen at random for evaluation across the different benchmarks. This shows that our performance estimation is accurate (otherwise the points in Figure IV.11 would show trends that do not follow the diagonal line drawn on the graph.)

Table IV.2 shows our results to further verify the independence of subproblems. The first column in the table, labeled *Fixed*, shows the subproblem under examination, and the second column, labeled *Vary*, is the subproblem to which independence is being evaluated. For example, the first row in the table compares the independence of the data cache miss rate from the instruction cache miss rate. For the first row this is determined by holding the data cache size constant, varying the size of the instruction cache, and analyzing the change in data cache miss rate. This is then repeated for several sizes of data cache. The

numbers reported are the average standard deviation and maximum standard deviation in miss rates across all fixed sizes evaluated. The results show that the subproblems are independent with less than a 0.1% standard deviation at maximum when holding one component constant and varying the other component. It can therefore be concluded from this graph that the data cache miss rate does not change significantly as the instruction cache miss rate varies for the cache sizes and processor model we are exploring.

We now examine the results of running the entire system on several programs. Figure IV.12 shows the resulting minimized area for each program for several performance design points. The area is shown in square features. The performance constraint is shown normalized to the base executable executed with no stalls for any components. For example, in order for `gs` to be executed in 1.5x the amount of time it would take to execute it with no stalls of any sort, the processor will need an area of 60M (square features).

As expected, relaxing the performance constraint reduces the area needed by the processor. All of the programs exhibit a very strong elbow in their performance area plot. Hence, the major working sets of the program are captured using a small amount of resources, and you will have to add significantly more more area on top in order to improve performance.

A closer look at an individual application shows how the constraint system trades off performance and area between all of the different sub-problems we examined. Figure IV.13 shows this breakdown for the program `ijpeg`. When the performance is tightly constrained, a great deal of area is devoted to the cache and branch predictor, as well as including the special integer multiply functional unit. As we relax the performance constraint, the optimizer balances off the different sub-problems in such a way that the performance criteria is met and the area is minimized.

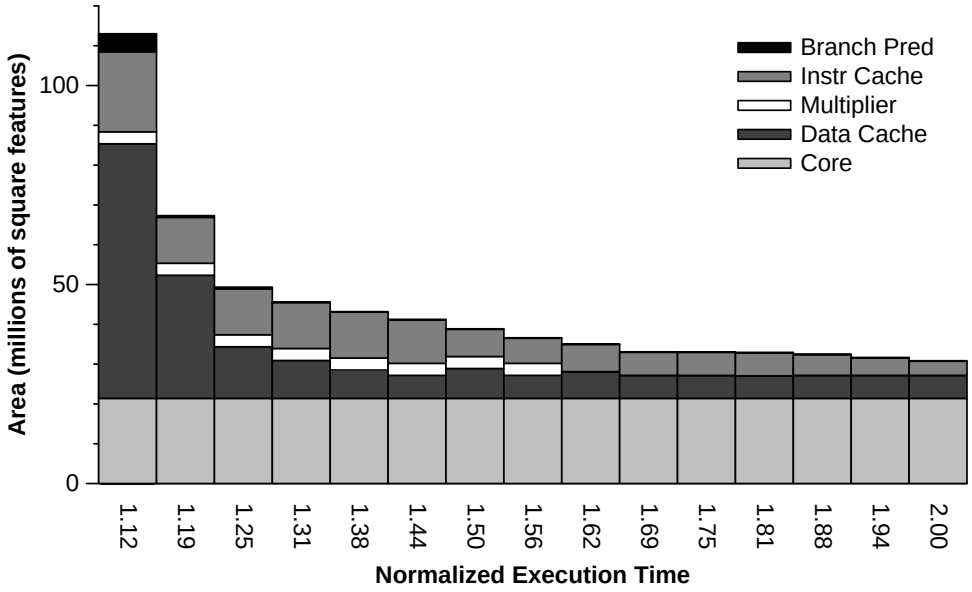

Figure IV.13: Tradeoffs made for the application `jpeg`.

This is a more detailed look at the tradeoffs made across all of the different types of sub-systems. The axis are the same as shown in Figure IV.12 but here only a small portion of the total constraints examined are shown. In addition, the area of each component has been broken out into a separate stack to show the relative importance of each sub-problem for this application.

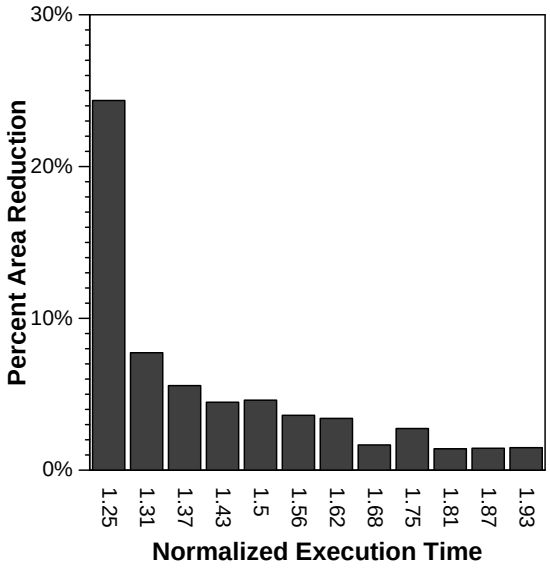

Figure IV.14: Overall area reduction for `gs` with custom FSM

Shown is the Percent reduction in area for `gs` when adding the option to use a custom finite state machine predictor for branch prediction into our Sherpa constraint system.

In relaxing the performance constraint, the first area component to be reduced is the large branch predictor, while both the data and instruction cache are significantly reduced in size. Over the performance range between 1.14 and 1.44 the instruction cache size does not change significantly and the data cache is reduced.

It is interesting to note that analysis of our design tradeoff graphs can be be used to find the working set sizes for the caches. Between 1.44 and 1.50 the performance constraint is relaxed to the point that the instruction cache can be shrunk to the next working set size. However to make up for this increase in instruction miss rate, the data cache actually has to increase in size. This sort of tradeoff can again be seen between 1.56 and 1.62 where the multiplier is cut out and the size of the data cache is again increased. It is these sorts of complex tradeoffs that our system has been designed to optimize.

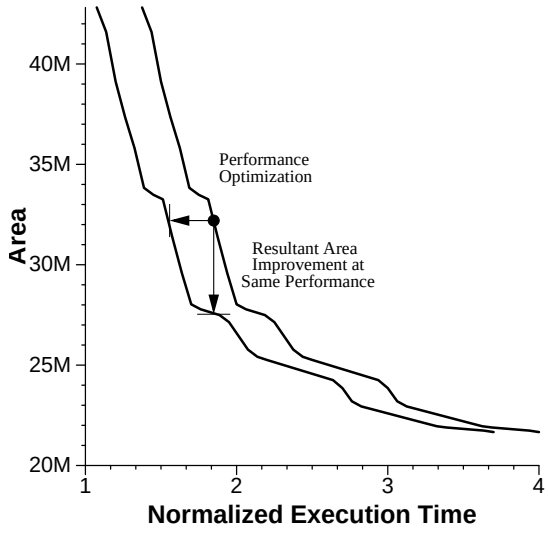

Figure IV.15: Diagram of how performance and area interact

This is an example to show the relationship between performance optimizations and their effect on area. Performance optimizations can reduce execution time and at the same time can significantly reduce the amount of area needed to meet a given performance constraint.

Optimization Tradeoffs

The goal of Sherpa is to use it to not only help guide the design of the architectural components already discussed in this chapter, but to be able to quickly evaluate new architecture optimizations. This allows the designer to see the significance of their ideas to the overall processor design when trying to minimize area.

To show this, we investigated the application specific optimization of automatically creating custom finite state machine (FSM) predictors for individual branches discussed in Chapter III. This optimization augments the branch predictor with custom branch prediction entries. These entries have hard coded to them given branch addresses, and a custom finite state machine tailored to that branch. We include the custom FSM predictor into the Sherpa constraint system as another possible option for the branch predictor. Figure IV.14 shows

the percent area reduction in relationship to performance when evaluating the use of custom finite state machines for branch prediction for `gs`.

The reason for the large reduction in area for the lower execution times comes from the increase in performance provided by the custom FSM branch predictor. Figure IV.15 graphically shows the reason for this by showing the relationship between performance optimizations and area. Optimizations that reduce execution time can at the same time significantly reduce the amount of area needed to meet a given performance constraint.

Performance/Area Versus Total Area Tradeoffs

Another sort of tradeoff analysis that can be provided by our system is computational efficiency analysis. Figure IV.E.2 shows a plot of performance per-unit area versus total area. In this plot we can see that the best performance efficiency for most of the programs we examined lies between 30 and 40 million square features, less than twice the size of the core functionality of the chip. While performance per unit area may not be useful to an embedded designer seeking to meet a given performance constraint, it is a result that would be very useful to chip multiprocessor designers in helping to maximize total processing power of a given chip real-estate. Application targeted chip multiprocessors, such as the Piranaha project [10], which was targeted at transaction processing, seeks to get higher performance by clustering together many simple processors with high computational efficiency onto a single chip. Our constraint system could be used to guide the design of the various subsystems on the chip to maximize the total performance.

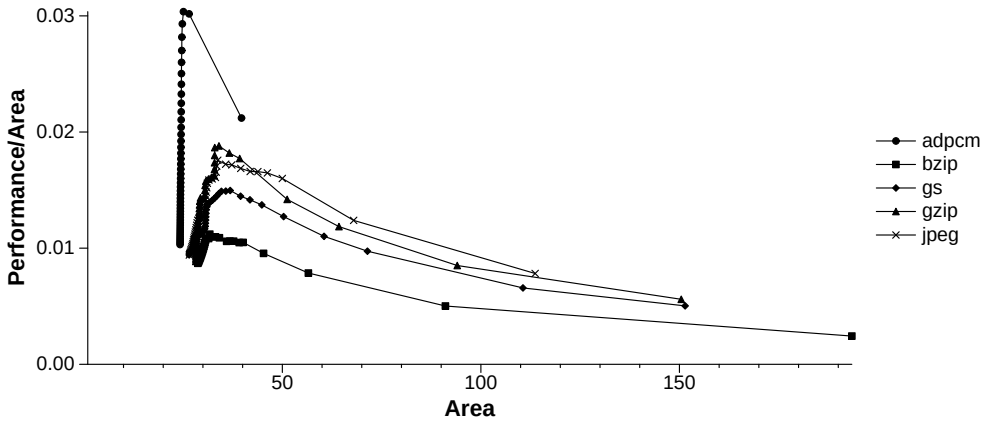

Figure IV.16: Performance versus area tradeoffs as analyzed by Sherpa

Shown is the Performance per unit area of the different benchmarks shown as function of total chip area. While performance per unit area may not be useful to an embedded designer, seeking to meet some performance constraint, it is a result that could be easily used by chip multiprocessor designers to help maximize to total processing power of a given chip real-estate.

IV.F Related Work

There is a great deal of related work in hardware/software co-design, application specific processor generators, and analytic performance estimation, but we have found none that address the problem we are attempting to solve. While a full listing of related work is not feasible in a chapter of this length, we do point out some representative chapters from each area and describe how our research fits into the broader context.

IV.F.1 Configurable Cores

There are currently many configurable core designs, that can be customized or otherwise modified to application specific requirements. The XTensa processor [49] allows the designer to input specifications that they want in a processor, and the XTensa tools generate the needed components ready for integration into a system-on-chip. Another example of a configurable RISC core is

the LX4380 [130] processor core. LX4380 supports adding up to 12 new instructions to the core as needed by a designer. The ARC processor core [7] is similar in design and intent to the LX4380. The tools available for these three configurable cores do not assist the designer in finding good design points for their application. We provide an automated system that will examine program behavior and suggest near-optimal design decisions, allowing the designer to make informed decisions early and experiment with different optimizations.

IV.F.2 Design Exploration for Application Specific Processors

An approach being examined for designing customized processors is to automatically generate trial processors (with a machine description language [54, 88], a GUI [62], or a template as described above) for a specific application, and then examine their performance and feed this evaluation back into the automated processor generator.

The Lx [43, 42, 41] customizable VLIW architecture builds custom hardware for loop nests in applications. They have a clustered VLIW architecture that can be customized to a given application domain with a semi-automated optimization step. The system starts by generating a trial architecture, for which it generates compilers and other tools. Key parts of this application code are then compiled and measured, and are used to generate a new trial architecture, and the process is repeated.

The PICO project [2, 3] uses a fully automated approach to creating a customized processor. Their system starts by designing a set of Pareto optimal memory hierarchies, processors, and custom hardware accelerators. Then, different combinations of these points are tested by assembling the processor and simulating its performance in detail. The best combinations are noted and combinations that are similar to these are evaluated. They report that the system

takes from 10 hours to several days.

The iterative approaches for Lx and PICO yield very good results, but take too long to be used interactively. We propose to use a high level modeling of performance first via Sherpa to allow the designer to very rapidly examine complex design tradeoffs in real time, and then feed this information into the iterative processor generation performed for Lx and PICO.

The Platune System [47, 46] has goals similar to ours, in that it too seeks to reduce the number of processor configurations that must be explored. The Platune System makes use of the fact that while some processor parameters are coupled, there are many others that are independent or are just one-way dependent. They then use this information entered by hand by the user to prune there design space, skipping over points that the user knows will not be Pareto optimal. The major difference between our two techniques is that after Platune prunes the design space, a brute force search of a parameters is still required, which limits the overall number of parameter settings that can be searched. Sherpa, on the other hand, uses linear programming to model many discrete points that are near linear as a line segment, which allows for the points to be searched analytically. We believe that a combination of the two techniques, using Platune to discover and group the dependent parameters and then exploring those parameters using Sherpa, rather than exhaustive simulation, can provide even better results than either technique by itself.

IV.F.3 Performance Estimation

Another common field of study that is closely related to our work is analytical cache performance estimation. Agarwal et. al. [4] present an analytical cache model that can capture the behavior of cache systems based on the analysis of a trace. Both Whalley [143] and Quong [101] present models for capturing the

expected instruction cache miss rate for a given application on different cache sub-systems either at compile time or through statistical analysis of a compressed trace. Jacob et. al. present work that uses a characterization of workload locality built from LRU stack distance to take an analytical approach to building memory hierarchies [64].

Many of these models could be incorporated into Sherpa to create the piece-wise linear functions for the different components we examined. Even so, we found using targeted simulation to quickly walk points in the design space to be efficient when generating the piece-wise linear functions.

IV.G Summary

The approach to exploring this design space presented in this chapter, is to first divide it into separate regions mapped by largely independent parameters. These regions form individual sub-problems that can be efficiently and accurately modeled using data-driven analytical techniques. We formulate the results of these models into a single large constraint-based integer-linear program that can be efficiently solved using conventional mathematical techniques.

We showed that the results from solving of the constraint system lined up very closely with the actual performance numbers obtained through detailed simulation. We examined the tradeoffs that were made between the different components included in optimization and saw how they can play off each other in a complex manner. We further demonstrated how the Sherpa framework can be applied to rapidly evaluate potential optimizations and their impact on *both* the performance and area of the processor.

A goal of this research is to provide a more scientifically sound methodology for evaluating novel architectural techniques in the embedded space. The traditional method of proposing a new technique, and then examining the per-

formance enhancement relative to a single baseline data-point is not very meaningful to overall system design in cost sensitive domains. An architect needs a way of visualizing the way new techniques trade-off against a range of potential design options. The methodology presented in this chapter provides the ability to perform this range analysis. It places architectural changes in a global setting allowing the architect to gain a full picture of its usefulness.

To our knowledge we are the first to apply integer-linear programming to total-processor design space exploration. The significant research advance we contribute in this chapter is the characterization of the processor design space into piece-wise linear models, the constraint formalism to combine them, our formation of piece-wise linear functions, and the validation of these techniques in this environment. The research in this chapter lays the foundation for optimization of more complex architectures.

While the techniques presented in this chapter are very powerful, and allow for the exploration of architectural tradeoffs, they are missing a very important peice of the puzzle, the algorithms themselves. In the next chapter we present techniques for exploring the tradeoffs between the algorithms and architecture in a more specific subset of the embedded design space, network processors.

V

Algorithmic Level Tradeoff

V.A Introduction

In the last chapter, we showed that Inter-component tradeoff is a powerful way of exploring, visualizing, and weighing the relative importance of the separate components of the chip. In this chapter, we take this idea of processor exploration to the next level of the design, the algorithm. To fully realize the potential of any customized processor solution, we must not only consider the interaction between the software and hardware, but the tradeoffs between the hardware and the algorithms themselves. In many application domains there are many choices of algorithm that can be used to accomplish the same tasks, and these algorithms often times have very important, and sometimes surprising, consequences to a hardware implementation.

To examine the tradeoffs at an algorithmic level, we need to dive into an specific application domain so that we can discuss the specifics of the algorithms involved and explore how they tradeoff with the other design factors. We also need am application domain where there is a both a rich set of algorithms and a significant computational demand. Network processors provide a perfect mix of these two attributes for us to explore.

The constant march of fiber optic link speeds has in recent years turned into a full speed sprint that has out paced even the exponential growth predicted for integrated circuits by Moore's law. Building backbone routers capable of handling these speeds currently requires an army of design and verification engineers, large amounts of silicon, and years of design time.

In order to handle a single 10 Gigabit per second fiber link (and many routers have 32 links), each line card must process up to 32 million packets every second[1]. Next generation products will have to handle upwards of 125 million packets per second. Even as bandwidth requirements are increasing, so is the complexity of protocols that need to be supported — from traditional forwarding functions such as IP lookup, to more recent requirements for doing intrusion detection.

Further, a competitive backbone router must not only support the most recent protocols, but must also take advantage of the most recent advances in network algorithm design. This is in stark contrast to more traditional architecture research where it is assumed that the benchmark is an unchanging standard. In the core router setting the *only* requirement is bandwidth, and anything in the design space can change to maximize bandwidth, including the architecture and the algorithms themselves.

In this chapter we describe a programmable system designed to efficiently execute a variety of network algorithms and show that it is a feasible alternative to custom ASICs. To achieve the high throughput required by core routers, we propose a novel memory design for our network processor and make use of a form of non-uniform wide word parallelism. In addition to this, we will generalize from our experiences to propose a novel method for the co-exploration of algorithms with hardware that will be of broad applicability to customized processor design in general.

[1] assuming a minimum packet size of 40 bytes

We begin the chapter in Section V.B explaining the design requirements of modern throughput-driven backbone routers and show how they differ from traditional latency-driven computing architectures. The unique requirements of processing network algorithms requires a new bandwidth-centric architecture that can deliver optimized worst-case performance. In Section V.C we summarize, at a very high level, the space of network processors and discuss how our design fits into this space. In Section V.D, we then describe and characterize our high throughput memory design, and contrast it with memory designs geared towards minimal latency. A key point of this research is the importance of understanding both architectural and algorithmic constraints at the same time. To this end, Section V.E focuses on co-exploring tradeoffs in both the architectural and algorithmic dimensions, and shows how a single architecture can perform well across a broad set of applications, if the algorithms are tuned in unison with the architecture. The chapter is then concluded in Section V.F with a summary of our contributions.

V.B Routing in the Backbone

The goal of this research within the application domain is to develop an architecture capable of implementing network algorithms at the speeds required by future backbone routers (10 Gb/s to 40 Gb/s), while remaining general purpose enough that new algorithms can be implemented completely in software or with a minimum of additional hardware complexity. In contrast, most programmable network processors on the market today target low performance (from 100 Mb/s to 10 Gb/s) low cost edge routers, leaving the task of routing in the backbone to ASICs. We know dive into some of the specifics of the application domain to provide context into the co-exploration process described later.

V.B.1 Backbone Router Model

The following model of a how a packet and router interact will help to ground the discussion that follows. A packet arrives at a router on an input link to an input line card that contains a router processor. This processor first does any security checks (e.g., screening for suspicious strings) and then performs a lookup. The lookup maps the destination address to the router output link that the packet should be sent to. The packet is then switched, using an internal crossbar, to the output line card specified by the matched prefix.

The processor has a forwarding table containing around 100,000 prefixes; these are bit strings of lengths 8 to 32 bits, representing networks around the world. The processor finds the longest prefix in the table that matches the 32 bit IP destination address in the packet.

The input processor may additionally classify the packet using a database of rules on header fields. At the output line card, classified packets are buffered in separate queues for each traffic class (e.g., low delay, high priority). The output line card also contains a processor which schedules packet queues for transmission on the output link in order to provide QoS (e.g., delay) guarantees.

The traditional approach to high speed router forwarding is to build custom ASICs that perform the required functions at these high speeds. Designing such ASICs is increasingly becoming a Herculean effort. As bandwidth requirements grow, network protocols become more complicated, and higher frequencies complicate physical design. Each new generation of routing ASIC is larger and more complex than the last, requiring larger teams from design to verification.

To combat this increasing complexity, a software based approach is needed that requires fewer engineers and less time. For software to run at the speeds required, we must create a new class of efficient, easily programmable, and scalable architectures to which we can map many different network algorithms.

Currently most backbone routers use ASICs for packet processors. Note that the ASICS handle packet forwarding. Routing control packets are forwarded to a separate line card that contains a route processor, a programmable processor which forwards packets two orders of magnitude slower, but can do general purpose computation required to recompute prefix lookup tables or to send out routing updates. Hardwiring the core forwarding path implies that protocol updates or advances in network algorithm design face serious deployment obstacles.

A generalized routing architecture has two major advantages over the traditional method that uses a myriad of ASICs. First, a major concern for router designers is to reduce Non-Recurring Engineering (NRE) costs (e.g., for masks, design time, etc.) which are becoming a dominant fraction of the overall costs for ASICs as speeds increase. Second, a programmable processor allows changes to the algorithms after installation. Such changes were not required for the traditional IP path that was cast in stone, but is becoming increasingly necessary today for security applications. This is a large part of the reason that network processors are expected to grow into a one billion dollar market within the next 4 years [53].

To address these issues, we focus on the design of a programmable architecture capable of handling modern network algorithms at backbone speeds. Note that our goal is *not* to replace the entire router with programmable parts. We continue to assume a hardwired internal crossbar switch, custom logic for buffer handling, etc.; the programmable processor only implements complex algorithms with large amounts of state such as IP lookups, classification, scheduling, and security checks.

V.B.2 Routing Requirements

Many initial attempts at designing network processors essentially retrofitted general purpose computer architectures to work in the networking space. While these are acceptable at low speeds, the goals of a programmable router processor are very different from a general purpose computer, and as such are governed by a different set of requirements. This leads to the following key differences in design assumptions that we address in our design:

- *Throughput is critical, not latency:* A single 40 Gigabit link, forwarding minimum size packets (40 bytes), requires that 125 million packets be handled each second (one every 8 ns). The handling of each packet requires multiple memory references, causing memory bandwidth to be of paramount importance. However; while the demand on throughput is very high, each individual packet may take many hundreds of nanoseconds from arrival to departure.

- *Design for the worst case:* This requirement is contrary to the current trends in processor designs that are based on optimizing for the average or expected case. In contrast, network vendors rate their products as supporting a particular wire rate. It is important that the performance delivered by their product be stable for any combination of test packets with which a customer chooses to test it. [2] This greatly complicates the design and analysis of a practical memory system.

- *Do not assume locality in the packet stream:* The final important point of difference follows from the previous two. Exploiting various forms of locality (e.g., spatial, temporal) in data sets is a tenet of computer architecture, one that has made possible the common case performance of desktop systems

[2] A router that cannot process packets at wire rates can drop (possibly important) packets before they are even examined and additionally may be susceptible to Denial-of-Service attacks using worst-case traffic.

today. Unfortunately, because backbone links merge together millions of individual packet streams [137], and because our goal is worst case throughput, we cannot rely on inter-packet locality to improve performance.

The standard worst case assumptions used by network designers [74] is that packets are minimum sized (roughly half the packets on the Internet are 40 byte TCP acknowledgments [137]), and there is no locality between packets.

V.B.3 Our Approach

Our approach to designing a 40 Gb/s router processor was to first understand the algorithms and techniques in use on many of today's high end router ASICs. We generalize from these examples to propose a programmable architecture capable of achieving high bandwidth for worst case performance. For each algorithm, we build an analytical model of its behavior and characterize the performance of the algorithm working in unison with the architecture. We use these models to explore design tradeoffs between the efficiency of the algorithm, and the circuit design options available for implementation. In our analysis and simulations we always assume a worst case packet stream for each algorithm. Likewise, all the results we present are lower bounds on the achievable performance across all possible packet streams.

V.C Network Processors

In this section we describe two popular network processor models (shared memory multi-threaded multi-processors, and pipelines of processors using partitioned memory). We then describe our underlying wide-word processing model, made possible by deeply pipelining on-chip memory. As with any interdisciplinary work, we can only touch upon a sampling of related work in network processors and refer the interested reader to [115] for a more comprehensive survey.

V.C.1 Independent Processors with Shared Memory

The top of Figure V.1 (part a) shows a common architecture for many network processors such as the Intel IXP1200. Internally, the processor contains several (e.g., 6 for the IXP) CPUs. The packet stream is divided among CPUs, with each processor working on multiple separate packets (threads) at the same time. When the processing for one packet stalls because of a memory reference, a CPU resumes processing for the next thread. Using fast context switching between threads, and 4 contexts per processor, the IXP1200 can process 21.4 M packets/second [133].

The main drawback of this design is that it is built around traditional memory, which is optimized for latency rather than throughput. To achieve high memory bandwidth, multiple processors and thread contexts are used to generate multiple pending accesses. The problem is that each processor does all of the forwarding for an assigned packet, and each independent processor needs independent access to a large amount of shared state (e.g., the global forwarding database.).

The only known way around this problem is to either replicate the state for each processor (e.g., replicate the 100,000 entry forwarding tables across all CPUs), or to share the state in some fashion. Replication is prohibitively expensive for the large problem sizes that routers need to deal with, especially given limited (e.g., 32 Mbits of SRAM) on-chip memory. The problem with sharing state is in providing *independent* access to the entirety of the state. In the worst case, if every data access of every processor was to the same bank of memory, all the benefit of multiprocessing would be lost because all memory accesses would be serialized. While these techniques work well on systems with small problem size or without worst-case performance constraints, neither of these options meets the requirements we set forth in Section V.B.2.

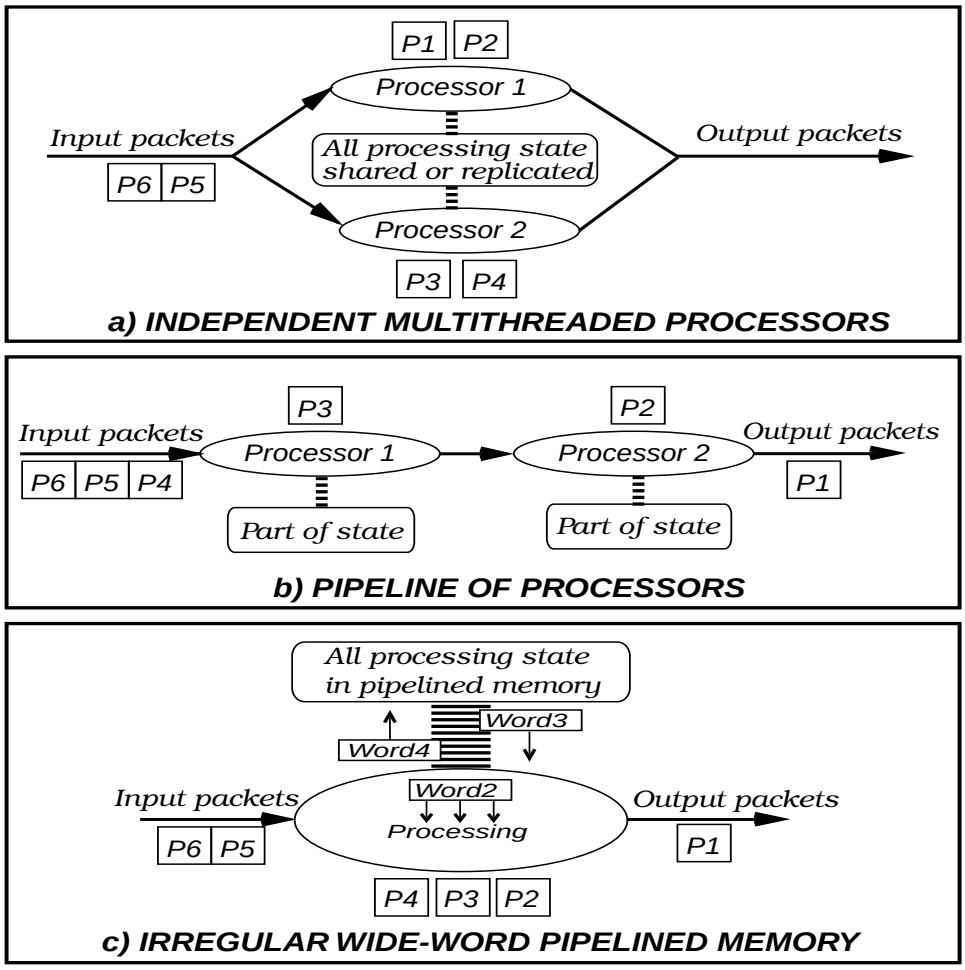

Figure V.1: Several ways of building network processors

Shown are two common approaches to extracting parallelism in network processors versus our pipelined memory design. In the top picture (a), input packets are distributed among multiple processing units to divide the load. Each processing element works on the complete handling of a packet but can work on multiple packets (threads) at a time. Thus each processing element must access all of the memory state. In the middle picture (b), input packets flow through a pipeline of processing elements. Each processing element performs a portion of the processing for each packet, one packet at a time. Thus each processing element need only access separate parts of the memory state. In our scheme (c), there is no need to partition the data structures, and memory throughput requirements are met by using wide words and pipelined memory.

V.C.2 Pipeline of Processors

The lack of memory bandwidth in the traditional shared memory model has led some companies like Intel and AMCC to move to a pipeline of processors, with multiple distributed memories. The model is that each processing element in the pipeline does part of the processing of a packet for *all* packets; for example, in the middle of Figure V.1 (b) processing element 1 can do the initial processing of a lookup and pass the results to processing element 2 for completion. The drawbacks of pipelined architectures are:

a) Memory Utilization/Contention Tradeoff: The simplest way to design the pipeline is to statically partition the processing state (e.g., IP lookup table) among processors to avoid contention, as shown in Figure V.1 (b). In algorithms such as IP lookup, the data structure is a tree, and the different levels of the tree are partitioned among the pipeline of processors. The work at the root of the tree is assigned to the first processing element, the root's children are assigned to the second element, and so on. Unfortunately, the trees are not balanced; the shape of the tree can vary from lookup database to database. Thus the memory needs of each CPU in the pipeline becomes unpredictable. This in turn means that static partitioning of the limited available on-chip memory among multiple distributed memories can be very wasteful of memory.

b) No support for writes: A pipeline of processors works best for functions such as lookups where writes are done only occasionally and not in real-time. However, there are important router forwarding functions that require forwarding time writes. For example, QoS scheduling involves real-time writes to update flow tracking registers, which then may have to be forwarded to all of the other independent memories. This make a worst case analysis on the mapped algorithm very difficult.

V.C.3 Our Solution: Pipelined Wide Word Memory

Our solution is motivated by the problems described above, which are all caused by the need for worst case memory bandwidth. We avoid these problems as follows. First, we make each memory access wide to increase the number of bits retrieved per access. Second, we observe that a memory subsystem consists of memory banks and an interconnect that ferries data read from a bank to the output. Traditional memory maintains the invariant that two readers do not access the same block by ensuring that *there is only one reader in the entire subsystem*. However, this is unnecessarily restrictive. In summary, our memory subsystem uses:

- *Wide Words*: We note that the performance of many network algorithms can be increased significantly by the use of wide data words. Instead of having to worry about multiple independent readers of memory having bank conflicts, we can instead provide aggressive memory bandwidth to one reader at a time through wide word access.

- *Internally Pipelined Memory:* Circuit design realities force traditional memories to sacrifice worst case throughput in order to reduce latency. Because our applications are incredibly latency tolerant, we can greatly increase the throughput by *internally* pipelining the memory to allow the memory system to work on multiple packets at once. For example, in the packet stream of Figure V.1, the memory can be delivering a word required to process packet $P1$, while at the same time fulfilling a read request for packet $P2$.

To best utilize our wide-word pipelined memory, our underlying processor model is also different from either of the architectures described above (shown in Figure V.1 (a) and (b)). Our design in Figure V.1 (c) has one multi-threaded processor working on multiple packets at the same time. When a wide word

is delivered from the memory system, multiple computation units work together over the whole word performing the calculations needed to process that word and generate the next address. Multiple packets have their processing interleaved, as our pipelined memory allows multiple memory requests to be outstanding at a single time.

V.D Memory Architecture

In this section, we describe the details of our approach, including a description of the tradeoffs of wide word and pipelined memories, the model of the machine presented to the programmer, and a characterization of the architecture.

V.D.1 Motivation for Wide Word Memory Architectures

Network algorithms traverse large data structures and are designed to provide worst-case bounds on both processing time and memory. The use of wide memory word accesses is very common within network ASIC designs as a means of increasing memory bandwidth and reducing processing times. An example of this can be found in the way that tree-based IP lookup algorithms are typically implemented. IP lookup can be efficiently implemented by storing prefixes in a *trie* that looks up one bit of the address at a time and goes either right or left through the trie based on this bit. In hardware trie algorithms, a state machine traverses the trie using bits in the destination address to obtain the output port. One technique to decrease search time is to reduce the tree height by packing multiple nodes in a trie path into a single wide word [134]. This allows more than one bit to be looked up at a time using a single memory access, but increases the size of the memory access and the fan-out of the trie. This is just one example of how wide-words can be used in network algorithms. In Section V.E, we briefly explain for each of our algorithms how a wide word

can be used to speed up the algorithm.

Because of the common use of wide words in a number of special purpose networking ASICs for lookups, classification and QoS, we believe it is important to abstract this ability in a programmable architecture by exploiting wide word access in the memory design and exposing wide word access at the programming level.

V.D.2 A High Throughput Pipelined Memory

While multithreading and pipelining are very standard ideas in the network processor context, one of our main contributions is showing how to get high worst case memory throughput by internally pipelining the memory. In the traditional memory layouts that prior network processors are designed around, only one or two (if multi-ported) memory lookups can be performed at the same time in the worst case. These designs use large memory tiles, trying to minimize the interconnect delay for latency, and rely on common case techniques such as locality and independent accesses to provide sufficient bandwidth. As such, the worst case throughput of such systems is limited to one or two transactions being processed by the memory at the same time, when, in the worst case, all the lookups fall into the same bank.

In contrast, our new memory design trades off latency for throughput. We propose pipelining the memory design using smaller memory tiles to process many transactions at the same time (see Figure V.2). This will result in a higher latency per transaction, but the number of transactions per unit time will be significantly higher than in a traditional memory design. To achieve this, we start by breaking the memory into smaller tiles, each of which is capable of reading out a wide word each memory cycle. The cycle times for these smaller banks are two to three times faster than monolithic memory banks, even assuming

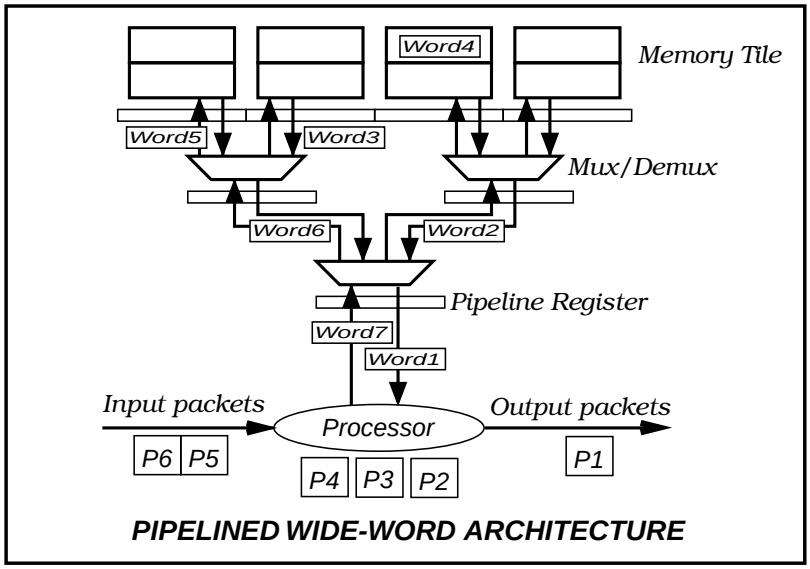

Figure V.2: Pipelined memory diagram

This figure shows how our pipelined tree-based memory can have many accesses flowing through it at the same time. At any given time, there is only one wide word access going to a memory bank, and all the memory banks are at the leaves of the tree. At each level in the tree (for the single ported version) there are at most two words in flight, one going up to the memory arrays and one coming back down.

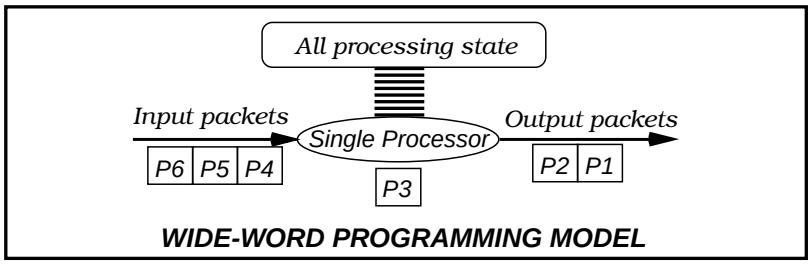

Figure V.3: Programming model for pipelined memory

While the actual architecture shown in Figure V.2 is quite complex, the programmer need only worry about programming their algorithms for a simple wide-word processor. This is because the architecture is designed to provide good performance even in the worst case, so there is no need to worry about the careful location of data structures in banks, or partitioning the program state between two separate memories as in Figure V.1(b).

optimal internal partitioning. Our design will have a larger interconnect delay, and hence a larger overall latency; however by pipelining the interconnect, we achieve better overall throughput because the delay of each stage is smaller than the overall latency of a traditional (non-pipelined) memory design. Therefore, the memory design can have N lookups in flight, where N is the number of pipeline stages (twice the height of the tree to go up and back down) of the memory layout.

Figure V.2 shows a high level diagram of our design. The memory has been divided into many small banks. At the bottom of each small bank is a pipeline latch that grabs the data we need. Data read from a bank then needs to travel from the tile (which could be quite far away) back to the processor. This is done over a deeply pipelined tree interconnect. When an access is made, a lookup is performed in parallel at the leaves of the tree, and each cycle the correct data item marches its way down through the pipelined tree to the root where it is read out by the processor. Since the tree is balanced and full, and there is only one memory access actively accessing the banks at a time, we are guaranteed that during the entire journey over the interconnect there will be no conflicts. This ensures that we will always be getting one wide word of data each cycle. The tree interconnect is not a trivial piece of real estate, because it needs to mux, demux, latch, and move large words of memory over a long distance. In fact, as we will see in Section V.E, the interconnection network accounts for between 10% and 30% of the total area of the memory system.

By multithreading applications, using small tiled memories with access to wide words, and including an efficient balanced interconnect with worst case bounds, our design effectively supports many in-flight accesses to memory to provide high throughput.

One final hardware optimization which we include in our memory design

trade off analysis, is the use of multi-ported memories. Multi-ported memory is slightly slower (Virage reports 10%), significantly less dense (75% larger for the same number of bits), and requires even more overhead in the form of interconnect area. Despite all of these draw backs, multi-ported memory provides two accesses to each memory bank even in the worst case. As we will see later, even though the penalty is significant, using more ports can be a good idea for more aggressive implementations.

V.D.3 Characterization of Architecture

Now that we have presented the main architectural ideas, we need a method of exploring and testing our ideas along with some intuition as to how the architecture reacts to different parameters.

In order to do this we need a way of quantifying the effect of memory bank delays and areas along with interconnect overheads. To calculate the memory bank parameters we used an analytical model together with a design walker that can estimate the area and delay of .13u SRAM for each bank. This was then validated against industry memory compilers (from Virage and Austria). For the interconnect, we modeled an H-Tree with additional area for pipeline registers and mux/demux modeled from VLSI layouts.

To gain some initial intuition as to how some of the key parameters of our memory design trade off against one another, we present Figures V.4 and V.5. These figures show the effect of increasing the number of pipeline stages and the width of the word used.

Figure V.4 shows the overhead and benefit resulting from using different numbers of pipeline stages for a word size of 128 bytes and 1 read-write port. The first line (drawn with a solid line) is the cycle time normalized to a traditional memory organization (which has been banked internally for latency). With only

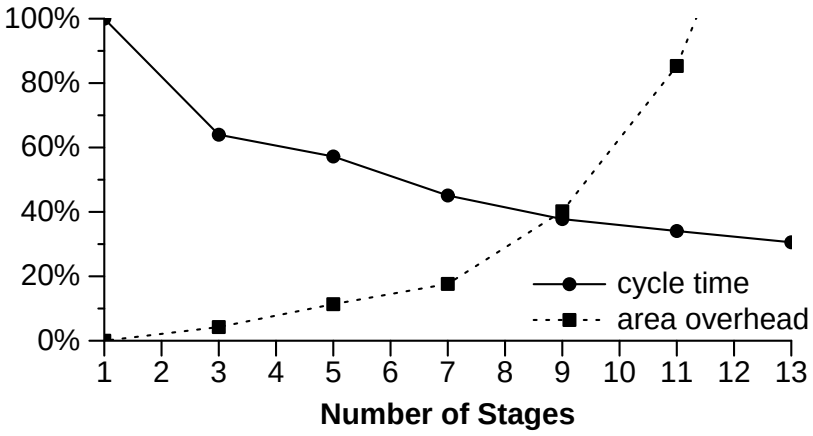

Figure V.4: Quantifying the overhead and benefit of pipeline memory stages.

For the line labeled `cycle time`, the y-axis is the percentage reduction in cycle time for access to a single memory bank. For the line labeled `area overhead` the y-axis is the percentage area overhead due to additional interconnect. Remember that the pipeline is not linear, but is instead a tree: this is why the pipeline area overhead increases non-linearly with number of stages.

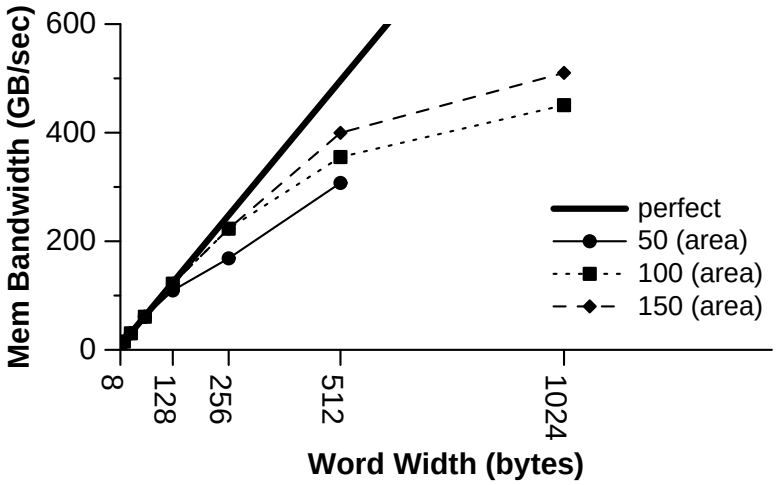

Figure V.5: Memory bandwidth versus size of word accessed

Show is the total memory bandwidth in Gigabytes per second versus the size of word accessed in bytes. The line labeled `perfect` assumes linear scaling of bandwidth with word size. In reality there are diminishing gains above words of 256 bytes caused by interconnect overhead cutting in on the the amount of area we can use for sub-dividing the memory banks.

one stage, the memory is not pipelined at all and thus the cycle time is 100%. As we increase the number of stages, the cycle time decreases as the memory is divided into smaller and smaller independent tiles. Doing this sub-division comes at a cost, and the second line quantifies this cost in terms of area overhead (in terms of percent increase in area over a single memory block). The overhead comes from both the cost of splitting a tile into 2 smaller tiles and the increased area due to interconnect.

Figure V.5 shows the total memory bandwidth versus the size of word accessed. If there are no circuit level limitations to increasing the size of memory words, the bandwidth should increase linearly (drawn as the `perfect` line in the graph). In reality there is an overhead in increasing the width of the words due to increased interconnect area and slower bank access times. In Figure V.5 there are 3 lines drawn, for different constraints on the area. The lines show the maximum memory bandwidth that can be achieved by our system under the given area constraints. Even up to quite large widths (over 128 bytes) there is not that much impact on bandwidth because it is all pipelined. We start to see smaller benefits (caused by interconnect overhead cutting in on the amount of area we can use for sub-dividing the memory banks, and by poor memory folding) when we get to widths above 256 bytes.

The above graphs are just a sampling of the tradeoffs involved in our architecture and do not even include any tradeoffs at the algorithmic level. In Section V.E, we provide a detailed examination of the tradeoffs we considered.

V.D.4 Programmer's View of Memory

We would like to *avoid* forcing the programmer to deal with wide word memory layout problems. For instance, lower speed network algorithms are often implemented using interleaved DRAM chips such as RAMBUS, where great care

must be taken by the programmer to lay out the data structure in appropriate banks to avoid bank conflicts. This clearly increases programming effort and in some cases makes efficient design impossible. The task of layout is further complicated by the fact that most of the algorithms are both dynamic and irregular in nature. This, as discussed in Section V.C, makes partitioning and replication a very difficult problem to solve, especially in the worst case.

Instead, the model which we would like to present to the programmer is that of a large single unified on-chip memory. It will be our task to make sure that the internal memory design details required for a high speed design are hidden from the programmer. Figure V.3 is a high level view of the model we present to the programmer. The figure is simple because the model presented is simple. The memory appears as a large uniform block of wide words. The processor performs a read, does a wide word operation, and then writes that word back to the memory (if it needs to). The challenge is to provide a high performance (high throughput) architecture that can still be programmed in this straight forward way.

V.D.5 Mapping an Algorithm to Wide Words

While the details of designing and programming algorithms for our architecture are outside the scope of this dissertation, we comment briefly on how this can be done. The first task is the transformation of the applications to take advantage of the wide-word accesses made available to the programmer. As mentioned earlier, our applications walk large data structures, and thus by default contain many dependent memory accesses. One way that this maps to our architecture is by taking multiple dependent memory references (such as a region of a tree) and flattening them into a single wide word access. However, different algorithms can use the wide words in different ways.

The next task is the processing of each wide word that comes back from memory using a rule based system. The wide word is partitioned into pieces (not necessarily at byte boundaries), and each piece has a small amount of computation performed on it in lock-step (similar to functional units for VLIW). This is followed by a merging step which takes the piece-wise results and makes final decisions or computations (such as which of many next-pointers to follow in a tree).

V.D.6 Related Architectures

Now that our architecture has been described at a high level, it is important to describe how our architecture differs from other well known architectures which are not specific to networking.

The Imagine stream processor [107] is a close relative to our design. Both are designed for memory bandwidth and focus on total throughput as a metric of performance. However, the Imagine stream processor is built for graphics and signal processing applications that are compute intensive. The Imagine model of computation is one of streaming data between dependent operations, and is built around the reuse and locality commonly found in applications in that domain. The applications we target, on the other hand, navigate through large data structures optimized for worst case performance. Thus Imagine uses reuse to tolerate large memory read times with standard memory designs, while we use a pipelined non-traditional memory to get large throughput without reuse assumptions. On the other hand, our architecture has a higher overall access latency which would be undesirable in many other contexts.

We also share similarities with vector architectures. Vector architectures, much as we propose, operate on very long words of data, or vectors. IRAM [72] is an example of such an on-chip vector architecture. Again, the

difference is both in the assumed reuse of a vector and in the applications. Vector machines were built with large regular scientific codes in mind. In order to perform a vector operation the programmer must write his application in the form of vectors. The problem with the network algorithms that we target is that the applications are not regular. Each word that we operate on describes a node in a tree (or a state in a state machine) rather than an array of numbers on which to perform arithmetic.

The final point of comparison is with VLIW architectures. VLIW architectures operate on long irregular *instruction* words. Each word is a set of independent instructions that can operate on any machine state. Our approach is more akin to a *Very Long Data Word* architecture in that we limit operations to the scope of one word at a time. However, our ideas can be combined with a very large instruction word that feeds the logic that works on long data words.

V.E Co-Exploring Network Algorithms and Memory Design

The computer architecture community tends towards a passive view of the software that it is tasked with executing. This is because it is simply not possible to make sweeping changes to the algorithms and data structures used in most general purpose environments. In an embedded enviornment this limitations may no longer apply, or at least may be more flexible. In the embedded space it may be possible to design, not only a processor customized to an application, but an algorithm customized to an application and processor.

Specifically, in the domain of network processors, we propose to take advantage of parameterized network algorithms and data structures that can exploit a wide word architecture. We do so to expose tradeoffs between the algorithm's need for resources, and the architectural and circuit level realities. In

this section we show that co-exploration in this domain space is both needed and useful and we present a new design methodology that can enable tradeoffs across both degrees of freedom.

V.E.1 Algorithm and Memory Design Co-Exploration

In Section V.D.3 we examined some of the behaviors of our proposed architecture to illustrate the nature of the tradeoffs we have to deal with, but to do so we made many simplifications in terms of the parameters examined. As we are about to show, in reality the interaction between the parameters is far more complex.

Figure V.6 is a graphical representation of the interaction of the parameters in both the algorithmic and architectural design spaces. Software elements are shown as ellipses and hardware elements are shown as boxes. The goal is to fix one of the two outputs (network throughput or area), and then to minimize the other. This graphical representation shows what makes this problem both difficult and interesting: there are many dependencies that interact in complex ways.

For example, if we fix the amount of area allowed, and we want to increase the amount of network bandwidth, we can try to reduce the total number of memory accesses required to process each packet. This can be done by extracting more wide-word parallelism from the application, but in order to do this we need to increase the size of the words read from memory. Increasing the word size affects almost everything else in the system including memory cycle times and interconnect overhead. This example also demonstrates another key point: algorithms are closely coupled with the choice of hardware parameters. Ignoring one, while attempting to optimize the other will be far from the best design.

The results in this section were generated by examining many different

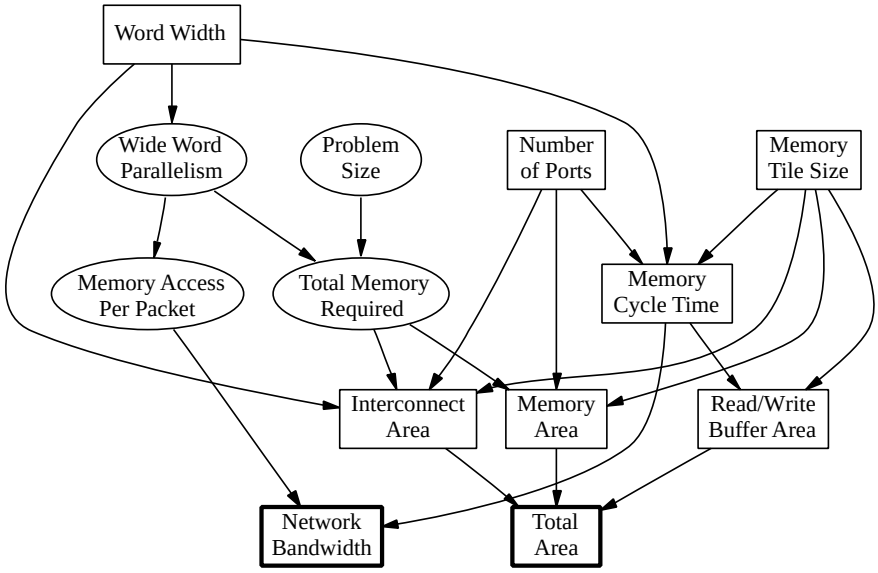

Figure V.6: Hardware and Algorithm tradeoff interaction

This is a graphical representation of the dependencies between hardware (drawn as boxes) and software (drawn as ellipses) when attempting to quantify wide-word pipelined memory network processor performance and area.

architectural and parameter sets, including but not limited to: the width of word in the memory and algorithm, the total size of the memory, the tiling of the memory, the number of read, write, and the read-write ports in the memory. Once the architecture model has been specified, the algorithm is modeled to take advantage of the architecture at hand. Each algorithm takes advantage of the memory width available, the number of ports (read and write) for access, and is limited by the amount of memory available.

While we are certainly not the first to propose design space exploration for network processors [29, 145], we believe that we are the first to systematically co-explore algorithmic and architectural tradeoffs together under worst case performance constraints.

V.E.2 Network Algorithms

Note that there are two different time scales at which routers process information: *forwarding time* and *route time*. The computations that occur in route time are those that set up the data structures used by forwarding time algorithms; route time computations happen relatively infrequently (on the order of every millisecond) compared to the rate at which forwarding happens (on the order of nanoseconds). For this reason we are not targeting the route time operations which form the bulk of router software. In contrast, computations that happen in forwarding time are the small amounts of code that need to be performed on every single packet that flows through the system. Making a chip that scales to these speeds while maintaining programmability and scalability is the real hardware challenge for backbone routers.

The tasks that need to be performed on each packet may be broken down into the following categories: Lookups, QoS, Classification, and Security. We define our benchmarks as a set of requirements, such as performing an IP lookup or classifying a packet stream rather than limiting ourselves to a particular implementation. Standard network processing benchmark suites either focus primarily on end user or edge functionality [30, 85, 146], or tie directly to a set of source code, neither of which is suited to our end goal of backbone router deployment. While the following algorithms are by no means an exhaustive set, they do provide a coarse generalization of the different types of algorithms used in modern and next-generation core routers. We use these to show the broad applicability of our architecture rather than as a hard proposal for a new benchmark suite. For each of these four algorithms we provide a description of the functionality, briefly explaining how it may take advantage of wide word accesses, and examine how well the algorithm maps to our architecture in terms of its estimated performance. We begin with a more in depth look at IP lookup, as it is one of the

Original		Port
1⋆	→	P1
0⋆	→	P2
01⋆	→	P3
011⋆	→	P4

Expanded		Port
10⋆	→	P1
11⋆	→	P1
00⋆	→	P2
01⋆	→	P3
0111⋆	→	P4
0110⋆	→	P4

Figure V.7: IP lookup table before and after controlled prefix expansion. The table on the left is smaller, but the table on the right can be more efficiently traversed because all of the patterns are aligned on 2-bit boundaries which allow 2-bits to be traversed at a time as opposed to 1 in the unexpanded table

canonical examples of network algorithms.

IP Lookup

The most fundamental activity that a router must engage in is the forwarding of packets. In order to determine which link any given packet must forward to, a lookup must be performed. IP lookup takes a stream of destination IP addresses[3] as input, and is tasked with finding the output port that each packet must be forwarded too. This is complicated by the rules for packet forwarding, which require a *longest prefix* operation.

We spend more time describing IP lookup than the other algorithms because we use IP lookup as an example of the various techniques we have developed, and with the hope of giving the reader an intuition behind the architectural and algorithmic decisions we examine. For a survey of IP lookup algorithms we refer the reader to [110].

The left table in Figure V.7 is an example IP lookup table. Any packet that enters the system is matched against the rule set, finds the longest match, and forwards the packet to the listed port. For example, a packet going to the address 1111 in our example will go out on port 1 (P1) because it matches 1⋆,

[3]Each destination IP address is either 32-bits (IPv4) or 128-bits (IPv6)

and it is the only match. If a packet was going to address 0110, it would match three different patterns, 0⋆, 01⋆, 011⋆, but only the longest pattern, in this case 011⋆, is used and the packet would be forwarded to Port 4 (P4).

One solution to this problem is a Content Addressable Memory, or CAM. While CAMs can work well in some instances, they have trouble scaling both to the number of prefixes in modern routing tables (due to the increased size as compared to SRAM) and in available bandwidth. Instead, most of today's high end routers use a tree-like representation of the list of prefixes, and then walk this tree at high speed in hardware. A Patricia-Tree [89] is an example of this sort of tree that many people are familiar with.

Using Wide Words: While the Patricia-Tree can be effective in lower speed software-only routing implementations, the state of network algorithms has advanced significantly since the creation of Patricia-Trees. Instead, several routers use a Multi-Bit Trie approach as is detailed in [134, 32]. In this approach, the lookup tree is expanded so that multiple bits can be searched in parallel. An example of this can be seen on the right side of Figure V.7. In the expanded table, you can see that all patterns are aligned on 2-bit boundaries which allows the table to be searched 2 bits at a time. At the same time you can see that the functionality of the table has not changed. The address 0110 now matches 0110⋆ (and in addition 01⋆), but the effect is the same (being forwarded to Port 4).

In general you can align the entries on any n-bit boundaries, enabling the lookup of n address bits at a time, and reducing the number of tree accesses (and memory access) by a factor of n. This optimization comes at the cost of expanded table sizes; in this example the table has gone from size 4 to 6, and generally the larger the number of bits that are looked up in parallel, the larger the table will get.

Because the memory expansion of multi-bit tries is large, the tables can

grow large (on the order of 100,000 prefixes) and the amount of on-chip SRAM is limited. Thus, many routers today use compressed multibit tries such as the Lulea [32] scheme. Such schemes compress away the redundant portions of multi-bit trie nodes by adding bitmaps indicating repeated values in a multi-bit node. Bitmap compression can reduce memory needs considerably, at the cost of a increased memory accesses per node to read the bitmaps.

However, our wide word architecture provides us with a simple alternative first described in [98]. We can represent all the prefixes (and their corresponding pointers to other nodes) that lie within a multi-bit trie node without expansion. Assuming a node fits in a wide word, the entire node can be read in one memory access and the destination address can be compared in parallel with all the prefixes in the word. The longest matching prefix within this node is selected and the corresponding pointer is followed to fetch the next multi-bit trie node in the tree. Essentially, we are doing a parallel associative lookup of the node's contents to finesse the need for expansion or bitmaps.

Co-exploration: In Figure V.8 we can see the overall tradeoffs that can be made by varying the parameters of both the architecture and the width of word used by the multi-bit trie algorithm. The x-axis is area in square millimeters, while the y-axis is the network bandwidth that can be supported by each algorithm. Every set of parameters that we examined is shown as a point on the graph. Pareto-optimal points are highlighted as circles. For this work a set of parameters is considered Pareto-optimal if there exists no other measured set of parameters that has both smaller area and better performance. For a reasonably sized system of 100 square millimeters, our technique can support throughputs approaching 100 Gbits/s assuming 40-byte packets. The key to achieving these bandwidths is the tradeoff between the extent to which IP lookup can exploit wide word access by searching multiple bits at a time, and the circuit level realities

discussed in Section V.D.

To search the space of IP addresses with wide words, you need to double the length of your word to search one more bit at a time, so there is a logarithmic relationship between the word size and the total number of memory accesses that the algorithm will need to make. In Figure V.10, we examine this tradeoff for a particular slice of the parameters. For each algorithm, we plot the points for the highest throughput for each word size for a memory with 1 port that is limited to an area of 50 square millimeters. The results show that for IP Lookup, a width of 128 bytes is the best balance for all of the constraints.

Quality of Service

Supporting quality of service (QoS) primitives is a standard requirement for any high speed router today. Based on packet classification at the input port, packets are separated into queues at the output port of the router. For example, all web traffic may be directed to one queue. QoS is provided by a scheduling policy that determines which queue is serviced next to send a packet on the output link.

The simpler scheduling functions allow queues to be prioritized (e.g., for Voice over IP), rate limits to be applied to some queues using so-called token buckets, and simple, weighted round-robin schemes such as Deficit Round Robin (DRR). A DRR scheduler services queues in round-robin order, allowing each queue to send a quantum Q of bits in each round-robin cycle. Since only whole packets can be sent, and packet sizes may not fit into the quantum Q, the DRR scheduler keeps track of the quantum not used by a queue in one opportunity and gives it back in the next. All of these schemes maintain some per queue state on which the scheduler must base its decision as to which queue to send from next. The number of queues can be large; some routers have around 64,000 queues per

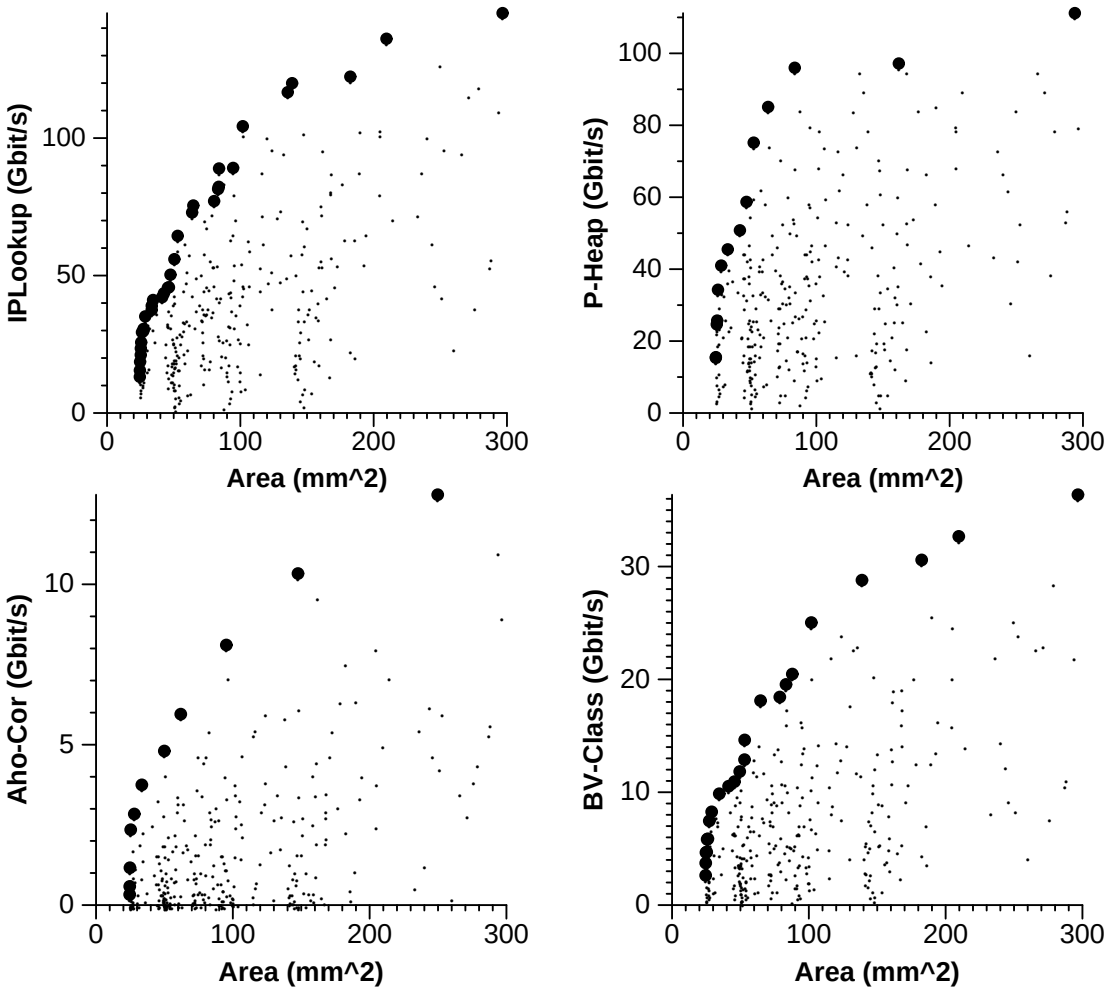

Figure V.8: Area-Performance tradeoffs for each algorithm
These tradeoffs are for the different algorithms, as discovered by exploring the algorithm and architecture in unison for each algorithm individually. The x-axis is area in square millimeters and the y-axis is the network bandwidth that can be supported by each algorithm. All points examined are shown and Pareto-optimal points are shown as circles.

output link.

More sophisticated schedulers, such as Weighted Fair Queueing [33, 12] and Virtual Clock [152, 153], have been designed to guarantee delays for delay-sensitive traffic, such as video, by implementing a scheme similar to earliest deadline schedulers. Thus each packet carries a deadline (or time stamp); the basic real time bottleneck is to select the lowest time stamp packet for transmission.

A natural data structure to compute the lowest time stamp is a heap. A heap is also useful in simple forms of QoS. For example, a heap can be used in rate limiting queues [36] and computing priorities, It can also be used in DRR to compute which queue to visit next in round robin order, while skipping over queues that have no data. An example of a data structure that allows the basic heap operations at high speeds and has a compact hardware implementation is the Pipelined-heap structure, or P-heap [14].

Using Wide Words: The P-heap is a structure that supports fully pipelined operation on a heap through the use of a side buffer that is used to communicate information between the stages, each of which is a level of the heap. The technique presented in [14] is designed with a hardware implementation in mind, but we adapt this technique to explore the algorithm space with our wide word memory design. We extend this work by supporting an even more aggressive heap structure that instead of being a binary heap (where each node has up to 2 children), can support heaps of arbitrary fanout. Through the use of wide words, we can support a higher out degree at each node in the heap, which means that the total height of the tree will be reduced. Note that wide word heaps, without the use of pipelining, have been implemented in custom ASICs in the past [36].

Co-exploration: The P-heap, unlike the Multi-bit Trie, is a balanced tree, which makes the analysis a bit simpler. In the case of this balanced heap, we can say that the height of the tree is strictly a function of the log of the number of

nodes and the log of the fanout. However the structure is complicated by the fact that there are forwarding time writes. The major tradeoff for this structure is the fan-out of the nodes versus the memory and width constraints of the system. Here there is a constant size overhead associated with each node for storing the *active* and *value* fields (discussed in [14]), a counter which grows logarithmically in size with the number of nodes (the *capacity*), and a list of pointers which grows linearly with the fanout and logarithmically with the size of the memory. All of these must be examined in conjunction with the architectural tradeoffs.

Intuitively, by increasing the width of the memory we can make fatter nodes which then allows us to flatten the height of the heap. A flatter heap means less accesses to the heap are necessary in the worst case (where we need to access a path to the leaf nodes in the heap). Less memory access per packet means that more packets can be processed with a fixed amount of memory bandwidth, which means better throughput.

In Figure V.8, we can see the results of performing the co-exploration of parameters. The pipelined heap application, as mapped to our architecture, can almost reach 100 Gbit/s with 100 square millimeters. This is similar to the IP lookup algorithm. The major difference between the way IP lookup and P-heap are implemented, is that in P-heap there is almost nothing to be gained in using more than 100 square millimeters. This is due to the fact that for P-heap it is expensive to increase the fan-out of the data structure (in terms of more overhead and control bits), which means that the algorithm benefits less from extremely wide words. Figure V.10 shows this point for a slice of the parameters. As can be seen in the Figure, P-heap performs better with word sizes around 64 bytes. Above 64 bytes, the algorithmic gains of wider words do not balance out against the performance of the memory sub-system.

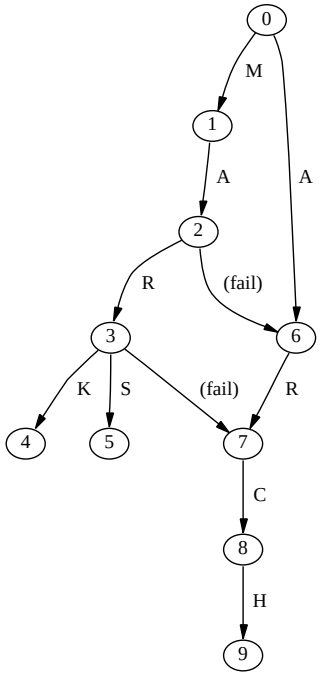

Figure V.9: Example state machine generated by the aho-corasick algorithm.

The nodes represent partial matches (the numbers are not representative of the state matched but are included only for ease of discussion), and the edges are traversed for the string being searched. Failure edges back to the root node (node number 0) are not shown.

Security

Another problem that network infrastructure providers are encountering is the growing need to be able to track and block particular types of data on the network, such as worms. The "Code Red II" worm, for example, is a piece of self-replicating malicious code that exploits a vulnerability in Microsoft IIS servers [23]. It has had an estimated world wide economic impact of over $2.6 billion dollars [1], and it can be stopped using a packet filter which checks for the string "scripts/root.ex-e?" in the URI content along with a quick examination of the packet length.

There are many existing software systems that are designed to detect

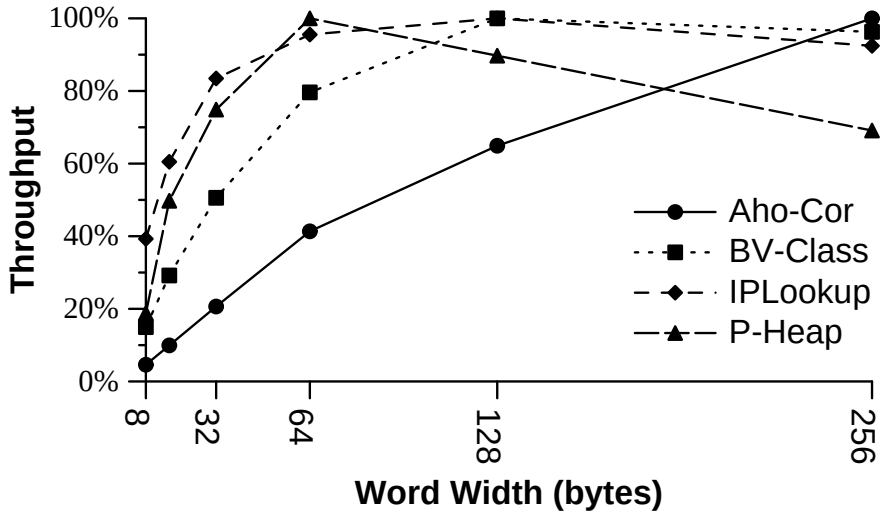

Figure V.10: Word width versus network bandwidth

This is a graph of word width versus realized network bandwidth (as a percentage of the best achieved on a per-algorithm basis) for memory with 1 port, and limited to an area of 50 square millimeters.

network intrusion or other malicious behavior. Unfortunately, the act of determining whether a particular packet is dangerous or not requires an examination of the payload of the packet. One operation in particular is essential, the ability to scan the packet contents for a set of strings which are known to cause trouble. Readers may be familiar with efficient algorithms for string matching such as Boyer-Moore [16], which are designed to find a *single* string in a long input.

Our problem is slightly different, we are searching for one of a *set* of strings from the input stream. Instead of simply iterating a standard one-string matching algorithm, the set of strings that we are looking for can be folded into a single state-machine. This method, the Aho-Corasick algorithm [5], is what is used in the `fgrep` utility as well as the latest versions of the `Snort` [108] network intrusion detection system.

The Aho-Corasick algorithm works by building up a state machine that is fed the string to be searched. The state machine is generated by building up a

tree of all the strings that need to be searched for, and a set of failure edges that are traversed when a search string on the path is not found.

This is another tree-like structure, but because of the failure edges (which point back to other parts of the tree), its traversal is not strictly tree-like. Another major difference between this algorithm and the others discussed thus far is the fact that this data structure must be accessed on *every single byte* of the packet, not just once for the packet header as in IP Lookup. This, as we will see, results in an order of magnitude less performance than the other three algorithms mentioned (as measured in total network bandwidth supported). However, we believe that this is a benchmark of growing importance and should be captured by our architecture.

Using Wide Words: To examine the behavior of the algorithm on real data, we generated the Aho-Corasick state machine for a set of strings used for actual intrusion detection and packet filtering. For this we used the default string set supplied with `Snort`, which includes, as part of it's rule base, a set of over 1000 suspicious strings resulting in an Aho-Corasick state machine with more than 10,000 nodes.

Because our data is no longer a proper tree (due to back edges), and because we want to ensure the worst case performance of the algorithm (across any packet stream), we now need to worry about the worst case cycle in the data structure. After building up a real state-machine from the `Snort` data, we found the major bottleneck to the system were several states at the top of the tree with very high fan-out. The root node in particular could have pointers to upwards of 150 possible next nodes, and the worst case cycle would always go through the root node. To combat this problem, we used our wide word resources to make these large nodes fit into less memory words. This handling of the large fan out nodes near the top of the state machine benefits the worst case performance as

the total number of memory access are reduced.

Co-exploration: In Figure V.8 we can see that our architecture, at 100 square millimeters, can only handle 8 Gbit/s. This poor performance (relative to IP lookup in terms of total supported bandwidth) is due to a combination of effects. By far the largest contributer to the poor performance is the requirement that one node must be traversed for every byte in the packet.

When we only have to worry about packet headers, life is much easier because we only need to do one operation (such as lookup the IP address) for every 40-bytes of network traffic. However, when we are forced to examine the payload, we need to process every byte of network traffic, which is time consuming by its very nature. The second effect is that our current mapping of the Aho-Corasick algorithm is not making full use of our wide-word architecture. We should be able to map large sections of the state machine to wide words, and developing better implementations of the Aho-Corasick algorithm is a topic of future research.

Classification

In recent years the trend has been for routers to rely less on traditional destination-based forwarding and more on the idea of service differentiation. The main idea is that services, such as allocations of bandwidth or reduced latency handling, are determined by the *type* of traffic that is being handled. Determining what type of traffic a particular packet qualifies as, and hence what rules apply to it, is driven by the header fields in the packet. For example, the destination address, source address and TCP/UDP port numbers could be used to classify priority voice over IP data.

Classification of a packet is done with a set of rules, each with an associated priority. Each rule specifies the type of packet that it applies to, typically

in the form of a range or wild-card, and the rule with the highest priority that matches is the only one that is applied. For example, one rule might match all traffic going from address 001$\star$ to 0100$\star$ on port 80 (web). Thus classification can be considered to be rule based forwarding, with priorities to handle conflicts. Even within the core, fairly large (e.g., 2000 rule) classifiers are commonly used for security. For example, many of the rules appear to be denying traffic that is from specified subnetworks and is directed at particular servers or subnetworks within an ISP.

Among the best performing classification algorithms are the HiCuts algorithm [52] and the Bit-Vector Linear Search (BVLS) [74]. The HiCuts algorithm is a decision tree which makes range checks at each node in the tree. It can benefit from a wide word architecture by merging multiple decision nodes into a single node to reduce tree height. While both algorithms can benefit from wide words, we chose to expose the tradeoffs using the conceptually simpler BVLS scheme. For a survey of classification schemes we refer the reader to [51].

Using Wide Words: In the BVLS algorithm, a separate trie is created for each of the fields (destination address, source address, port numbers, etc.) that can be used for rule matching. At the leaves of each of these tries is a long bit vector. Each element (bit) in the vector corresponds to a rule (from the classification rule set), and the vector is a list of rules that are eliminated from possibly matching because they do not match in this particular field. All of the the tries are walked based on the packet header information, leaving us with a set of vectors (one for each field). Each vector is the set of rules that have been eliminated, and if we take the logical OR of this set of vectors, we will be left with one vector that tells us which rules have not been eliminated yet, and thus, are matched.

There are two places where we can use wide word access to speed the

data structures involved. The first way is to speed the walking of the separate tries used for each of the fields. The wide word access can be used in a way similar to that described in Section V.E.2 and we do not discuss the tradeoffs again here.

The other advantage to be had from wide words is in operating on the long bit vectors themselves. Because there is one bit in the vector for each rule, and we need to examine over 2000 rules, and the bit vector OR operation requires many accesses to the memory. Note, our 2000 rule data set is similar to a rule set used by a major ISP. Using wide words here is a fairly simple matter of needing less separate words per vector when the vectors are longer that the word length. By using the wide word, we can operate on more of the bits at a time, and as such, we achieve a near linear increase in the bandwidth from an algorithmic standpoint.

Co-exploration: In Figure V.8 we can see that using the BVLS algorithm can support approximately 20 Gbit/s at 100 square millimeters. While this number is short of the goal of 40 Gbits/s we set for ourselves, there are several areas that can be improved in terms of the implementation of the algorithm. Our current implementation of the algorithm is limited primarily by the initial trie lookups, which is why the shape of the graph is similar to that for IP lookups. In addition, we can further improve these numbers by using wide words to target the initial trie lookups (currently not done), or by using an algorithm such a HiCuts [52].

V.E.3 Combined Analysis

Now that we have examined how each of the algorithms map to our full architecture, it makes sense to break down the effect of each of our contributions. We examine the best performance that can be obtained with and without the

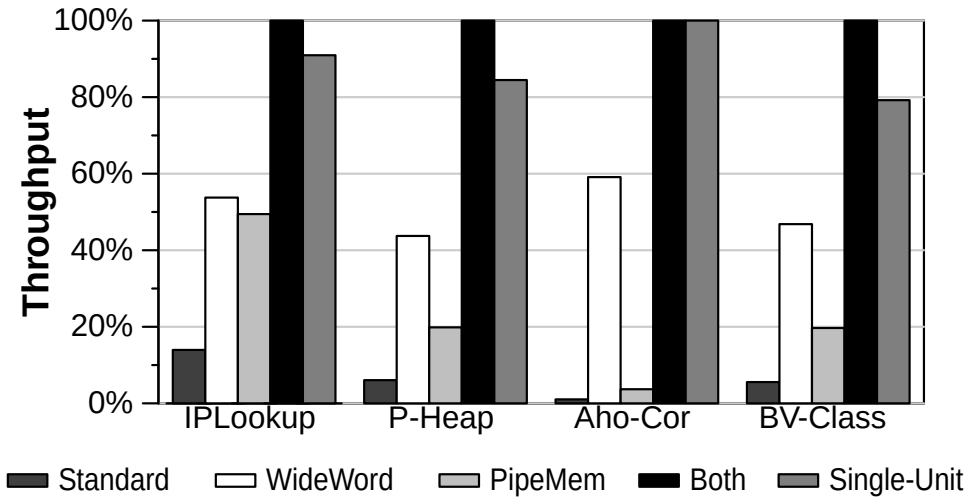

Figure V.11: Examination of the best configurations found under different assumptions.

The first 4 bars assume the architecture is tuned to a given architecture, while the last bar show what happens if a single architecture is tuned to the suite of applications. Results are normalized to the best configuration found for *each* algorithm.

presented techniques for a constrained area in order to show this breakdown. Figure V.11 was generated by taking a hard area limit, in this case 100 square millimeters, and by trying to find parameters of the algorithms and architecture such that total bandwidth supported is maximized, but the total area budget is not over-spent. Because of this, all these configurations have close to 100 square millimeters.

There are 5 bars on the graph. The first 4 are bars, where we view each algorithm independently (as was done in the previous sections), show results for tuning the architecture individually for each algorithm. The first bar, labeled *Standard*, is the base case where we assume a fairly simple algorithm that is limited to using 8-bytes of memory for each node, and where the memory access time is similar to a standard on-chip memory (with the decode logic separated from the actual read). While this base case may or may not be an accurate

representation of the state of the art, it does provide a point of comparison to show the importance of our other techniques.

The next bar on the graph shows the effect of having a memory system that has support for wide-word memory access, but it is not pipelined past the standard method. Note that, while this memory is not tiled into pipeline stages, internally the memory has been designed such that the internal delays are optimized using memory sub-banking to break up the otherwise exorbitantly long row and column lines. The third bar is the effect of just using a pipelined memory, without any use of wide words to deliver more parallelism or memory bandwidth.

The bar labeled *Both* is the most aggressive solution that we found for a given algorithm, but with the same constraint on the total area. Because this represents the best we could do using the techniques presented, and because the actual values for this can be seen in Figure V.8, we have chosen to normalize all results for each algorithm to this bar. This bar represents, for a given algorithm, the worst case throughput of the best configuration tailor made to that algorithm.

In addition to these first four bars, which are tuned to each application separately, it is important to note the performance of the *best overall* performing configuration. The final bar, labeled *single-unit*, shows what happens if we pick a single best configuration, and map all of the algorithms to that configuration. This means that the architecture is tuned to optimize for the suite of applications rather than to a specific individual. The tuning is done by picking the architecture that has the highest geometric mean across all of the benchmarks. The difference between this bar and the *Both* bar shows the amount we lose from using a single system as opposed to systems custom tailored to each application, which is surprisingly only a small amount. This presents the hope that a *single* implementation could be used to achieve high performance from all of the

applications we have examined.

V.F Summary

In this chapter we have examined the potential of exploiting tradeoffs at the algorithmic level with architectural design realities. In doing so, this chapter focused on creating a programmable and scalable network processor solution for next generation backbone routers to achieve 40+ Gb/s. The two traditional network processor models we discussed have issues with scaling. The first model uses independent processors with shared memory; the use of traditional memory designs optimized for latency limits the achievable memory throughput. The second model uses pipelined processors with partitioned memory to obtain high memory throughput. However, the second approach does not support writes and does not balance memory across partitions, both of which are important for router functions such as QoS and lookups.

To solve these issues we propose using a unified pipelined wide word memory. One of the contributions of this chapter is to achieve high memory throughput for a single, unified memory by internally pipelining the memory. This allows our design to process many transactions at the same time. In addition, the use of wide words allows our design to increase memory bandwidth, leveraging the fact that many network algorithms can be sped up using wide words.

To exploit this wide word architecture, a new methodology of co-exploration is needed so that the algorithm design can be tradedoff with design of the wide word pipelined memory architecture. We performed this co-exploration for several important network algorithms, which include IP lookup, QoS, Security, and Classification. We performed co-exploration by creating a parameterized version of these network algorithms and data structures that can exploit our wide word architecture. We then examined the tradeoffs between the different memory

configurations varying the memory bandwidth achieved, word width and overall memory size. Through this co-exploration, we showed that a programmable architecture can efficiently exploit behavior inherent to many common network algorithms. In particular, we found that a single set of parameters worked nearly optimally across all of the algorithms we examined, suggesting that these parameters could be used as the basis for a unified network processor design.

Working across two abstraction levels (algorithm and memory design) simultaneously allows us to explore a richer variety of tradeoffs and produce more efficient designs than if we were to explore each abstraction level separately, as done normally. Note that redesigning software every time a new architecture is designed would be an engineering nightmare in a conventional CPU design. However, in a core router design, the key forwarding algorithms that must be designed under real-time constraints (e.g., lookups) are only a small fraction of the overall software effort (programs to compute routes, management code, etc.), and are often redesigned to fit each new generation of hardware.

VI

Adaptive Architectures

VI.A Introduction

As has been shown in the earlier chapters, that design time customization can exploit application specific information for embedded application. But hardware customization is not the only way to take advantage of program behavior. Many times we would like to get the advantages of customization into high performance general-purpose (desktop/workstation/mainframe) computers where customization is not a realistic design choice. The techniques that are developed in this dissertation have broader application than just customization, and often we can use the techniques in an adaptive manner.

It is worth clarifying the distinction between adaptive and customized processors. A customized processor is one that has been *physically* altered in design before fabrication. In this way area savings can be achieved by make smaller structures that have equivalent, but specialized, functionality. Adaptive processors adjust to the application that is running *dynamically*.

Both require a set of techniques that take advantage of program behavior in processors, first by quantifying the program behavior and then by adjusting to its needs. In an adaptive processor we need to rely on run-time data gather-

ing techniques that use hardware profiling rather than static analysis, and then configure and tune the hardware without user or compiler interaction.

In this chapter, we present a set of techniques for gathering program phase information at run time. This is the first and most crucial step in developing adaptive versions of any customization optimizations as this sets the level of granularity on which optimization can be effectively applied. Modern processors can execute upwards of 5 billion instructions in a single second, yet most architectural features target program behavior on a time scale of hundreds to thousands of instructions, less than half a μS. While these optimizations can provide large benefits, they are limited in their ability to see the program behavior in a larger context.

Recently there has been a renewed interest in examining the run-time behavior of programs over longer periods of time [35, 60, 116, 121, 9]. It has been shown in Chapter II that programs can have considerably different behavior depending on which portion of execution is examined. More specifically, it has been shown that many programs execute as a series of phases, where each phase may be very different from the others, while still having a fairly homogeneous behavior within a phase. Taking advantage of this time varying behavior can lead to, among other things, improved power management, cache control, and more efficient simulation. The primary goal of this research is the development of a unified run-time phase detection and prediction mechanism that can be used to guide any optimization seeking to exploit large scale program behavior.

A phase of program behavior can be defined in several ways. Past definitions are built around the idea of a phase being an interval of execution during which a measured program metric is relatively stable. We extend this notion of a phase in Chapter II to include all similar sections of execution regardless of temporal adjacency. Simply put, if a phase of execution is correctly identified, there

should only be small variations between any two execution intervals identified as being part of the same phase. We have further shown that a phase behavior in any program metric is directly a function of the way the code is being executed. If we can accurately capture this behavior at run-time through the computation of a *single* metric, we can use this to guide many optimization and policy decisions without duplicating phase detection mechanisms for *each* optimization.

In this chapter, we present an efficient run-time phase tracking architecture that is based on detecting changes in the *proportions* of the code being executed. In addition, we present a novel phase prediction architecture that can predict, not only when a phase change is about to occur, but also the phase to which it is will transition. Since our phase tracking implementation is based upon code execution frequencies, it is independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique, building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics allows us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

We demonstrate the effectiveness of our hardware based phase detection and classification architecture at automatically partitioning the behavior of the program into homogeneous phases of execution and to identify phase changes. We show that the changes in many important metrics, such as IPC and energy, correlate very closely with the phase changes found by our metric. We then evaluate the effectiveness of phase tracking and prediction for value profiling, data cache reconfiguration, and re-configuring the width of the processor.

The rest of the chapter is laid out as follows. In Section VI.B, prior work related to phase-based program behavior is discussed. Simulation methodology and benchmark descriptions can be found in Section VI.C. Section VI.D

describes our phase tracking architecture. The design and evaluation of the phase predictor are found in Section VI.E. Section VI.F presents several potential applications of our phase tracking architecture. Finally, the results are summarized in Section VI.G.

VI.B Related Work

In this section, we describe work related to phase identification and phase-based optimization.

In [116], we provided an initial study into the time varying behavior of programs, showing that programs have repeatable phase-based behavior over many hardware metrics – cache behavior, branch prediction, value prediction, address prediction, IPC and RUU occupancy for all the SPEC 95 programs. Looking at these metrics over time, we found that many programs have repeating patterns, and that important metrics tend to change at the same time. These places where the metrics change represent phase boundaries.

In chapter II and in [121], we proposed that by profiling only the code that was executed over time we could automatically identify periodic and phase behavior in programs. The goal was to automatically find the repeating patterns observed in [116], and the lengths (periods) of these patterns. We then extended this work in chapter II (also published in [123]), using techniques from machine learning to break the complete execution of the program into phases (clusters) by only tracking the code executed. We found that intervals of execution grouped into the same phase had similar behavior across all the architecture metrics examined. From this analysis, we created a tool called SimPoint, which automatically identifies a small set of intervals of execution (simulation points) in a program to perform architecture simulations. These simulation points provide an accurate and efficient representation of the complete execution of the program.

The work of Dhodapkar and Smith [35, 34] is the most closely related to ours. They found a relationship between phases and instruction working sets, and that phase changes occur when the working set changes. They propose that by detecting phases and phase changes, multi-configuration units can be re-configured in response to these phase changes. They have used their working set analysis for instruction cache, data cache and branch predictor re-configuration to save energy [35, 34].

The work we present in this chapter identifies phases and phase changes by keeping track of the proportions in which the code was executed during an interval based upon the profiler used in [121]. In comparison, Dhodapkar and Smith [35, 34] track the phase and phase changes solely upon what code was executed (working set), without weighting the code by its frequency of execution. Future research is needed to compare these two approaches.

Additional differences between our work include our examination of architectures for predicting phase changes, and different uses from [35, 34], such as value profiling and processor width reconfiguration. We provide an architecture that can fairly accurately predict what the next phase will be, along with predicting when there will be a phase change. In comparison, Dhodapkar and Smith do not examine phase-based prediction [35, 34], but concentrate on detecting when the working set size changes, and then reactively apply optimization.

Merten et al. [86] developed a run-time system for dynamically optimizing frequently executed code. Then in [9], Barnes et al. extend this idea to perform phase-directed complier optimizations. The main idea is the creation of optimized code "packages" that are targeted towards a given phase, with the goal of execution staying within the package for that phase. Barnes et al. concentrate primarily on the compiler techniques needed to make phase-directed compiler optimizations a reality, and do not examine the mechanics of hardware phase

detection and classification. We believe that using the techniques in [9] in conjunction with our phase classification and prediction architecture will provide a powerful run-time execution environment.

VI.C Methodology

To perform our study, we collected information for ten SPEC 2000 programs `applu`, `apsi`, `art`, `bzip`, `facerec`, `galgel`, `gcc`, `gzip`, `mcf`, and `vpr` all with reference inputs. All programs were executed from start to completion using SimpleScalar [18] and Wattch [17]. Because of the lengthy simulation time incurred by executing all of the programs to completion, we chose to focus on only 10 programs. We chose the above 10 programs since their phase based behavior represents a reasonable snapshot of the SPEC 2000 benchmark suite, along with picking some of the programs that showed the most interesting phase-based behavior. Each program was compiled on a DEC Alpha AXP-21164 processor using the DEC C, and FORTRAN compilers. The programs were built under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`).

The timing simulator used was derived from the SimpleScalar 3.0 tool set [18], a suite of functional and timing simulation tools for the Alpha AXP ISA. The baseline microarchitecture model is detailed in Table VI.1. In addition to this, we wanted to examine energy usage optimizations, so we used a version of Wattch [17] to capture this information. We modified all of these tools to log and reset the statistics every 10 million instructions, and we use this as a base for evaluation.

I Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
D Cache	16k 4-way set-associative, 32 byte blocks, 1 cycle latency
L2 Cache	128K 8-way set-associative, 64 byte blocks, 12 cycle latency
Main Memory	120 cycle latency
Branch Pred	hybrid - 8-bit gshare w/ 2k 2-bit predictors + a 8k bimodal predictor
O-O-O Issue	out-of-order issue of up to 4 operations per cycle, 64 entry re-order buffer
Mem Disambig	load/store queue, loads may execute when all prior store addresses are known
Registers	32 integer, 32 floating point
Func Units	2-integer ALU, 2-load/store units, 1-FP adder, 1-integer MULT/DIV, 1-FP MULT/DIV
Virtual Mem	8K byte pages, 30 cycle fixed TLB miss latency after earlier-issued instructions complete

Table VI.1: Baseline Simulation Model for Phase Tracker.

VI.D Phase Capture

In this section we motivate the occurrence of phase-based behavior, describe our architecture for capturing it, and examine the accuracy of using the program behavior in our phase-tracking architecture to identify phase changes for various hardware metrics.

VI.D.1 Phase-Based Behavior

The goal of this research is to design an efficient and general purpose technique for capturing and predicting the run-time phase behavior of programs for the purpose of guiding any optimization seeking to exploit large scale program behavior. Figure VI.1 helps to motivate our approach to the problem. This figure shows the behavior of two programs, `gcc` and `gzip`, as measured by various different statistics over the course of their execution from start to finish. Each point on the graph is taken over 10 million instructions worth of execution. The metrics shown are the number of unified L2 cache misses (ul2), the energy con-

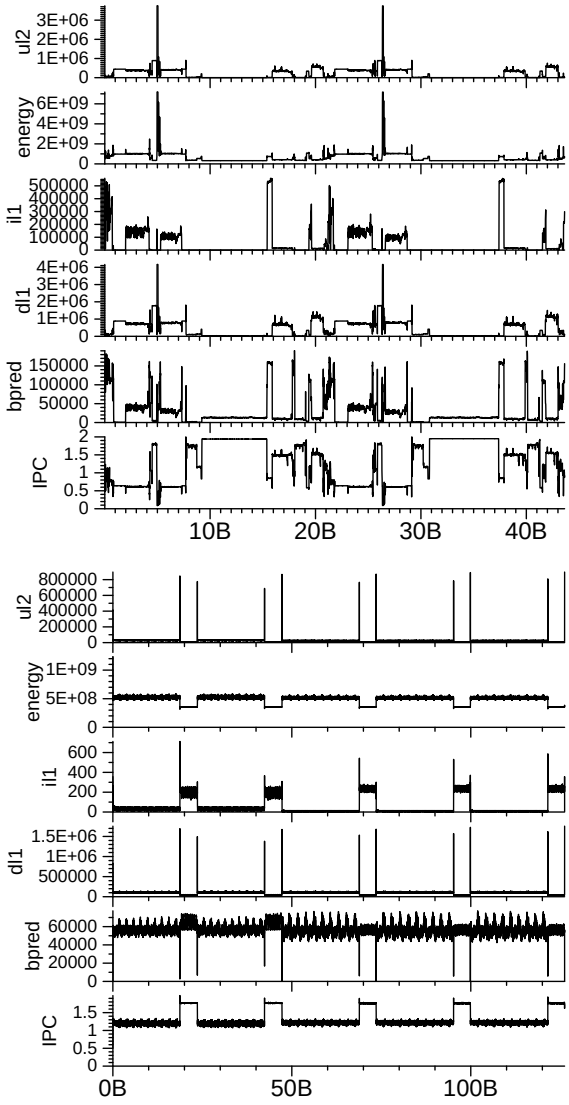

Figure VI.1: Application behavior changing over time

To illustrate the point that phase changes happen across many metrics all at the same time, we have plotted the value of these metrics over billions of instructions executed for the programs `gcc` (shown left) and `gzip` (shown right). Each point on the graph is an average over 10 million instructions. The number of unified L2 cache misses (ul2), the energy consumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC are plotted.

sumed by the execution of the instructions, the number of instruction cache (il1) misses, the number of data cache misses (dl1), the number of branch mispredictions (bpred) and the average IPC. The results show that all of the metrics tend to change in unison, although not necessarily in the same direction. In addition to this, patterns of recurring behavior can be seen over very large time scales.

As can be seen from these graphs, even at a granularity of 10 million instructions (which is at the same time scale as operating system time slices) there can be wildly different behavior seen between intervals. In this chapter, we concentrate on a granularity of 10 million instructions because it is both outside the scope of normal architectural timing and is small enough to allow for many complex phase behaviors to be seen.

VI.D.2 Tracking Phases by Executed Code

Our phase tracker architecture operates at two different time scales. It gathers profile information very quickly in order to keep up with processor speeds, while at the same time it compares any data it gathers with information collected over the long term. Additionally, it must be able to do all that while still being reasonable in size.

Our phase profile generation architecture can be seen in Figure VI.2. The key idea is to capture basic block information during execution, while not relying on any compiler support. Larger basic blocks need to be weighed more heavily as they account for a more significant portion of the execution. To *approximate* gathering basic block information, we capture branch PCs and the number of instructions executed between branches. The input to the architecture is a tuple of information: a branch identifier (PC) and the number of instructions since the last branch PC was executed. This allows us to roughly capture each basic block executed along with the weight of the basic block in terms of the number

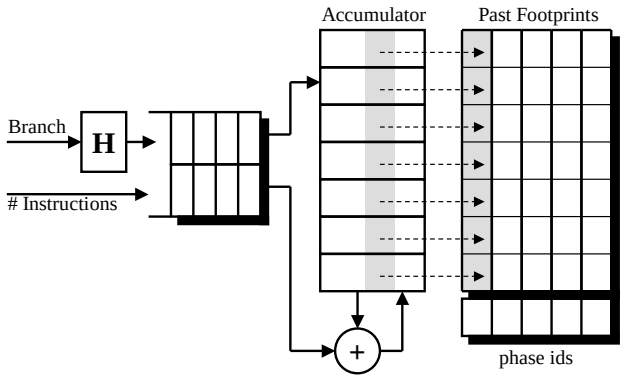

Figure VI.2: Phase classification architecture.

Each branch PC is captured along with the number of instructions from the last branch. The bucket entry corresponding to a hash of the branch PC is incremented by the number of instructions. After each profiling interval has completed, this information is classified, and if it is found to be unique enough, stored in the past footprint table along with its phase ID.

of instructions executed, as we did in [121, 123] for identifying simulation points.

Classifying phases by examining only the code that is executed allows our phase tracker to be independent of any individual architecture metric. This allows our phase tracker to be used as a general profiling technique building up a profile or database of architecture information on a per phase basis to be used later for hardware or software optimization. Independence from architecture metrics is also very important to allow us to consistently track phase information as the program's behavior changes due to phase-based optimizations.

At this point it is worth making more explicit the differences between our technique and that of Dhodapkar and Smith [35, 34]. Dhodapkar and Smith use a bit vector to track the working set of the code for a particular interval. While our technique is based on the basic block vectors used in [121]. The bit vectors of Dhodapkar and Smith track a metric that is related to which code blocks were *touched*, whereas our metric tracks the *proportion* of time spent executing in each code block. This is a subtle but important distinction. We have

found that in complex programs (such as `gcc` and `gzip`) there are many instructions blocks that execute only intermittently. When tracking the pure working set, these infrequently executed blocks can disguise the frequently executed blocks that dominate the behavior of the application. On the other hand, by tracking the frequency of code execution it is possible to distinguish important instructions (basic blocks) from a sea of infrequently executed ones. Examining these differences in more detail is a topic of future research.

Another advantage of tracking the proportions in which the basic blocks are executed is that we can use this information to identify not only when different sections of code are executing, but also when those sections of code are being exercised differently. A simple example is in a graphics manipulation program running a parameterized filter on an input image. If you run a simple 3x3 blur filter on an image you get very different behavior than if you run a 7x7 filter on the same image despite the fact that the same filter code is executing. The 7x7 filter will have many more memory references and those memory references conflict very differently in the cache than in the 3x3 case. We have seen this very behavior in examining the interactive graphics program `xv`. Using the proportion of execution for each basic block can distinguish these differences, because in the 3x3 filter the head of the loop is called more than twice as frequently as in the 7x7 filter.

The same general idea applies to other data structures as well. Take for example a linked list. As the number of nodes in the linked list traversal changes over different loop invocations, the number of instructions executed inside the loop versus the time spent outside the loop also changes. This behavior can be captured when including a measure of the proportion of the code executed, and this can distinguish between link list traversals of different lengths.

VI.D.3 Capturing the Code Profile

To index into the accumulator table in Figure VI.2, the branch PC is reduced to a number from 1 to *Nbuckets* using a hash function. We have found that 32 buckets is sufficient to distinguish between different phases even for some of the more complex programs such as `gcc`. A counter is kept for each bucket, and the counter is incremented by the number of instructions from the last branch to the current branch being processed. Each accumulator table entry is a large (in this study 24-bit), saturating counter, which will not saturate during our profiling interval of 10 million instructions. Updating the accumulator table is the only operation that needs to be performed at a rate equivalent to the processor's execution of the program (once for every branch executed). In comparison, the phase classification described below needs to only be performed once every 10 million instructions (at the end of each interval), and thus is not nearly as performance critical.

We note that the hashing function we use is fundamentally the same as the random projection method we used to generate phases in Chapter II. In this prior work, we make use of random projections of the data to reduce the dimensionality of the samples being taken. A random projection takes trace data in the form of a matrix of size $L \times B$, where L is the length of the trace and B is the number of unique basic blocks, and multiplies it by a random matrix of size $B \times N$, where N is the desired dimensionality of the data which is much smaller than B. This creates a new matrix of size $L \times N$, which has clustering properties very similar to the original data. The random projection method is a powerful technique when used with clustering algorithms, and for capturing phase behavior as we showed in Chapter II. The hashing scheme we use in this chapter is essentially a degenerate form of random projection that makes a hardware implementation feasible while still having low error. If all the elements of the

random projection matrix consist of either a 0 or a 1, and they are placed such that no column of the matrix contains more than a single 1, then the random projection is identical to this simple hashing mechanism. We have designed our phase classification architecture around this principle.

Figure VI.3 shows the effect of applying the above mentioned technique for capturing the phase behavior of the integer benchmark `gzip`. The x-axis of the figure is in billions of instructions, as is the case in Figure VI.1. Each point on the y-axis represents an entry of the phase tracker's accumulator table. Each point on the graph corresponds to the value of the corresponding accumulator table entry at the end of a profiling interval. Dark values represent high execution frequency, while light values correspond to low frequency. The same trends that were seen in Figure VI.1 for `gzip` can be clearly seen in Figure VI.3. In both of these figures, when observing them at the coarsest granularity, we can see that there are at least three different phases labeled `A`, `B` and `C`. In Figure VI.3, the phase tracker table entries `2, 5, 7, 13 and 17` distinguish the two identical long running phases labeled `A` from a group of three long running phases labeled `C`. Phase table entries `12 and 20` clearly distinguish phase `B` from both `A` and `C`. This figure is pictorial evidence that the phase tracker is able to break the program's execution into the corresponding phases based solely on the executed code, and that these phases correspond to the behavior seen across the different program metrics in Figure VI.1.

VI.D.4 Forming a Footprint

After the profiling interval has elapsed, and branch block information has been accumulated, the phase must then be classified. To do this we keep a history of past phase information.

If we fix the number of instructions for a profiling interval, then we can

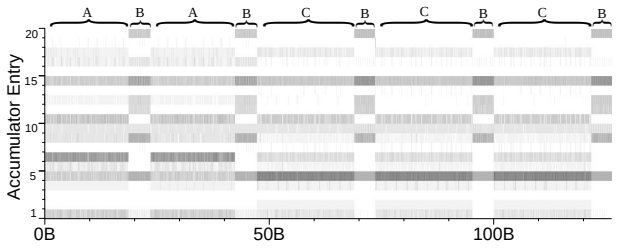

Figure VI.3: Accumulator table occupancy over time

This is a visualization of the accumulator table used to track program behavior for `gzip`. The x-axis is in billions of instructions, while the y-axis is the entry of the accumulator table. Each point on the graph corresponds to the value of the accumulator table at the end of a profiling interval where dark values correspond to more heavily accessed entries. The same trends that were seen in Figure VI.1 can be clearly seen in Figure VI.3.

divide each bucket by this fixed number to get the percentage of execution that was accounted for by all instructions mapped to that bucket. However, we do not need to know the exact percentages for each bucket. Instead of keeping the full counter values, we can instead compress phase information down to a couple of the most significant bits. This compressed information will then be kept in the Past Footprint table as shown in Figure VI.2.

The number of counter value bits that we need to observe is related to *Nbuckets*. As we increase the number of buckets, the data is spread over more buckets (table entries), making for less entries per bucket (better resolution) but at the cost of more area (both in terms of number of buckets and more bits per bucket). To be on the safe side, we would like *any* distribution of data into buckets to provide useful information. To achieve this we need to ensure that even if data is distributed perfectly evenly over all of the buckets, we would still record information about the frequency of those buckets. This can be achieved by reducing the accumulator counter by:

$$(bucket[i] \times Nbuckets)/(intervalsize)$$

If the number of buckets and interval size are powers of two, this is a simple shift operation. For the number of buckets we have chosen (32), and the interval size we profile over, this reduces the bucket size to 6 bits, and thus requires 24 bytes of storage for each unique phase in the Past Footprint table. In practice we see that the top 6 bits of the counter are more than enough to distinguish between two phases. In the worst case, you may need one or two more bits to reduce quantization error, but in reality we have not seen any programs that cause this to be an issue.

If too few buckets are used, aliasing effects can occur due to the hashing function, where two different phases will appear to have very similar Footprints. Therefore, we want to use a sufficiently large number of buckets to uniquely identify the differences in code execution between phases, while at the same time use only a small amount of area.

To examine the aliasing effect and determine what the appropriate number of buckets should be, Figure VI.4 shows the sum of the differences in the bucket weights found between all sequential intervals of execution. The y-axis shows the sum total of differences for each program. This is calculated by summing the differences between the buckets captured for interval i and $i-1$ for each interval i in the program. The x-axis is the number of distinct buckets used. All of the results are compared to the ideal case of using an infinite number of buckets (or one for each separate basic block) to create the Footprint. On the program `gcc` for example, the total sum of differences with 32 buckets was 72% of that captured with an infinite number of buckets. In general we have found that 32 buckets was enough to distinguish between two phases.

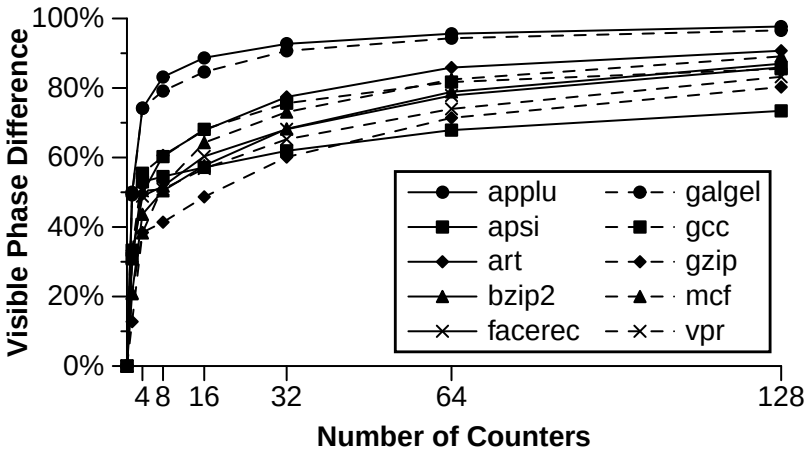

Figure VI.4: Determining an adequate number of counters

This is the percent difference found between Footprints from sequential intervals of execution, when varying the number of counters used to represent the footprints. The results are normalized to the difference between intervals found when having an infinite number of buckets to represent the footprint; 32 buckets captures most of the benefit.

VI.D.5 Classifying a Footprint to a Phase ID

After reducing the vector to form a footprint, we begin the classification process by comparing the footprint to a set of representative past footprint vectors. We compare the current vector to each vector in the table. The next section details how we perform the comparison and determine what a match is. If there is a match, we classify the profiled section of execution into the same phase as the past footprint vector, and the current vector is not inserted into the past footprint table. If there is no match, then we have just detected a new phase and hence must create a new unique phase ID into which we may classify it. This is done by choosing a unique phase ID out of a fixed pool of IDs. When allocating a new phase ID, we also allocate a new past footprint entry, set it to the current vector, and store with that entry the newly allocated phase ID. This allows future similar phases to be classified with the same ID. In this way only

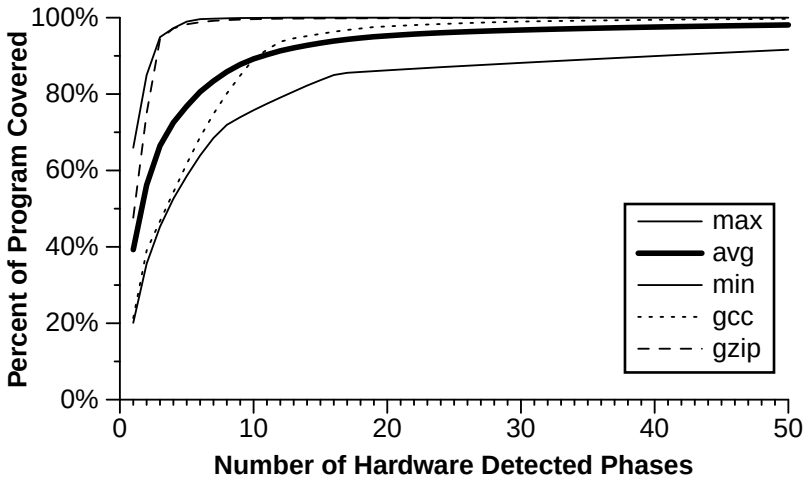

Figure VI.5: Phase coverage

Results of the minimum number of phases that need to be captured versus the amount program execution they cover. The y-axis is the percent of program execution that is covered. The x-axis is the minimum number of phases needed to capture that much program execution.

a single vector is kept for each unique phase ID, to serve as a representative of that phase. After a phase ID is provided for the most recent interval, it is passed along to prediction and statistic logging, and the phase identification part of our algorithm is completed.

To examine the number of phase IDs we need to track, Figure VI.5 shows the percentage of execution that can be accounted for by the top p phases, where p is shown on the x-axis. Results are graphed for the programs that had the min (`galgel`) and max (`art`) coverage, `gcc`, `gzip`, and the overall average. These results show that most of the program's phase behavior can be captured using a relatively small number of phase IDs. If we only track and optimize for the top 20 phases in each application, we will capture and be able to accurately apply phase prediction/optimizations to over 90% of the program's execution on average. In the worst case (min), we are able to optimize most of the program (over 80%) by only targeting a small number (20) of important recurring phases.

Finding a Match

We search through the Footprint histories to find a match, but this query is complicated by the fact that we are not necessarily searching for an exact match. Two sections of execution that have very similar footprints could easily be considered a match, even if they do not compare exactly. To compare two vectors to one another, we use the Manhattan distance between the two, which is the element-wise sum of the absolute differences. This distance is used to determine if the current interval should be classified as the same phase ID as one of the past footprint intervals.

If we set the distance threshold too low, the phase detection will be overly sensitive, and we will classify the program into many, very tiny phases which will cause us to lose any benefit from doing run-time phase analysis in the first place. If the threshold is too high, the classifier will not be able to distinguish between phases with different behavior. To quantify this effect, we examine how well our hardware technique classifies phases for a variety of thresholds compared to the phases found by the off-line clustering algorithm used in SimPoint [123].

The SimPoint tool is able to make global decisions to optimize the grouping of similar intervals into phases. The off-line algorithm makes no use of thresholds, instead its decisions are based solely on the structure found in the distribution of program behaviors. Our technique must be far more simplistic because it must be performed on-line and with limited computational overhead. This reduction in complexity comes at the cost of increased error.

The Different Phases line in Figure VI.6 shows the ability of our hardware technique to find phase changes (transitions between one phase and the next) when different thresholds are used to perform the phase classification. For example, when using a Manhattan distance of 1 million as our threshold (shown as 20 on our x-axis because it is in $\log_2$), our hardware technique identified 80% of

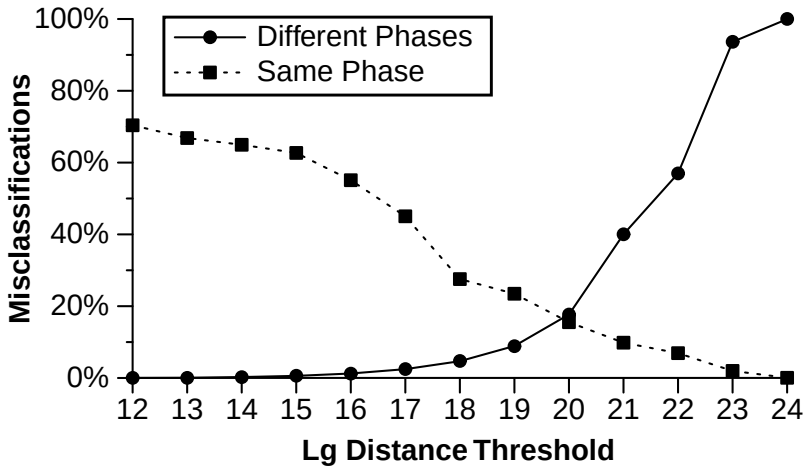

Figure VI.6: Sensitivty versus Accuracy in the phase tracker

Results showing how well our hardware phase tracker classifies two sequential intervals of execution as being from "Different" or the "Same" phase of execution. The percent of misclassifications are shown in comparison to the phase classifications found using the off-line clustering SimPoint tool [123].

the phase changes that occurred in the more complex off-line SimPoint analysis. Conversely, 20% of the phase changes were incorrectly classified as having the same phase ID as the last interval of execution.

Likewise, the Same Phases line in Figure VI.6 represents the ability of our hardware technique to accurately classify two sequential intervals as being part of the same phase as a function of different thresholds (again as compared to the off-line clustering analysis). For example, when using a Manhattan distance of 1 million (shown as 20 on the x-axis), our hardware technique identified 80% of the intervals that stayed in the same phase as correctly staying in the same phase, but 20% of those intervals were classified as having a different phase ID from the prior phase.

A misclassification occurs when two sequential intervals of execution are classified as being in the same phase or in different phases using our hardware approach when the off-line clustering analysis tool found the opposite for these

two intervals.

If we are too aggressive and our hardware phase analysis indicates that there are phase changes when there are actually no noticeable changes in behavior, then we will create too many phase IDs that have similar behavior. This can create more overhead for performing phase-based optimization. On the other hand, if we are too passive in distinguishing between different phases, we will be missing opportunities to make phase specific optimizations.

In order to strike a balance between having a high capture rate and reducing the percent of false positives, we chose to use a threshold of 1 million. When comparing this with the interval size of 10 million instructions, this means that a difference in the phase behavior will be detected if 10% of the executed instructions are in different proportions. In choosing 1 million, we have on average a 20% misclassification rate. Note, that a misclassification does not necessarily mean that an incorrect optimization will be performed. For example, if we have a "Same Phase" misclassification (the two intervals were really from the same phase, but were classified into different phases), then a phase change is observed using our hardware technique when there was not one in the baseline classifier. If the two hardware detected phases have the same optimization applied to them, then this misclassification can have no effect.

VI.D.6 Per-Phase Performance Metric Homogeneity

Using the techniques presented above, we can perform phase classification on programs at run-time with little to no impact on the design of the processor core. One of the goals of phase classification is to divide the program into a set of phases that are fairly homogeneous. This means that an optimization adapted and applied to a single segment of execution from one phase, will apply equally well to the other parts of the phase. In order to quantify the extent to

	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gcc	full	1.32	(43.4%)	27741	(135.5%)	445083	(110.7%)	50763	(203.2%)	6.44E+08	(90.0%)	227912	(139.7%)
	18.5%	0.61	(1.6%)	34665	(22.0%)	753382	(5.4%)	125091	(23.2%)	1.03E+09	(1.8%)	395997	(5.3%)
	18.1%	1.95	(0.3%)	13048	(3.9%)	28112	(15.1%)	43	(73.9%)	3.22E+08	(0.2%)	1006	(5.6%)
	7.2%	0.64	(0.2%)	843	(15.1%)	885081	(0.1%)	75	(215.5%)	9.78E+08	(0.3%)	443655	(0.1%)
	4.0%	1.49	(1.2%)	10145	(7.6%)	703554	(6.8%)	15591	(5.2%)	4.20E+08	(1.1%)	354084	(7.0%)
	3.9%	1.76	(1.6%)	2015	(13.6%)	98947	(5.9%)	102	(45.1%)	3.57E+08	(1.6%)	15595	(12.6%)
	phase	IPC	(stddev)	bpred	(stddev)	dl1	(stddev)	il1	(stddev)	energy	(stddev)	ul2	(stddev)
gzip	full	1.33	(16.3%)	56045	(11.1%)	90446	(58.2%)	60	(138.1%)	4.82E+08	(13.5%)	22880	(112.0%)
	17.1%	1.24	(3.4%)	53300	(10.8%)	96960	(10.1%)	12	(44.2%)	5.05E+08	(3.5%)	24218	(8.6%)
	9.4%	1.23	(3.8%)	54973	(11.5%)	99523	(11.3%)	11	(45.5%)	5.09E+08	(3.8%)	24518	(9.3%)
	8.8%	1.76	(0.6%)	56449	(4.8%)	37331	(5.6%)	241	(8.4%)	3.55E+08	(0.6%)	5617	(15.6%)
	8.0%	1.22	(4.3%)	54791	(6.8%)	99671	(11.9%)	40	(25.7%)	5.14E+08	(4.4%)	28153	(11.0%)
	7.4%	1.24	(3.1%)	55215	(11.1%)	96701	(9.6%)	12	(35.4%)	5.04E+08	(3.2%)	23701	(8.4%)

Table VI.2: Phase Homogeneity for `gzip` and `gcc`

Examination of per-phase homogeneity compared to the program as a whole (denoted by `full`). For the two programs and each of the top 5 phases of each program, we show the average value of each metric and the standard deviation. The name of the phase is the percent of execution that it accounts for in terms of instructions. These results show that after dividing up the program into phases using our run-time scheme the behavior within each phase is quite consistent.

which we have achieved this goal, we need to test the homogeneity of a variety of architectural statistics on a per-phase basis.

Figure VI.2 shows the results of performing this analysis on the phases determined at run-time. Due to space constraints we only show results for two of the more complicated programs `gcc` and `gzip`. For both programs, a set of statistics for each phase is shown. The first phase that is listed (separated from the rest) as `full`, is the result of classifying the entire program into a single phase. The results show that for `gcc` for example, the average IPC of the entire program was 1.32, while the average number of cache misses was 445,083 per ten million instructions. In addition to just the average value, we also show the standard deviation for that statistic. For example, while the average IPC was 1.32 for `gcc`, it varied with a standard deviation of over 43% from interval to interval. If the phase-tracking hardware is successful in classifying the phases, the standard deviations for the various metrics should be low for a given phase ID.

Underneath the phase marked `full` are the five most frequently executed

phases from the program as identified by our phase tracker. The phases are weighted by the percentage of the program's executed instructions they account for. For `gcc`, the largest phase accounts for 18.5% of the instructions in the entire program and has an average IPC of 0.61 and a standard deviation of only 1.6% (of 0.61). The other top four phases have standard deviations at or below this level, which means that our technique was successful at dividing up the execution of `gcc` into large phases with similar execution behavior with respect to IPC. Note, that some metrics for certain phases have a high standard deviation, but this occurs for architecture features/metrics that are unimportant for that phase. For example, the phase that occurs for 7.2% of execution in `gcc` has only 75 L1 instruction cache misses on average. This is an L1 miss rate of 0.00075%, so an error of 215% for this metric will not likely have any effect on the phase.

When we look at the energy consumption of `gcc`, it can be observed that energy consumption swings radically (a standard deviation of 90%) over the complete execution of the program. This can be seen visually in Figure VI.1, which plots the energy usage versus instructions executed. However, after dividing the program into phases, we see that each phase has very little variation within itself. All have less than 2% standard deviation. By analyzing `gcc` it can also be seen that the phase partitioning does a very good job across all of the measured statistics even though only *one* metric is used. This indicates that the phases that we have chosen are in some way representative of the actual behavior of the program.

VI.E Phase Prediction

The prior section described our phase tracking architecture, and how it can be used to classify phases. In this section we focus on using phase information to predict the next phase. For a variety of applications it is important to be able

to predict future phase changes so that the system can configure for the code it will soon be executing rather than simply reacting to a change in behavior.

Figure VI.7 shows the percentage interval transitions that are changes in phase, for our set of benchmarks. For all of these programs, phase changes come quite often, but it should be noted that this statistic alone cannot gauge the complexity of the program behavior. The program `gcc` switches less than 10% of the time, but switches between *many* different phases. The other extreme is `art` which switches almost half the time, but it is only switching between a few distinct phases. In this case, large repeating patterns can be observed. No two phases executing sequentially are that similar, but there is an order to the sequence. By adding in a prediction scheme for these cases, we not only take advantage of stable conditions as in past research, but actually take advantage of any repeating patterns in program behavior.

VI.E.1 Markov Predictor

The prediction of phase behavior is different from many other systems in which hardware predictors are used. Because of this new environment, a new type of predictor has the potential to perform better than simply using predictors from other areas of computer architecture (branch and address prediction, memory disambiguation, etc.).

After observing the way that phases change, we determined that two pieces of information are important. First, the set of phases leading up to the prediction are very important, and second, the *duration* of execution of those phases is important.

A classic prediction model that is easily implementable in hardware is a Markov Model. Markov Models have been used in computer architecture to predict both prefetch addresses [68] and branches [24] in the past. The basic idea

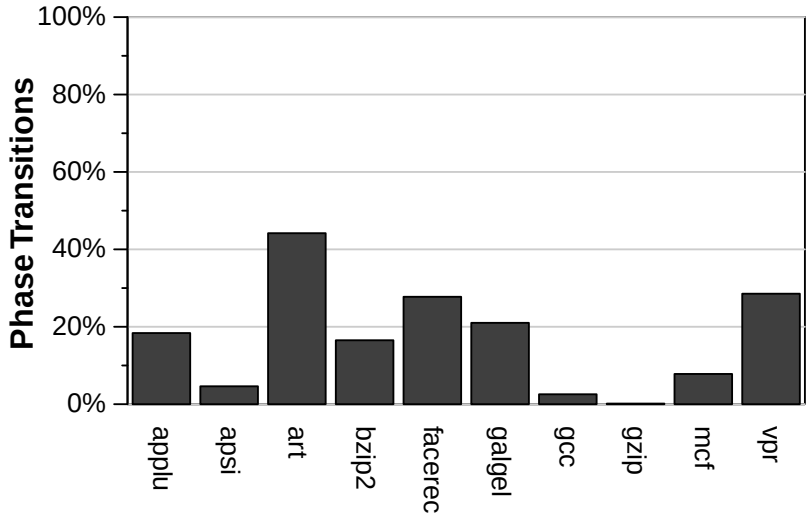

Figure VI.7: Phase transition activity

The percent of execution intervals that transition to a different phase from the prior execution interval's phase as found by our phase tracking architecture with 32 footprint counters using a 1 million Manhattan threshold.

behind a Markov Model is that the next state of the system is related to the last set of states.

The intuition behind this design is that phase information tends to be characterized by many sections of stable behavior interspersed with abrupt phase changes. The key is to be able to predict when these phase changes will occur, and to know ahead of time what phase they will change to. The problem is that the changes are often preceded by stable conditions, and if we only consider the last couple of intervals we will not be able to tell the difference between sections of stable behavior that precede a phase change, and those sections that will continue to be stable. Instead, we need a way of compressing down stable phase information into a piece of information that we can use as state.

VI.E.2 Run Length Encoding Markov Predictor

To compress the stable state we use a *Run Length Encoding* (RLE) Markov predictor. The basic idea behind the predictor is that it uses a run-length encoded version of the history to index into a prediction table. The index into the prediction table is a hash of the phase identifier and the number of times the phase identifier has occurred in a row.

Figure VI.8 shows our RLE Markov Phase ID prediction architecture. The the lower order bits of the hash function provide an index into the prediction table, and the higher order bits of the hash function provide a tag. When there is a tag match, the phase ID stored in the table provides a prediction as to the next phase to occur in execution. When there is a tag miss, the prior phase ID is assumed to be the next phase ID to occur in the program's execution. We found that predicting the last phase ID to be 75% accurate on average.

We only update the predictor when there is (1) a change in the phase ID, or (2) when there is a tag match. We only insert an entry when there is a phase ID change, since we want to predict when the phase is going to change. Execution intervals where the same phase ID occurs several times in a row do not need to be stored in the table, since they will be correctly predicted as "last phase ID", when the there is a table miss. This helps table capacity constraints and avoids polluting the table with last phase predictions. For the second update case, when there is a tag match, we update the predictor because the observed run length may have potentially changed.

VI.E.3 Predictor Comparison

We compare our RLE Markov phase predictor with other prediction schemes in Figure VI.9. This Figure has four bars for every program, and each bar corresponds to the prediction accuracy of a prediction architecture. The first

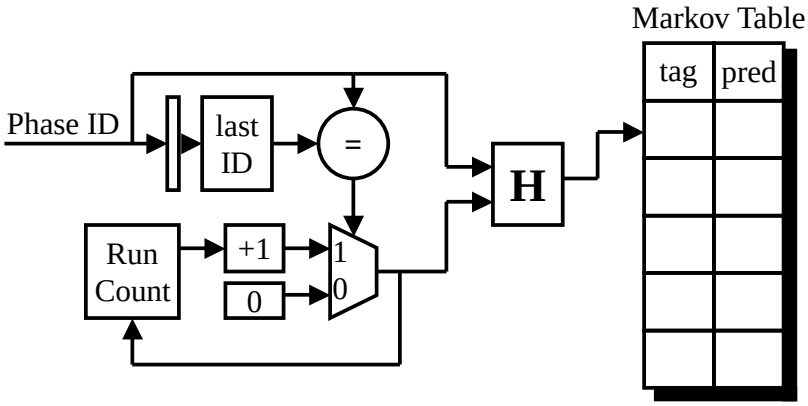

Figure VI.8: Phase prediction architecture

Phase Prediction Architecture for the Run Length Encoded (RLE) Markov predictor. The basic idea behind the predictor is that two pieces of information are used to generate the prediction, the phase id that was just seen, and the number of times prior to now that it has been seen in a row. The index into the prediction table is a hash of these two numbers.

and simplest scheme, `Last Phase`, simply predicts that the next phase is the same as the current phase, in essence always predicting stable operation. The prediction accuracy of this scheme is inversely proportional to the rate at which phases change in a given benchmark. For the program `gzip` for example, there are long periods of execution where the phase does not change, and therefore predicting no-change does exceptionally well.

In order to insure that we were not simply providing an expensive filter for noise in the phase IDs, we also compared against a simple noise filter which works by predicting that the next phase will be the most commonly occurring of the last three phases seen. This is not shown, as it actually performed worse on all of the programs.

Additionally we wanted to examine the effect of a simple Markov model predictor for history lengths of 1 and 2. The Markov model predictor does a better job of predicting phase transitions than `Last Phase`, but it is limited by the fact that long runs will always be predicted as infinitely stable due to the

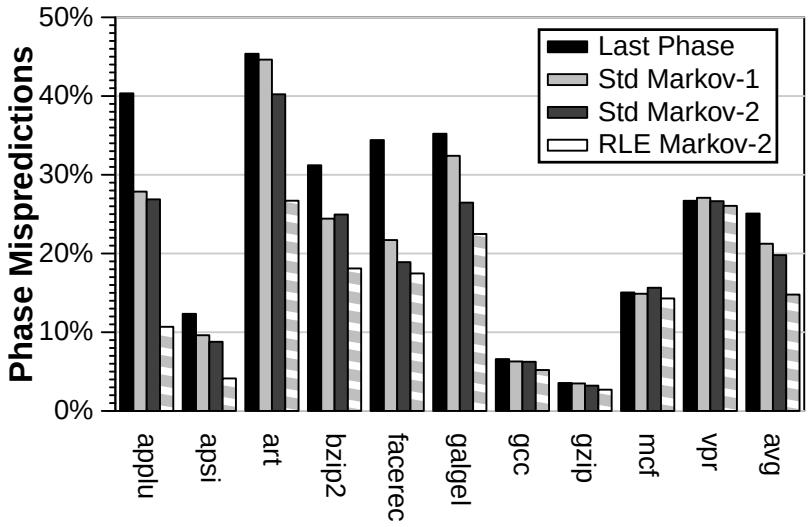

Figure VI.9: Phase ID Prediction Accuracy.

This figure shows how well different prediction schemes work. The most naive scheme, `last`, simply predicts that the phases never change. The bars marked `Markov` and `RLE Markov` show how well we can predict the phase identifiers if we use a Markov prediction scheme with a Markov table size of 256 entries.

history filling up. However, it is still very effective for `facerec` and `applu`, but does not provide much benefit for either `art` or `galgel`.

The final bar, `RLE Markov`, is our improved Markov predictor which compresses stable phases into a tuple of phase id and duration. All of the Markov predictors simulated had 256 entries taking up less than 500 bytes of storage. Using RLE Markov outperforms both the `Last Phase` and traditional `Markov` on all the benchmarks. It performs especially well compared to other schemes on both `applu` and `art`. Overall, using a Run-Length Encoded Markov predictor can cut the phase mispredictions down to 14% on average.

VI.F Applications

This section examines three optimization areas in which a phase-aware architecture can provide an advantage. We begin by examining the relationship between phase behavior and value locality. We then demonstrate ways to reduce processor energy consumption by adjusting the aggressiveness of the data cache and the instruction front end.

VI.F.1 Frequent Value Locality

Prior work on value predictors has shown that there is a great deal of value locality in a variety of programs [79, 22]. Recently, researchers have started to take advantage of frequently loaded values for the purpose of optimizing caches. For example, Yang and Gupta [148] proposed a data cache organization that compresses the most frequently used program values in order to save energy. Another way of exploiting value locality is through value specialization, which can be done either statically or dynamically [20, 93, 87] to create specialized versions of procedures or code-regions based upon the values frequently seen. These techniques are built on the idea of finding the most frequent values for loads over the whole program, and then specializing the program to those frequent values.

We examine the potential of capturing frequent values on a per-phase basis and compare this to the frequent values aggregated over the entire program, as would be used in value code specialization [20]. To perform this experiment we first gathered the top 16 values that were loaded over the complete execution of the program and stored them into a table. We then examined the percentage of executed loads that found their loaded value in this table. This result is shown as `Static` in Figure VI.F.1. While significant portions of some programs are covered by just these few top values (such as `applu`), over half of the programs

have less than 10% of their loaded values covered by these top values.

The question is: can we do better by exploiting hardware-detected phase information? To answer this question we take the top 16 values for each phase, as detected by the hardware phase tracker. These top values will be shared across a single phase even if it is split into two or more different sections of execution. Each load in the program is then checked against the top values for its corresponding phase. The `Phase Coverage` bar in Figure VI.F.1 shows the percent of all load values in the program that were successfully matched to it's per-phase top value set.

Without any notion of loads or values, our method of dividing up phases is very successful at assisting in the search for frequent values. By just tracking the top 16 values of each phase, we are able to capture the values from almost 50% of the executed loads on average. The `Perfect` bar shows percentage of loads covered if one captures the top 16 load values for each and every interval (i.e., 10 million instructions) separately. This is in effect the best that we could hope to achieve for an interval size of 10 million instructions, because the 16 entries in the value table are custom crafted for each interval individually. As shown in Figure VI.F.1, the phase-tracker compares favorably with the optimal coverage. Two thirds of the total possible benefit from per-interval value locality can be captured by per-phase value locality. It is important to point out that this graph by itself is not a good indicator of usefulness as near perfect coverage could be achieved simply by making every interval a separate phase. However, as shown in Figure VI.5 only a few phases (around 20) are used to cover at least 80% of the program's execution.

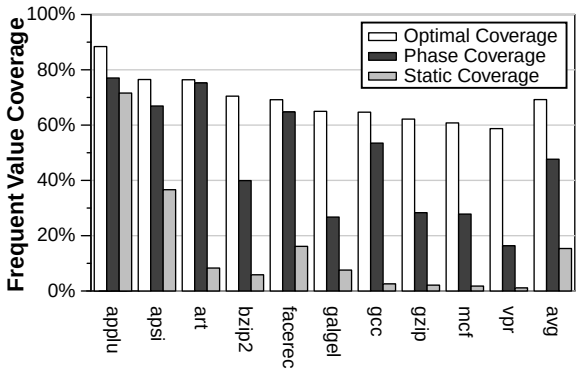

Figure VI.10: Phase aware value locality study

The percent of the program's load values that are found in a table of the most frequently values loaded over the whole program (Static Coverage), on a per-phase basis (Phase Coverage), and on a per execution interval basis (Optimal Coverage).

VI.F.2 Dynamic Data Cache Size Adaptation

In a modern processor a significant amount of energy is consumed by the data cache, but this energy may not be put to good use if an application is not accessing large amounts of data with high locality. To address this potential inefficiency, previous work has examined the potential of dynamically reconfiguring the data caches with the intention of saving power. In [8], Balasubramonian et. al. present two different schemes with which re-configuration may be guided. In one scheme, hardware performance counters are read by re-configuration software every hundred thousand cycles. The software then makes a decision based on the values of the counters. In another scheme, re-configuration decisions are performed on procedure boundaries instead of at fixed intervals. To reduce the overhead of re-configuration, software to trigger re-configuration is only placed before procedures that account for more than a certain percentage of execution.

Another form of re-configurable cache that has been proposed dynamically divides the data cache into multiple partitions, each of which can be used for

a different function such as instruction reuse buffers, value predictors, etc [102]. These techniques can be triggered at different points in program execution including procedure boundaries and fixed intervals. The overhead of re-configuration can be quite large and making these policy decisions only when the large scale program behavior changes, as indicated by phase shifts in our hardware tracker, can minimize overhead while guaranteeing adequate sensitivity to attain maximum benefit.

We examined the use of phase tracking hardware to guide an energy aware, re-sizable cache. The energy consumption of the data cache can be reduced by dynamically shifting to a smaller, less associative cache configuration for program phases that do not benefit significantly from more aggressive cache configurations. By targeting only those phases that are predicted to have energy savings due to cache size reduction, our scheme is able to reduce power with very little impact on the performance.

We examined an architecture with two possible cache configurations, 32KB 4-way associative and 8KB direct mapped. In Figure VI.F.2, the trade off between these two configurations is plotted. For each program, we use the 32KB cache configuration as the baseline result. The labeled circles in Figure VI.F.2 show the total processor energy savings and performance degradation for each program if only the smaller (8KB) cache size is used. For example, a processor with a smaller cache configuration for the program `applu` is both 5% slower and uses 5% less energy. Two programs, `vpr` and `apsi`, actually use more energy with a smaller cache due to large slow downs. These two points are off the scale of this graph and are not shown.

While examining energy savings and slow down is interesting, it is important to note that there is more than one way to reduce both energy and performance. Voltage scaling in particular has proven to be a technology capable

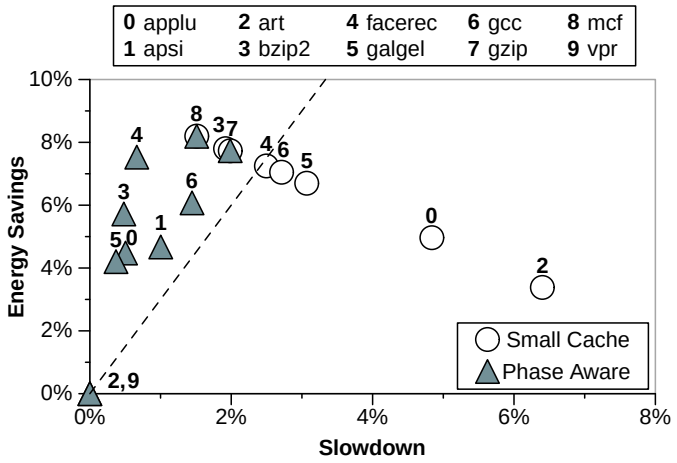

Figure VI.11: Data cache reconfiguration

The tradeoff between energy savings and slowdown for two different cache policies. All results are relative to a 32KB 4-way associative cache. The circles in the graph (each labeled with a number for the program the data point is from) show the energy and performance of an 8KB direct mapped cache. The triangles show the tradeoff of intelligently switching between an 8KB direct mapped and a 32KB 4-way data cache based on phase classification and prediction.

of reaping large energy savings for a relative reduction in performance. For our results, we assume that for voltage scaling a performance degradation of 5% will yield an approximate energy saving of 15%. We use this rule of thumb as our guideline for determining when to reduce the active size of the cache. In Figure VI.F.2, this simple model of voltage scaling is plotted as a dashed line. When the cache size is reduced, most programs fall far short of this baseline, meaning that voltage scaling would provide a better performance-energy tradeoff. There are a couple of exceptions, in particular `mcf`, `bzip`, and `gzip` do well even without any sort of phase-based re-configuration.

The shaded triangles in Figure VI.F.2 show what happens if we use phase classification and prediction to guide our re-configuration. When a new phase ID is seen, we sample the IPC and energy used for a few intervals using the 32KB 4-way cache, and a few intervals for the 8KB direct mapped cache. These

samples could be kept in a small hardware profiling table associated with the phase ID. After taking these samples, if we find that a particular phase is able to achieve more than three times the energy savings relative to the slow down seen when using the 8KB cache, we then predict for this phase ID that the smaller cache size should be used. This heuristic means that the small cache size is used only if re-configuration would beat voltage scaling for that phase. After a decision has been made as to the configuration to use for a phase ID, the corresponding cache size is stored in the phase profiling table/database associated with that phase ID. The phase classifier and predictor are then used to predict when a phase change occurs. When a phase change prediction occurs, the predicted phase ID looks up the cache size in the profiling table, and re-configures the cache (if it is not already that size) at the predicted phase change.

For all programs, our re-configuration is able to beat or tie voltage scaling. For example, using phase-based re-configuration results in a slowdown of 0.5% for `applu`, while the total energy savings is 4.5%. Even the program `apsi`, which had *increased* energy consumption in the small cache configuration, is able to get almost 5% energy savings with only a 1% slowdown.

VI.F.3 Dynamic Processor Width Adaptation

One way to reduce the energy consumption in a processor is to reduce the number of instructions entering the pipeline every cycle [63, 6]. We call this adjusting the width of the processor. Reducing the width of the processor reduces the demand on the fetch, decode, functional units, and issue logic. Certain phases can have a high degree of instruction level parallelism, whereas other phases have a very low degree. Take for example the top two phases for `gcc` shown in Figure VI.2. The intervals classified to be in the first phase consisting of 18.5% of execution have an IPC of 0.61 with a high data cache miss rate. In comparison,

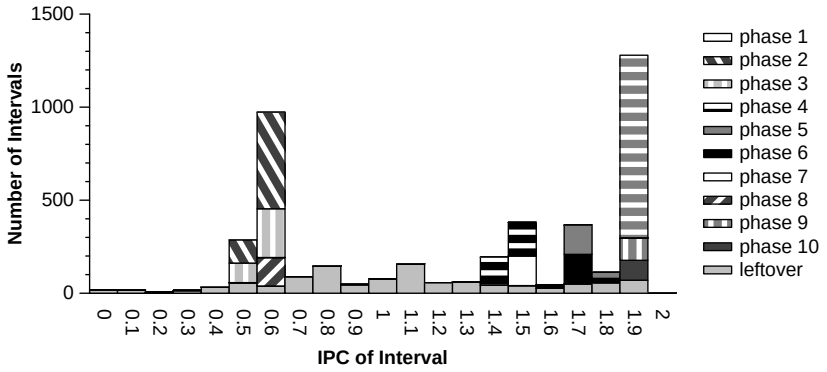

Figure VI.12: The IPC of `gcc` as split up by the phase detector.

the intervals in the second most frequently encountered phase (accounting for 18.1% of execution) have an IPC of 1.95 and very low data cache miss rates. We can potentially save energy without hurting performance by throttling back the width of the processor for phases that have low IPC, while still using aggressive widths for phases with high IPC. This idea can be seen more clearly in Figure VI.F.3. Figure VI.F.3 shows for `gcc` a histogram of IPC values for all the intervals in the run of the program. It further shows how the top 10 phases divide up that histogram. You can see that the phase tracker is able to find phases with similar IPC even though there it does not use any information about the performance of the executing program.

In the current literature, decisions to reduce or increase the fetch / decode / issue bandwidth of the processor are made either at fixed intervals (relatively short intervals such as 1,000 cycles) [63] or, as in the case of branch confidence based schemes, when a branch instruction is fetched [6]. It can very difficult to design real systems that save energy by reconfiguring at these speeds, but a hardware phase-tracker can help make these decisions at a coarser granularity while still maintaining performance and energy benefits.

We examined an architecture that could be configured with 2 different widths - one where up to 2 instructions are decoded and up to 2 issued per cycle,

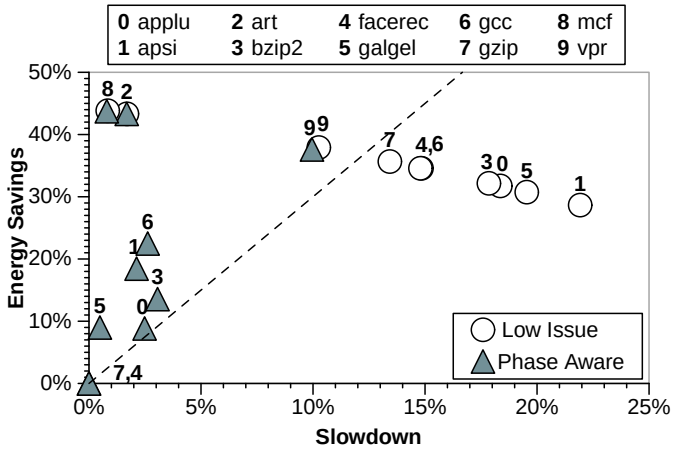

Figure VI.13: Processor Width Adaptation.

The tradeoff between energy savings and slowdown for two different front end policies. All results are relative to an aggressive 8–issue machine. The circles in the graph (each labeled with a number for the program) show the energy and performance of a less aggressive 2–issue processor. The triangles show using the phase classifier and predictor for switching between 2–issue and 8–issue based on phase changes.

and one where up to 8 instructions are decoded and up to 8 issued per cycle. When a new phase ID is seen by the phase tracker, we sample the IPC for three intervals with a width of 2 instructions, and three intervals with a width of 8 instructions. If there is little difference in the IPC between these two widths, then we assign a width of 2 instructions to this Phase ID in our profiling table, otherwise we assign a width of 8 instructions. During execution, we use the phase ID predictor to effectively predict the best width for the next interval of execution (based on past samples) and adjust the processor's width accordingly. Our results show that the chosen configuration for a given phase can be trained (1) with only a few samples, and (2) only once to accurately represent the behavior of a given phase ID. This requires very little training time due to the fact that 20 or fewer phase IDs are needed to capture 80% or more of a program's execution as shown in Figure VI.5.

Figure VI.F.3 is a graph of the results seen when applying phase-directed width re-configuration. The white circles in the graph show the behavior of running the programs on only a 2–wide machine relative to the more aggressive 8–wide machine. The dotted line again shows what could potentially be achieved if voltage scaling was used. While `mcf` and `art` save a lot of energy with little performance degradation on a 2–wide machine, the other programs do not fair as well. The program `apsi`, for example, has a slowdown of over 22% with an energy savings of around 30%. This does not compare favorably to voltage scaling (as discussed in Section VI.F.2). On the other hand, if we use phase-directed width throttling on `apsi`, a total processor energy savings of 18% can be achieved with only 2.2% slowdown.

For all of the programs we examined, with one exception, the slowdown due to phase aware width throttling was less than 4%, while the average energy savings was 19.6%. This result demonstrates that there is significant benefit to be had in the re-configuration of processor front end resources even at very large granularities. In the worst case, this will mean a re-configuration every 10 million instructions, and on average every 70 million instructions. This should be designable even under conservative assumptions.

VI.G Summary

In this chapter we present an efficient run-time phase tracking architecture that is based on detecting changes in the code being executed. This is accomplished by dividing up all instructions seen into a set of buckets based on branch PCs. This way we approximate the effect of taking a random projection of the basic block vector, which was shown in Chapter II to be an effective method of identifying phases in programs.

Using our phase classification architecture with less than 500 bytes of

on-chip memory, we show that for most programs, a significant amount of the program (over 80%) is covered by 20 or less distinct phases. Furthermore, we show that these phases, while being distinct from one another, have fairly uniform behavior within a phase, meaning that most optimizations applied to one phase will work well on all intervals in that phase. In the program `gcc`, the IPC attained by the processor on average over the full run of execution is 1.32, but has a standard deviation of more than 43%. By dividing it up into different phases, we achieve much more stable behavior, with IPCs ranging between 0.61 and 1.95, but now with standard deviations of less than 2%.

In addition to this, we present a novel phase prediction architecture using a Run Length Encoding Markov predictor that can predict not only when a phase change is about to occur, but to which phase ID it will transition to. In using this design, which also uses less than 500 bytes of storage, we achieve a phase prediction miss rate of 10% for `applu` and 4% for `apsi`. In comparison, always predicting that the phase will stay the same results in a miss rate of 40% and 12% respectively.

We also examined using our phase tracking and prediction architecture to enable new phase-directed optimizations. Traditional architecture and software optimizations are targeted at the average or aggregate behavior of a program. In comparison, phase-directed optimizations aim at optimizing a program's performance tailored to the different phases in a program. In this chapter, we examined using phase tracking and prediction to increase frequent value profiling coverage, and to provide energy savings through data cache and processor width re-configuration.

We believe our phase tracking and prediction design will open the door for a new class of run-time optimization that targets large scale program behavior. Even though we present a hardware implementation for phase tracking, a similar

design can be implemented in software to perform phase classification for run-time optimizers, just-in-time compilation systems, and operating systems. Hardware and software optimizations that can potentially benefit the most from phase classification and prediction are (1) those that need expensive profiling/training before applying an optimization, (2) those where the time or cost it takes to perform the optimization is either slow or expensive, and (3) those that can benefit from specialization where they have the same code/data being used differently in different phases of execution. By using our dynamic phase tracking and prediction design, phase-behavior can be characterized and predicted at the largest of scales, providing a unified mechanism for phase-directed optimization.

VII

Summary and Future Work

Programs can have very different performance and resource needs, and by tailoring a processor architecture to the task at hand we can increase performance, reduce power consumption, and improve area efficiency. This tailoring can be done at design-time by customizing the physical design of the processor, or it can even be done adaptively at run-time. Because of the complexity of the applications we would like to run, and because analyzing their behavior on a given processor can be difficult, we need a set of automated techniques that can take a processor design template and then customize it to the application that is running on it.

VII.A In This Disseration

In this dissertation, we provide several such novel architectural techniques based on methods from algorithms, machine learning, and automaton theory. In this chapter we will reiterate some of the important contributions from each chapter and discuss the algorithms involved. In addition, in this chapter we will try to place the results in a larger context by discussing some of the future work that can and has been built upon these ideas and techniques.

VII.A.1 Phase Behavior in Programs

As was pointed out in chapter II, the understanding of program behavior is one of the most important aspects of computer architecture and program optimization. It used to be the case that the performance of a processor running an application could be estimated with relative ease, but with the increasing complexity of modern designs this is no longer the case. For example, in the simple case of a single issue machine with blocking caches, you can estimate the performance by counting the number of instructions executed and the number of cache misses. This type of simplistic analytic model is no longer an accurate way of measuring performance because there are now many components, each with a complex behavior all acting in unison. For example, to keep performance increasing at the desired rate we now need extremely long pipelines (on the order of 40 stages), prefetching, branch prediction, trace caches, and multiple levels of cache. Trying to model the behavior of all of these structures requires very careful simulation of a running program's interaction with the underlying hardware.

The problem is that many programs have wildly different behavior on even the largest of scales. This greatly complicates the steps of profiling and simulation which are crucial to both research in processor customization and general purpose computer architecture research. On todays processors, the detailed simulation of a single minute of execution requires on the order of 50 days of simulation time. This number is already outside the scope of reason for any interactive results that can be used for guiding design exploration, and it will grow in future generations of processor. This is because while processors are getting faster (which means both more instructions per second and faster simulators) they are also getting more complex at an exponential rate (which slows down simulators significantly).

Chapter II presents automatic techniques that are capable of finding

and exploiting large-scale program behavior for the purpose of faster simulation. More techniques, such as these will be needed in the future, and in fact there has already been significant adaptation of the presented techniques within the architecture community. The program phase analysis in chapter II is based on a reduction to a known problem in machine learning, clustering. This reduction has been shown to be both very powerful and robust, but there is still room for future work in this new area.

The most natural set of extensions to this work are improvements to the algorithms that are used to generate the simulation points for this work. The reduction in chapter II is an approximation to a more complete and complex description of the problem. The main idea of the extension is in the fact that the clustering method used in chapter II is based on the idea that each time slice is independent from it's neighbors in terms of it's performance. This means that any *sequence* data, or information about the ordering in which the intervals occur, is not used in the determination of the simulation points. This information could be used in our simulation methodology to pick larger *contiguous* sections of execution for simulation, or it could be used to build up a hierarchy of phases changes, representing both small-scale changes and large-scale changes under a single unified method of description. In order to achieve this we must consider our input data, not as a set of discrete points, but rather as a set of paths where a patch connects points/intervals that are adjacent in time. Clusters of paths are then needed to represent the data.

Another extension to this work is expanding it to work in multi-threaded and multi-processor environments. This is a very difficult problem, and a solution to this does not fall cleanly out of this phase analysis work. However, phase analysis may still prove to be a useful tool. The problem is that in a multi-threaded environment, the performance of the machine is not only a function of the code

that is executing from a single thread, but it is rather multiple independent sections of code working together. Because of this there is a complex feedback involved. The performance of the system on each thread at time t determines what code is executed at time $t + 1$, which in turm determines the performance at time $t + 1$. One possibility may be to have a hybrid approach which mixes analytic modeling based on phase analysis with measurements gathered through detailed simulation.

VII.A.2 Automatic FSM predictor design

In chapter III new algorithms for the design of FSM predictors are proposed. Finite state machines (FSM) are one of the fundamental building blocks in the design and optimization of computer systems and they are often used in the control of prediction and speculation. For example, the classical 2-bit saturating up-down counters in a standard branch predictor design are simple state machines with the sole purpose of predicting a particular branch pattern. These simple saturating counters are also used in confidence estimation, as accuracy counters, and in cache replacement schemes.

The power of the tools from chapter III is that a hardware design for a customized branch predictor can be produced directly from application profiles without the need for user interaction. The state machines are built using efficient non-randomized algorithms from combinatorial optimization, statistical modeling, and regular expressions, in addition to some novel optimization algorithms based on observed behaviors. These state machines can then be built into the hardware of the processor to help with particularly troublesome branches.

In addition to being a powerful tool for generating state machines for inclusion in hardware designs, the techniques in chapter III provide a new way of analyzing and thinking about program behavior at the small scale (on the order

of instructions rather than on the large scale as in chapter II). One typical way to approach a problem in computer architecture is to try a bunch of different schemes, see which ones work, and from the results of that hypothesize about the underlying mechanism that is making some of the schemes work. While this can be a fast way to achieve results it has the downside that very few problems in computer architecture have been formalized to the extent that they can be reasoned about outside of the immediate context of the problem. One of the contributions of chapter III is to show that it *is* possible to be both formal and useful at the same time. The formulation of the problem in chapter III is still dependent on the craft and intuition of the designer to use the state machines effectively, but at the same time leverages the idea of design automation to make automatic customization a realistic design option.

In addition to a proof of the usefulness of algorithmic analysis in computer architecture, the actual tools developed in chapter III provide a new way of reasoning about branches and sequence prediction in processor design. The state machines generated from a particular sequence provide a new way to visualize the behavior of a given branch, set of branches, load confidence mechanism, or any other behavior that results in a stream of binary decisions. One natural extension to this work is to modify the use of state machines for non-binary decisions, for example in a prefetcher. A prefetcher needs to decide not only when to prefetch but also where to prefetch from. The design methodology could possibly be extended to capture this sort of decision as well. Another improvement that could be made to the state machine design methodology is to take into account the cost of implementing a particular path of the sequence earlier in the design process.

VII.A.3 The tradeoff of different components

In chapter IV we explore the effect of customizing portions of the processor on the performance of the processor as a whole. With the ability to customize portions of a chip comes the problem of how to combine them in ways that are the most beneficial to the end user or designer. Without a way of judging the tradeoffs between different components it is impossible to gauge the benefit of any optimization done on a specific subset of the chip unless it's effect on the system as a whole is examined.

The approach we used is to decompose the overall problem into a set of mostly linear sub-problems that can be traded off of one another with integer-linear programming. While integer programming is a computationally hard problem, we get around this by decomposing the problem into a problem that is for the most part linear-programming. The integer variables are only used to control curves with concavities. Because this occurs only infrequently (once or twice per component) the problem can be solved in on the order of seconds on a modern processor. The power of this system is that by calculating good tradeoffs, the system blurs the line between performance and area optimizations allowing designers to rapidly see the effect of decisions and optimizations on the design space.

This type of analysis is currently lacking from most embedded architecture research. As the embedded market continues to grow we expect to see that the techniques presented in chapter IV will be of growing importance. There is a growing gap between cycle level simulation and high level design decisions because the cost of simulation is growing so quickly. In chapter II we show how this can be reduced, but the two problems are in many ways different. In chapter II a faster way of validating the performance of a processor on a particular benchmark is presented, but even this takes on the order of an hour per configuration.

In order to help a designer make informed choices we need a performance model that can be used interactively. The techniques in chapter IV start along this path by providing such a model in a somewhat restricted architectural space (the class of in-order machines).

The future of this work is in the automatic creation of high-level yet accurate performance models. In chapter IV the performance model is automatically generated from both intuition about the behavior of the processor, a hand built parameterized model, and a series of experiments that help to set the parameters of the model to most accurately represent the behavior of the processor. In order for this work to achieve wide spread adaptation, these steps must be automated. What is needed is a more complex timing model that can still be fit to data using statistical techniques. The complicating factor is that the model must be *extensible* to be able to handle new additions to the machine. Building a model of a modern processor that is extensible, accurate, and fast has applications far outside just computer architecture research. For instance, compiler writers need a way to determine if an optimization should be applied or not and for this the current best approach is to simply execute the optimized code at run-time, and measure it's performance. Another example is in supercomputing where the code itself can be changed during a run to increase performance, but how to best change the code often requires trial and error. By building on this work, a unified architecture model may be within the grasp of computer architecture researchers, and we consider it to be one of the most important problems in systems research today.

VII.A.4 Architecture-Algorithm Co-exploration

In chapter V the idea of customized processor design exploration is expanded to take into consideration not only architectural changes, but algorithmic

ones as well. The algorithms and implementations used can have very different hardware resource needs, and by taking this into consideration we can effectively balance the needs of the software and the hardware. In chapter V this idea is explored in the context of network processors.

Designing ASICs for each new generation of backbone routers is a time intensive and fiscally draining process. In chapter V we focus on the design of a programmable architecture for backbone routers, based on the manipulation of wide irregular memory words, that can provide a feasible design alternative to custom ASICs. We propose a pipelined memory design that emphasizes worst-case throughput over latency, and co-explore architectural tradeoffs with the design of several important network algorithms. Through this co-exploration, we show that a programmable architecture can efficiently exploit behavior inherent to most common network algorithms to keep up with next generation network speeds.

The network processor techniques described in chapter V are based in algorithms as in the previous chapters. The difference is that the algorithms here are being run on a processor, rather than run to design a processor. However, many of the same ideas apply. For example, the Aho-Corasick string searching algorithm ported to our network processor has parallels with the finite state machine design used in chapter III. Both are based on the idea of the precomputation of a graph based representation of a set of input patterns, and both take as input a sequence of characters at run-time and find matches online.

While the network processor research has flowed out of the processor customization research in this dissertation, it has opened up a new door to be explored by future work. There are many open questions left here. Is a wide word architecture really the best solution? Is there a more efficient way of mapping a string matching algorithm to a wide word architecture? What are the

power ramifications of our design choices? What additional current and future applications will map onto our architecture? These are just some of the questions to be answered in the near future about our network processor design. Farther out, there are questions about what else we can use our processor design for. Can we build a processor along the same lines that is built for network security applications or intrusion detection? Is it possible to build a processor that has worst case bounds on throughput, but can still efficiently capture such complex problems?

Even further out there is a question of the applicability of algorithm and architecture co-exploration to other domain areas. For example, with the encapsulation of data structures becoming more and more common, is it possible to design architectures that are specialized to a given set of data structures? For example, can we design a processor that is specialized to the associative arrays of Perl or Python? It may be possible to change the way that people attack there problems if that is the case.

VII.A.5 Processor Adaptation

Design time customization works very well for application-specific processors, but many times we would like to be able to get the benefits of customization without requiring specialized hardware. In these cases it may be possible to adapt the processor to the application at run-time. Many of the techniques that we have developed in this dissertation for customization are applicable to adaptive systems.

We can take advantage of program behavior in processors, first by quantifying the program behavior and then tailoring to its needs. This is true of both the static and dynamic case. The difference between the two is that in the adaptive model we need to rely on run-time data gathering techniques that use

hardware profiling rather than static analysis, and then configure and tune the hardware without user or compiler interaction. The centerpiece to any dynamic optimization is an accurate model of program behavior that can be generated online. In chapter II we show how such a model can be built by analyzing a program and input set pair statically. In chapter VI we show how we can approximate the same information dynamically by adding only small amounts of hardware to the processor. This small amount of hardware can detect the phase changes in programs automatically by tracking only the code that is being exercised at any given interval of time. This is the first and most crucial step in developing any adaptive versions of optimizations based on customization.

The algorithms in chapter VI are based on approximations of the more complex cluster algorithms presented in chapter II. For example, instead of requiring the k-means clustering algorithm, which is an iterative process, we instead approximate this with an online algorithm which has been trained on the more complex version. Even with these simplifications we found that the phase analysis performed by the online technique still enabled many new optimizations such as further value locality and cache power reduction.

Phase analysis opens up a whole new arena of research in the optimization of programs on the large-scale. Without the phase analysis it is very difficult to perform any expensive optimizations because the local history available at the architectural level is very limited. With phase analysis hardware in place new optimizations can be applied, and this will be an active area of future work. There are two areas of future work for phase analysis. The first is in making phase analysis systems more accurate and robust so that they can deal with realistic computing systems. For example some of the assumptions made in the original work may not hold true in a real environment. We assume that during the run, only a single process has access to the machine. However a real com-

puter processor may be context switching between many different threads. This may further complicate the analysis as two or more thread may be competing for resources. There is no reason why phase analysis should not be able to detect context switches, and then use this information to further optimize the execution of individual threads. In addition, the phase analysis could be made more robust by providing feedback online. This would entail testing at run-time whether the phase analysis is doing a good job at dividing up the program into homogeneous sections. Currently this is not done because of the scale of the simulation involved. In order to enable this work we need an infrastructure which is capable of analyzing the phases of programs running on real hardware.

The other type of future work that stems from the phase analysis is other uses of small code monitoring assistance. Including phase analysis hardware is a form of introspective computing and it may be possible to extend this methodology to include other forms of introspection such as anomaly detection. Currently in the security community anomaly detection is based on the idea of detecting when the behavior of a program is doing something that has not been seen before. If we are able to develop a model of a running server at run-time, we could then analyze it to see if this behavior changes drastically from the norm using phase analysis hardware. Phase analysis hardware could additionally be used to guide software or compiler optimizations. For example the compiler may tag each section of code with the phase identifier that it could use to determine if the executing program is similar to the profile that was used during optimization. In fact, it may be able to support all of these ideas with a single peice of hardware.

VII.B Algorithms and Computer Architecture

I would like to finish this dissertation with some thoughts on the future of my research and computer architecture. Throughout this dissertation, and my graduate work as a whole, I've made use techniques from algorithms, machine learning, theory, and statistical modeling on problems from computer architecture. We began with the application of clustering, a technique for machine learning, on modeling program phase behavior. We then showed how to use algorithms from the theory of regular expressions to model branch predictors and conference estimators. This is followed by integer-linear programming solution to design space exploration, and algorithm-architecture co-exploration for network processors. Outside of this dissertation I have shown how to use dynamic programming to solve the problem of finding minimal patches for architectures with fixed instruction ROMs [119], graph coloring approximations to find a low conflict page placement [120], Markov models to improve prefetching accuracy [125], and bloom filters to enhance value profiling [94]. Almost all of these contributions have foundations in algorithms with applications in architecture.

Computer architecture is a very applied field. It is easy to become seduced by the quick pay off of fast results by a quick hack. But for computer architecture to advance further as a field we need to temper the desire for immediate results with the understanding that can be brought by formalization. Our research over the past five years has convinced us that there is great potential in the algorithmic treatment of problems from computer architecture. The formalization of problems in computer architecture does not mean that we should move away from being highly applied. Any formalization should be rigorously checked against reality, but this does not prevent it from being useful. Indeed, the abstraction required by formalization frees the researcher to consider new possibilities which may not be apparent while buried under the details of a fully

complex system.

In addition to the future work which have I have laid out in the above sections, I hope to further explore the interaction between algorithms and architecture. This is still a very young area of research and it is beginning to gain momentum. Following some of the work presented in this dissertation, other researchers have proposed program models based on the Fourier Transform and Wavelets. Other researchers have applied techniques from control systems to the problems of power management at the architectural level. All of these ideas point to a renaissance in computer architecture, a reawakening to the ideas generated from other fields. This is truly an exciting time to be a computer architect and I hope to have some small part in helping to shape its future.

Bibliography

[1] Malicious code attacks had $13.2 billion economic impact in 2001. Computer Economics, Jan 2002.

[2] S. Abraham, B. Rau, R. Schreiber, G. Snider, and M. Schlansker. Efficient design space exploration in pico. In *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 71–79, San Jose, California, November 2000.

[3] S. G. Abraham and S. A. Mahlke. Automatic and efficient evaluation of memory hierarchies for embedded systems. In *32nd International Symposium on Microarchitecture*, 1999.

[4] A. Agarwal, M. Horowitz, and J. Hennessy. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2):184–215, 1989.

[5] A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[6] J.L. Aragon, J. Gonzalez, and A. Gonzalez. Power-aware control speculation through selective throttling. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture*, February 2003.

[7] ARC. Whitepaper: Customizing a soft microprocessor core. http://www.arccores.com, 2001.

[8] R. Balasubramonian, D. H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *33rd International Symposium on Microarchitecture*, pages 245–257, 2000.

[9] R. D. Barnes, E. M. Nystrom, M. C. Merten, and W. W. Hwu. Vacuum packing: Extracting hardware-detected program phases for post-link optimization. In *35th International Symposium on Microarchitecture*, December 2002.

[10] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th Annual International Symposium on Computer Architecture*, Vancouver, Canada, June 2000.

[11] A. Ben-Dor, R. Shamir, and Z. Yakhini. Clustering gene expression patterns. *Journal of Computational Biology*, 6:281–297, 1999.

[12] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[13] M. Berkelaar. lp solve: a mixed integer linear program solver. ftp://ftp.es.ele.tue.nl/pub/lp_solve, September 1997.

[14] R. Bhagwan and B. Lin. Fast and scalable priority queue architecture for high-speed network switches. In *Proc. of IEEE Infocomm*, March 2000.

[15] C. M. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[16] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):761–772, 1977.

[17] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[18] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[19] M. Burtscher and B.G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *Journal of Instruction-Level Parallelism*, 1, 1999.

[20] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[21] B. Calder and G. Reinman. A comparative survery of load speculation architectures. *Journal of Instruction Level Parallelism*, 2, 2000.

[22] B. Calder, G. Reinman, and D.M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–74, June 1999.

[23] CERT/CC. Code Red worm exploiting buffer overflow in IIS indexing service DLL. CERT Advisory CA-2001-19, Jan 2002.

[24] I.-C. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 128–137, October 1996.

[25] I.-C. Chen, J.T. Coffey, and T.N. Mudge. Analysis of branch prediction via data compression. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[26] T-F. Chen and J-L. Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 5(44):609–623, May 1995.

[27] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 International Conference on Computer Design (ICCD)*, October 1996.

[28] Standard Performance Evaluation Corporation. Spec cpu2000. 2000.

[29] P. Crowley and J. L. Baer. A modeling framework for network processor systems. In *in Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[30] P. Crowley, M. E. Fiuczynski, J. L. Baer, and B. N. Bershad. Workloads for programmable network interfaces. In *IEEE 2nd Annual Workshop on Workload Characterization*, Austin, Texas, October 1999.

[31] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pages 143–151, San Francisco, CA, 2000. Morgan Kaufmann Publishers.

[32] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *Proceedings of SIGCOMM*, pages 3–14, 1997.

[33] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. In *Proc. SIGCOMM '89*, September 1989.

[34] A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, February 2002.

[35] A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *29th Annual International Symposium on Computer Architecture*, May 2002.

[36] Z. D. Dittia, G. M. Parulkar, and J. R. Cox. The APIC approach to high performance network interface design: Protected DMA and other techniques. In *Proc. of IEEE INFOCOM '97*, April 1997.

[37] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *31st International Symposium on Microarchitecture*, December 1998.

[38] R. J. Eickemeyer and S. Vassiliadis. A load instruction unit for pipelined processors. *IBM Journal of Research and Development*, 37:547–564, July 1993.

[39] J. Emer and N. Gloy. A language for describing predictors and its application to automatic synthesis. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[40] G. Ezer. Xtensa with user defined dsp coprocessor microarchitectures. In *Proceedings of the International Conference on Computer Design, 2000 (ICCD2000)*, pages 335–342, September 2000.

[41] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoli, and F. Homewood. Lx: a technology platform for customizable vliw embedded processing. In *27th Annual International Symposium on Computer Architecture*, pages 203–213, 2000.

[42] J. A. Fisher. Customized instruction-sets for embedded processors. In *Proceedings of the Design Automation Conference, 1999*, pages 253–257, 1999.

[43] J. A. Fisher, P. Faraboschi, and G. Desoli. Custom-fit processors: Letting applications define architectures. In *29th International Symposium on Microarchitecture*, pages 324–335, December 1996.

[44] M. J. Flynn and R. I. Winner. Asic microprocessors. In *23th International Symposium on Microarchitecture*, pages 237–243, 1990.

[45] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, November 1996.

[46] T. Givargis and F. Vahid. Platune: A tuning framework for system-on-a-chip platforms. *IEEE Transactions on Computer Aided Design*, 21(11), November 2002.

[47] T. Givargis, F. Vahid, and J. Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *International Conference on Computer Aided Design*, November 2001.

[48] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[49] R. E. Gonzalez. Xtensa: A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, March-April 2000.

[50] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[51] P. Gupta and N. McKeown. Algorithms for packet classification. *IEEE Network Magazine*, 15(2), 2001.

[52] Pankaj Gupta and Nick McKeown. Packet classification using hierarchical intelligent cuttings. In *Proceedings of Hot Interconnects VII*, August 1999.

[53] L. Gwennap. $1 billion and still counting. EETimes 1/4 issue, January 2002.

[54] G. Hadjiyiannis, P. Russo, and S. Devadas. A methodology for accurate performance evaluation in architecture exploration. In *In Proceedings of the Design Automation Conference (DAC 99)*, pages 927–932, 1999.

[55] G. Hamerly and C. Elkan. Learning the k in k-means. Technical Report CS2002-0716, University of California, San Diego, 2002.

[56] J. Haskins and K. Skadron. Minimal subset evaluation: Rapid warm-up for simulated hardware state. In *Proceedings of the 2001 International Conference on Computer Design*, September 2001.

[57] J. Haskins and K. Skadron. Memory reference reuse latency: Accelerating sampled microarchitecture simulations. Technical Report CS-2002-19, U of Virginia, July 2002.

[58] T.H. Heil and J.E. Smith. Selective dual path execution. Technical report, University of Wisconsin - Madison, November 1996.

[59] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*, chapter 2, pages 37–81. Addison-Wesley, 1979.

[60] M. Huang, J. Renau, and J. Torrellas. Profile-based energy reduction in high-performance processors. In *4th Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2001.

[61] Sematech International. Design twg update. In *International Technology Roadmap for Semiconductors*, Hsinchu, Taiwan, December 2000.

[62] M. Itoh, S. Higaki, J. Sato, A. Shiomi, Y. Takeuchi, A. Kitajima, and M. Imai. Effectiveness of the asip design system peas-iii in design of pipelined processors. In *In Proceedings of Asia and South Pacific Desing Automation Conference 2001 (ASP–DAC 2001)*, pages 649–654, 2001.

[63] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the DATE 2001 on Design, automation and test in Europe*, pages 190–196, 2001.

[64] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge. An analytical model for designing memory hierarchies. *IEEE Transactions on Computers*, 45(10):1180–1194, 1996.

[65] E. Jacobsen, E. Rotenberg, and J.E. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, December 1996.

[66] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[67] J.-M. Jolion, P. Meer, and S. Bataouche. Robust clustering with applications in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):791–802, 1991.

[68] D. Joseph and D. Grunwald. Prefetching using markov predictors. In *24th Annual International Symposium on Computer Architecture*, June 1997.

[69] R. E. Kass and L. Wasserman. A reference Bayesian test for nested hypotheses and its relationship to the schwarz criterion. *Journal of the American Statistical Association*, 90(431):928–934, 1995.

[70] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[71] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. In *Proceedings of the International Conference on Computer Design*, September 2000.

[72] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *Computer*, 30(9):75–78, 1997.

[73] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. In *Workload Characterization of Emerging Applications, Kluwer Academic Publishers*, September 2000.

[74] T. V. Lakshman and Dimitrios Stiliadis. High-speed policy-based packet forwarding using efficient multi-dimensional range matching. In *Proceedings of SIGCOMM*, pages 203–214, 1998.

[75] E. Lawler and D. Wood. Branch and bound methods: A survey. *Operations Research*, 14(291):699–719, 1966.

[76] S. Leibson. Xscale (strongarm-2) muscles in. *Microprocessor Report*, September 2000.

[77] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, December 1996.

[78] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[79] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138–147, October 1996.

[80] J. MacQueen. Some methods for classification and analysis of multivariate observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, CA, 1967. University of California Press.

[81] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[82] S. McFarling. Program optimization for instruction caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 183–191, April 1989.

[83] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[84] S. McFarling and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. Association for Computing Machinery, 1986.

[85] G. Memik, W. H. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference Computer-Aided Design (ICCAD)*, 2001.

[86] M. Merten, A. Trick, R. Barnes, E. Nystrom, C. George, J. Gyllenhaal, and Wen mei W. Hwu. An architectural framework for run-time optimization. *IEEE Transactions on Computers*, 50(6):567–589, June 2001.

[87] M. Mock, C. Chambers, and S.J. Eggers. Calpa: a tool for automating selective dynamic compilation. In *33rd International Symposium on Microarchitecture*, pages 291–302, December 2000.

[88] T. Morimoto, K. Saito, H. Nakamura, T. Boku, and K. Nakazawa. Advanced processor design using hardware description language aidl. In *In Proceedings of Asia and South Pacific Desing Automation Conference 1997 (ASP–DAC 1997)*, pages 387–390, 1997.

[89] D. R. Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

[90] H. Mulder and R. J. Portier. Cost-effective design of application specific vliw processors using the scarce framework. In *22th International Symposium on Microarchitecture*, 1989.

[91] J. Mulder. An area model for on-chip memories and its applications. *IEEE Journal of Solid States Circuits*, 26(2):98–106, February 1991.

[92] E. Musoll. Predicting the usefulness of a block result: a micro-architectural technique for high-performance low-power processors. In *32nd International Symposium on Microarchitecture*, November 1999.

[93] R. Muth, S.A. Watterson, and S.K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[94] S. Narayanasamy, T. Sherwood, S. Sair, B. Calder, and G. Varghese. Catching accurate profiles in hardware. In *9th International Symposium on High-Performance Computer Architecture (HPCA-9)*, February 2003.

[95] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[96] M. Oskin, F. T. Chong, and M. Farrens. HLS: Combining statistical and symbolic simulation to guide microprocessor designs. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[97] D. Pelleg and A. Moore. X-means: Extending K-means with efficient estimation of the number of clusters. In *Proceedings of the 17th International Conf. on Machine Learning*, pages 727–734. Morgan Kaufmann, San Francisco, CA, 2000.

[98] Radia Perlman. *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 2nd edition, October 1999.

[99] M. Postiff, G. Tyson, and T. Mudge. Performance limits of trace caches. *Journal of Instruction Level Parallelism*, 1, 1999.

[100] M. Puig-Medina, G. Ezer, and P. Konas. Verification of configurable processor cores. In *Proceedings of the Design Automation Conference (DAC2000)*, pages 426–431, 2000.

[101] R. W. Quong. Expected i-cache miss rates via the gap model. In *21st Annual International Symposium on Computer Architecture*, pages 372–383, 1994.

[102] P. Ranganathan, S. V. Adve, and N.P. Jouppi. Reconfigurable caches and their application to media processing. In *27th Annual International Symposium on Computer Architecture*, pages 214–224, June 2000.

[103] B. Ramakrishna Rau and Michael S. Schlansker. Embedded computing: New directions in architecture and automation. In *7th International Conference on High-Performance Computing (HiPC2000)*, 2000.

[104] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable functional units. In *27th International Symposium on Microarchitecture*, pages 172–180, 1994.

[105] G. Reinman and B. Calder. Predictive techniques for aggressive load speculation. In *31st International Symposium on Microarchitecture*, 1998.

[106] G. Reinman and N. Jouppi. Cacti version 2.0. http://www.research.digital.com/wrl/people/jouppi/CACTI.html, June 1999.

[107] S. Rixner, W. J. Dally, U. J. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. D. Owens. A bandwidth-efficient architecture for media processing. In *31st International Symposium on Microarchitecture*, pages 3–13, November 1998.

[108] Martin Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of LISA'99: 13th Systems Administration Conference*, pages 229–238, November 1999.

[109] R. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer Aided Design*, 6(5):727–750, 1987.

[110] M. Sanchez, E. Biersack, and W. Dabbous. Survey and taxonomy of IP address lookup algorithms. *IEEE Network Magazine*, 15(2):8–23, 2001.

[111] S. Santhanam. Strongarm 110: A 160mhz 32b 0.5w cmos arm processor. In *Proceedings of HotChips VIII*, pages 119–130, 1996.

[112] Y. Sazeides and J. E. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, pages 248–258, December 1997.

[113] Manfred Schlett. Trends in embedded microprocessor design. *IEEE Computer*, 31(8):44–49, August 1998.

[114] R. Schreiber, S. Aditya, B.R. Rau, V. Kathail, S. Mahlke, S. Abraham, and G. Snider. High-level synthesis of nonprogrammable hardware accelerators. Technical report, Hewlett Packard Reseach Labs, 2000. HPL-2000-31.

[115] Niraj Shah. Understanding network processors. Ver 1.0, September 2001.

[116] T. Sherwood and B. Calder. Time varying behavior of programs. Technical Report UCSD-CS99-630, UC San Diego, August 1999.

[117] T. Sherwood and B. Calder. Loop termination prediction. In *3rd International Symposium on High Performance Computing*, October 2000.

[118] T. Sherwood and B. Calder. Automated design of finite state machine predictors for customized processors. In *28th Annual Intl. Symposium on Computer Architecture*, June 2001.

[119] T. Sherwood and B. Calder. Patchable instruction rom architecture. In *International Conference on Compiler, Architecture, and Synthesis for Embedded Systems (CASES 2001)*, November 2001.

[120] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *International Conference on Supercomputing (ICS)*, June 1999.

[121] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[122] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[123] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[124] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[125] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, December 2000.

[126] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th International Symposium on Computer Architecture (ISCA 2003)*, June 2003.

[127] T. Sherwood, G. Varghese, and B. Calder. A pipelined memory architecture for high throughput network processors. In *30th International Symposium on Computer Architecture (ISCA 2003)*, June 2003.

[128] K. Skadron, P. Ahuja, M. Martonosi, and D. Clark. Branch prediction, instruction-window size, and cache size: Performance tradeoffs and simulation techniques. *IEEE Transactions on Computers*, 48(11):1260–81, November 1999.

[129] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[130] C. Snyder. Synthesizable core makeover: Is lexra's seven-stage pipelined core the speed king? In *Microprocessor Report*, June 2001.

[131] C.D. Snyder. Fpga processors cores get serious. *Microprocessor Report*, 14(9), September 2000.

[132] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, pages 414–425, June 1995.

[133] T. Spalink, S. Karlin, L. L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Symposium on Operating Systems Principles*, pages 216–229, 2001.

[134] V. Srinivasan and G. Varghese. Fast address lookups using controlled prefix expansion. *ACM Transactions on Computer Systems*, 7(1):1–40, February 1999.

[135] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[136] R. A. Sugumar and S. G. Abraham. Set-associative cache simulation using generalized binomial trees. *ACM Transactions on Computer Systems*, 13(1):32–56, 1995.

[137] Kevin Thompson, Miller Gregory J, and Rick Wilder. Wide-area internet traffic patterns and characteristics. *IEEE Network Magazine*, 11(6):10–23, November 1997.

[138] J. Tubella and A. Gonzalez. Control speculation in multithreaded processors through dynamic loop detection. In *4th International Symposium on High Performance Computer Architecture*, February 1998.

[139] G. Tyson, M. Farrens, J. Mathews, and A. Pleszken. Managing data caches using selective cache line replacement. *International Journal of Parallel Programming*, 25(3), 1997.

[140] G. Tyson, K. Lick, and M. Farrens. Limited dual path execution. Technical Report CSE-TR 345-97, University of Michigan, 1997.

[141] S. Wallace, B. Calder, and D.M. Tullsen. Threaded multiple path execution. In *22nd Annual International Symposium on Computer Architecture*, June 1998.

[142] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, December 1997.

[143] D. B. Whalley. Fast instruction cache performance evaluation using compile-time analysis. In *Proc. 1992 ACM SIGMETRICS and PERFORMANCE '92 Int'l. Conf. on Measurement and Modeling of Computer Systems*, page 13, Newport, Rhode Island, USA, 1-5 1992.

[144] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. In *IEEE Journal of Solid-State Circuits*, May 1996.

[145] T. Wolf. A network processor performance and design model with benchmark parameterization. In *in Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[146] T. Wolf and M. A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software*, pages 154–163, April 2000.

[147] Lisa Wu, Chris Weaver, and Todd Austin. Cryptomaniac: a fast flexible architecture for secure communication. In *28th Annual International Symposium on Computer Architecture*, pages 110–119, 2001.

[148] J. Yang and R. Gupta. Frequent value locality and its applications. *Special Issue on Memory Systems, ACM Transactions on Embedded Computing Systems*, 1(1):79–105, November 2002.

[149] T. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.

[150] O. Zamir and O. Etzioni. Web document clustering: A feasibility demonstration. In *Research and Development in Information Retrieval*, pages 46–54, 1998.

[151] C. Zhang, F. Vahid, and W. Najjar. A highly-configurable cache architecture for embedded systems. In *30th Annual International Symposium on Computer Architecture*, June 2003.

[152] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *Proceedings of SIGCOMM '90*, 1990.

[153] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. *ACM Transactions on Computer Systems*, pages 101–125, May 1991.

[154] A. Zhi, A. Moshovos, S. Hauck, and P. Banerjee. Chimaera: A high performance architecture with a tightly-coupled reconfigurable functional unit. In *27th Annual International Symposium on Computer Architecture*, June 2000.