2000


Kirk William Cameron

Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_disstheses/7247

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
EMPIRICAL AND STATISTICAL APPLICATION MODELING
USING ON-CHIP PERFORMANCE MONITORS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by

Kirk W. Cameron
B.S., University of Florida, 1994
August 2000
Dedication

This dissertation is dedicated to:

my beautiful and loving wife

Melissa.

You are my compass and my light, without which

this would have never been possible.

And to God for strength, reason, and a wonderful life.
Acknowledgements

In such an enormous, time-consuming endeavor, there are invariably many people to thank for their guidance, support, friendship, and wisdom. Prof. Xian-He Sun, my advisor, mentor and friend, saw ambition and drive in me I didn't know I had. Without his vision, intelligence, and compassion, I'd never have made it this far. While he could not convince me to move to Chicago, he did inspire me to at least attempt the academic life. Most of my academic success is in some way attributable to Xian-He for which I thank him.

We all gain from experience. For that I must thank the LSU faculty, the 82450NX Chipset Validation Team at Intel Corporation, and the PA Team at Los Alamos National Laboratory for invaluable real-world experiences that undoubtedly prepared me for my professional career. I am in particular debt to Yong Luo, my laboratory mentor and friend, for his rebuttals, thoughts, and contributions. I would also like to thank my doctoral committee members for their suggestions and support.

They say a man's stature and character can be measured by observing those he calls friends. If so, I can think of no higher complement than being able to call the following individuals my friends. Dr. Kasidit Chanchio has been there since the beginning, and continues to provide me with support and insight on just about everything. Thanks to Steve O'Neal, without whom I'd have never passed the general exams. Ken Shell and Kimberly Schneider, if there was a beginning to this endeavor, you were there before that even started - I wish you both great success and thank you for always believing in me. Dave and Kara Heckman, I treasure your friendship and hope that one day our kids will play together. To the "Italian mafia", Fabrizio, Mariella,
Allesandro and Federico, thanks for all the advice and especially the tiramisu and accompaniments. To my sister, Tiffany, all my love and one more thing: you're next.

To my family and all my wonderful in-laws, your unending support and encouragement have always inspired me and I am thankful for your thoughts and prayers. And to the unmentioned multitude of others that have helped in many, many ways; thank you.
Table of Contents

Dedication ................................................................................................................................iii
Acknowledgements ................................................................................................................iv
List of Tables ........................................................................................................................... ix
List of Figures ........................................................................................................................ xi
Abstract .................................................................................................................................... xiii

Chapter 1 Introduction ........................................................................................................1
  1.1 Performance Analysis ............................................................................................... 1
  1.2 Measuring Performance ........................................................................................... 2
  1.3 Our Approach ............................................................................................................ 4
  1.4 Motivation ................................................................................................................... 5
  1.5 Thesis ........................................................................................................................... 6
  1.6 Organization .............................................................................................................. 6

Chapter 2 Literature Review .............................................................................................. 8
  2.1 Superscalar Architectures ....................................................................................... 8
  2.2 Motivation ................................................................................................................... 10
  2.3 Our Analytical Approach ....................................................................................... 11
  2.4 Historic Approaches to Modeling .......................................................................... 14
  2.5 Related Work ........................................................................................................... 19
  2.6 Summary ................................................................................................................... 27
  2.7 References ................................................................................................................. 28

Chapter 3 Tools, Testbeds and Workloads ..................................................................... 34
  3.1 Measurements .......................................................................................................... 34
  3.2 Hardware Monitors ............................................................................................... 35
  3.3 Current Tools .......................................................................................................... 37
  3.4 Common Problem Set ............................................................................................ 40
  3.5 Testbeds ..................................................................................................................... 42
    3.5.1 MIPS R10000 Machines ................................................................................... 42
    3.5.2 Intel Machines ................................................................................................... 45
    3.5.3 ASCI Codes ....................................................................................................... 46
    3.5.4 Codes for Architectural Evaluation .............................................................. 48
  3.6 Workload Characterization .................................................................................. 49
    3.6.1 CPI Characterization of ASCI Codes ........................................................... 49
    3.6.2 Steady-State Characterization of ASCI Codes ............................................. 52
    3.6.3 Simulator Characterization of SPEC Codes ............................................... 53
  3.7 Summary ................................................................................................................... 62
  3.8 References ................................................................................................................. 62
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6.3 Scalability Comparison</td>
<td>142</td>
</tr>
<tr>
<td>5.6.4 Evaluation of Memory Components</td>
<td>144</td>
</tr>
<tr>
<td>5.7 Hybrid Model Analysis</td>
<td>147</td>
</tr>
<tr>
<td>5.7.1 Level 1 Results</td>
<td>148</td>
</tr>
<tr>
<td>5.7.2 Level 2 Results</td>
<td>149</td>
</tr>
<tr>
<td>5.8 Summary</td>
<td>152</td>
</tr>
</tbody>
</table>

Chapter 6 Conclusions ................................................. 153
6.1 Overall Summary ..................................................... 154
6.2 Scientific Contributions ................................. 155
6.3 Future Work .......................................................... 156

Appendix: Performance Monitor Survey .................................. 158

Vita ...................................................................................... 162
List of Tables

Table 3.1 Common problem instruction set ............................................................. 41
Table 4.1 Accuracy of the validation method .......................................................... 73
Table 4.2 Validation of t2 and tm for the memory model ........................................ 73
Table 4.3 Results for synthetic instruction streams on MIPS R10000 .................. 80
Table 4.4 Results for ideal synthetic instruction streams on MIPS R10000 ............ 81
Table 5.1 Model parameters for PowerChallenge .................................................. 107
Table 5.2 Memory access times for PC, O2K, and Intel ASCI Red ....................... 108
Table 5.3 Branch and icache characteristics for measured codes ....................... 112
Table 5.4 Utilization factors for the measured codes ............................................ 113
Table 5.5 Ideal cpi0 calculated using Equation 4.12 ............................................. 114
Table 5.6 MFU mutation penalty ......................................................................... 125
Table 5.7 Architecture based on time of analysis and mutation ........................... 127
Table 5.8 Mutation frequency ............................................................................ 134
Table 5.9 Percent of time RS-MFU is full ......................................................... 136
Table 5.10 Class level information ..................................................................... 137
Table 5.11 Mean effects table ............................................................................ 138
Table 5.12 Contrast method for pairwise comparison .......................................... 140
Table 5.13 LSD post hoc comparison for code .................................................. 141
Table 5.14 LSD post hoc comparisons for machines ........................................... 141
Table 5.15 Scalability comparison of HEAT ...................................................... 142
Table 5.16 Scalability comparison of Sweep ...................................................... 144
Table 5.17 L1 hit ratio comparison for HEAT ..................................................... 145
Table 5.18 L2 hit ratio comparison for HEAT .......................................................... 145

Table 5.19 L2 hit ratio comparison for HYDRO.......................................................... 146
List of Figures

Figure 3.1 Topology of SGI Origin 2000 at Los Alamos ........................................... 43
Figure 3.2 Performance of HEAT as a function of linear problem size .................... 51
Figure 3.3 Performance of SWEEP as a function of linear problem size ................. 51
Figure 3.4 Performance of HYDRO as a function of linear problem size ............... 51
Figure 3.5 Performance of HYDROT as a function of linear problem size ............... 52
Figure 3.6 Performance of NEUT as a function of linear problem size ................. 52
Figure 3.7 Instruction distances for Heat on O2K and PC ........................................ 54
Figure 3.8 Instruction distances for Sweep on O2K and PC .................................... 54
Figure 3.9 Instruction distances for Dsweep on O2K and PC .................................. 54
Figure 3.10 Instruction distances for Hydro on O2K and PC .................................... 55
Figure 3.11 Instruction distances for Hydro-t on O2K and PC ................................. 55
Figure 3.12 Instruction distance occurrences for Swim ............................................. 58
Figure 3.13 Instruction distance occurrences for Wave5 .......................................... 58
Figure 3.14 Instruction distance occurrences for Su2cor ......................................... 59
Figure 3.15 Instruction distance occurrences for Compress95 .................................. 59
Figure 3.16 Instruction distance occurrences for Ijpeg .............................................. 60
Figure 3.17 Instruction distance occurrences for li .................................................... 60
Figure 3.18 Instruction distance occurrences for kmeans ......................................... 61
Figure 4.1 Memory model times .............................................................................. 69
Figure 4.2 Latency effect on model ......................................................................... 69
Figure 5.1 PTERA modules ..................................................................................... 100
Figure 5.2 MIPS R10000 SynBAD specification and resulting assemble .................. 102
Figure 5.3 Memory stall and overlap parameters (PowerChallenge) ................. 109
Figure 5.4 Memory stall and overlap parameters (Origin 2000) ...................... 109
Figure 5.5 Memory stall and overlap parameters (Intel ASCI Red) .................. 110
Figure 5.6 Performance variation in relation to link-length and link-width ....... 119
Figure 5.7 Performance variation in relation to link-length and link-width ....... 121
Figure 5.8 Performance variation in relation to link-length and link-width ....... 122
Figure 5.9 MIPS R10000 functional unit and reservation station .................. 126
Figure 5.10 RS-MFU modification to MIPS R10000 ....................................... 129
Figure 5.11 IPC of various architecture schemes ........................................... 132
Figure 5.12 IPC results for varying number of entries in RS-MFU ................. 135
Figure 5.13 Machine mean distribution ......................................................... 139
Figure 5.14 Code mean distribution ............................................................... 139
Figure 5.15 $m_0$ values calculated on the Origin 2000 ................................. 150
Figure 5.16 $m_0$ values calculated on the PowerChallenge ............................ 151
Abstract

To analyze the performance of applications and architectures, both programmers and architects desire formal methods to explain anomalous behavior. To this end, we present various methods that utilize non-intrusive, performance-monitoring hardware only recently available on microprocessors to provide further explanations of observed behavior. All the methods attempt to characterize and explain the instruction-level parallelism achieved by codes on different architectures. We also present a prototype tool automating the analysis process to exploit the advantages of the empirical and statistical methods proposed. The empirical, statistical and hybrid methods are discussed and explained with case study results provided. The given methods further the wealth of tools available to programmer's and architects for generally understanding the performance of scientific applications.

Specifically, the models and tools presented provide new methods for evaluating and categorizing application performance. The empirical memory model serves to quantify the hierarchical memory performance of applications by inferring the incurred latencies of codes after the effect of latency hiding techniques are realized. The instruction-level model and its extensions model on-chip performance analytically giving insight into inherent performance bottlenecks in superscalar architectures. The statistical model and its hybrid extension provide other methods of categorizing codes via their statistical variations. The PTERA performance tool automates the use of performance counters for use by these methods across platforms making the modeling process easier still. These unique methods provide alternatives to performance modeling and categorizing not available previously in an attempt to utilize the inherent
modeling capabilities of *performance monitors* on *commodity processors* for scientific applications.
Chapter 1

Introduction

1.1 Performance Analysis

Today's superscalar architectures have become very complex. Every new generation promises higher speeds and higher throughput, leading to better performance. It is not always clear what "better performance" means, however. To some it simply means certain applications perform better. For instance, over the past several years, the demand for faster graphics rendering has been the thrust of processor advances resulting in special hardware modifications specifically targeted at boosting the performance of these types of applications. Here, performance was improved by decreasing the amount of time necessary to perform typical graphics-related instructions. What about every other type of application? This drives home the point that although improvements generally have targeted workloads, those are not always the same workloads in which you are interested. This leads us to wonder how particular enhancements affect the codes in which we are interested.

A simple way to see what effect new architectures have on our codes is to simply run our codes on the new architecture. This is a full-proof method of observing performance. The problem is that this only tells us the overall performance, not why the performance is such. Furthermore, such results don't generally give an indication of how other improvements might affect our codes. For example, would it be worth it to purchase more cache so our applications would run faster? With simple overall timings, we cannot estimate performance unless we measure runs on all the different configurations. There must be a better way to provide performance analysis.
There is a better way. We can develop analytical and empirical models that seek to describe the performance of code and machine. Models of this type typically have parameters that take into consideration code and machine separately. In this way we can anticipate how changes will affect performance without the need of executing every permutation of code and machine. Unfortunately, comprehensive models of this type are very difficult to develop. To make things easier, we can focus on certain types of workloads and certain types of machines that are more easily modeled.

In our work, we chose to focus on scientific workloads. These types of workloads tend to use matrices and tend to be computationally intensive. For this reason their performance is quite often loop-dominated making them somewhat more conducive to modeling. Focusing on codes of this type, we can provide more complicated modeling of the underlying architecture. Our models pay particular attention to the underlying architecture while maintaining the ability to model most superscalar architectures and memory hierarchies. We do not focus on other performance contributors like I/O as they will certainly complicate our modeling efforts. Our approach is piece-wise to some extent in that we must fully understand the microprocessor performance and memory performance before we "muck the water" with other complicated performance estimates for other contributing factors.

1.2 Measuring Performance

We must define our method of gauging performance. Since we focus on single processor and memory hierarchy performance in our modeling techniques, our parameters must incorporate the characteristics of both levels of focus while providing some amount of comparability within architectures. For these reasons, we chose cpi
(cycles per instruction) as our gauge for performance. The overall cpi for a particular code-architecture combination is indicative of the achieved instruction-level parallelism. It shows the extent to which a superscalar processor is good at performing superscalar activities in a quantifiable way. Obviously, on-chip architectural changes will affect cpi for better or worse. Furthermore, effective performance gain or loss in the memory hierarchy translates into a decrease or increase in cpi, respectively. So, we begin with cpi as our key indicator of performance. When we focus on cpi, it opens up other intriguing possibilities due to its nature.

In the second edition of *Computer Architecture: A Quantitative Approach*, John Hennessy and David Patterson describe an interesting method of itemizing the overall cycles per instruction rate (cpi) of an application-architecture combination.

To determine the cpi for an instruction in a modern processor, it is often useful to separate the component arising from the memory system and the component determined by the pipeline, assuming a perfect memory system. This is useful both because the simulation techniques for evaluating these contributions are different and because the memory system contribution is added as an average to all instructions, while the processor contribution is likely to be instruction specific. Thus, we can compute the cpi for each instruction, i, as:

$$ cpi_i = \text{pipeline cpi}_i + \text{memory system cpi}_i $$

Following this generalization about the overall cpi for a code, in this dissertation we present several modeling methods and their resulting conclusions and extensions that follow Hennessy and Patterson's assertion that pipeline cpi and memory cpi can be evaluated separately. Each of the modeling techniques discussed in this text follow this general approach by attempting to further refine and isolate the individual terms that collectively describe the overall cpi of a code.
1.3 Our Approach

The empirical memory model focuses on memory system cpi. It provides an analysis method allowing empirical inference of average incurred stall time due to memory accesses. The statistical and hybrid models provide further insight into the memory system cpi particularly for architectures that vary in their implementation of memory hierarchies. The instruction-level model provides a novel technique of evaluating the on-chip or pipeline cpi for a particular code. Bottleneck analysis using this method leads to suggestions for architectural improvement.

As shown in Chapter 2, approaches of this type are not completely new. Simulators have been available for quite some time to allow performance analysts to model the behavior of code-architecture combinations. Simulations are severely limiting in their ability to run computationally large codes in a short period of time. For this reason, empirical methods are more practical for scientifically, computationally large codes. An empirical approach to modeling (or a statistical one for that matter) involves gathering measurable data during code execution and attempting to understand performance through interpretation using advanced modeling methods.

Until recently, empirical measurements for superscalar architectures were limited at best. The majority of empirical and analytical modeling techniques used time measured in seconds as model inputs to analyze performance. In the last few years however, performance monitoring hardware has become fairly common among processor architectures including Intel, Compaq, and IBM chips to name a few. It is this recent advance that the methods described herein attempt to exploit, immediately
qualifying them as novel approaches. By limiting the events necessary to measurement, the intent is to provide methods that are somewhat applicable across architectures.

1.4 Motivation

Application developers are concerned with producing a product on time. They focus on quickly creating bug-free code that meets predetermined specifications. These developers do not typically have the time or the inclination to develop complex performance models of their particular application for evaluation and prediction over various platforms. Due to time and cost constraints, code developers depend on others to provide the cross-platform performance analysis techniques and tools necessary to quickly identify bottlenecks in applications.

Architectural designers focus on the general performance of their hardware on applications. In development they focus on low-level characteristics such as die area, gate complexity, power, and clock rate. In practice of course, the resulting chip design was created to run code efficiently. Unfortunately, the general applicability of a microprocessor design limits individual code performance. Today's superscalar microprocessors have grown increasingly more complex as designers attempt to hide memory latency, and increase instruction-level parallelism. Certain enhancements made to current generation processors are difficult to quantify. The resulting complexities and performance enhancements demand more comprehensive methods of quantifying individual contributions to performance degradation.

It is the goal of performance analysis to produce general models that quantify, evaluate and sometimes predict performance of applications on a given architecture. Such models often incorporate qualities of the code and architecture and provide
formulas for evaluation of the interaction between a particular application and hardware platform. While comprehensive models are rare, models of this type tend to focus on communication, memory, or architecture performance. This dissertation focuses on models that attempt to give insight into single processor memory and architecture performance.

1.5 Thesis

Our thesis is as follows. Performance monitors have recently become available on commodity processors. Previous methods primarily used overall timings to develop empirical and analytical methods for categorizing the performance of applications. We seek to provide methods that utilize the strengths of performance monitors to provide means of categorizing and analyzing scientific applications. The measurements available allow particular quantification of the memory and on-chip performance analytically, empirically, and statistically. Our intention is to provide the first fairly comprehensive attempt at performance analysis utilizing performance monitors only. Such techniques promise useful, readily available methods for gauging code performance of both applications and architectures.

1.6 Organization

This dissertation is organized as follows: Chapter 2 provides a literature review of past modeling approaches in relation to the current work. We discuss the multitude of approaches to performance modeling that exist and attempt to provide arguments for the empirical and statistical approaches of our models. Chapter 3 provides the context of the performance models. We attempt to provide simply the nuts and bolts of the different models we have developed in Chapter 4. The following chapter provides the
detailed case studies for each particular model. Lastly, in Chapter 6 we present overall conclusions and future directions of this work.
Chapter 2

Literature Review

2.1 Superscalar Architectures

In 1971, the Intel 4004 microprogrammable computer on a chip was born, forever changing the world [1]. From humble beginnings, microprocessors have advanced far beyond what was imaginable in the early 1970s. As technology has evolved, we have been able to add greater complexity on a single chip through the ability to fit more and more transistors into a fixed die area. Increasing the available number of transistors translates into more complex logic at the architectural level. Performance drives architectural development. So, as the number of transistors available increased, the complexity of the microprocessor followed suit [2, 3].

Early microprocessors issued and executed a single instruction per cycle [1]. These types of architectures are known as scalar processors. A natural extension to scalar processing is superscalar processing, or the ability to issue more than one instruction per clock cycle [4]. Superscalar processors were first created in the early 1990s possibly as a consequence of the reduced instruction set computing (RISC) movement [5]. Superscalar processing is not necessarily limited to RISC architectures, as some CISC architectures (complex instruction set computers) incorporate superscalar abilities as well [6].

Superscalar execution evolved from pipelined execution in the 1950s and 1960s [7-9]. Pipelined execution broke instruction execution into pieces that could be overlapped for greater performance. With pipelining, the goal of achieving instruction-level parallelism or ILP (multiple instructions issued and executed in a single cycle)
was realized. However, it would not be until the mid-to-late 1980s that superscalar processors would appear [10, 11]. In the early 1980s, multiple instruction issue and execution were open problems. A number of significant research advances would lead to a viable superscalar architecture including branch prediction, register renaming, out-of-order execution, and non-blocking loads [4, 12-14]. Particularly in the early 1980s, research at Stanford, Berkeley, and IBM was focused on the RISC architecture [5, 15-18]. Initial implementations of superscalar architectures were predominantly RISC based [10, 11]. As the complexity of processors increased, superpipelined superscalar processors evolved while striving to increase and exploit the amount of ILP available in code and architecture. Jouppi eventually showed superpipelining and superscalar approaches to be similar methods of obtaining instruction-level parallelism [19]. Today, architectural development is essentially focused on achieving a maximum degree of ILP while maintaining a high clock frequency and a reduced instruction set [2, 20, 21]. Superscalar architectures of today implement the following general techniques to achieve greater performance according to Smith and Sohi at the University of Wisconsin-Madison:

- Improved instruction fetch capabilities (stemming usually from advanced branch prediction)
- Methods for isolating true dependences between instructions (obtained via register renaming)
- Methods for issuing multiple instructions in parallel (icache, preliminary decoding)
• Resources for parallel execution of instructions (multiple pipelined functional units, non-blocking loads, increased instruction window sizes and depth of speculation)
• Methods for communicating data through memory hierarchies (data consistency from cache to memory, stream buffers, replacement algorithms)
• Methods for committing the process state in order (out-of-order execution while maintaining correctness)

The reader is referred to two Patterson and Hennessy texts on the subject of computer architecture for detailed explanation of these and other microprocessor architecture topics [22, 23]. We mention the above topics merely for completeness in our discussion of microprocessor performance analysis.

2.2 Motivation

The metric of primary interest to a performance analyst is time. Both application developers and architectural designers want to know how fast codes will complete on particular architectures. While their needs are similar, their goals are quite different. A code developer would like feedback on how to write an application to take full advantage of the underlying architecture. The architectural designer is interested in isolating bottlenecks in the architecture that may provide further avenues of performance improvement in succeeding generations of the architecture. To determine performance across platforms, architectural designers use benchmarks (like SPEC [24] and lmbench [25]). Unfortunately, there are limitations to such comparison [26]. The result, however unsettling, is that (at least at present) future architectures are greatly influenced by the measured performance of particular benchmark codes on simulated...
designs. On the other hand, user applications are not typically represented by benchmark applications. Users are left to "tweak" their codes until they achieve acceptable performance against implemented designs that were probably not created with their code in mind.

Thus, there exists a great need for models that can represent the interaction between code and architecture in an intuitive manner providing insight to both hardware designers and application developers. There is a wide gap however between models created to identify bottlenecks in architecture and those created to analyze code. We will discuss this in greater detail later in this document, but for now let us just say that architectural models tend to require extensive knowledge of architecture and codes at a very low level. In fact to analyze hardware, more often than not simulators are used. On the other side, vendors typically provide high-level tools to help users analyze code. These tend to be very architecturally specific however, and require code developers to learn a different tool for different architectures. There exists a need for models and tools that bridge this gap between existing models. This is the focus of our approach at model development. In the next sections we provide details of past modeling efforts focused on architectural design modeling. We'll attempt to highlight differences to our approach while providing a glimpse of our modeling efforts.

2.3 Our Analytical Approach

The focus of our performance analysis methods will be on discovering and dissecting the instruction-level parallelism inherent to code and architecture and their interaction. For completeness, let us begin with a simple discussion of program execution time. As mentioned, time is the primary metric used to determine application performance over a
specific architecture. Microprocessor customers invariably want their applications to run faster and faster so productivity is at least enhanced by time saved waiting on things like backups, disk access time, and computation. In today's computer systems there are a large number of places performance can be studied. Typically, the overall execution time of an application is the sum of the computational time (on-chip), the time spent accessing memory (off-chip to memory), the communication time between nodes (parallel processing or client-server context) and the time spent on I/O (time to disk or output). Each of these pieces of execution time has warranted large amounts of research. In the context of this document, we will focus only on computational time (on-chip) and time spent accessing memory (off-chip) for applications executing on a single processor. As will be apparent, these problems are complex enough to support years of useful research.

Now that we are focusing on overall execution time for on-chip computation and memory accesses, we can use a fairly common formula to express the execution time of a particular application. Following [27], we express time as the following product:

$$\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$$ (2.1)

Following Bhandarkar and Clark in [27], we can describe the terms of this equation in regard to the system aspects that influence them. The number of instructions resulting from a program is a function of the compiler and instruction set architecture (ISA). The cycle time is a direct function of the underlying VLSI technology and architectural design (such as degree of pipelining, ISA, etc.). The cycles per instruction is a function of many things including the architectural design and the compiler (in essence the code). So the cycles per instruction (or cpi) gives an indication of the interaction...
between code and architecture performance. Furthermore, the cpi is related to the achieved instruction-level parallelism of a particular code-architecture combination. In fact, achieved cpi is of great interest at the beginning stages of the architectural design process [28-30]. So by focusing on cpi values, we can infer performance differences for different codes on the same architecture and for the same codes on architectures with the same ISA but minor enhancements. We will use cpi to compare the achievable ILP of particular code-machine combinations throughout this document.

Most performance analyses using cpi values have the objective of evaluating the architecture only. As a result, models of this type do not typically break the term itself down any further than overall cpi. But, the ILP (and hence cpi) of a program varies greatly across the duration of a program [19]. We feel that great insight can be gathered into application and architecture performance if we break down cpi into contributing pieces. Following [22] and [31], we initially break cpi down into two parts corresponding to the pipeline and memory cpi.

\[ cpi = \text{pipeline cpi} + \text{memory system cpi} \]  

(2.2)

If we decouple memory and pipeline cpi we can focus on individual contributions to the overall cpi. Another advantage to this approach is that by separating the two terms, we can derive models that independently attempt to model each piece. This can lead to an iterative design process allowing us to replace obsolete models with more accurate ones or updated versions reflecting new architectural changes. Furthermore, as Emma adeptly describes in [31], cpi is intuitive in nature when we try to explain performance degradation in terms of lost instruction-level parallelism. In fact, Emma alludes to the development of models that dissect cpi into even more terms describing pieces of the...
overall cpi. Emma's paper provides an exhaustive discussion on the properties and usefulness of cpi modeling formulations. Our models use mean-value analyses, classic statistics, and elementary queuing theory to achieve acceptable accuracy and to allow realistic analyses [32-35]. Our novel approach to modeling memory system cpi (as described later in the memory model section) uses mean-value analysis techniques. Our instruction-level model that attempts to estimate pipeline cpi is unique as well incorporating elementary queuing analysis and allowing for iterative statistical approximations of the modeled code. Our statistical approach uses regressive techniques to identify variations in the cpi formula across similar architectures. Finally, as will become evident in later sections, all of our approaches utilize performance counter values only while still providing analysis capabilities. Perhaps more importantly, by limiting the input parameters of our models, the techniques themselves can be used across many platforms that support similar types of event counting.

2.4 Historic Approaches to Modeling

Since the Intel 4004, successive generations of microprocessors have attempted to outperform their predecessor's [3]. In previous sections, we discussed the particular innovations that brought about the development of the superscalar microprocessor of today along with our approach to quantifying instruction-level parallelism. But, the key to understanding the aforementioned enhancements and our approach can be found in the methods of performance evaluation prevalent today and throughout the 1980s and 1990s.

Later in this document, we will address current platform specific tools in the context of counters. These types of tools focus on presentation of architecture-specific
information gathered from a variety of sources including performance monitors, system-level interrupts, etc. They do not typically attempt to model or provide conclusions or suggestions regarding application performance. Analysis is left up to the user. In contrast, cpi models such as those presented in this document, aim to give qualitative and quantitative information to the user regarding code performance. In the context of historic ILP modeling, previous work has been accomplished for both the purpose of architectural evaluation and code performance analysis. Few provide useful information to both architects and application developers. These types of models (such as Saavedra et al. [36-42], and our current work) tend to sacrifice low-level details or accuracy for the sake of simpler formulas and general applicability. This is an acceptable tradeoff in many cases. At other times, simulators or platform specific tools are necessary to pinpoint particular details for architecture. Our models are made specifically to be useful across platforms for code-architecture analysis while sacrificing the ability to analyze some architectural details particular to one platform.

Microprocessors like the Intel 4004 were created by small teams of experts in relatively short periods of time. However, as processor complexity has increased, the number of people involved in the creation of a new processor has grown to keep pace. What used to be accomplished by a small team of engineers is now done by many separate teams with diverse tasks such as design, performance, validation, etc. With the increased complexity and cost of development, architects have developed methodologies for determining the features of new processors. These methodologies typically involve analyzing important benchmarks against trace-driven simulations of the new architecture. This is a time consuming task, and elementary models have been
developed to ease the process. But basically, the most interesting models of the last 25 years began with the development of trace-driven simulation to augment notoriously time-consuming direct-simulation [43]. We should note we use these terms quite generally in our discussion. For trace-driven simulation, we refer to any time-driven approach with the purpose of evaluating performance more than correctness by not actually simulating architecture completely. Typically this is accomplished via a certain level of abstraction (many times at the instruction-level) such that timings for these abstractions are estimated resulting in execution time estimates fairly close to the genuine architecture. Many times trace-driven simulations accept compiled execution traces. These are particularly used for determining features of next generation architectures that use the same instruction set architecture. Direct simulation involves a gate-level simulation of architecture (usually in some kind of hardware design language). Obviously, direct simulations are typically orders of magnitudes slower than their trace-driven counterparts. Architectural development these days typically involves iterative versions of trace-driven simulation for evaluating architectural tradeoffs [21, 28-30, 44].

As mentioned, trace-driven simulations became common in the mid-to-late 1970s [43]. The idea was to model performance while sacrificing some accuracy and reducing completion time. Studies regarding accuracy and usability have been numerous, as trace-driven simulators have continued to be popular methods of architectural performance evaluation [45-49]. Two recently successful trace-driven simulators are SimpleScalar [50] and MINT [51].
The advent of trace-driven simulations, led to new methods of performance modeling. One method that was very useful for early microprocessors was what we shall refer to as the analytical dot-product approach. These methods involve counting certain occurrences within an application's execution. The types of occurrences are sorted and execution timings are applied to each occurrence. In this way, a dot product of the number of each type of occurrence multiplied by the timing of each occurrence can be used as an accurate estimate of overall performance time. A classic paper by Peuto and Shustek [52] is an excellent representation of such a model. In 1978, Clark and Levy [53] managed to perform the same type of analysis using a performance-monitoring device instead of the trace-driven simulation approach. A few years later, MacDougall argued that a combination of trace-driven simulation and monitors gave an even better performance estimate [54]. Now, since processors were becoming more and more complicated resulting in large amounts of data from trace-driven simulation, Emma and Davidson proposed trace-reduction adding another layer of abstraction above excessive trace-driven results [55]. These types of methods are still prevalent today, as statistical methods become more and more important to performance analysis so as to reduce simulation results to meaningful subsets of information. From the late 1980s to mid 1990s, Saavedra et al. proposed an interesting variation on the trace-driven dot-product method common in earlier research [42]. He and his colleagues realized they could create small codes or micro-benchmarks that could be measured for execution time and applied to an analysis of the original source code. By creating a vector of micro-benchmarks, measuring source code for occurrences, and profiling the resulting executable, they could predict code performance on other machines. This was
an excellent innovation that led to several extensions to their methods to extend the accuracy and usefulness [36-41]. The problems in their method and similar dot-product methods we discuss in the next few paragraphs. Basically, strict application of such models becomes less useful as processors become more complicated and the effect of code-interaction begins to severely inhibit performance prediction and direct measurement.

Two very interesting threads of thought have resulted from early trace-driven modeling research. This underscores the importance of such modeling techniques in fostering further understanding of the interaction of code and architecture. The first is the reduced instruction set computing (RISC) movement [5, 15-18]. Papers such as Peuto and Shustek [52] resulted in the conclusion that although instruction sets tended to be large and complex (CISC), a smaller subset of instructions accounted for 80-90% of the overall set of instructions of a program. Just a few years later, the RISC movement was in full swing. The second major movement was the push from superpipelined to superscalar architectures mentioned previously. This was to come later, but some argue it was a direct result of the RISC movement. In any case, instruction-level parallelism was suddenly of major concern to the performance community.

While many researchers hoped for unlimited performance potential from instruction-level parallelism, others were not convinced of its usefulness. Jouppi and Wall were particularly pessimistic regarding the potential of ILP. They argued that code is inherently sequential allowing a maximum of two to three instructions per cycle to execute simultaneously [19]. More importantly, they proposed the metric of average
degree of superpipelining and showed superpipelining to be equivalent in its exploitation of instruction-level parallelism to superscalar processing [19]. Their pessimistic outlook for ILP did not deter development of techniques to improve achievable ILP in processors. As superscalar processors became available, compiler improvements and architectural advances such as register-renaming, increased window size, branch prediction, and out-of-order execution continued to increase the achievable ILP in processors. The result of these complicated additions to microprocessor logic was to make performance modeling and prediction much more difficult. The historical dot-product formulas mentioned would no longer accurately estimate processor performance since the processor was attempting to execute instructions in parallel. The most important complication was the dynamic property of code interaction. To put it simply, a single instruction no longer has a predictable execution time. You must consider the adjacent instructions and how they influence the performance. Furthermore, innovations such as non-blocking loads allow useful work to be accomplished while waiting on cache misses. Whether this event occurs close to the context of the instruction to be timed for a dot-product model will greatly influence the timing. It thus became necessary to take a different approach to performance modeling - a variation on the simple dot-product method.

2.5 Related Work

A variation on the original dot-product approach is necessary to model the execution overlap common in today's superscalar architectures. In our performance model, we separate memory and pipeline cpi as mentioned. By doing this, we can isolate contributions to performance with and without memory effect. Utilizing this separation,
we can use two distinct methods for modeling each portion of CPI. We use a queuing theory based method for analyzing the performance of pipeline CPI. This will be discussed in greater detail later in this document, but its approach is similar to other approaches at instruction-level modeling predominant in trace-driven analysis. Separately, we use a mean-value analysis technique to infer non-overlapped memory latencies in our memory model. The initial separation of pipeline and memory CPI is what allows the combination of our two separate modeling approaches into a single model for performance analysis. Our approach attempts to take the usefulness of the instruction-level modeling approach of on-chip bottleneck analysis and combine it with the overlap analysis method of inference described above. In this way, we can minimize the error of inference techniques while maximizing its usefulness in quantifying overlapped execution. We do not know of a model in the current literature that incorporates both of these modeling techniques in the interest of performance at the instruction-level. Furthermore, our models rely solely upon the output of performance monitors making them practical and applicable across platforms due to their inherent abstraction. We further analyze performance at this level through a statistical analysis approach based on the overall model to corroborate and elaborate on results obtained with the original model. We discuss related approaches highlighting the differences with our work in this section.

All of the approaches we discuss in this section (including our own) are variations on the dot-product method of performance evaluation. Basically, in each method, parameters for code and machine are defined, timings are measured, and performance is calculated from the resulting vector product. In fact, many of these
methods seem quite complicated at first glance. Nonetheless, they are all variations on the dot-product method. The major differences among these methods are the parameters defined and the derivation of the coefficients of the resulting vectors. In trace-driven approaches, parameter coefficients are derived from measurements within the trace-driven simulator. In empirical approaches, somehow measurements are obtained directly. This could be from direct timing routines, micro-benchmarks, or performance monitors. Whether empirical or trace-driven approaches are used, statistical reductions can be performed to minimize the data produced. There is also a separate trend to use statistical approaches as a third way of directly obtaining measurements. This is not through reduction, but through direct sampling. We see this as a future direction in its beginning stages of research development that we won't address here. Some related work can be found in [46, 56, 57]. These have no direct relation to our statistical method. Our method focuses on statistically isolating differences between separate results given by the full performance model (the combination of the instruction-level and memory model).

We can identify several types of analytical dot-product approaches to performance analysis. Generally, these are either layered or non-layered approaches. These two approaches can require trace-driven or empirically derived input parameters. A layered approach is a dot-product model that attempts to break performance measurements into contributing pieces. For example, the MACS approach [58] attempts bounding of performance by isolating contributions to degradation at the compiler and scheduler levels. In this way, their model uses step-wise refinement of performance estimates to focus on bottlenecks and offer suggestions for improvement.
They use direct measurements as inputs to their models making this an empirical version of the layered approach. Another empirical layered method is micro benchmarking [36-42]. This dot-product method utilizes direct empirical measurements of small micro-codes to estimate individual contributors to performance. Vectors of machine and application characteristics are used to estimate and predict performance of different machine-application combinations. This method at first focuses on simple source code compilations, then incorporates optimizations, followed by cache effects, etc. In this way, it is a layered model of performance analysis as well. We know of no trace-driven layered approaches currently. This makes some sense since layered approaches tend to need multiple different runs of code for complete coverage. Since trace-driven simulators are inherently slow, such methods would only compound the greatest drawback of trace-driven approaches. Non-layered approaches offer a less compartmentalized version of the layered approach. These types can be trace-driven or empirical as well. These are not layered because, while some assumptions may be present to ease modeling efforts, methods focus on resulting code instead of iterative versions incorporating individual contributions (such as compiler, scheduler, etc.). While these methods could be used to isolate performance contributors, they are not dependent on such methods as the layered approaches previously discussed. There are many examples of trace-driven non-layered modeling efforts [19, 52, 55, 59-63]. Non-layered trace-driven methods are the technique of choice for those evaluating next generation hardware implementations in detailed simulations. They usually rely on the exceptional detail provided through simulation allowing for histogram statistics and other measurements impossible via direct measurement. This results in strong, detailed
models of ILP and performance that are useful in evaluating the performance of future architectures and tradeoffs. Empirical non-layered approaches have been somewhat common as well [31, 53, 54, 64, 65]. Our memory, instruction-level and statistical models should be considered non-layered empirical versions of the dot-product approach. This is due to the fact that we obtain the coefficients for our vectors following direct measurement using performance monitors. Since performance counters monitor events on-chip, their results incorporate all variations implied by compilers, schedulers, etc. Hence, this must be a non-layered approach and is certainly empirical in nature.

There are pros and cons for each of these approaches. Layered empirical approaches offer a way of isolating individual contributors to performance [36-42, 58]. These can vary from compiler optimizations to scheduling and caching policies. Simplistic dot-product models of this type that do not take into account dynamic code-interaction can even be predictive in nature. Typically however, these models tend to suffer from architectural dependence. This can be in the form of compiler dependent results, or ISA dependent analysis techniques. Also, dynamic code interaction in the form of work overlap (very common to superscalar processors where work is accomplished in parallel at the instruction-level) is typically ignored in these approaches. Non-layered trace-driven approaches, as mentioned, are primarily useful in evaluating architectures. These methods are not usually focused on offering insight to code performance. Since they rely on simulation, they tend to be very time-consuming and tend to ignore memory influence in resulting models. Some of these could be used empirically, but this is not the case in the current literature [19, 52, 55, 59-63]. Non-layered empirical approaches must suffer from a loss in accuracy and analysis.
capabilities resulting from the abstractions necessary to utilize empirically derived results only. These can also suffer from architectural dependence if not enough abstraction is built into the models. The benefits result from the direct measurement qualities. The inaccuracy of the model abstraction can be offset by the use of actual measured results from the real system. Furthermore, with the advent of performance monitoring on chip and at the system level, such techniques can require very little overhead and can provide useful analysis while requiring only the normal runtime of an application. We believe these methods are also more easily extended to the advances in dynamic performance execution. This is where applications are given access to performance measurements on the fly enabling them to adapt their execution to avoid bottlenecks. While these approaches include binary translation, multi-threaded processing, and processors in memory, these are architectures in their infancy. Nonetheless, empirical on-chip measurements will most likely be enhanced for these different architectures warranting further development of models that exploit resulting event monitoring. The dynamic nature of such advancements leads to further dependence of models on the dynamic nature of workloads and hence will render other simpler dot-product models even more obsolete. It is for these reasons that we believe non-layered empirical techniques are very likely to gain popularity with inevitable architectural advances beyond superscalar technology.

Since ours is an empirical non-layered approach to application and architecture evaluation and modeling, we should contrast the related efforts just mentioned in some detail. We previously mentioned the work of Peuto and Shustek [52], Clark and Levy [53], Emma and Davidson [55], Jouppi [66], and Jouppi and Wall [19]. Their
accomplishments laid the groundwork for dot-product approaches primarily via trace-driven simulation. The models they introduced tend to ignore some of the complications resulting from the dynamic aspects of code interaction as previously mentioned. This provides a major contrast to our approach where we attempt to infer the overlapped work provided in today's superscalar architectures. MacDougall [54], Mangione-Smith et al. [64], and Emma [31] offer later versions of these types of models with slightly empirical slants (although MacDougall's approach also necessitates the use of a simulator). While Emma and MacDougall focus on dot-product formalizations for reasons such as instruction set analysis and measurement techniques, respectively, Mangione-Smith attempts to simplify the scope of analysis (to loop-based scientific kernels only) in an attempt to model pipeline performance. Mangione-Smith's work has some similarities to our work in its isolated focus on scientific codes, but our approaches are somewhat different and their work tends to ignore the influence of memory on performance of their codes since they all fit into cache.

The trace-driven non-layered approaches of Stephens et al. [59], Rauchwerger et al. [60], Kamin et al. [62], Dubey et al. [63] and Noonburg and Shen [56] share some commonality with our instruction-level approach to modeling. None use formalized queuing theory as we do, and none use the approach to cpi breakdown we use that is derived from [22] and [31]. Furthermore, each of these is focused on architecture performance only without offering code analysis (although this could sometimes be derived, but from a macro perspective). The modeling assumptions and the approach to bottleneck estimation apparent in several of these approaches are similar to our queuing-based bottleneck analysis in the instruction-level model. The contribution of
these papers individually is of great importance to these types of modeling techniques. Stephens presents potential parallelism as a metric for quantifying machine and code characteristics. They address pessimistic and optimistic approximations of cpi that have some application in the context of our modeling techniques. This paper is the first in a series of ILP bounding papers where the focus is on quantifying ILP through a comparison of optimal and achievable ILP in the form of a ratio. Rauchwerger, Noonburg, Kamin, and Dubey each offer variations to this bounding approach. Mangione-Smith attempts bounding as well only focusing more on the timing bounds of idealized architecture-code combinations in the context of scientific codes. These approaches tend to focus on modeling instruction streams in ways similar to our queuing-based approach, but they suffer from their avoidance of the issue of memory influence. Our techniques are focused also on instruction-level on-chip performance only, but allow extension and inclusion of our memory modeling methods. In this way, we combine the best of both approaches. Herein lies the strength of our approach. We use a traditional dot-product method to break down performance into recognizable pieces. By separating memory cpi from pipeline cpi, we can isolate the methods of approximation. In other words, we can use traditional queuing theory to approximate pipeline cpi and separately utilize a curve fitting approach to identify overlap performance in memory cpi. This is the strength of our technique, but also its weakness. Because we do a linear fitting for memory cpi, we can focus only on codes that scale in memory performance linearly. This is fine for scientific codes that are primarily loop-based, but for other types of codes this may not be as effective. This is a limitation we accept for now, but plan to concentrate on eliminating in the future. The
instruction-level model portion has no such constraints and can be performed on any type of code.

The memory-modeling portion of our model uses a basic mean-value analysis. There are approaches that extend this type of approach to non-linear approximations. Krishnaswamy and Scherson [65] proposed an analytical method of non-linear approximation of performance vectors. This approach is related to our work as well, in that it is the extreme version of coefficient inference techniques. This is primarily what our memory model does. It attempts to infer the actual latencies suffered after successful superscalar overlap of useful work. Krishnaswamy provides a technique using centroid approximations to achieve a similar type of coefficient inference. The context of this approach is in deriving coefficients for performance vectors that (in their example) are instruction-level elements. The problem with this approach is the loss of accuracy and again, an apparent neglect of memory influence parameters. While this technique could be very useful, it is easy to question the accuracy of such an approach. It attempts to infer all its coefficients, hence the more codes measured the better the accuracy. For these reasons, it requires multiple runs of different codes in an effort to analyze the architecture only. It does not seem useful for application performance analysis.

2.6 Summary

We have established the historic context of superscalar architecture performance analysis that is appropriate for the types of application and architectural analyses presented in this document. Particularly, we have focused on the current motivations for such approaches, their historic usefulness and practical nature. While many
attempts have been accomplished using techniques similar to ours, none provide the same performance picture attainable through our models. Our models provide two important improvements over past and current analysis techniques. Mean-value analysis allows quantification of instruction-level performance overlap. The decoupling of memory and pipeline performance allows formal dissection of cpi through differing modeling approaches incorporating the aforementioned mean-value approach for memory analysis and a queuing-based approach for on-chip performance estimates. Extensions to these basic techniques provide further analysis using statistical approaches and further dissection of cpi to incorporate contributions such as branching and dependences. Furthermore, empirical non-layered approaches such as ours show promise for future applications in dynamic architectures. The next chapter provides detailed discussions of the workloads and testbeds utilized in our experiments.

2.7 References


[36] R. H. Saavedra, "CPU performance evaluation and execution time prediction using narrow spectrum benchmarking," in *Computer Science Division of the..."


Chapter 3

Tools, Testbeds and Workloads

3.1 Measurements

There are three general methods for obtaining empirical performance measurements. These include using software, trace-driven simulators, and performance monitors. Empirical measurements obtained by software typically involve somewhat intrusive processes that monitor system-level interrupts. These approaches suffer by sometimes affecting the code they are trying to evaluate, throwing off measurements through context switches, cache usage, etc. Trace-driven simulators are designed with complete versatility in mind. They allow measurement of just about anything, but at the price of extended execution time. Regardless of this limitation, as previously mentioned they provide a useful means of performance evaluation of simulated architectures. Performance monitors cannot possibly provide the detail of a simulator, but they are inherently non-intrusive and do not add significantly to run-time. They can however be difficult to use and few general models exist to fully utilize the results provided.

Due in part to these pros and cons, non-simulator implementations of these methods typically manifest as hybrids combining both software and performance monitors. CXperf [1] from HP is a good example of a hybrid implementation with limited low-level performance monitoring capabilities that focuses primarily on system-level analysis. This implementation consists of monitoring software interrupts providing inferred results to a user along with very limited use of underlying performance monitors. Implementations of this type are typically not sufficient for low-level modeling due to their intrusive nature and poor accuracy. More complete versions
that utilize many low-level performance counter measurements across multiple platforms would be preferable in our analysis techniques. Trace-driven simulators are an excellent resource for providing low-level control and detail for performance analysis purposes. Unfortunately, the limitations previously discussed for modeling memory coupled with the extended execution time for computationally large problems and their sometimes-limited availability do not make trace-driven approaches optimum for our methods. Furthermore, promised advances such as dynamic code translation and the ability for performance analysis without reliance on source code provides great incentive for monitor-based modeling techniques. Of course, for certain problems accurate measurements from trace-driven simulators could be captured in the same way as performance monitors collect information and results could be used in the models we propose. Our method of choice for data collection is extended use of the performance monitor or counter. We discuss its use in detail in the following section.

3.2 Hardware Monitors

Classically, vendor validation and performance measurements would be accomplished via creation of a special hardware chip/card interface that was physically attached to a processor to provide measurement of certain necessary events. Perhaps due to the complexity increase in today's designs, more and more vendors have begun sacrificing valuable silicon for the purpose of providing special controlled registers with the sole function of capturing performance data in the form of event counts. Such registers provide a non-intrusive method of measuring code performance via selective processor-dependent events. Despite this fact, researchers have yet to completely harness the power of processor performance monitors.
There are several very good reasons for a lack of enthusiasm toward the usefulness of performance monitors. They are typically very difficult to use and require kernel modifications to support existing interfaces. Vendors provide access to on-chip counters only out of courtesy. Performance monitors do not directly contribute to profits. In fact, they are probably quite costly in both silicon and design effort. Interfaces and provided events often go undocumented to the outside world for fear of calls for support. Using performance monitors often requires an underlying knowledge of the hardware. A user must understand exactly what he/she is counting and what the counts mean. To complicate matters, there is no standard set of counted events. PAPI [2], part of the PTOOLS Consortium, and PCL [3], the European version, are attempts to standardize counter events across machines. Vendors are skeptical of such attempts, but some express the desire for such a standard. Whether due to architectural differences or cost limitations, vendors gather different information from their counters and thus provide their own versions of monitors. Performance measurements are not directly comparable, except in rare circumstances. For example, one may be tempted to compare total cycles from one machine to the next for a specific code. This is a trap: these results, even on the same architecture are not necessarily comparable possibly due to different cache sizes, compilers, clock rates, etc. Raw counts then, are not very useful for cross-platform comparison. Analytical, statistical or empirical modeling must be used to interpret resulting information obtained from performance monitors. Hard data measurements are not directly comparable, as mentioned, but extrapolated conclusions from higher level models can allow comparisons among machines with
differing architecture over the same application. Each of our models provides analysis of raw counts to help understand application and architecture performance.

3.3 Current Tools

There are two types of tools that utilize performance monitors. These include high-level analysis tools such as VTune [4] that provide analysis of the underlying data gathered for a particular processor-family implementation. Low-level tools such as prof are available across platforms, but provide only minor analysis of the underlying data and require a proprietary tool to actually gather performance measurements on the counters.

We discuss a sample of performance monitoring tools readily available for event counting on seven representative mainstream processors, namely the MIPS R10000, Alpha EV Family, Intel PPRO Family, the IBM 604e, the HP PA-RISC, the Cyrix 6x86MX, and the Sun UltraSPARC III. In the Appendix, we provide the performance monitor events useful to our models that are available for these processors.

The MIPS R10000 is a 32-bit 4-way superscalar microprocessor that provides optimization techniques such as speculative execution, multiple branch prediction and register renaming [5-7]. It contains two performance monitors. Tools such as Speedshop [8, 9] from SGI provide system level and instruction level analysis via sampling at the system level and use of performance counters, respectively. perfex and libperfex [8, 10] (a library version for calls within code) are provided as part of Speedshop to allow programmers to access performance monitors themselves.

The Compaq Alpha EV Family most recently consists of the 21064 (EV4), 21164 (EV5), and the 21264 (EV6) processors. The EV4 and EV5 are in-order
processors that provide 2 and 3 performance counters respectively [11-14]. The EV6 is a 64-bit 4-way superscalar microprocessor that provides optimization techniques such as speculative execution, multiple branch prediction and register renaming [14, 15]. DCPI [12, 16] is the tool of choice for the EV4 and EV5, but was developed primarily for in-order processor performance monitoring. ProfileMe [12, 17] has recently been developed to monitor the out-of-order EV6 processor with its two counters. It requires additional hardware to obtain full functionality. Both provide combinations of low-level and high-level monitoring using performance counters and sampling combined.

The Intel PPRO processor family includes the Pentium II and Pentium III. They are quite similar in architecture differing mostly in extensions such as MMX and actual interface to the motherboard, so we discuss them generally based on the original Pentium Pro. The Pentium Pro processor is a 32-bit 3-way superscalar microprocessor that provides optimization techniques such as speculative execution, multiple branch prediction and register renaming [18, 19]. For processors running Linux, users may access the two Machine Specific Registers (MSR’s) or performance monitors with tools such as pperf and libpperf [20]. These provide low-level access to performance monitors in the same way as perf and libperf from SGI’s Speedshop. In Windows’ environments, Intel tools such as VTune [4], IPEAK [21], and PCT [22] provide functionality similar in detail to that of Speedshop. They utilize underlying counters and system level monitoring to give the user access to performance measurements of interest at the desired level of detail. Of course, there are major differences with Speedshop since Intel processors provide many more event counts [23, 24].
The IBM 604e is part of the PowerPC Family from IBM. This processor is a 64-bit 4-way superscalar processor that provides optimization techniques such as speculative execution, multiple branch prediction and register renaming [25]. It provides four performance monitors for event counting. On typical AIX platforms, low-level API’s include sPM604e or the PMAPI [26]. IBM is currently reworking its performance tools and so API’s such as these and a few created by fellow users provide the only access to performance counters generally. A new PMAPI is under development and many await its release. Until then, counters for this family of processors can most easily be accessed via freeware available online. GUI tools at the system level are available, but few if any provide access to low-level results from performance counters; most focus on memory access and SMP traffic.

HP PA RISC [27], Cyrix 6x86MX [28], and Sun UltraSPARC IIi [29] processors have few usable tools to take advantage of low-level performance measurements via performance monitors. While each provide some counters, they are very limited in use and most tools are for system level sampling only such as HP’s CXperf [1]. These processors are a poor choice for performance analysis using monitor measurements due to the limited events countable and/or the lack of readily available tools (see Appendix).

Currently, we know of no tools (including of course the tools mentioned) that provide both cross-platform high-level single-processor performance analysis and access to low-level performance monitors. There are lots of problems in using performance monitors across platforms. Portability of a tool would be compromised if every single event able to be counted for a single platform was provided to the user and...
used in analysis methods. In some cases understanding what a particular counter represents is very difficult to decipher without detailed knowledge of the processor and in other cases vendor specific counts have no equal on another processor. Performance counters are platform specific despite attempts such as PAPI [2] and PCL [3] mentioned earlier. For this reason, tools created for a particular platform tend to provide analysis based on the extent to which measurements can be provided for that platform. For example, HP’s CXperf [1] uses software measurements (event sampling) to gather results probably due to limited counter resources readily available on the PA RISC chip. On the other hand, Intel’s VTune [4] is capable of very low-level instruction analysis using its multitude of performance counters.

3.4 Common Problem Set

Provided we are willing to sacrifice a certain level of detailed analysis, we can construct a common problem set of events that are countable using monitors across a usable set of microprocessors. This would not be an attempt to standardize, such as PAPI [2] and PCL [3], but merely an abstraction of the counts necessary for use by the empirical and statistical models in our work. These analysis techniques require automated implementation to take full advantage of the conclusions they provide. Performance counters provide a practical alternative to simulators for such measurements on both computationally intensive codes and codes for which source code is not readily available. Our empirical hierarchy analysis technique promises answers to utilization of current latency hiding techniques. Our instruction-level analysis technique promises bottleneck estimations at the queue level of microprocessor architectures. Our
statistical techniques provide advanced methods of performance isolation. Each of these requires results obtained by performance counters over the defined problem set.

In Table 3.1 we give a problem set necessary for our analysis techniques. The set is generic enough to support at least four, possibly five platforms currently. The downside to such a set is our limited ability to go below instruction-level analysis (to pipeline analysis). We do not consider this a goal of our modeling techniques and discount this fact. However, those desiring such analysis would probably be better served using a simulator or a very advanced, processor-specific performance tool such as ProfileMe [17]. This problem set could conceivably be modified to incorporate more measurements, but for now there are no other counts necessary to our modeling techniques. The Appendix shows the available counters for a cross-section of current microprocessors. These results show single implementations of a cross-platform, counter-based analysis tool are possible for both RISC and CISC architectures using our problem set abstraction.

Table 3.1 Common problem instruction set

| Cycles | clock ticks for a particular cpu |
| Graduated Instructions | instructions that are committed to the program state |
| Loads | accesses by cpu with the result of bringing date into the L1 cache |
| Stores | accesses by cpu to store data in the L1 cache |
| L1 misses | loads that require access to the next level of memory (L2 cache) |
| L2 misses | loads that require access to the next level of memory (memory) |
| Graduated fl pt instr | fl pt instr calculated by the FPU for a processor (not ops) |
| Branches | # of times a branch stmt is decoded giving alternate control flow |
| Branch misses | # times an incorrect branch is chosen resulting in delays |
| TLB misses | # times translation for current address not found in TLB |
| Icache misses | # times instructions are not found in the instruction cache |
3.5 Testbeds

The testbeds utilized throughout this document are primarily of interest to the national laboratory community. The Accelerated Strategic Computing Initiative (ASCI) is a tri-lab project aimed at improving capabilities in nuclear weapon simulation. Machines with the ability to achieve tera-flop computing exist at each of the laboratories. Sandia National Laboratory maintains the Intel ASCI Red computer, an MPP currently utilizing the Intel Xeon processor. Los Alamos National Laboratory has the SGI Origin 2000, an SMP currently utilizing the MIPS R10000 processor. Lawrence Livermore National Laboratory maintains the IBM SP-2 using the IBM RS6000 processor. For many reasons, not the least of which was practicality; our machine testbeds consist of the Sandia and Los Alamos machines, along with other machines readily available. Our performance analysis is in the context of single processor performance, and this mix of machines covers both a RISC and CISC implementation to show the generality of our approaches. We discuss these machines in some detail for completeness, but our models do not cover the network and shared memory aspects of these machines. The reader is referred to [30] for a shared memory extension to the empirical memory model.

3.5.1 MIPS R10000 Machines

The SGI PowerChallenge is an SMP architecture that employs a central bus to interconnect memories and processors [31]. The bus bandwidth (1.2 Gbytes/sec) does not scale with more processors. Cache coherence is maintained through a snoopy bus protocol, which broadcasts cache information to all processors connected to the bus. The SGI Origin 2000, on the other hand, is a distributed shared memory (DSM)
architecture which uses a switch interconnect that improves scalability by providing interconnect bandwidth proportional to the number of processors and memory modules [32]. Coherence is maintained by a distributed directory-based scheme. Figure 3.1 shows a network view of the machine. Each router in the hypercube topology connects two nodes to the network. Each node contains two processing elements and one local memory unit. A 128-processor system, for example, consists of a fifth-degree hypercube with 4 processors per router.

The processing elements of both the Origin 2000 and PowerChallenge systems use a 200MHz MIPS R10000 microprocessor. The processor is a 4-way super-scalar architecture which implements a number of innovations to reduce pipeline stalls due to data starvation and control-flow [7]. For example, instructions are initially decoded in-order, but are executed out-of-order. Also, speculative instruction fetch is employed after branches. Register renaming minimizes data dependencies between floating-point
and fixed-point unit instructions. Logical destination register numbers are mapped to the 64 integer and 64 floating point physical registers during execution. The two programmable performance counters track a number of events [8] and were a necessity for this study. The most common instructions typically have one- or two-clock latencies.

While the processing elements of the PowerChallenge and Origin 2000 systems are identical, there are major differences in the memory architecture and corresponding performance of the two systems. The PowerChallenge is an UMA architecture with a latency of 205 clocks (1025 ns). Latencies to the memory modules of the Origin 2000 system, on the other hand, depend on the network distance from the issuing processor to the destination memory node. Accesses issued to local memory take about 80 clocks (400 ns) while latencies to remote nodes are the local memory time plus 33 clocks for an off-node reference plus 22 clock periods (110 ns) for each network router traversed. In the case of a 32 processor machine, the maximum distance is 4 routers, so that the longest memory access is about 201 clocks (1005 ns) which is close to the uniform latency of the PowerChallenge. This unique feature of Origin 2000 systems provides us a good opportunity to adjust the memory access latency by placing memory and execution thread on different nodes.

In addition, improvements in the number of outstanding loads that can be queued by the memory system were made. Even though the R10000 processor is able to sustain four outstanding primary cache misses, external queues in the memory system of the PowerChallenge limited the actual number to less than two. In the Origin 2000, the full capability of four outstanding misses is possible. The L2 cache sizes of these
two systems are also different. A processor of PowerChallenge can be equipped with up to 2MB L2 cache while a CPU of Origin 2000 system always has a L2 cache of 4MB.

3.5.2 Intel Machines

The Intel ASCI Red machine originally used the 200Mhz Pentium Pro microprocessor. Following an upgrade in early 1999, all these chips were replaced with Pentium II Xeon microprocessors. The Pentium II Xeon architecture uses the Pentium Pro core architecture with some additions including MMX, larger and faster cache, faster clock rate and a revised IC package on a processor card. Our modeling techniques focus on architectural characteristics mostly within the core (except for cache sizes), meaning the specified parameters for our models are applicable to any Pentium Pro based architecture. The models themselves are general, what we mean here is that the same parameters can be entered for the original Pentium Pro and the Pentium II Xeon. This means our models work equally well for Pentium Pro or the Pentium II Xeon. Since the upgraded machine uses the Pentium II Xeon, we will describe the architecture in this context.

The ASCI Red Supercomputer is a Massively Parallel Processor (MPP) with a distributed memory Multiple-Instruction, Multiple Data (MIMD) architecture [33]. All aspects of this system architecture are scalable, including communication bandwidth, main memory, internal disk storage capacity, and I/O [34]. The ASCI Red maintains communication through an Interconnection Facility (ICF) in a 38x32x2 topology with a peak (sustainable) bi-directional bandwidth of 800 MB/sec [33]. A Kestrel board holds two compute nodes connected through a Network Interface Chip (NIC) and attached to
a Mesh Router Chip (MRC). The memory subsystem on an individual compute node is implemented using the Intel 82453 Chipset with 128 MB/node.

ASCI Red is composed of 9,216 processors providing 4,536 compute nodes. Each compute node consists of two 333 MHz Pentium II Xeon Processors. The 333 MHz Pentium II Xeon processor is a 3-way super-scalar architecture that reduces pipeline stalls utilizing features such as out-of-order execution, speculative execution of branches, and register renaming. Two programmable performance counters are also available, providing the data used in our studies. Each processor includes separate 16KB data and instruction caches along with 512KB secondary L2 cache. This L2 cache is located on a separate die in the same package closely coupled via a dedicated 64-bit full-clock-speed backside cache bus. The L1 data cache can handle as many as four outstanding misses and has a miss latency of three cycles, whereas the L2 cache miss latency is about 50 cycles [18]. Only one CPU on a node is used in our experiments.

3.5.3 ASCI Codes

Four applications that form the building blocks for many nuclear physics simulations were used in this study. A performance comparison of the Origin and PowerChallenge architectures has been done using these codes [35].

SWEEP3D is a three dimensional solver for the time independent, neutral particle transport equation on an orthogonal mesh [36]. In SWEEP3D, the main part of the computation consists of a "balance" loop in which particle flux out of a cell in three Cartesian directions is updated based on the fluxes into that cell and on other quantities such as local sources, cross section data, and geometric factors. The cell-to-cell flux
dependence, i.e., a given cell cannot be computed until all of its upstream neighbors have been computed, implies a recursive or wavefront structure. The specific version used in these tests was a scalar-optimized "line-sweep" version [36] that involves separately nested, quadrant, angle, and spatial-dimension loops. In contrast with vectorized plane-sweep versions of SWEEP3D, there are no gather/scatter operations and memory traffic is significantly reduced through "scalarization" of some array quantities. Because of these features, L1 cache reuse on SWEEP3D is fairly high (the hit rate is about 85%). A problem size of N implies N^3 grid points. DSWEEP is a vectorized version of the same code.

HYDRO is a two-dimensional explicit Lagrangian hydrodynamics code based on an algorithm by W. D. Schulz [37]. HYDRO is representative of a large class of codes in use at the Laboratory. The code is 100% vectorizable. An important characteristic of the code is that most arrays are accessed with a stride equal to the length of one dimension of the grid. HYDRO-T is a version of HYDRO in which most of the arrays have been transposed so that access is now largely unit-stride. A problem size of N implies N^2 grid points.

HEAT solves the implicit diffusion PDE using a conjugate gradient solver for a single time-step. The code was written originally for the CRAY T3D using SHMEM. The key aspect of HEAT is that its grid structure and data access methods are designed to support one type of adaptive mesh refinement (AMR) mechanism, although the benchmark code as supplied does not currently handle anything other than a single-level AMR grid (i.e. the coarse, regular level-1 grid only). A problem size of N implies N^3 grid points.

47
NEUT is a Monte-Carlo particle transport code. It solves the same problem as SWEEP3D but uses a statistical solution of the transport equation. Particles are individually tracked through a three dimensional mesh where they have some probability of colliding with cell material. The output from the particle tracking is a spatial flux discretized over the mesh. Vector (or data parallel) versions of this type of code exist which track particle ensembles rather than individual ones. A problem size of \( N \) implies \( N^3 \) grid points and 10 particles per grid point.

### 3.5.4 Codes for Architectural Evaluation

In the results chapter of this work, we will present some architectural enhancements suggested by our modeling techniques to show their usefulness. In the architecture community it is necessary to use well-accepted codes when comparing the performance of new architectures. The scientific ASCI codes are used in many of our performance studies since laboratory personnel typically reference results. In the interest of publishing within the architecture community and to receive constructive criticism, we discuss the new architecture implementation in terms of achievable performance on SPEC codes. SPEC codes are created and provided by the Standard Performance Evaluations Corporation (SPEC) in order to allow independent confirmation of vendor performance claims. Unfortunately, many vendors target SPEC code performance when developing new architectures since results become high profile if measured performance is good. We will briefly mention the codes we use from this suite.

For architectural performance comparisons, we use 3 integer and 3 floating point applications from the SPEC95 benchmark [38] plus kmeans [39]. Kmeans is a laboratory code that utilizes an iterative clustering algorithm. Clustering algorithms are
often used in image processing or computer vision applications. The SPEC95 integer applications include compress (compresses and decompresses files in memory), li (a LISP interpreter), and ijpeg (graphics compression and decompression). The SPEC95 floating point applications include swim (shallow water model with a 513 x 513 grid), su2cor (quantum physics Monte Carlo simulation), and wave5 (plasma physics electromagnetic particle simulation). For Spec95 applications, we use the associated training data sets since we are primarily concerned with on-chip performance. For kmeans, we use “-D3 -N10000 -K30 -n50” as the parameters.

3.6 Workload Characterization

It was necessary to be certain that the codes we are interested in studying meet certain criteria. For instance, we define a parameter as the average distance between two like instructions in an executing instruction stream. For the ASCI scientific codes we want to be sure these distances are convergent on average. For SPEC codes (and kmeans), we want to make sure our coverage of certain performance enhancements is warranted for the codes under study. We have already mentioned how we characterize the quality of a particular architecture by how well it supports our minimum problem set. The results are given in the Appendix. In this section, we present three separate workload characterization studies accomplished to identify code properties important to our techniques or performance improvements.

3.6.1 CPI Characterization of ASCI Codes

There are certain assumptions in the empirical memory model that are necessary for simplified modeling. More importantly, we find the ASCI codes have characteristics common among scientific codes such as high branch prediction. In this section we
present some single-processor characteristics of the benchmark codes as obtained from performance counters on the MIPS R10000 for the Origin 2000 and PowerChallenge machines previously mentioned. Note that the maximum MFLOPS observed may, in some cases, be obtained from unreasonably-small problem sizes relative to actual production runs; the data are presented here merely as a reference for the normalized Mflop curves in Figures 3.2-3.6. Detailed performance characteristic data for these codes were collected on a 2-MB L2 PowerChallenge system and a 4-MB L2 Origin2000 system. Performance data as a function of problem size for the Power Challenge and Origin are illustrated in Figures 3.2-3.6. MFLOPS curves are normalized such that the maximum rate for each code is one.

The codes' overall cpi curves are generally the inverse of their corresponding MFLOPS curves; that is, an increasing cpi corresponds to a decreasing MFLOPS at nearly the same slope and vice versa. The cpi of three of the codes (HEAT, HYDRO and SWEEP) is strongly dependent on problem size. The figures show that normalized MFLOPS curves (except for HYDRO-T) follow the tendencies of the L2_hit curves. On the PowerChallenge system, a drop in L2_hit rate causes much more impact to MFLOPS than it does on the Origin system. This is due to lower memory latency (both actual and effective) on the Origin2000 system. Although not shown in the figures, we calculated TLB hit ratio and branch prediction hit ratio. The calculation shows that MIPS R10000 processor can do a good job of speculative branch prediction. All four benchmark codes (HEAT, HYDRO, HYDRO-T and SWEEP) have branch prediction hit ratios over 99%. This means that over 99% of speculated branch predictions are
Figure 3.2 Performance of HEAT as a function of linear problem size. Right axis shows cache hit rates and normalized megaflops.

Figure 3.3 Performance of SWEEP as a function of linear problem size.

Figure 3.4 Performance of HYDRO as a function of linear problem size.
3.6.2 Steady-State Characterization of ASCI Codes

We will save formal definitions to be presented in the context of the model discussions later in this document. For the instruction-level model of on-chip performance, we are generally interested in the average distance (in number of instructions) between two instructions of the same type. In this workload characterization, we want to be assured
that the average distances achieve a steady state to allow modeling based on this parameter. For this reason, we measure these values (on average) using the performance counters of the MIPS R10000 and plot them verse problem size for the three primary types of instructions that make up most of the performance of these codes (i.e. integer, floating point, and memory operations). The fact that these are the primary contributors to performance is measured as well, but not presented here. This involves a count of the total instructions executed verse the summation of each of the identified types. For all of our codes, these instructions account for roughly 99% of the associated performance.

Figures 3.7-3.11 present the results of this study. As can be seen in every case, average arrival distances of like instructions converge to fairly consistent values as problem sizes increase. This confirms our suspicion that these values will be useful in modeling performance as problem size increases to the point of steady state. Minor fluctuations at small problem sizes are easily explained as anomalous due to cold misses as caches are yet fully utilized. Performance levels out as problem sizes begin utilizing the cache more regularly.

3.6.3 Simulator Characterization of SPEC Codes

Another method of analyzing codes for instruction distribution involves counting the number of occurrences of certain distances between instructions. This describes the "clustering" of instructions by type. If certain occurrences happen often enough, they become a bottleneck to performance due to limitations on service on a given microprocessor. This type of analysis requires the use of a simulator for profiling the code. We use the SPEC codes because results of this analysis will help dictate the
Figure 3.7 Instruction distances for Heat on O2K and PC

Figure 3.8 Instruction distances for Sweep on O2K and PC

Figure 3.9 Instruction distances for Dsweep on O2K and PC

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
functionality of our modeling-inspired architectural enhancements to the original MIPS R10000 design. While the modeling methods we discuss later inspired the changes we suggest, this workload characterization quantitatively confirms our inclinations as to the properties of these codes with respect to instruction clustering. What follows is a discussion of the results in the context of on-chip architecture for the measured SPEC codes using the SimpleScalar simulator tool-set [40].
Later in this document we will present a method for obtaining qualitative conclusions regarding the loss of performance due to instruction and functional unit mismatch. A major conclusion is that significant performance gain is possible using architecture that can change its functional unit allocation dynamically. In developing the hardware logic necessary to implement such a dynamic architecture, it is necessary to minimize the subset of "important" instructions. "Important" in this context means those instructions that will have primary influence on the performance of applications of interest.

A processor services instructions. Instructions enter the processor, and are eventually committed to program-state. But the processor has a limited amount of resources available on-chip. The functional units themselves are typically hardwired allowing only a finite number and type of instructions to be executed per cycle resulting in stalls if they are overwhelmed. Furthermore, stalls resulting from this mismatch and of course memory latency cause instructions to be backed up to the fetch/decode stage. This again results in stalls on-chip since only a finite number of instructions can be active at any one time due to limits in registers, queue sizes, etc. For any particular processor, these limitations vary.

So, contemplating the qualities of a typical processor, we measure the distance between consecutive instructions of each type. In other words, we directly count the number of instructions between two identical instruction types. Why is this interesting? The frequencies within the instruction stream itself determine the number of times a certain distance between a certain type of instruction occurs, thus giving a good representation of the original application. Furthermore, assuming an architecture uses
the same instruction set and compiler, such a profiling scheme is comparable across architectural improvements to the physical limitations previously described. Lastly, this approach directly quantifies the qualities that affect this instruction and functional unit mismatch. Particularly, we want to be able to focus on instructions that exhibit high frequencies of small distances between like instructions. Such instructions will inevitably have an adverse effect on performance since static functional unit allocation will result in on-chip stalls.

By providing quantitative comparisons between each instruction type measured, we can directly compare all instruction types for a particular code. Also, we can highlight the most "important" instructions for the codes measured and compare the codes themselves. Not surprisingly, the same instructions tend to be "important" across codes while magnitudes will vary. By augmenting the profiling capabilities of the SimpleScalar tool-set [40], which simulates a MIPS R10000 architecture, we are able to measure inter-arrival distances between instructions. We view the committed instruction sequence as a sequential stream of instruction types that are executed by the processor. This stream is the entity we analyze.

We profile the instruction stream as follows: if we encounter an integer-add instruction, we count the number of other instructions that occur prior to the next occurrence of an integer-add instruction. We keep track of the number of times distances of this length occur and plot length on the x-axis and number of occurrences on the y-axis of our graphs. Figures 3.12-3.18 show the resulting sets of most frequent instructions for all the codes of interest.
Figure 3.12 Instruction distance occurrences for Swim

Figure 3.13 Instruction distance occurrences for Wave5

We first utilize this technique to provide a list of the most frequently occurring clusters of instructions. For floating-point intensive applications, namely Swim, Wave5 and Su2cor, the list of significant instructions is similar to the integer intensive codes
Compress95, jpeg, li, and k-means. These are the "important" instructions for these particular codes. A reconfigurable unit that provides support for these types of
Instructions is likely to achieve performance gain provided the switching penalty is minimal. These instructions are listed in each of the figures.
Figure 3.18 Instruction distance occurrences for kmeans

Integer-add operations are quite common among all the codes. This is expected in integer-intensive codes, but perhaps the magnitude of their presence in floating-point intensive codes is not so intuitive. Nonetheless, the floating-point codes Swim, Wave5, and Su2cor each show frequency distributions that outweigh their floating-point add counterparts significantly. This shows that a reconfigurable unit providing extra bandwidth to integer-add operations should provide a performance boost. Also, the penalty incurred by switching from integer-add to floating-point add resulting in cycle delay could be canceled out by the gain in integer performance afforded by a reconfigurable unit. In other words, a tradeoff is possible between switching penalty and integer bandwidth performance gain since the quantity of these integer operations is typically two or three times larger than the quantity of floating-point add operations.

This discussion provides the motivation behind our choices of including and excluding functionality for the reconfigurable unit. Particularly, the goal is to provide

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
extended integer execution bandwidth while maintaining the power provided from reconfiguring as a floating-point unit.

3.7 Summary

The focus of this chapter was the preliminary work necessary to understand the codes and machines for which we wish to analyze and develop performance analysis methods. We discussed in detail the functionality of performance monitors, their availability on commodity processors, and a common problem set for cross-platform-based analytical methods. We provided the details of the SGI Origin 2000 and PowerChallenge machines and the Intel ASCI Red machine. The processors under scrutiny are the MIPS R10000 and the Intel PPRO-based Xeon, respectively. Finally, we developed workload characterization methods to initially characterize the scientific codes used in our studies. These studies are of great importance since they provide the basic arguments for the characteristics of our models. We discussed and gave results for three separate methods that give the underlying concepts necessary for successfully modeling our applications. In the next two chapters we'll use our workload characterization results to create and apply our empirical and statistical models.

3.8 References


62


[31] Silicon Graphics Computer Systems, M. Galles and E. Williams, "Performance optimizations, implementation, and verification of the SGI Challenge


Chapter 4
Performance Analysis Methods

4.1 The General CPI Model

There are two parts to the general cpi model. The two parts focus on separate aspects of application performance. The memory model attempts to infer the average stall times incurred after the effects of latency hiding have been taken into account. The instruction-level model attempts to quantify the on-chip performance without the effects of memory influence. Together they attempt to provide users with more information about the performance of code as it interacts with the underlying hardware. We begin with a discussion of the general approach to this type of modeling, followed by the individual models themselves along with some validation work. We should note that the two separate models should be thought of as works in progress with the effective goal of complete analysis of the on-chip and off-chip performance. Future work involves providing more functionality to these models at both levels. For now the models provide useful performance analysis for scientific applications.

Typically, the overall execution time of an application is the sum of the computational time (on-chip), the time spent accessing memory (off-chip to memory), the communication time between nodes (parallel processing or client-server context) and the time spent on I/O (time to disk or output). Each of these pieces of execution time has warranted large amounts of research. In the context of this document, we will focus only on computational time (on-chip) and time spent accessing memory (off-chip) for applications executing on a single processor. As will be apparent, these problems are complex enough to support years of useful research.
Now that we are focusing on overall execution time for on-chip computation and memory accesses, we can use a fairly common formula to express the execution time of a particular application. Following [1], we express time as the following product:

\[
\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}
\]  

(4.1)

Following Bhandarkar and Clark in [1], we can describe the terms of this equation in regard to the system aspects that influence them. The number of instructions resulting from a program is a function of the compiler and instruction set architecture (ISA). The cycle time is a direct function of the underlying VLSI technology and architectural design (such as degree of pipelining, ISA, etc.). The cycles per instruction is a function of many things including the architectural design and the compiler (in essence the code). So the cycles per instruction (or cpi) gives an indication of the interaction between code and architecture performance. Furthermore, the cpi is related to the achieved instruction-level parallelism of a particular code-architecture combination. In fact, achieved cpi is of great interest at the beginning stages of the architectural design process [2-4]. So by focusing on cpi values, we can infer performance differences for different codes on the same architecture and for the same codes on architectures with the same ISA but minor enhancements. We will use cpi to compare the achievable ILP of particular code-machine combinations throughout this document.

Most performance analyses using cpi values have the objective of evaluating the architecture only. As a result, models of this type do not typically break the term itself down any further than overall cpi. But, the ILP (and hence cpi) of a program varies greatly across the duration of a program [5]. We feel that great insight can be gathered into application and architecture performance if we break down cpi into contributing

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
pieces. Following [6] and [7], we initially break cpi down into two parts corresponding to the pipeline and memory cpi.

\[ cpi = \text{pipeline cpi} + \text{memory system cpi} \]  

(4.2)

If we decouple memory and pipeline cpi we can focus on individual contributions to the overall cpi. Another advantage to this approach is that by separating the two terms, we can derive models that independently attempt to model each piece. This can lead to an iterative design process allowing us to replace obsolete models with more accurate ones or updated versions reflecting new architectural changes. Furthermore, as Emma adeptly describes in [7], cpi is intuitive in nature when we try to explain performance degradation in terms of lost instruction-level parallelism. In fact, Emma alludes to the development of models that dissect cpi into even more terms describing pieces of the overall cpi. Emma's paper provides an exhaustive discussion on the properties and usefulness of cpi modeling formulations.

4.1.1 Empirical Memory Model

To analyze the memory system cpi of Equation 4.2, we use a simplified mean value parameterization [8] to separate CPU execution time from stall time due to memory loads/stores. Figure 4.1 is a pictorial description of the times in the model. The model projects the overall cpi of an application as a function of CPU execution time and average memory access times.

\[ cpi = cpi_0 + \sum_{i=2}^{n\text{levels}} h_i * t_i \]  

(4.3)
where cpi₀ is defined to be the cpi of the application assuming that all memory accesses are from an infinite L1 cache and take 1 CP (i.e. the i=1 term is included in cpi₀), and hᵢ and tᵢ are, correspondingly, the hits per instruction and average non-overlapped access times for the iᵗʰ level in the memory hierarchy. Measured access times at the iᵗʰ level correspond to access time from level i to the registers. The second term of Equation 4.3 is also referred to as cpi_stall or memory system cpi from Equation 4.2 (the first term, cpi₀, is equivalent to pipeline cpi in Equation 4.2). If no overlap of CPU execution and memory accesses occur, every memory access to the iᵗʰ level incurs the full round-trip latency, which we denote as Tᵢ. We define (following definitions by Larson at SGI) a measure of the overlap of memory accesses with computation as m₀, where
\[ cpi = cpi_0 + (1 - m_0) \sum_{i=2}^{n\text{levels}} h_i \cdot T_i \]  

(4.4)

and \( m_0 \) is one minus the ratio of the average memory access time to the maximum memory access time:

\[ m_0 = 1 - \frac{\sum_{i=2}^{n\text{levels}} h_i \cdot t_i}{\sum_{i=2}^{n\text{levels}} h_i \cdot T_i} \]  

(4.5)

We note here that the separation of computational time from memory access time in this model implies that the two can be treated independently (i.e. that \( cpi_0 \) is constant). In fact, the out-of-order execution of the R10000 processor means that different dynamic instruction sequences will be seen for different size problems. For the codes examined and for representative kernel-codes in general that increase in computational iterations as problem sizes increase, thus making them easier to model, a constant \( cpi_0 \) is maintained and \( cpi \) is primarily affected by the memory hierarchy -- satisfying our criteria. The effect of increasing the round-trip memory latency to \( dT_m \) is depicted in Figure 4.2. Once the latency hiding ability of the architecture on a particular code has been exhausted, any additional main memory latency will simply add to the non-overlapped time \( t_m \).

In this case, the new \( cpi \) (from Equation 4.3, where the sum is over the L2 cache and main memory) will be:

\[ cpi' = cpi_0 + h_2 t_2 + h_m (t_m + dT_m) \]  

(4.6)

This equation predicts a linear relationship between \( dT_m \) and slope \( h_m \). If any additional memory latency incurred by \( dT_m \) can be hidden, the increase in \( cpi \) will be strictly less
than that predicted by Equation 4.6. That is, the relationship is an upper bound for the increase in time due to memory latency. Of particular interest is the model's use in quantifying the effect of memory latency on cpi. It can also be used to separate individual contributions of latency hiding techniques in an empirical manner.[9]

4.1.2 Validation of the Empirical Memory Model

Validating such a model is a difficult task at best. The results obtained by the model cannot currently be measured using hardware counters or other direct means. At first we used indirect methods such as manipulating where memory resides in the SMP to predict performance and thus indirectly validate the model. Critics did not like this indirect method of validation, so we endeavored to directly validate the model using the SimpleScalar simulator with some modifications to meet our assumptions. The result was direct validation of the memory model along with insight into its overall accuracy.

First, we made modifications to simulate the MIPS R10000 processor. The modifications were made under consultation with Daniel Citron, a SimpleScalar expert. Yan Solihin at LANL is also to be thanked for his modifications to the simulator. Primarily we made sure we matched the architecture of the current MIPS R10000 processor precisely. Second, we inserted instrumentation to count the number of cycles stalled due to data cache misses. Since the R10000 is a superscalar processor, such that the processor is able to hide some cache miss latencies, we need a special method of calculating this stall time. The method that we use to calculate the stall cycles at the commit stage is the same method used by the RSIM processor simulator [10]. If during commit the processor cannot commit as many instructions as the commit width, we can observe which instruction cannot be committed. If the instruction is a memory
instruction (load or store) and it has caused a cache miss, we increase the stall cycle by the number of wasted commit slots:

\[ \text{stall} = \frac{\text{committed}}{\text{commitwidth}} \]  

(4.7)

For example, if the commit width is 4 and during that cycle 2 instructions preceding a stalled memory instruction can commit, we increase the stall cycle by 2/4. Furthermore, we categorize the stall cycles into stalls due to L1 cache miss (L1stall), and stalls due to L2 cache miss (L2stall). Since we also know the number of cycles (cycles), graduated instructions (inst), misses in the L1 cache (L1miss) and L2 cache (L2miss), we can use these to calculate cpi_o, t_2, and t_m:

\[ \text{cpi}_o = \frac{\text{cycles} - \text{L1stall} - \text{L2stall}}{\text{inst}} \]  

(4.8)

\[ t_i = \frac{\text{L1stall}}{\text{L1miss} - \text{L2miss}} \]  

(4.9)

\[ t_m = \frac{\text{L2stall}}{\text{L2miss}} \]  

(4.10)

There are a few things to note: first, we ignore the instruction cache misses in our calculations. This is not a problem since instruction cache misses account for a very small portion of total cache misses in scientific codes [11]. Second, t_2 and t_m may be larger than isolated cache miss latencies (T_2 and T_m) due to the effect of TLB misses. For example, although a hit on the L2 cache should only take a few cycles, for a TLB miss the penalty may be as high as 100 cycles. The accuracy of the method is demonstrated by calculating t_2 and t_m of lmbench [12] and comparing them with the specified parameters of the simulator. Lmbench is basically a micro-benchmark that accesses array elements in a specified stride. By controlling the size of the array and the
Table 4.1 Accuracy of the validation method

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Calculated</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_2$</td>
<td>11.35</td>
<td>11.86</td>
</tr>
<tr>
<td>$t_m$</td>
<td>81.68</td>
<td>80.68</td>
</tr>
</tbody>
</table>

Table 4.2 Validation of $t_2$ and $t_m$ for the memory model

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sweep</th>
<th>Hydro</th>
<th>Hydro-t</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Computed</td>
<td>Predicted</td>
<td>Computed</td>
</tr>
<tr>
<td>$t_2$</td>
<td>8.5669</td>
<td>9.289</td>
<td>8.7626</td>
</tr>
<tr>
<td>$t_m$</td>
<td>48.0369</td>
<td>46.5925</td>
<td>29.7071</td>
</tr>
</tbody>
</table>

By setting `lmbench` to produce a cache miss on every access, we can predict the cache miss penalty in isolation ($T_2$ and $T_m$) and compare these with the actual $T_2$ and $T_m$ supplied as parameters to the simulator. Table 4.1 shows the numbers calculated using Equations 4.9 and 4.10 for the simulator output and the actual $T_2$ and $T_m$. Supplied parameters are $T_2 = 11$, $T_m = 80$. The numbers in Actual are somewhat different because we take into account array access wraparounds and TLB misses. Overall, the numbers are very close, with 4.3% error for $T_2$ and 1.2% error for $T_m$, which is mostly caused by ignoring instruction misses in the calculation. Thus, $t_2$ and $t_m$ measurement by the simulator is reasonably accurate for our purposes.

The model first predicts $cpi_0$, then $t_2$ and $t_m$, using separate techniques. Our purpose is to validate the $t_2$ and $t_m$ prediction. To do this, we first use the average value of $cpi_0$ as a constant $cpi_0$ as problem size increases to predict $t_2$ and $t_m$. Then we compare the predicted values with the values output by the simulator.
Three Los Alamos scientific applications: sweep, hydro, and hydro-t are validated. The results are shown in Table 4.2. The table shows that the predicted \( t_2 \) and \( t_m \) are very close to the values computed by the simulator. Thus, we have validated the memory hierarchy model, showing \( t_2 \) and \( t_m \) predictions within 10% of the average computed values of \( t_2 \) and \( t_m \) output by the processor simulator.

To get the most accurate \( t_2 \) and \( t_m \) prediction, the applications must have sufficiently large data set sizes (defined as the size of the working set of the applications, also referred to as problem size). In addition, a loop-based application needs a large number of iterations. The data set size of the applications must overflow the L2 cache so that we get steady values for \( t_m \), while the number of iterations must be large enough for the value of \( cpi_0 \) to converge. These qualities are corroborated on the actual machines by our previously discussed workload characterization, and the performance results presented later.

4.1.3 Instruction-Level Model

The empirical memory model gives insight to the performance of the hierarchical memory scheme. As we saw in Equation 4.2, this tackles modeling of the memory system cpi term. Our research on the empirical memory model highlighted the importance of modeling the pipeline cpi as part of the general cpi model. The instruction-level model is a first attempt at such a model [13]. As discussed in Chapter 2, the instruction-level model makes certain assumptions that are fairly common among other attempts at modeling the pipeline cpi. The main differences between our approach and previous approaches involve the use of performance monitors and elementary queuing theory in our models. Saying our approach is better or worse is an
academic argument. In truth, ours is simply different. Providing some conclusions similar to other approaches, but not depending on simulators for the results. Furthermore, the formalization using queuing theory provides a more substantial basis in accepted mathematical theory. As queuing theory is widely accepted in mathematical circles, we feel some of the performance explanations become more intuitive. Moreover, by using queuing theory, we minimize the validation efforts to simply whether or not our resulting models are able to acceptably model performance. Another advantage is its relation to the general cpi model, allowing it to be used in conjunction with the memory model and (as will be shown) the hybrid model. Because of the mathematical nature of this model, we discuss the parameters and implementation in terms of a series of definitions.

Allow an instruction stream to be represented as a series of instructions, \( I = \{i_1, i_2, i_3, \ldots, i_n\} \). Let the following define the set \( I \) even further: \( n \) = total number of instructions; \( k \) = total number of instruction types; \( T = \{t_1, t_2, t_3, \ldots, t_k\} \). The set \( I \) is such that each element in \( I \), \( i_j \), is an element of one and only one type from the set \( T \). There are two properties followed by these expressions. The first is that there exists a finite set of instruction types. The second is that each instruction in the instruction stream, \( I \), is only one of these types.

Let \( P = \{p_1, p_2, p_3, \ldots, p_k\} \) express the probability distribution of each corresponding instruction type in set \( T \). \( \sum_{i=1}^{k} p_i = 1 \) must hold since these represent the probability distribution of the instruction stream \( I \).

Following Kleinrock [14], we revise some of these terms further to coincide with standard terminology for queuing theory. Let \( C = I \), such that every element \( i_j \) of \( I \)
is equal to the corresponding element $c_j$ of $C$ ($C$ is a simple copy of $I$). This is common in queuing theory since elements entering a queuing system are typically referred to as customers. Now, for each element $c_j$ of $C$, we define $\tau_j = j$ as the associated arrival of element $c_j$ of $C$. Thus, the arrival of instructions is dictated by the ordering of elements in $C$ and the associated arrival value, $\tau_j$, for a particular element. Having established the initial parameters for our revised version of Kleinrock's queuing theory approach, we can begin to create definitions that will contribute to explaining the instruction-level model.

*Inter-arrival distance* ($d$): We define the inter-arrival distance between two consecutive instructions $(c_a, c_b)$ of the same type ($c_a \in t_c$, $c_b \in t_c$) as $d = \tau_b - \tau_a$. This implies there are $d-1$ instructions between $c_a$ and $c_b$ that are not of type $t_c$. We refer to $c_a$ and $c_b$ as an adjacent pair of type $t_c$.

*Average inter-arrival distance* ($\bar{d}$): We define the average inter-arrival distance for all instructions of type $t_c$ in $C$, as the average inter-arrival distance of all adjacent pairs of type $t_c$ in $C$. $\bar{d}$ is the average number of instructions that occur following an instance of type $t_c$ up to and including the next instance of type $t_c$.

*Service time* ($x$): We define the service time of an instruction, $c_a$ of type $t_c$, as the number of cycles necessary to fully execute $c_a$.

*Average service time* ($\bar{x}$): We define the average service time of an instruction, $c_a$ of type $t_c$, as the average number of cycles necessary to fully execute an instruction of type $t_c$.

We now have a sufficient number of definitions and terminology to describe the instruction-level model. Since queuing theory generally uses arrival times as wall clock
time, where customers arrive at certain rates that are determined by the associated wall
clock arrival time as they enter the system, we need a time scale that is applicable to
microprocessors. First, we cannot use a running timer, our system's overall time is
predefined as a total number of cycles; this can then be converted into seconds based on
the clock rate of a particular processor. We are more interested in the performance on a
cycle by cycle basis, in other words the achieved ILP. By using an arbitrary definition
for a cycle (i.e. one that does not have an associated nano-second duration), we can
compare machines with the same ISA, but different clock rates through their instruction
level parallelism rather than their duration. This is similar to the established argument
for comparing cpi values found earlier. So, we use cycle as our atomic unit during
which arrivals can take place. The important difference here (with that of normal
queuing theory) is that we are not slaves to time, making our comparisons in terms of
ILP instead of duration.

This being said, with our atomic parameter cycle, there is an associated
maximum arrival rate we define as \( \beta \). In other words, a maximum of \( \beta \) instructions per
cycle can enter the queuing system. This parameter is given typically as the achievable
superscalar width of a processor. The associated probabilities \( P \) of different instruction
types of the instruction stream \( I \), determine the composition of \( \beta \) at each succeeding
cycle for the model.

\( \lambda \) is the number of instructions (or customers) that are introduced into the system
every cycle. For instructions of type \( t_c \), \( \lambda_c = \beta p_c \). \( p_c \) is determined by the probability that
the next instruction is of type \( t_c \). \( \bar{d} \) is the number of instructions that occur following an
instruction of type \( t_c \) while looking for the next occurrence of the same type. So, \( 1/\bar{d} \) is
the probability $p_c$ of encountering an instruction of type $t_c$ on average. We thus revise our original definition of $\lambda = \beta p_c = \beta / \overline{d}$ . We now require more definitions.

**Utilization factor ($\rho_c$):** We define the utilization factor as the average arrival rate of the customer to the system ($\lambda_c$) times the average service time ($\overline{x_c}$) divided by the number of potential servers ($s_c$). Here $s_c$ is the number of servers available at the $\overline{x_c}$ service time rate in the system. On a processor these are the number of functional units for a particular instruction type. To calculate $\rho_c$, we use:

$$\rho = \frac{\lambda \overline{x_c}}{s_c} = \frac{f \overline{x_c}}{d \cdot s_c}$$  \hspace{1cm} (4.11)

**Limiting factor ($t_c$ where $\rho_c > 1$):** The limiting factor of the queuing system describing a microprocessor is instruction type $t_c$ associated with the highest utilization factor greater than one. $\rho_c < 1$ indicates type $t_c$ instructions do not fully utilize the system resources. $\rho_c = 1$ indicates type $t_c$ instructions fully utilize the queuing system. $\rho_c > 1$ indicates saturation of the systems resources by type $t_c$ instructions.

Once the limiting factor is established (say type $t_c$), the throughput of instructions is bottlenecked by instructions only (assuming uniform distribution, perfect cache, no branch influence, no dependence influence, no icache misses) and is thus limited by this instruction type throughput combined with the probability an instruction of type $t_c$ will occur. We call this value ideal CPI $0$.

$$\text{ideal CPI}^0 = \left( \frac{1}{d^0} \right) \left( \frac{\overline{x_c}}{s_c} \right) = p_c \cdot \frac{\overline{x_c}}{s_c}$$  \hspace{1cm} (4.12)

In the next section, we will discuss the direct validation of this portion of the model with the given assumptions. This portion only estimates ideal CPI, however, and we must discuss what portion of the pipeline CPI this estimates. Equation 4.12 gives us our
first glimpse of the performance limitations based on functional-unit and instruction mix mismatch. Results presented later, give the first clues to architectural enhancements alluded to in the simulator-based Workload Characterizations of Chapter 2.

4.1.4 Instruction-Level Model Validation

To validate our model, we chose to use synthetic codes on real processors using hardware performance counters to provide necessary counts as inputs. In this way, we hope to underscore the practicality of our modeling technique and the time saved using our characterization method. The modeling technique discussed so far is general in nature and easily modified for different architectures.

Both the Origin 2000 (O2K) and PowerChallenge (PC) use the MIPS R10000 RISC based microprocessor as discussed in Chapter 3. The R10000 processor is a 4-way superscalar CPU with an integer, floating point, and memory queue each containing 16 entries. Ignoring branch and icache effects, stalls during execution are typically attributed to: 1 of 3 queues full, outstanding misses full (4 for L1 on R10K), maximum 32 outstanding instructions reached, renaming registers consumed, and back-to-back write-backs from L1. Architectural characteristics stipulate consumption of all renaming registers and back-to-back write-backs are very rare, so we focus on the other constraints. As a good first-order approximation, at each cycle, the load/store unit can execute up to one memory instruction, and the two integer and two floating-point units can each execute two instructions. (Actually, we approximate the floating point service rate using observed measurements and based on the mix of additions and multiplications since each unit can provide only one of these operations.)
We have created code that we can modify to ensure certain instruction streams are fed to the microprocessor in the interest of validation. We use direct hardware counter measurements to ensure synthetically created code meets all assumptions. In Table 4.3, we present a series of uniformly distributed instruction mixes and measured results to show our method works. The pattern descriptions consist of one or two parts. The first part describes the repeated sequence of instructions. For example, miii refers to a memory instruction followed by three integer instructions. This series constitutes a synthetic stream repeated to the point of stability (in the millions of instructions). If a stream contains more than two f's (i.e. floating point operations), we specify the types of operations after the "underscore". For example, fff_** refers to a repeated sequence of floating point instructions of the type "multiply", "add", "multiply". We specify these in order to account for the fact that while claiming two floating point units for the MIPS R10000, in reality there is one floating point servicing only additions and another servicing only multiplication operations. The mix of addition's and multiplication's thus affects cpi via a change in service rate as apparent in Table 4.3. For these we also use an

---

**Table 4.3 Results for synthetic instruction streams on MIPS R10000**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Utilization Factors</th>
<th>Limiting Factor</th>
<th>1/d</th>
<th>x/</th>
<th>Meas CPI</th>
<th>Calc CPI</th>
<th>Rel Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>fff <strong>+</strong></td>
<td>2.6385 0.0055 0.0091</td>
<td>f</td>
<td>0.9894</td>
<td>0.6667</td>
<td>0.6622</td>
<td>0.6596</td>
<td>-0.40%</td>
</tr>
<tr>
<td>iff <strong>+</strong></td>
<td>1.9841 0.0041 0.5039</td>
<td>f</td>
<td>0.7440</td>
<td>0.6667</td>
<td>0.5192</td>
<td>0.4960</td>
<td>-4.47%</td>
</tr>
<tr>
<td>iii</td>
<td>0.0000 0.0082 1.9820</td>
<td>i</td>
<td>0.9910</td>
<td>0.5000</td>
<td>0.5057</td>
<td>0.4955</td>
<td>-2.01%</td>
</tr>
<tr>
<td>iiii</td>
<td>0.9921 0.0041 1.4949</td>
<td>i</td>
<td>0.7475</td>
<td>0.5000</td>
<td>0.5962</td>
<td>0.5192</td>
<td>-5.67%</td>
</tr>
<tr>
<td>miff <strong>+</strong></td>
<td>1.9841 0.9962 0.0068</td>
<td>f</td>
<td>0.7440</td>
<td>0.6667</td>
<td>0.4989</td>
<td>0.4960</td>
<td>-0.57%</td>
</tr>
<tr>
<td>miii</td>
<td>0.0000 0.9962 1.4949</td>
<td>i</td>
<td>0.7475</td>
<td>0.5000</td>
<td>0.5960</td>
<td>0.3737</td>
<td>-5.63%</td>
</tr>
<tr>
<td>mm</td>
<td>0.0000 3.9450 0.0136</td>
<td>m</td>
<td>0.9863</td>
<td>1.0000</td>
<td>1.0010</td>
<td>0.9863</td>
<td>-1.47%</td>
</tr>
<tr>
<td>mmiff <strong>+</strong></td>
<td>0.9921 1.9882 0.0068</td>
<td>m</td>
<td>0.4971</td>
<td>1.0000</td>
<td>0.5044</td>
<td>0.4971</td>
<td>-1.45%</td>
</tr>
<tr>
<td>mmif</td>
<td>0.9921 1.9882 0.5029</td>
<td>m</td>
<td>0.4971</td>
<td>1.0000</td>
<td>0.5072</td>
<td>0.4971</td>
<td>-2.01%</td>
</tr>
<tr>
<td>mmii</td>
<td>0.0000 1.9882 0.9989</td>
<td>m</td>
<td>0.4970</td>
<td>1.0000</td>
<td>0.5070</td>
<td>0.4970</td>
<td>-1.97%</td>
</tr>
<tr>
<td>mmiff</td>
<td>0.9921 2.9803 0.0068</td>
<td>m</td>
<td>0.7451</td>
<td>1.0000</td>
<td>0.7553</td>
<td>0.7451</td>
<td>-1.35%</td>
</tr>
<tr>
<td>mmiii</td>
<td>0.0000 2.9803 0.5029</td>
<td>m</td>
<td>0.7451</td>
<td>1.0000</td>
<td>0.7526</td>
<td>0.7451</td>
<td>-1.01%</td>
</tr>
</tbody>
</table>
average floating point service rate of 1.5 instructions per cycle. We chose a mix of instructions to cover most possible permutations for a four-instruction mix without providing every single permutation. This provides us with a concise list of instances with excellent coverage.

There are several interesting observations to be made in Table 4.3. When a certain instruction is not present, its associated utilization factor is equal to (or very near) zero since none of the resources are being utilized. When this happens for floating-point instructions we get exactly zero since no extraneous floating point instructions will be executed. Sometimes integer instructions are necessary to calculate an address for example, giving values very close to zero instead of exactly zero for the utilization factor. In each of these examples, a single utilization factor ($\rho$) greater than 1 is found indicating a single limiting factor. We use this factor's associated queue to calculate ideal $\text{cpi}_0$ (shown as Calc CPI) in this chart using Equation 4.11. In Table 4.3, all of these instruction streams contribute directly to $\text{cpi}_0$ while our assumptions are met and there are no other contributors to $\text{cpi}$; thus $\text{cpi}_0=\text{cpi}$ in this context.

Table 4.3 shows our calculated and measured $\text{cpi}_0$ are within the tolerance of the counters themselves, implying they are quite accurate. Table 4.4 shows the results of perfect instruction mix giving the ideal cpi (calculated as $1/\beta$) of the MIPS R10000. All

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Utilization Factors</th>
<th>$\beta$</th>
<th>Meas CPI</th>
<th>Calc CPI</th>
<th>Rel Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>iliff +*</td>
<td>$0.9920$</td>
<td>$0.0041$</td>
<td>$0.9989$</td>
<td>$4.0000$</td>
<td>$0.2576$</td>
</tr>
<tr>
<td>mfiff +*</td>
<td>$0.9920$</td>
<td>$0.9962$</td>
<td>$0.5029$</td>
<td>$4.0000$</td>
<td>$0.2580$</td>
</tr>
<tr>
<td>miiff</td>
<td>$0.4960$</td>
<td>$0.9962$</td>
<td>$0.9989$</td>
<td>$4.0000$</td>
<td>$0.2577$</td>
</tr>
</tbody>
</table>
of these results directly validate our model on the MIPS R10000. Thus, with our assumptions, we are able to model ideal $cpi_0$ with a great deal of accuracy. Since our theory is general in nature, we believe validation on other processors will support these findings.

4.1.5 Extended Instruction-Level Model

We have extended the functionality of the instruction-level model to get a closer estimate of the pipeline CPI. Unfortunately, complete validation has not been accomplished for these extensions to the previously validated instruction-level model of ideal $cpi_0$. The extended model does incorporate data dependencies and branch prediction to some extent. For synthetic instruction streams the model is valid, but the mapping from these synthetic instruction streams to our codes is future work.

Nonetheless, the extended model gives insight into the architectural performance of a particular architecture while showing the practicality of extending the basic instruction-level model.

Recalling Equation 4.2, there are two parts to the overall CPI: memory system CPI and pipeline CPI. The instruction-level model was developed to estimate a lower bound for pipeline CPI, which it has certainly done. We call this lower bound ideal $cpi_0$ because of the assumptions associated with it and the fact that computing it is based on measured code parameters. We thus know that ideal $cpi_0$ is a portion of the pipeline CPI, but what makes up the rest of this term? Assuming ideal cache, we propose the following formula to estimate $cpi_0$ or pipeline CPI.

$$cpi_0 = cpi_{mem} + cpi_{op} + cpi_{mem} + cpi_{mem} + cpi_{mem}$$  \hspace{1cm} (4.13)
Each of the terms of this equation requires explanation. $cpi_0$, of course, is the pipeline $cpi$ term of Equation 4.2. $cpi_{ideal}$ is the peak $cpi$ of the processor. For a 4-way superscalar processor like the MIPS R10000, $cpi_{ideal}$ is .25. This is the underlying theme of this formula. The premise is that there exists an achievable maximum rate ($cpi_{ideal}$) of $cpi$, that is increased as an instruction stream takes on attributes no longer resembling the perfect stream required to achieve $cpi_{ideal}$. Each attribute adds to the ideal $cpi$ ($cpi_{ideal}$) causing performance degradation. So the rest of the terms of Equation 4.13 represent attributes of the code that negatively impact $cpi_{ideal}$.

$cpi_{res}$ is the $cpi$ gained from stalls due to resource conflicts on-chip. This can be calculated using the ideal $cpi_0$ from Equation 4.12 and the measured $cpi_0$ from the memory model.

$$cpi_{res} = cpi_0 - ideal\ cpi_0$$  \hspace{1cm} (4.14)

It might seem that estimating $cpi_{res}$ this way defeats the purpose of attempting to model $cpi_0$. In a sense, that is correct, but in reality we are not as interested in being able to directly predict $cpi_0$ as we are in interested in dissecting $cpi_0$ to find the main contributors to performance degradation. It is this second goal that we focus on. This will become clearer after we define the rest of Equation 4.13.

$cpi_{dep}$ is the $cpi$ gained from stalls due to unsolvable data dependencies between instructions. We will not be able to measure this directly for now. There are no counters available that provide insight to such characteristics, and we still wish to keep our methods practical by relying on performance monitors. As will be apparent, this is actually the term we are most interested in quantifying. Obtaining values for this term would allow us to quantify the data dependence properties of an application. To the
best of our knowledge, this has not been achieved. As will be shown in the result chapter, we can quantify dependences for synthetic code, but the mapping to our codes has not been accomplished and will be the subject of future work.

\( cpibraceh_{\text{branch}} \) is the overhead involved for a branch. There are methods for estimating such occurrences, such as using a worst case prediction (i.e. that every mispredicted branch incurs the maximum latency resulting from swapping out an entire thread). For now, however we simply use this as a placeholder with a value of zero when we assume perfect branch prediction. Correctly predicted branches can be modeled since they basically impede the fetch/decode bandwidth based on the probability distribution of branches throughout the code.

\( cpi_{\text{non\_uniform}} \) is another placeholder. It is only present if we do not assume non-uniform distribution. Uniform distribution gives the processor an ideal mix of instructions and makes things much easier to model. But realistically speaking, we want to be able to extend the model to non-uniform distribution eventually. We believe it is possible to quantify the contribution of a non-uniform stream in a single term.

Overall, Equation 4.13 is an over-simplification of pipeline cpi since we do not take into account such things as TLB or icache misses, and because of its many assumptions. However, it is a good first step toward approximating the performance contributions of pipeline cpi. By manipulating the unknowns, and eliminating terms based on our assumptions, we provide the following equation for preliminary dependence analysis of synthetic instruction streams.

\[
\begin{align*}
cpi_{\text{syn}} &= cpi - [cpi_{\text{dep}} + cpi_{\text{mis}} + cpi_{\text{non\_uniform}}] \\
&= cpi - [cpi_{\text{dep}} + cpi_{\text{mis}} + cpi_{\text{non\_uniform}}]
\end{align*}
\]  

(4.15)

Using the assumptions of uniform distribution and perfect branch prediction, we get:
\[ cpi_{op} = cpi_0 - [cpi_{ideal} + cpi_i] \]  \hspace{1cm} (4.16)

\[ cpi_{op} = cpi_0 - [cpi_{ideal} + (cpi_0 - \text{ideal } cpi_0)] \]  \hspace{1cm} (4.17)

\[ cpi_{op} = \text{ideal } cpi_0 - cpi_{ideal} \]  \hspace{1cm} (4.18)

This is the model portion of the extended instruction-level model incorporating data dependences. Since we have already validated the ideal cpi\(_0\) formula, cpi\(_{ideal}\) is a constant, and since we can control the addition of data dependence only in our instruction streams, no further validation for the extended instruction-level model is necessary provided we model only synthetically built instruction streams for now. As mentioned, the dependence mapping to our codes in future work does seem possible since we have derived formal methods for quantifying the dependences. We need to conduct mapping that is not overly time-consuming or complicated and that provides us with sufficient information for describing the types of dependencies found in context. This unfortunately will be rather time-consuming and is beyond the scope of this thesis.

4.2 Statistical Analysis Method

The general cpi method broke down cpi into pieces for further evaluation. There are ways of evaluating cpi without breaking the term down further. Traditional statistics provides us with the tools necessary for such performance evaluation. The idea here is to complement the general cpi model with other avenues for evaluation allowing the use of our models to determine the type of analysis required.

This methodology consists of four levels of evaluation [15]. All of the four levels of evaluation are based on two-factor factorial statistical methods [16]. While the first two levels of the methodology focus on the mean performance over problem sizes, the last two level evaluations show the performance variation when problem size
increases. The combination of these four levels of evaluations provides a feasible solution for predicting the performance when problem scales up and to suggest further memory system improvements. The strength of this methodology lies in its ability to classify code-machine combinations at a high level, providing insight to the probable bottlenecks as problem and system size scale up. It also presents a step-wise refinement approach toward focusing on the direct causes of performance deviations among similar problems across varying machines. Algorithms are given to facilitate understanding, but readers are referred to [16] for details regarding general statistical terminology and methods.

4.2.1 Background

Some background knowledge of scalability and statistics is needed for understanding all methodologies. We introduce our terminology and the memory scalability concept in order to facilitate discussion of the statistical model for memory hierarchy evaluation.

Multiple treatment factors: In our experimental design, we use two-factor factorial design. Problem size and machine are the two factors used for scalability study and code and machine are the two factors used in data reference pattern study for the statistical model. Each factor has multiple levels.

Factorial experiment: An entity that is used for the experiment is called an experimental unit. For example, one combination of the different levels of the code and machine factors, is an experimental unit.

Cell: Cell refers to the measurement made to an experimental unit. The value of cpi measured could be considered a cell. A cell may include an observation.
Main effects: Main effects are the differences in the mean response across the levels of each factor when viewed individually. For instance, code and machine are two main effects for a study.

Interaction effects: Interaction effects are differences or inconsistencies of the main effect responses for one factor across levels of one or more of the other factors. In our experimental design, both code and machine may have effects on the experimental units. If code influences the performance of a machine, or, vice versa, machine influences the performance of code, then interaction effects exist.

4.2.2 Definitions

A goal of high performance computing is to solve large problems fast. Considering both execution time and problem size, what we seek from parallel processing is speed, which is defined as work divided by time. The average unit speed is a good measure of parallel processing. It measures the computation performed in each processor per second.

Average Unit Speed (or average speed): The achieved speed of the given computing system divided by the number of processors.

Isospeed Scalability: Formally defined in [17] as the ability to maintain the average speed in parallel processing when the number of processors increases. A code-machine combination is scalable if the achieved average speed of the code on the given machine can remain constant with increasing numbers of processors, provided the problem size can be increased with the system size.

Data Scalable for single system: We say a code-machine combination is data scalable, if either the speed of the code-machine combination does not decrease with the problem
size increase or the cpi of the code-machine combination does not increase with the problem size.

4.2.3 Four-Level Statistical Method for Evaluating Memory Systems

Readers will notice that a validation section does not follow this method. This is due to the fact that exhaustive references for the statistical methods used have been written as referenced earlier in this section. In particular, the validation comes into play when the results are presented. In other words, when used correctly, do the statistical methods discussed provide us with further information about code-machine interaction? In general, the applied statistical methods mentioned determine if variations exist that warrant investigation. When we observe certain variations between code-machine combinations that differ in implementation of the memory hierarchy, culprits for performance differences can be identified. So to summarize, we are confident in the methods themselves when applied correctly, and the truth is we simply need to determine whether or not they provide useful insight into code performance. The results chapter provides the proof of the usefulness of our approach.

4.2.3.1 Level One Evaluation: Main Effect

Level one evaluation uses the two-factor factorial experiment to find the effects of code and machine. Using the two factors, code and machine, it detects the overall effect of code, machine, and their interaction on the final performance. The dependent variable for the two-factor factorial design is cpi. If code effect exists, we conclude that the codes have different memory reference patterns that diverge memory access time. When machine effect exists the memory system difference on the machines does make a difference in performance. Finally, when code-machine interaction effects exist the
memory system difference has a different impact on different memory reference patterns. Notice that all these effects are overall effects of codes and machines. Any of the effects that exist deserve further investigation to identify the source or sources.

Algorithm of Main Effects:

- Compute Cell Means, Machine Means, Code Means and Overall Mean
- Compute TSS, SScells, SSW, SSA, SSC, and SSAC
- Compute all the degrees of freedom
- Compute MScells, MSA, MSC, MSAC, and MSW
- Get F values by using SS divided by the degree of freedom

4.2.3.2 Level Two Evaluation: Code/Machine Classification

We would like to know the contribution of each code/machine toward effects and to identify the outstanding code/machine for more detailed study. The key technique to single out outstanding contributors is to find the relative performance of a code/machine with that of others. Statistical classification methods provide a means to group code/machine based on their relative performance.

In general, there are a! comparisons for a factor with a levels. If two machines belong to the same category, then statistically they are the same, for the set of codes and under the interested range of problem sizes. If two codes belong to two different categories, then they have different memory reference/computation patterns. A good general-purpose machine should not deliver a wide cpi distribution among codes.

Algorithm of Contrast Method:

- Obtain statistical data for code-machine combos
- Compute $\sum \alpha * mean(y_i)$ and $\sum \alpha_i^2$
- Compute \( t = \frac{\sum \alpha \cdot \text{mean}(y_i)}{\sqrt{\frac{\text{MSW}}{n \sum \alpha_i^2}}} \)

- Judge the testing hypothesis by using the probability of t-value.

4.2.3.3 Level Three Evaluation: Scalability Comparison

The third step of our evaluation methodology is individual evaluation for outliers. It compares the data scalabilities of a given code on different machines. Memory scalability evaluation is a new approach. It evaluates the ability of a memory system in handling large data sizes. The same or a better initial performance combined with a better scalability guarantees a code will have a better performance when problem size scales up. A code with a smaller initial cpi and a better scalability has the potential to become superior as problem size scales up.

Using cpi as the measurement, with the same code on two different machines, if the interaction of the two variations is negative then the second machine has a better scalability. If the interaction of the two variations is zero, the two machines have the same scalability; otherwise, the first machine has a better scalability. The algorithm for the statistical scalability evaluation is given below.

Algorithm of Scalability Comparison:

- Assign a value for each of the factor levels and construct an index table
- Substituting values in the index table to equation
  \[ \text{cpi} = \mu + \beta X_r + \beta X_e + \beta \cdot J_e \]
- Solve the linear system generated.
- Judge the term \( \beta \cdot J_e \) by the probability of t-value.
4.2.3.4 Level Four Evaluation: Memory Hierarchy

The last step of our evaluation methodology is designed to locate memory components that cause the variation. Level four evaluation compares the performance variation of primary components of the underlying memory systems. Combined with the level two evaluation, this evaluation determines the ability of each memory component in handling different memory reference patterns and suggests possible improvements at the component level.

Algorithm for Memory Structure Evaluation:

- Assign a value to each of the factor levels and construct an index table
- Substituting values in the index table into equations
  \[ L1 = \mu + \beta X_x + \beta X_\cdot + \beta X \cdot \cdot \text{ and } L2 = \mu + \beta X_x + \beta X_\cdot + \beta X \cdot \cdot \cdot \text{ separately.} \]
- Solve the two linear systems generated individually.
- Judge the term \( \beta \cdot \cdot \) by the probability of t-value.
- Determine the performance variation of each of the three primary components.

4.3 The Hybrid Method

As mentioned in Chapter 2, statistics have provided reduction techniques for simulated data in the context of single microprocessor performance [18, 19]. The previously discussed statistical method has also focused on regressive techniques for studying scalability and variations in like architectures statistically with promising results. Generally speaking, if we were to combine the strength of such comparisons with a strong empirical or analytical technique, we could conceivably provide more
information furthering the usefulness of the original model. The hybrid method combines the strengths of both the empirical memory model and the statistical method.

4.3.1 The Hybrid Approach: Level 1

We again begin with Equation 4.2 that breaks down cpi into pipeline and memory system cpi. Level one of the hybrid approach focuses on using two-factor factorial experiments to identify the combinations that show differences in performance that warrant further investigation. Following the statistical analysis method, we identify codes and machines as observations to be used in the two-factor factorial experiments. Once all measurements have been obtained, we can perform the experiments for the factors code and machine. Using statistical methods with the help of the SAS statistical tool [20], we gather results relating to the variations present among codes, machines and their interactions. We accomplish this via a series of hypothesis experiments where statistically we determine whether or not a hypothesis is true or false. This is the essence of the two-factor factorial experiment. This allows us to identify within a certain tolerance, the differences among code-machine combinations.

Hypothesis: Overall effect does not exist. For this experiment, the dependent variable is the overall average cpi measured across codes for the machines. With these parameters, disproving the hypothesis indicates that in fact, differences between the architectures for these codes exist. If this hypothesis is not disproved, then we believe with some certainty, that there are no statistical differences among the two architectures for these codes. If this hypothesis is rejected, then the next three hypotheses should be visited.
Hypothesis: Code effect does not exist. For this experiment, the dependent variable is the pipeline cpi term from the decoupled cpi of Equation 4.2. In practice, this term is experimentally measured when using the empirical model. If the hypothesis holds in this experiment, no difference is observed statistically for these codes on these machines at the pipeline level. Conversely, if the hypothesis is rejected, code effect does exist indicating differences at the pipeline level for this application on these architectures. In the empirical model context, if this occurs, further analysis of the cpi pipeline term is warranted.

Hypothesis: Machine effect does not exist. For this experiment, the dependent variable is the cpi memory term from the decoupled cpi of Equation 4.2. This term can be derived experimentally as well. If the hypothesis holds in this experiment then no discernible difference between these machines statistically is apparent for these codes. Otherwise, rejecting this hypothesis indicates machine effect does exist. In the case of the empirical memory model, this warrants further investigation since it implies variations in the memory performance across code-architecture combinations.

Hypothesis: Machine-code interaction does not exist. For this experiment, the dependent variable is overall cpi measured across individual codes and individual machines. If this hypothesis is held, then no machine-code interaction effects are apparent statistically. Otherwise, rejecting the hypothesis begs for further investigation of the individual codes and machines to determine why machine-code interaction changes the performance across machines. Such performance differences indicate that codes behave differently across different machines in an unexpected way, hence requiring further investigation.
4.3.2 The Hybrid Approach: Level 2

If code effect exists, study pipeline cpi. This indicates fundamental differences at the on-chip architectural level. The empirical memory model does not provide insight to such performance differences, treating pipeline cpi as a black box. The instruction-level model could be used to provide more insight to performance variations for such a code.

If machine effect exists, study memory cpi. If machine effect exists, statistical variations are present between different codes at the memory hierarchy level across machines. This is exactly the purpose of the empirical memory model: to analyze contributions to performance from the memory hierarchy. At this point, the statistical method has provided an easy method for determining when further analysis using the memory model is necessary. This requires a more detailed look at the decoupled cpi in Equation 4.2.

Equations 4.4 and 4.5 of the empirical memory model indicate that \( m_0 \) reflects the performance variations in cpi when pipeline cpi is constant over increasing problem sizes. Calculating \( m_0 \) is costly since it requires a least square fitting first to obtain each \( t_i \) term. By applying the statistical method and through direct observation, we have isolated the conditions under which it is worthwhile to calculate the terms of Equation 4.4. For conditions where machine effect exists, \( m_0 \) will provide useful insight to the performance of the memory latency hiding effects mentioned. We can also use \( m_0 \) statistically to describe the scalability of a code in regard to how predictable the performance is as problem size increases. We can use other variations on the original statistical method to study the variations of \( m_0 \). This is somewhat less costly than determining \( m_0 \) for each problem size and machine combination. Nonetheless, actually
calculating $m_0$ values provides validation to the conclusions obtained using this technique (this will be shown in the results section). If $m_0$ values show no statistical variations or are constant as problem sizes increase, performance scales predictably and $m_0$ can be used for performance prediction of problem sizes not measured. If $m_0$ values fluctuate statistically or are not constant as problem size increases, performance does not scale predictably and $m_0$ cannot be used for performance prediction.

$m_0$ values across machines can also provide insight into performance. If statistical differences across machines for the same problem are non-existent or if $m_0 - m_0'$ is constant as problem size increases, where each $m_0$ represents measurements for the same code over different machines, then the memory design differences make no difference for the codes being measured.

If machine-code interaction exists, study cpi. This corresponds to the fourth hypothesis of level one. If machine-code effect exists, statistical variations are present when machine-code interactions occur. This indicates further study of the resulting cpi is necessary since there exist unexplained performance variations. This scenario is outside the scope of the hybrid method, but exactly what the statistical method was intended to help analyze. Further focus on particular code and architecture combinations should be carried out using the statistical method.

4.4 Summary

In this chapter we provide the theoretical models we use for analysis. Where necessary we discussed the direct validation of the methods as proof of correctness. The general cpi model is the common thread among our methods for analyzing the instruction-level parallelism of codes. The statistical method allows analysis and focus on the overall cpi
of codes while the empirical memory model and instruction-level model focus on analyzing portions of the overall CPI. The hybrid method combines the conclusions of the statistical method to narrow down the focus and further application of the empirical and statistical methods generally. Each method requires gathering of counts on performance monitors in order to provide inputs to the models. Simple measurements using the common problem set for these codes gives the information necessary to all the techniques discussed in this chapter and applied in Chapter 5. Together the models give analysis of the memory hierarchy, inherent architectural bottlenecks, and pair-wise statistical variations for the codes and machines of interest.

4.5 References


96

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


Chapter 5
Application and Experience

5.1 SynBAD and PTERA

5.1.1 Overview of PTERA

We have incorporated cross-platform measurement capabilities for both the memory model and the instruction-level model in a Performance Tool for the Evaluation of Realistic Applications (PTERA), pronounced “tera”. This tool provides automation of measurement gathering and code creation techniques while accessing the underlying performance counters. The PTERA tool is actually comprised of several key functional units that can be improved independently making it both modular and adaptive to a user’s needs. User interaction is available via PTERA’s user interface. Here users can either specify a code of particular interest or use the SynBAD tool within PTERA to create SYNthetically Built Assemble Directives as per the user’s specifications.

SynBAD allows users to specify desired instruction mix patterns to be synthetically created for the target architecture. In this way, users can provide PTERA with existing code or code with desired characteristics to be analyzed. After assemble code is generated via the user’s specifications for synthetic applications, PTERA uses available hardware monitors and appropriate interfaces to gather measurements for a minimum problem set. Actual measurements are then sent to the PTERAnalyzer for analysis using the aforementioned modeling techniques.

The SynBAD portion of PTERA is designed with the experimenter in mind. In particular, researchers will have absolute control over the instruction stream produced by SynBAD. While the user will be able to control mixes of individual instruction
types such as floating point and integer instructions, users will also be able to define data dependence relations. Through automation of this assemble code generation, we plan to extend instruction-level analysis techniques to incorporate branch and data dependence influence on performance.

5.1.2 The PTERA Prototype

To prove that a tool with the described functionality can be developed for multiple platforms, we have implemented a prototype version with a good portion of functionality and automation over a RISC and CISC platform with usable results. The current version is a compilation of developed code and shell scripts that provide the measurements necessary to analyze codes using all the models discussed in Chapter 4. It is our sincere hope that since we have developed a working version of the tool with a good deal of functionality, that future work will target a tool with similar functionality built from the ground up. While it does not currently automate the calculations within the models themselves, it functions to provide the measurements necessary to all the models discussed in Chapter 4. We generally discuss the PTERA prototype in this section so readers understand the tool used to generate all the results found in these experiments. The models themselves are semi-automated, requiring separate calculations for now via spreadsheets, separate programs, and SAS, but we hope they will also be part of the final tool-set.

Figure 5.1 shows an overview of module interaction in PTERA. There are three distinct layers in the software that have fixed interaction. The "completely independent" layer in Figure 5.1 contains modules that have no dependence on the particular hardware being studied. These modules include the User Interface and the
Figure 5.1 PTERA modules

PTERAnalyzer, which respectively provide user interaction with the tool and analysis of the performance measurements. The “layered dependent” layer indicates modules that themselves have layered implementation. For example, portions of both the SynBAD module and the User Application module are completely independent of the underlying architecture as they interact with the User Interface module. In contrast, portions of SynBAD and the User Application modules must provide information to the Performance Monitor module in the “completely dependent” layer. In the design of PTERA, to promote portability, we will minimize the hardware dependent software in modules in the “layered dependent” layer by minimizing interaction across module layer boundaries throughout the tool.

The PTERA prototype currently contains working versions of the User Interface module, SynBAD, the User Application module, and the Performance monitor module. We briefly discuss the functionality provided in the current prototype module by module. The first phase of PTERA was a multi-part feasibility study. This completed
study included surveys of existing tools, microprocessor candidates for implementing versions of the PTERA tool, and preliminary results. These results have already been discussed in previous chapters. Furthermore, PTERA made the extended instruction-level model a reality through experimentation discussed later in this chapter. The feasibility study indicated the MIPS R10000 and Intel Xeon were worthy of initial implementation for example RISC and CISC architectures respectively.

5.1.2.1 User Interface Module

For now, the user interface module requires users to manipulate script templates that explain the desired use of the tool. For user applications, executables, makefiles, and compilation details must be specified. For SynBAD user's the particulars of the instruction stream must be defined. Scripting requires fundamental understanding of dependence relations discussed in a later section. SynBAD users can specify such things as loop-carried dependences, loop-lengths, and data dependence relations among any allowable combinations of instruction types. The important advance in this implementation is a robust method for intuitively describing data dependence relations via a simple matrix of dependences. Users simply enter values in a matrix where a row-column relation represents a dependence relation. All such relations are checked for validity and created based on user specifications if allowed. Template makefiles are available for specifying applications to be measured at the program level. Both are designed to be intuitive in nature, but currently require understanding of the underlying code functionality.

Figure 5.2 gives an example portion of the specification file along with its corresponding assemble code as created by PTERA via SynBAD. The code exhibits all
the assumptions of the instruction-level model in Chapter 4. Furthermore, the
dependence relations model both regular and loop-carried dependences as shown. The
first line of the specification simply identifies the number of instructions that will be
specified. For now, the limit is 100 instructions maximum. The next 4 lines then
indicate the type of instructions and their order. The 4x4 matrix is always n x n where n
is the number of instructions specified. If we assign the four instructions numbers from
1 to 4, then the rows and columns of the 4x4 matrix describe the dependences present in
our synthetic stream. We only define forward data dependences for now, so any
element of the matrix \((i,j)\) is set to "U" for undefined when \(i \geq j\). Otherwise, for \(i < j\) we
indicate 3 different values. An "X" indicates no dependence between corresponding
instructions \(i\) and \(j\). "I" indicates an infinite dependence or "loop-carried" dependence
from instruction \(i\) to \(j\) to \(i\) to \(j\), etc. And a number indicates a finite dependence from \(i\) to
\(j\) to \(i\) to \(j\), where the length will be the number specified. Using this robust method, we
can intuitively describe many of the data dependences characteristic in scientific codes.
Later in this chapter, we discuss some ideal experiments created to quantify dependence
contribution to performance as these parameters are varied. Figure 5.2 describes a
dependence of length 3 between the first integer add instruction and the second integer add instruction. The "I" indicates an infinite loop-carried dependence allowed from the floating point multiply to the floating-point store instruction ad infinitum.

5.1.2.2 SynBAD and User Application Modules

The current implementation of SynBAD allows any data dependence relation of arbitrary length and loop-carried qualities to be defined for many integer, floating point, and memory operations as shown in Figure 5.2. Relationships are checked to ensure validity and errors are reported. SynBAD guarantees correctness for user specified instruction streams. The User Application module simply allows wrap-around measurement of entire code segments. If portions of code are to be measured, counter measurements must be embedded directly in the code for now. The focus of initial implementation of PTERA was on SynBAD functionality.

5.1.2.3 Performance Monitor Module

This module is implemented as generally as possible allowing indirect descriptions of instruction types for interaction with the SynBAD module. This allows the lowest level of implementation to be changed easily for cross-platform compatibility. The same benefit will be observed for the User Application module once isolated performance monitoring is implemented. This was particularly difficult to implement generally since different instructions have different rules in interacting with other instructions via dependences. These particular problems were eventually resolved, but the time and effort was much more involved than anticipated. Nonetheless, correctness at the SynBAD level is held at the performance monitor level ensuring accurate synthetic code creation. This module is simplified by use of the problem set for performance
measurements discussed in Chapter 3. Measurements are obtained for problem set elements only.

5.1.2.4 PTERAnalyzer Module

For now, this module is implemented by hand utilizing the counter output that is automatically formatted for analysis in spreadsheet format. The measurements provided by each implemented portion of the PTERA tool allow each of the models presented in Chapter 4 to be used for analyzing both applications and synthetic code created by SynBAD.

5.1.3 RISC and CISC Implementation of PTERA

Results presented in this chapter have been gathered on both the RISC-based MIPS R10000 and the CISC-based Intel Xeon processors. All the functionality described in the previous section is contained in the RISC version for the MIPS R10000 on the SGI Origin 2000 machine described previously. This prototype is quite robust and offers nearly all functionality described in the PTERA proposal.

The CISC version is still in the preliminary stages for two primary reasons. First of all, successful modeling at the instruction-level as shown later in this chapter was the basis for a 3-year funded proposal (for $110,000 per year through LANL LDRD CSSE ER #2000022). The proposal was to develop a variant of a RISC-based microprocessor that takes advantage of the performance attributes discovered by the instruction-level model. This proposal was one of 6 awarded funding from a pool of about 40 beginning September 1999. This changed the focus of current research substantially allowing for a more limited implementation of the PTERA tool for the CISC platform. It is for this reason that we present a limited portion of the results of the
new architecture in this dissertation primarily to show the usefulness of our modeling technique in providing insight to architecture performance as well as code performance.

The second reason for limited current implementation on the CISC system is problematic in nature. The system chosen was the Intel ASCI Red machine discussed earlier. Of late, this system has become much less stable than previous versions. Instability is a serious problem when performing counter measurements since often multiple runs for a single code must be accomplished. This is a particular problem with SynBAD since many runs are required to achieve the type of dependence analysis presented later in this chapter for the MIPS R10000. Unfortunately, the ASCI Red machine no longer seems to be a viable platform, and we will be required to port the semi-ported PTERA to a more stable system to complete implementation for the CISC architecture. The version implemented did allow for several feasibility studies to be completed along with results for the empirical memory model presented later in this chapter. SynBAD has been partially implemented for some integer operations. Floating-point operations have required more work than anticipated due to the stack-based implementation of floating point register allocation on the Intel architecture. Other minor problems such as particular reserved registers for certain operations have also delayed implementation.

The overall conclusion supports the initial feasibility study. The PPRO architecture will allow full implementation of PTERA and SynBAD, but a more stable platform must be available to eliminate unnecessary delays. It is our intention, as future work to port this semi-implemented version to a Linux stand-alone platform. The reason we did not do this initially involves certain compiler dependences for the ASCI
codes that are not readily available on Linux architectures. As we will no longer be dependent on ASCI funding, we can eliminate this dependence as we attempt the new implementation.

5.2 Empirical Memory Model

5.2.1 Methodology

The empirical memory model described in Chapter 4 provides the foundation for an analysis of the architectural features of the Origin 2000, PowerChallenge, and Intel ASCI Red application performance. The key issue is determination of the amount of memory access time that is overlapped by computation. Although this overlap is not directly measurable using the R10000 or the PPRO (Xeon) performance counters, we can infer the overlap for an individual application by fitting empirical performance data obtained from its execution using different problem sizes.

R10000 and PPRO (Xeon) performance counters supply measurements of the total execution cycles and total graduated instructions via the PTERA performance tool. The ratio of these two measurements gives the overall cpi of the application. The hit ratios (coming from the same application executing on different problem sizes) are also directly measurable and the unknowns in Equation 4.3 (for a two-level cache scheme) become the average times, $t_2$, $t_m$, and $cpi_0$. The value of $cpi_0$ can be obtained by measuring the cpi of a problem that fits entirely in the L1 cache. We have confirmed these values via the simulator for the R10000 coupled with the experiments discussed in the validation section of the empirical memory model in Chapter 4. The remaining unknowns are inferred from the measured data by a least squares fit constrained such that $0 \leq t_i \leq T_i$. 

106
Table 5.1 shows the model parameters for each of the LANL benchmark codes determined from a data set of executions on the 2-MB L2 PowerChallenge. The least square fit generally has errors that are less than 6%, in line with errors measured on the simulator. The maximum latencies, Ti, are measured with lmbench and are found to be consistent with numbers published by SGI.

<table>
<thead>
<tr>
<th></th>
<th>( t_2 )</th>
<th>( t_m )</th>
<th>cpi₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat</td>
<td>2</td>
<td>128</td>
<td>0.74</td>
</tr>
<tr>
<td>Hydro</td>
<td>3</td>
<td>117</td>
<td>0.89</td>
</tr>
<tr>
<td>Hydro-t</td>
<td>0</td>
<td>69</td>
<td>0.9</td>
</tr>
<tr>
<td>Sweep</td>
<td>11</td>
<td>145</td>
<td>0.88</td>
</tr>
<tr>
<td>Neut</td>
<td>2.2</td>
<td>205</td>
<td>0.77</td>
</tr>
</tbody>
</table>

5.2.2 Analysis of Stall Time Due to Memory Accesses

Table 5.2 compares the memory access times, \( t_i \), for the benchmark codes on the PowerChallenge, the Origin 2000, and the Intel ASCI Red supercomputer. In general, L2 cache accesses are completely overlapped with computation (low values of \( t_2 \)) for the comparable Origin 2000 and PowerChallenge. Additionally, the observed values of \( t_m \) suggest that about one-half of the main memory latency is hidden on the PowerChallenge, and Origin 2000. The exception is SWEEP (not measured on Intel Red) where the value of 11cps for \( t_2 \) indicates that accesses to the secondary cache are not overlapped. The reason that SWEEP stands out may be due to loop-carried dependencies in the inner loops. These dependencies present less prefetch opportunities for the compiler and result in less overlap of processor execution with memory.
Table 5.2 Memory access times for PC, O2K, and Intel ASCI Red

<table>
<thead>
<tr>
<th></th>
<th>$t_2$ PowerChal</th>
<th>$t_m$ PowerChal</th>
<th>$t_2$ Origin 2000</th>
<th>$t_m$ Origin 2000</th>
<th>$t_2$ Intel Red</th>
<th>$t_m$ Intel Red</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAT</td>
<td>2</td>
<td>128</td>
<td>0</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HYDRO</td>
<td>3</td>
<td>117</td>
<td>2.4</td>
<td>50</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>HYDRO-T</td>
<td>0</td>
<td>69</td>
<td>0</td>
<td>11</td>
<td>5.1</td>
<td>5.8</td>
</tr>
<tr>
<td>SWEEP</td>
<td>11</td>
<td>145</td>
<td>11</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEUT</td>
<td>2.2</td>
<td>205</td>
<td>11</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LMBENCH</td>
<td>11</td>
<td>205</td>
<td>11</td>
<td>80</td>
<td>7</td>
<td>37</td>
</tr>
</tbody>
</table>

accesses. We believe that the model parameters for NEUT may be inaccurate. There is so little time associated with the memory accesses for NEUT (due to high cache-hit ratios) that small absolute least square errors can result in large relative changes to the parameters.

Table 5.2 also shows effects attributed to the number of usable registers on the two different microprocessors, namely the MIPS R10000, and Intel Xeon. The 200MHz R10000 provides 64 registers whereas the 333 MHz Pentium Xeon allows at most 40 registers for general use. This gap in registers available degrades overlap performance as expected leading to a higher percentage of overlap work performed by both the Origin 2000 and PowerChallenge. This is directly confirmed by the higher percentage of non-overlapped access time (out of nominal full latency) for HYDRO on Intel ASCI Red in both L2 and memory levels.

Figures 5.3, 5.4, and 5.5 show graphs of cpi\textsubscript{stall} relative to the overall cpi for all machines on most codes. The second half of each figure shows the corresponding overlap parameter, m\textsubscript{0}. A number of general observations are apparent from the graphs. The overall cpi on the Origin is typically less by factors of up to three on the PowerChallenge and consistently less than those measured on the Intel ASCI Red. The
percentage of CPI represented by stall time on the Origin can be less than 40%, while, on the PowerChallenge, it can be as large as 80%. Two codes, HYDRO-T and NEUT, exhibit high locality of reference and CPU stalls due to memory accesses are less than
Figure 5.5 Memory stall and overlap parameters (Intel ASCI Red)

10% of the total time. A study of the algorithms/implementations of these codes would lead one to expect this. NEUT has a modest number of scalar variables per particle that are used many times before another particle is computed (high temporal locality).

HYDRO-T is a 2D code and was re-coded from the original HYDRO so inner loops have stride-1 vectorizable loops (high spatial locality). The success of the transposition can be seen by comparing each version in the figures.

Memory overlap parameters are higher on the Origin than on the PowerChallenge, indicative of the better latency hiding capability of the Origin. As discussed previously, and confirmed by the overlap parameters, the Intel ASCI Red maintains an even lower hiding capability than both the Origin and the PowerChallenge. Two extreme examples are given: HYDRO-T with very high overlap, and SWEEP (not shown for the Intel ASCI Red), with very low overlap. The high spatial locality of HYDRO-T means that there is a great deal of parallelism between L1, L2 and main memory accesses. Additionally, on the Origin 2000, major portions of this 2-D
algorithm fit entirely in the 4-MB L2 cache. In contrast, SWEEP shows much less overlap on either the PowerChallenge or the Origin. The results for NEUT, where the PowerChallenge shows high overlap and the Origin shows very low overlap, are again due to the large parameter changes associated with the least-squares fit mentioned above.

5.3 Instruction-Level Model

5.3.1 Bottleneck Analysis of MIPS R10000

We must show the assumptions of the instruction-level model are met. There are two assumptions that need some explanation. Uniform distribution is obviously not going to be found in our codes. In our technique, we extract the average inter-arrival distance \( \bar{d} \) values from the measured codes using the PTERA tool. These values are used to create (theoretically) a synthetic, uniformly distributed, instruction stream for input into the queuing theory based instruction-level model. Our actual codes also contain dependencies. We do not model dependencies in our equations for the regular instruction-level model. Instruction streams created with \( \bar{d} \) values are (again theoretically) independent of time as discussed in the instruction-level model section in Chapter 4. We can also intuitively infer that dependencies will not influence the instruction sequence committed to machine-state. Dependencies will affect the overall number of cycles for an application, but not the order in which instructions graduate from the processor. In other words, the \( \text{cpi}_0 \) calculated is a lower bound for \( \text{cpi}_0 \) that does not incorporate the effect of dependencies and instruction clustering. The argument holds for the infinite cache assumption as well. This is actually confirmed in the validation section of the instruction-level model since all errors in measurement are
negative indicating the characteristic of underestimation. In this case, we will again be modeling a best-case scenario.

To discount the effect of branch misprediction and the overhead impact of branch instructions, we also need to obtain the ratios of branch instructions and branch mispredictions to ensure the applications can be simplified as three major instruction flows (floating point, integer, and memory). On the other hand, the instruction cache miss ratio is also considered to see if the instruction fetch effect can be significant. The key to this methodology is to estimate the $\bar{d}$. values that cause stall of the microprocessor due to the limitation of architectural constraints.

Table 5.3 exhibits branch ratios, branch misprediction ratios, and the instruction cache miss ratios for all these codes. It is clear from Table 5.3 data that both branch and instruction cache effect are negligible. Under this condition, the performance study of these codes can focus on the impact of the three major instruction flows (floating point, integer, and memory).

In Chapter 3 we showed the stability of inter-arrival distances for these codes. These figures demonstrate that they converge to constant values with increasing problem sizes. This is understood as the instruction flow pattern of a problem that reaches its steady state. This phenomenon proves that $\bar{d}$. can be used in characterizing
these codes once they reach the steady state. We have now shown that the assumptions for the instruction-level model are fairly well adhered to. We can now apply the models to the measurements obtained on the Origin 2000 and PowerChallenge. Utilizing these instruction-level characteristics, we calculate the utilization factors for each code over both machines in Table 5.4. Due to their architectural similarity, the utilization factors are identical across PowerChallenge and Origin 2000.

For Sweep, Dsweep, and Heat the only utilization factor greater than one is $\rho_m$, indicating memory instructions are the bottleneck. This leads us to declare the memory instruction utilization as our limiting factor for these codes on these machines. A limiting factor is the key contributor to stalls within the microprocessor (excluding dependencies and memory latency as we assume infinite L1 cache). For these codes, it is very likely the memory queue will fill, leading to stalls in decoding as entries graduate slower than they arrive. For Hydro and Hydro-t, we have utilization factors greater than one for the memory and integer queues. This leaves us two possibilities for the limiting factor. The queue associated with the maximum of the two utilization factors in the ideal case would fill first, namely either the integer or memory queue (statistically the measurements are the same). These scenarios can only happen however, if the maximum instruction threshold $K$ is not reached. As mentioned earlier,
K=32 for the MIPS R10000. Since the memory and integer queue lengths are both 16, we cannot reach the maximum number of instructions prior to stalling on a single queue. Thus, the limiting factor for both of these codes will be either integer or memory instructions.

Sweep, Dsweep, and Heat have utilization factors that are a good deal greater than one. This would indicate that the performance of these codes as implemented and compiled immediately loses a good deal of performance from the ideal case. This inference is confirmed by the ideal cpi\(_0\) calculations given in Table 5.5. Values for Sweep, Dsweep and Heat vary from 30%-50% over the ideal cpi of the MIPS R10000 processor. This is not to say codes should ever achieve ideal cpi. Rather the conclusion here is that inherent characteristics of these codes cause performance loss even when ideal assumptions about the code are made. Hydro and Hydro-t, two codes that achieved fairly good performance at the memory hierarchy level as seen in the previous section, do very well at the instruction-level as well. Under ideal assumptions, the instruction-mix coupled with the architecture of the MIPS R10000 allows for almost ideal performance. Thus Hydro and Hydro-t are well structured for the MIPS architecture. Again, these ideal cpi\(_0\) values are lower bounds on performance, and it is

Table 5.5 Ideal cpi\(_0\) calculated using Equation 4.12

<table>
<thead>
<tr>
<th></th>
<th>Sweep</th>
<th>Dsweep</th>
<th>Heat</th>
<th>Hydro</th>
<th>Hydro-t</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ideal cpi(_0)</strong></td>
<td>0.35</td>
<td>0.46</td>
<td>0.36</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td><strong>PowerChallenge</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Origin 2000</strong></td>
<td>0.36</td>
<td>0.47</td>
<td>0.36</td>
<td>0.27</td>
<td>0.27</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
our hope that extensions such as the extended instruction-level model will incorporate more and more functionality to closer model CPI0.

5.4 Dependence Analysis of MIPS R10000

5.4.1 Methodology

This analysis technique incorporates data dependence in the instruction-level model creating an extended technique that can be validated for synthetic streams as discussed in Chapter 4. First we discuss the methodology built upon the extended instruction-level analysis to provide the context for our discussion. Next we show how using these definitions in conjunction with the extended instruction-level model allows us to quantitatively describe dependence performance on the MIPS R10000 for synthetically created instruction streams. Again, the PTERA prototype was essential to all data collection on synthetic instruction streams. The results here required hundreds of separate runs and measurements that would have been nearly impossible to do without the automation provided in PTERA.

Using the same terminology first described in section 4.1.3 we add a few more terms and discuss the relationships between data dependences formally. Let \( m \) be the total number of types of different reservation stations or queues on a processor. Then the set \( Q = \{ q_1, q_2, q_3, \ldots, q_m \} \) provides the number of each type of available queue. Earlier we preliminarily defined \( s_c \) as the number of type of servers of type \( c \) available in a system. Similarly, there are \( m \) types of servers available. The set \( S = \{ s_1, s_2, s_3, \ldots, s_m \} \) describes the number of each type of available server. In the case of the MIPS R10000, \( m = 3 \) since we model integer, floating point, and memory types of instructions and there exists one queue for each of these types of instructions. The earlier
description of the MIPS R10000 detailing the functional units available describes the set S (with m=3 types, of course). With these and earlier descriptions, we can define the types of characteristics we vary for our dependence experiments.

*flow dependence:* This is a true data dependence that exists between $i_a$ and $i_b$ if $i_a$ precedes $i_b$ ($a < b$) and $i_a$ sets a register value that $i_b$ uses.

*inter-queue dependence:* This type of dependence exists between instructions $i_a$ and $i_b$, where $i_a \in t_a$ and $i_b \in t_b$, if $i_b$ has flow dependence on $i_a$ and $t_a \in q_c$, $t_b \in q_c$.

*intra-queue dependence:* This type of dependence exists between instructions $i_a$ and $i_b$, where $i_a \in t_a$ and $i_b \in t_b$, if $i_b$ has flow dependence on $i_a$ and $t_a \in q_c$, $t_b \in q_d$ where $c \neq d$.

*link-length of dependence:* Given flow dependence from $i_a$ to $i_b$, if no further dependences exist from $i_b$ to a succeeding instruction, we say a link-length of 1 exists from $i_a$ to $i_b$. Given the same flow dependence from $i_a$ to $i_b$ with another flow dependence from $i_b$ to a succeeding instruction $i_c$, we say a link-length of 2 exists from $i_a$ to $i_c$. Generally, given succeeding flow dependences from $i_a$ to $i_b$ to $i_c$ to ...$i_n$, when 1 instructions are linked by dependences, the link-length is said to be $l-1$.

*link-width of a dependence:* Given a flow dependence from $i_a$ to $i_b$, if $c = b - a$, then there exists a link-width of $c-1$ instructions between $i_a$ and $i_b$.

### 5.4.2 Ideal Experiments for Dependence Analysis

PTERA and SynBAD have proven exceptionally useful in the validation and extension of the original instruction-level model. Combined experimentation and modeling efforts led to the extended instruction-level model discussed in Chapter 4. While further work needs to be accomplished to find the relationship between the frequency of branches and performance in the ideal case, data dependence impact on ideal
performance can be quantified using PTERA and the definitions discussed in the
previous section. In the series of ideal experiments encompassing hundreds of different
synthetic codes and measurements, we are able to vary independently the parameters of
link-length, link-width, and the defined dependence for full coverage of instruction
mixes representing all ideal cases. We vary these parameters in an effort to quantify
their influence on performance degradation. We believe the size of these parameters in
relationship to the architectural constraints on-chip such as queue length, will allow
predictive performance of uniformly distributed instruction streams. This follows our
plan of creating models of this type by modifying rudimentary models to incorporate
more and more detail. Again, this process has been completely automated for the MIPS
R10000 using the PTERA tool.

The "ideal experiments" consist of the 256 possible combinations of instruction
sequences of length 4 that can achieve a CPI of 1/β when no dependences or branches
are present, icache and TLB misses are minimized, and all accesses are to registers
(excluding cold misses). We created and measured instances of all these instruction
sequences in the form of synthetic streams. Runs were measured for the perfect case
and then varying each of the mentioned parameters. Dependences were made from the
first to the fourth instructions in the four-instruction sequence. SynBAD eliminated any
dependences not allowed (for instance a flow dependence from integer add to floating
point multiply) leaving 16 usable combinations. For each combination we measured 17
runs varying the link-length of the data dependence from 0 to 16, for the first 16 runs
with an infinite dependence as the 17th run. We also combined these runs with 9 other
runs where we created different widths between the dependences by inserting
independent instructions of the same sequence (4 at a time) following the initial 4 instructions that contain the dependence. For the 16x16 case we added 3 sets of the same four instructions after the first four instructions that contained the dependence. This gives a repeated sequence of 16 instructions, and the 16x16 label refers to the defined dependence matrix as shown in the first section of this chapter regarding SynBAD specifications. The added instructions have no dependences causing the processor to work at the ideal rate for those extra instructions. In the case of a linked dependence, the width is increased by 12 instructions in this case. These widths are specified following the instruction sequence labels in the figures that are presented in this section. Inter-queue and intra-queue dependences are determined according to instruction type. So overall we completed 16 x 17 x 9 = 2448 runs in less than a week using the PTERA tool. Actually the runs did not take as long as collecting, organizing and analyzing the data. This data is shown in Figures 5.6-5.8 and explained later in this section. We proceed with an example to clarify our discussion of the results.

Figure 5.6 gives some results from PTERA for instruction sequences meeting the constraint of ideal cases. In particular, Figure 5.6 plots the link-length on the x-axis versus the cpi on the y-axis. Figure 5.6 shows results for an instruction stream consisting of a floating point add (fa), followed by two integer adds (ia), and a floating point multiply (fm). This constitutes a basic block. A link-length of 3 describes the case where two basic blocks totaling 8 instructions contains three links. These 8 instructions would be fpadd-iadd-iadd-fpmul-fpadd-iadd-iadd-fpmul, in that order. The three dependences would be from the first fpadd to the first fpmul, the first fpmul to the second fpadd, and the second fpadd to the second fpmul. A dependence of length 2
would be the same without the last dependence. This set of instructions with such dependences is repeated approximately 1 million times and cpi is measured using performance monitors on the MIPS R10000 via the PTERA tool. This dependence describes and inter-queue relationship. The resulting code purposely contains no cache, branch or icache misses so performance degradation is attributable to our variables. This figure describes only this instruction stream case.

The 4x4 plot indicates no extra instructions are introduced between basic blocks. The 8x8 plot means the basic block was extended to include 4 extra independent instructions of the same mix as the original 4 instruction basic block. By varying the number of dependence links (the x-axis) we see an increasing cpi trend for the 4x4, 8x8, and 12x12 cases. This indicates performance degradation caused by these dependences.
As the number of independent instructions introduced between linked-chains increases or to use our introduced terminology, the link-width increases (i.e. the 16x16 case and all larger cases), this trend disappears. It is also apparent that performance is better in these larger cases generally regardless of the dependence link-length.

We can easily quantify the actual $cpi_{dep}$ given in Equation 4.18. We do not show those calculations here since the graphs indicate trends in $cpi_{dep}$ that would not be seen as easily in a very large table of $cpi$ values. One can simply draw a horizontal line at $1/\beta=.25$ and compare the achieved $cpi$ visually. Equation 4.18 gives a simple method for this calculation and allows us to quantify the $cpi_{dep}$ contribution formally if necessary. For describing the performance of these streams however, we feel the charts provide a more intuitive way of describing a large amount of information succinctly.

The reader may notice achieved $cpi$ in the ideal cases without performance degradation is about .27 instead of .25. This is tolerable variance in the counters due to overhead and accuracy issues, but is close enough for our measurement purposes. Closer values can be obtained requiring longer runs, but since increasing the run time over almost 2500 runs is significant, we use these measurements under compromise.

These types of conclusions can be drawn for simply one instruction stream case. Other cases show some or no influence on performance due to dependences. These types of results are exactly what were initially intended by the concept of SynBAD. To minimize redundancy, we present only two other figures to enable description of all 16 instruction mixes. Figure 5.6 is also representative of the stream described by $fmiaiafa$ (fpmul-intadd-intadd-fpadd), which simply shows that reversing the positions of the
dependent instructions gives nearly identical performance on average as link-lengths and -widths vary.

Figure 5.7 shows the results for the instruction sequence iaafmia (intadd-fpadd-fpmul-intadd), another inter-queue relationship. As shown in the chart, dependence has influence only when the width is small, in the 4x4 case. Inserting independent instructions washes away the effect of the linked dependences. The link length does progressively influence performance in the 4x4 case indicating overall that the processor does a good job of handling integer-add dependences, but is still influenced by loop-carried dependences and their associated lengths. The performance degradation is not of the same magnitude as in the previous figure since integer-add throughput is
Figure 5.8 Performance variation in relation to link-length and link-width.

better overall than floating point throughput. The two fully functional integer-add
functional units of the MIPS R10000 dictate this. Only when instructions with
dependences are sufficiently close together, are these units overwhelmed into degraded
performance. Switching the order of the independent instructions fa (fpadd) and fm
(fpmul) has no influence on performance. This figure is not shown for brevity.

The last instruction sequences in the 16 combinations include 12 combinations
that have identical performance. Figure 5.8 shows the results for one of these 12
combinations, the case of fsfaiafm (fpstore-fpadd-intadd-fpmul). Here and in the other
11 combinations, an add or multiply (floating point or integer) is dependent on a store
of the same type. There are also variants on the interior independent instructions. But
overall, as shown in Figure 5.8, no dependence influence occurs for all the varied link-lengths and -widths. This would indicate that the MIPS R10000 architecture handles these instruction sequences at the ideal rate regardless of the dependence lengths and widths. But does this make sense? The answer is yes. This phenomenon is due to the fact that the linked dependences described are not exactly continuous in this case. While the dependence from instruction 1 to instruction 4 is accomplished, register allocation for the store instruction is such that a continued dependence is not allowed from instruction 4 in the first basic block to instruction 1 in the second basic block. The processor is able to shift fetch and decode such that it keeps the single recurring dependence from affecting performance by shifting which four instructions are fetched and decoded each cycle. So this is not a truly linked dependence, and the performance does not degrade because the broken dependence link allows shifting back into the ideal fetch and decode performance rate of \(1/\beta\). This illustrates the robust ability of SynBAD to allow dependences just as specified within the parameters of the assemble code for a particular architecture.

5.5 Architecture Advances via Modeling

5.5.1 Mutable Functional Unit (MFU)

The instruction-level model, as presented provides insight into the performance bottlenecks caused by a mismatch between the instruction-mix and the functional unit allocation on a processor. The results of our modeling efforts indicated processors could conceivably benefit from an architecture that supported dynamic allocation of functional unit resources. The workload characterization of SPEC codes presented in Chapter 3 reinforced our preliminary conclusions. With the help of some key
collaborators in architecture and compiler performance, we developed a successful grant proposal to pursue such architectural development. We present a brief synopsis of the initial results for the purpose of supporting our contentions as to the usefulness of such modeling techniques. Yan Solihin, Yong Luo, Maya Gokhale, and Dominique Lavenier are to be thanked for their contributions to this large body of research. The writer of this dissertation was responsible for much of the underlying theory, the initial idea, workload characterization, algorithm development, and overall analysis. We discuss performance variations as the inverse of cpi, ipc. This is customary in the architecture community.

Our goal is to augment a superscalar processor with reconfigurability without requiring specialized compilers, large investment in custom fabrication technology, and complex synchronization between subsystems running at very different clock rates. We modify the R10000 floating-point adder so that it is able to additionally perform integer operations. In choosing which integer operations are to be executed by the MFU, our priority is to accommodate frequently executed integer instructions. Instruction profiles of Spec95 using SimpleScalar compilation show that integer addition, followed by integer shift and logic operations, are the most frequent integer instructions. In addition, memory instructions are as frequent as integer operations. This approach identifies the relative frequency of certain instruction types over the entire code. However, we are also interested in profiling the clustering behavior of instructions of the same type. In Chapter 3, results of characteristic profiling are given for the Spec95 codes. In particular, we identify the clustering of instructions by distance between two consecutive instructions of the same type. The results confirm the simple frequency
profiling showing integer-additions and memory operations are the most "clustered" and dominant instructions for the codes in this study. This provides further evidence that only limited modification of the original floating point adder is necessary to achieve performance improvement.

Thus, based on both profiling results, we designed the MFU to be able to execute integer addition, shift, and logic operations, plus address generation for memory operations. The design requires widening the adder data path to 64 bits and adding a few switches to the floating-point adder to enable the unit to mutate into an integer adder/shifter. The resulting hardware design revealed that an MFU roughly has the same number of gates as a floating-point adder.

One important aspect that affects the performance of the MFU is its mutation penalties, which are shown in Table 5.6. The penalty is particularly high (2 cycles)

<table>
<thead>
<tr>
<th>Mutation Category</th>
<th>Current Instruction</th>
<th>Next Instruction</th>
<th>Instruction After Next</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer to FP mutation</td>
<td>Logic/Add</td>
<td>FP-ADD</td>
<td>FP-ADD</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Shift</td>
<td>FP-ADD</td>
<td>FP-ADD</td>
<td>1</td>
</tr>
<tr>
<td>FP to integer mutation</td>
<td>FP-ADD</td>
<td>Logic</td>
<td>Not-ADD</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>FP-ADD</td>
<td>Logic</td>
<td>ADD</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FP-ADD</td>
<td>Shift</td>
<td>All-Integer</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>FP-ADD</td>
<td>ADD</td>
<td>All-Integer</td>
<td>2</td>
</tr>
</tbody>
</table>

when we switch from floating-point capability to integer addition capability. This penalty is due to the need to wait for the floating-point pipeline to partially drain before we are able to use it for integer addition. Thus, it is important to reduce the frequency of mutation so as to avoid such high mutation penalties.
5.5.2 Alternative Architectures

We chose to use the MIPS R10000 architecture as the basis of our study. As a reference, the architecture of MIPS R10000 is shown in Figure 5.9. The functional units consist of 2 integer ALUs. One is capable of performing basic operations (add/sub, logic) plus branch and shift operations, and the other is capable of performing basic plus integer multiplication and division. There is one Address Generation Unit (AGU) which is embedded into the Load Store Unit (LSU). Finally, there are 2 floating-point units (FPUs). FPU1 is capable of performing addition, and FPU2 is capable of performing multiplication, division, and square root operations. There are three reservation stations: integer, floating point, and memory/address reservation stations. Each reservation station has 16 entries, and issues instructions in an out-of-order manner to the respective functional units.
The basic modification needed is to replace the floating point adder (FPU1 in R10000) with an MFU, which is able to perform floating point addition, integer addition, logic, shift operations, and address generation. Note that memory operations are sent to MFU for address generation only, while the actual loads and stores are performed by the LSU. Next, we must design an architecture that exploits the MFU. We need to determine the pipeline stage in which to perform analysis of the instruction stream and the actual mutation of the MFU. Based on this, we consider three schemes as shown in Table 5.7.

In fetch profiling (FProf), analysis is performed on the instructions that are fetched from the instruction cache. When the fetched instruction mix shows a need for more integer execution or address generation bandwidth, the MFU is mutated to serve integer and memory operations. Otherwise, the MFU is mutated to perform floating point addition operations.

In Reservation Station Monitoring (RSMon), the analysis is performed by inspecting the fullness of the reservation stations. If one reservation station is full of instructions, more bandwidth is needed to service instruction types of that reservation station, thus the MFU is mutated to serve the reservation station. The architecture modification needed for this scheme is minimal, as it only needs to detect whether one

Table 5.7 Architecture based on time of analysis and mutation

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Fetch Profiling (Fprof)</th>
<th>RS Monitoring (RSMon)</th>
<th>Dedicated RS (RS-MFU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Fetch</td>
<td>Dispatch</td>
<td>Dispatch</td>
</tr>
<tr>
<td>Mutation</td>
<td></td>
<td></td>
<td>Issue</td>
</tr>
<tr>
<td>Modifications to R10000 functional unit configuration</td>
<td>FPU-&gt;MFU</td>
<td>FPU1-&gt;MFU</td>
<td>FPU1-&gt;MFU</td>
</tr>
<tr>
<td></td>
<td>Augment FPU2 with fp-adder</td>
<td>Augment FPU2 with fp-adder</td>
<td></td>
</tr>
</tbody>
</table>
reservation station becomes full, and react to it. The drawback is high switching frequency: when the MFU is mutated to serve a reservation station, it reduces the execution bandwidth of other reservation stations, which may quickly become full, necessitating a new mutation.

In both FProf and RSMon schemes, there is a possibility of starving floating point addition operations as the MFU is the only unit that is capable of executing floating point additions. This happens when there is a change of instruction stream from floating point operation intensive to integer operation intensive. When the schemes detect such a change, they mutate the MFU to serve integer operations. However, it is possible that floating-point addition operations in the floating-point reservation station have to stall for execution because the MFU is serving integer operations. Stalling these instructions in the end stall other instructions that have finished execution but cannot be committed. Thus, to avoid starving floating-point operations, it is necessary to augment the original FPU2 (multiplier/divider) with an adder, as shown in Table 5.7.

In the third scheme (RS-MFU), analysis of instructions is performed at the dispatch stage right after fetched instructions are decoded for operands. After decoding the operands, register renaming is performed in parallel with a steering logic that selects a subset of instructions to be executed by the MFU. The selected instructions are dispatched to a new reservation station (RS-MFU) that will issue the instructions only to the MFU, as illustrated in Figure 5.10. If the MFU detects that the new instruction has a different type compared to the one it is serving, it performs the mutation and executes that instruction. The steering is performed in parallel with register renaming to avoid any effect on clock frequency. Memory instructions that are sent to RS-MFU are
also sent to the address reservation station. MFU only performs the address generation, then passes the results to the address reservation station.

Since all floating-point additions have to be executed by the MFU, we do not add floating-point execution bandwidth while potentially reducing it, we expect to see performance improvement for integer applications, but we have to demonstrate that the performance of floating-point applications does not suffer.

One issue with the RS-MFU scheme is the die area occupied by the new reservation station. Fortunately, since the reservation station will accommodate all floating-point addition operations, the floating-point reservation station only contains floating-point multiplication, division, and square root operations. Thus, the number of entries in the floating-point reservation station can be reduced from 16 entries to 8
entries, while the RS-MFU contains 8 entries. Thus, the total area for reservation stations for the RS-MFU scheme is roughly the same as that of R10000.

Another issue is whether the reservation station (RS-MFU) should implement in-order or out-of-order issue. The potential benefits of using an out-of-order RS-MFU are: reduction of stalls due to larger instruction issue window and reduction of mutation frequency by prioritizing the issue of instructions of the same type. However, at this moment we believe that out-of-order issue advantages are not worth the added complexity. First, reducing mutation frequency by prioritizing instructions of the same type has the potential drawback of delaying the execution of an earlier instruction in the RS-MFU, causing all dependent instructions in other reservation stations to stall. Furthermore, reducing mutation frequency can be achieved more effectively by tuning the steering logic. Second, the advantage of reducing stalls with a large instruction window may have the adverse effect of increasing mutation frequency. Reordering instructions before they enter the RS-MFU can approximate the benefit of out-of-order issue. This is beyond the scope of this work and for now we simply use an in-order implementation.

5.5.3 Comparisons of the Alternative Architectures

In FProf and RSMon schemes, the hardware modifications include replacing FPU1 with an MFU, augmenting an adder into FPU2, and wiring all reservation stations to the MFU so that the MFU can receive instructions from all reservation stations. The hardware increase of augmenting an adder into FPU2 seems non-trivial. It may in fact be comparable to a brute-force approach of adding extra functional units. In addition, requiring the MFU to be connected to all reservation stations is not a scalable approach.
The RS-MFU scheme has two advantages compared to FProf and RSMon schemes. First, it does not add die area to the existing functional unit area because FPU2 is not augmented with a floating-point adder. Second, it is more scalable since the MFU is connected only to one reservation station.

The hardware cost for the RS-MFU scheme is a new reservation station, steering logic, and extra read and write ports in the integer register file. As mentioned earlier, the die area resulting from the new reservation station is compensated by reduction in the number of entries of floating point reservation station. In a later section, we show that this configuration yields the same performance as having a 16-entry RS-MFU and 16-entry floating point reservation station. In addition, the steering logic is simple and does not add a lot of die area. It also performs in parallel with register renaming so that clock frequency is not affected.

In our study, we make an assumption that providing data path from both the integer register file and the floating-point register file to and from the MFU is feasible. In real implementation, the two register files may be located physically apart. Thus, actual MFU implementation may require layout change or a multi-cycle read and write from one of the register files. We have completed a study regarding the steering logic and an associated algorithm. For brevity in this document we simply state that our algorithm efficiently steers instructions to appropriate queues for the purposes of the multiple functional unit.

5.5.4 Results

Figure 5.11 shows the IPC of the base architecture (first bar), RSMon scheme (second bar), FProf scheme (third bar), RS-MFU scheme with 8-entry RS-MFU and 8-entry
floating point reservation station (fourth bar), and the addition of an integer ALU that also calculates addresses for memory operations (fifth bar). The fifth bar is provided for comparison to assess the effectiveness of the FProf, RSMon, and RS-MFU schemes compared to a less-scalable brute-force approach of simply adding an extra ALU unit that also performs address generation for memory operations (AGU). Though it is not a scalable approach, the brute-force approach provides the maximum attainable performance for the other schemes.

For integer applications, we have interesting results. All schemes improve the IPC for all applications. However, the improvement of FProf and RSMon schemes is consistently lower than the improvement from the RS-MFU scheme, especially for compress and ijpeg. FProf outperforms RSMon for ijpeg, while RSMon outperforms FProf for li.
The RS-MFU scheme improves the IPC of integer applications from 8.3% for compress to 14.3% for kmeans. The load balancing due to the steering of integer and memory instructions to the RS-MFU explains this performance improvement. Since there are few or no floating point addition operations, the MFU will provide more integer execution and address generation bandwidth for all or most of the time. The figure also shows that for integer applications, RS-MFU scheme achieves comparable IPC with a brute-force approach of adding an extra ALU/AGU.

For floating-point applications, the IPC of the base architecture is slightly higher than the architecture with an additional ALU/AGU, showing that floating point applications do not need additional integer execution or address generation bandwidth. Consequently, adding an MFU using any scheme, which adds integer execution and address generation bandwidth, will have little impact on IPC, the fact that is shown in Figure 5.11.

The figure shows that the IPC of floating-point applications for all schemes are comparable. This is due to the fact that the MFU does not add any additional floating-point execution bandwidth. Thus, the MFU is only beneficial when there is no floating-point addition, for example, during initialization phase. For RS-MFU scheme, this extra bandwidth gives su2cor and swim a little bit of IPC improvement (1.3% and 0.9%, respectively). However, for su2cor, the additional IPC is apparently offset by the cost of mutation (IPC decreases by 0.6%), due to frequent mutation as shown in Table 5.8.

The table shows that the mutation frequencies for ijpeg and li are very low because there are no floating-point addition instructions, thus the MFU always serve integer and memory instructions. On the other hand, Kmeans and compress have 0.7%
and 0.5% floating-point addition instructions, resulting in a higher mutation frequency compared to jpeg and li, although still much lower compared to the mutation frequency of floating point applications.

**Table 5.8 Mutation frequency**

<table>
<thead>
<tr>
<th>Application</th>
<th>Average # instr. Per mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>40.5</td>
</tr>
<tr>
<td>wave5</td>
<td>16.5</td>
</tr>
<tr>
<td>su2cor</td>
<td>21.1</td>
</tr>
<tr>
<td>compress</td>
<td>370</td>
</tr>
<tr>
<td>jpeg</td>
<td>7219576</td>
</tr>
<tr>
<td>li</td>
<td>7065704</td>
</tr>
<tr>
<td>kmeans</td>
<td>325</td>
</tr>
</tbody>
</table>

Compared to the additional ALU/AGU architecture, the RS-MFU scheme achieves over 97% of the IPCs for floating-point applications (98.9% for swim, 97.7% for wave5, and 99.4% for su2cor). Overall, we have shown that RS-MFU is the most effective scheme compared to FProf and RSMon. We also have shown that the RS-MFU improves the performance of integer applications significantly (as significant as adding an extra ALU/AGU), while maintaining the performance of floating point applications. All the performance gain is obtained with very little hardware cost.

The effect of the size (number of entries) of RS-MFU is shown in Figure 5.12. In addition to the 8-entry RS-MFU with 8-entry floating point reservation station that is shown in Figure 5.12 (base RS-MFU scheme), Figure 5.12 shows the IPC of a 16-entry RS-MFU, an 8-entry RS-MFU, and a 4-entry RS-MFU, all with the original 16-entry floating-point reservation station. We found that there is virtually no difference in IPC.
between the base RS-MFU scheme with the 16-entry RS-MFU with 16-entry floating-point reservation station. The reason that we don’t lose performance when reducing the number of entries in the floating point reservation station to 8 is that the reservation station now only holds multiplication, division, and square root instructions, with all addition operations sent to RS-MFU.

However, one interesting result is that for some applications (su2cor and kmeans), the 8-entry RS-MFU achieves better performance than a 16-entry RS-MFU. The reason for this is that the 8-entry RS-MFU provides better instruction distribution balance across the reservation stations. When there are no floating-point instructions, placing an integer or memory operation in RS-MFU or other reservation stations (because the RS-MFU is full) can make a difference in performance.

In this case, the performance is higher when we dispatch the operations to other reservation stations. For a 4-entry RS-MFU, however, IPC is lost compared to an 8-
entry RS-MFU, most notably for swim and kmeans. The reason for this is that instruction distribution worsens when we can only put 4 instructions in the RS-MFU. In particular, more instructions are sent to other reservation stations, giving other functional units increased loads, increasing load imbalance.

Table 5.9 Percent of time RS-MFU is full

<table>
<thead>
<tr>
<th>Applications</th>
<th>16-entry RS-MFU</th>
<th>8-entry RS-MFU</th>
<th>4-entry RS-MFU</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>0.00%</td>
<td>12.24%</td>
<td>56.96%</td>
</tr>
<tr>
<td>wave5</td>
<td>0.00%</td>
<td>6.97%</td>
<td>44.33%</td>
</tr>
<tr>
<td>su2cor</td>
<td>0.00%</td>
<td>7.26%</td>
<td>42.67%</td>
</tr>
<tr>
<td>compress</td>
<td>0.00%</td>
<td>0.96%</td>
<td>22.15%</td>
</tr>
<tr>
<td>ijpeg</td>
<td>0.00%</td>
<td>2.44%</td>
<td>25.19%</td>
</tr>
<tr>
<td>li</td>
<td>0.00%</td>
<td>0.24%</td>
<td>18.09%</td>
</tr>
<tr>
<td>kmeans</td>
<td>0.00%</td>
<td>0.61%</td>
<td>31.15%</td>
</tr>
</tbody>
</table>

Table 5.9 shows the percent of execution time the RS-MFU is full. Swim and kmeans are the applications that lost IPC the most when using a 4-entry RS-MFU. They also maintain high percentages of time full for the RS-MFU, 56.96% and 31.15% respectively.

This concludes our discussion of the mutable functional unit. The relation to the instruction-level modeling efforts and workload characterization should be clear. The modeling efforts enable us to see the benefit of such a unit while actually designing and implementing the unit provided the proof of concept.

5.6 Memory Hierarchy Evaluation Using the Statistical Method

We have applied this method to the SGI Origin 2000 and PowerChallenge Machines due to their similarities and differences as discussed in Chapter 3. All four levels of evaluation have been used to evaluate these ASCI machines and benchmarks.
Experimental results show the adaptive statistical method is feasible and effective. In our experimental testing, the two machines, PowerChallenge and Origin2000, are denoted as machine level 1 and level 2, respectively. The five codes, HEAT, HYDRO, SWEEP, DSWEEP, and HYDROT, are denoted as 1, 2, 3, 4, 5, respectively. We have used SAS throughout the experimental evaluation. The problem sizes used in the experiment range from \( N=50 \) to memory/time constraints. The corresponding range for the codes are: HEAT = \([50, 100]\), HYDRO = \([50, 300]\), SWEEP = \([50, 200]\), DSWEEP = \([50, 200]\), HYDROT = \([50, 300]\). All the experimental data are measured from single node sequential executions using the SGI hardware performance counters via the PTERA performance tool.

### 5.6.1 Main and Interaction Effects

The relationship between code and machine is first investigated. To catch the mean relationship over the range of problem sizes, replicate measurements have been taken for different problem sizes for a given experimental unit. The two-factor factorial experiment is used to find the effects. The GLM procedure of SAS is used to analyze the two-factor factorial experiment for level one evaluation. Tables 5.10 and 5.11 show results from GLM. Table 5.10 lists the GLM model class level information. Table 5.11 is the mean effects table of the factorial experiment. It consists of two sectors separated

<table>
<thead>
<tr>
<th>Class</th>
<th>Levels</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>2</td>
<td>1 2</td>
</tr>
<tr>
<td>CODE</td>
<td>5</td>
<td>1 2 3 4 5</td>
</tr>
</tbody>
</table>

Number of observations in data set = 113
Table 5.11 Mean effects table

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Sum of Squares</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Pr&gt;F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>9</td>
<td>112.5410006</td>
<td>12.5045556</td>
<td>27.44</td>
<td>0.0001</td>
</tr>
<tr>
<td>Error</td>
<td>103</td>
<td>46.9436516</td>
<td>0.4557636</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corrected Total</td>
<td>112</td>
<td>159.4846523</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R-Square: 0.705654  C.V.: 34.64445  Root MSE: 0.675103

<table>
<thead>
<tr>
<th>Source</th>
<th>DF</th>
<th>Type I SS</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Pr&gt;F</th>
</tr>
</thead>
<tbody>
<tr>
<td>MACHINE</td>
<td>1</td>
<td>14.39563307</td>
<td>14.39563307</td>
<td>31.59</td>
<td>0.0001</td>
</tr>
<tr>
<td>CODE</td>
<td>4</td>
<td>93.17895152</td>
<td>23.29473788</td>
<td>51.11</td>
<td>0.0001</td>
</tr>
<tr>
<td>MACHINE*CODE</td>
<td>4</td>
<td>4.96641604</td>
<td>1.24160401</td>
<td>2.72</td>
<td>0.0334</td>
</tr>
</tbody>
</table>

by the double-line. The upper table is for overall effect, and the lower table is for individual effects. Look at row four of Table 5.11. The F value is 27.44 and the probability of F (Pr > F ) is 0.0001, which is less than 0.05. The hypothesis that an overall-effect does not exist is rejected. This means that code or machine effects exist. The lower table is a continuation of the upper table to locate the potential effects. Look at row two of the lower table. The probability of F is 0.0001 < 0.05, which suggests that a machine main effect exists. The same conclusion can be drawn for code.

For machine and code interaction, the probability of F is 0.0334, which is again smaller than 0.05. An interaction effect for code and machine also exists. Evaluation should be continued to understand these effects. The mean effect analysis can be explained visually. As depicted in Figure 5.13, the code performance crosses over the two machines between code 2 and code 3. This line crossing indicates the existence of an interaction effect of machine and code. It confirms the results given by the Contrast
method (see Table 5.11). However, codes 2 and 3 have very similar performances on the two machines.

If we can take code 2 and 3 as one code through classification, then there is no code performance crossing over the two machines and, therefore, no interaction effect for machine and code. Classification of code and machine is important for understanding measured performances. In fact, based on our level 2 evaluation, codes 2
Table 5.12 Contrast method for pairwise comparison

<table>
<thead>
<tr>
<th>Contrast</th>
<th>DF</th>
<th>Contrast SS</th>
<th>Mean Square</th>
<th>F Value</th>
<th>Pr &gt; F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heat vs. Dsweep</td>
<td>1</td>
<td>18.73737434</td>
<td>18.73737434</td>
<td>41.11</td>
<td>0.0001</td>
</tr>
<tr>
<td>Heat vs. Sweep</td>
<td>1</td>
<td>6.48938939</td>
<td>6.48938939</td>
<td>14.24</td>
<td>0.0003</td>
</tr>
<tr>
<td>Heat vs. Hydro</td>
<td>1</td>
<td>8.44857266</td>
<td>8.44857266</td>
<td>18.54</td>
<td>0.0001</td>
</tr>
<tr>
<td>Heat vs. Hydro-t</td>
<td>1</td>
<td>25.87993484</td>
<td>25.87993484</td>
<td>56.78</td>
<td>0.0001</td>
</tr>
<tr>
<td>Dsweep vs. Sweep</td>
<td>1</td>
<td>42.24375672</td>
<td>42.24375672</td>
<td>92.69</td>
<td>0.0001</td>
</tr>
<tr>
<td>Dsweep vs. Hydro</td>
<td>1</td>
<td>51.96661369</td>
<td>51.96661369</td>
<td>114.02</td>
<td>0.0001</td>
</tr>
<tr>
<td>Dsweep vs. Hydro-t</td>
<td>1</td>
<td>84.81327756</td>
<td>84.81327756</td>
<td>186.09</td>
<td>0.0001</td>
</tr>
<tr>
<td>Sweep vs. Hydro</td>
<td>1</td>
<td>0.00268119</td>
<td>0.00268119</td>
<td>0.01</td>
<td>0.939</td>
</tr>
<tr>
<td>Sweep vs. Hydro-t</td>
<td>1</td>
<td>4.41163307</td>
<td>4.41163307</td>
<td>9.68</td>
<td>0.0024</td>
</tr>
<tr>
<td>Hydro vs. Hydro-t</td>
<td>1</td>
<td>5.40337655</td>
<td>5.40337655</td>
<td>11.86</td>
<td>0.0008</td>
</tr>
<tr>
<td>Machine1 vs. Machine2</td>
<td>1</td>
<td>19.78987372</td>
<td>19.78987372</td>
<td>43.42</td>
<td>0.0001</td>
</tr>
</tbody>
</table>

and 3 are statistically the same (see Table 5.13). The two lines between code 2 and code 3, therefore, statistically are merged to one line.

Figure 5.14 plots the codes performance over the two machines. We can see that machine 2 always outperforms machine 1, so a machine effect does exist. Based on two-factor factorial mechanisms the GLM procedure systematically finds the main and interaction effects, which sometimes, but not always, can be determined easily through visual display.

5.6.2 Code and Machine Classification

The codes and machines have been classified based on the Contrast and Post Hoc comparisons common in statistical methods. The Contrast procedure of SAS is used for the Contrast comparison. The result of the pairwise code/machine Contrast comparison is given in Table 5.12. All the probabilities of rejection are less than 0.05, except at row nine. Code HYDRO and SWEEP are in the same group. They have similar performance variations caused possibly by the computational pattern and/or the data reference.
pattern. All other codes, namely HEAT, DSWEEP, and HYDROT, have their own signatures. They each belong to different groups. The two machines are also in two different groups.

The LSD procedure of Post Hoc comparison is also applied to classify the sets of codes and machines. Tables 5.13 and 5.14 give the result of the code and machine classification respectively. From Table 5.13 we can see that HEAT belongs to group B; DSWEEP belongs to group A; HYDROT belongs to group D; and HYDRO and SWEEP belong to group C. The result is the same as that of Contrast comparison. In the Post Hoc comparison, the grouping distance used is 0.4072. The groups are ordered according to their mean cpi values. The group with the highest cpi value (worst in performance) is listed first. The group with the second highest cpi value is listed second, and so on.

<table>
<thead>
<tr>
<th>T Grouping</th>
<th>Mean</th>
<th>N</th>
<th>CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3.7324</td>
<td>17</td>
<td>4(DSWEEP)</td>
</tr>
<tr>
<td>B</td>
<td>2.4568</td>
<td>22</td>
<td>1(HEAT)</td>
</tr>
<tr>
<td>C</td>
<td>1.6287</td>
<td>28</td>
<td>2(HYDRO)</td>
</tr>
<tr>
<td>C</td>
<td>1.6048</td>
<td>18</td>
<td>3(SWEEP)</td>
</tr>
<tr>
<td>D</td>
<td>1.0074</td>
<td>28</td>
<td>5(HYDROT)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T-Grouping</th>
<th>Mean</th>
<th>N</th>
<th>MACHINE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2.3217</td>
<td>54</td>
<td>1(PowerChallenge)</td>
</tr>
<tr>
<td>B</td>
<td>1.65552</td>
<td>59</td>
<td>2(Origin2000)</td>
</tr>
</tbody>
</table>

Table 5.13 LSD post hoc comparison for code

Table 5.14 LSD post hoc comparisons for machines

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
It is interesting to note the implications of these simple results for code classification. We observe that with the exception of HYDRO and SWEEP, each code has a unique performance variation pattern that warrants further investigation. As will be shown, these unique patterns can be further broken down into individual effects contributed by differences in the memory hierarchy in this particular test environment. These patterns directly contribute to the inherent scalable performance across machines for these particular codes. As shown in Table 5.14, PowerChallenge and Origin2000 are classified into two different groups. The distance between the two groups is larger than 0.2522 (least significant difference = 0.2522 cpi). The Origin2000 is always better than PowerChallenge for the set of codes under consideration. This result again matches that of Contrast comparison.

**5.6.3 Scalability Comparison**

Using a regression method we conduct scalability comparisons on all of the five codes over the two machines. Recall that this third step in our methodology compares the data scalabilities of a given code on different machines, whereas the level two evaluation grouped codes based on their average performance over the range of problem sizes. As we discussed in the previous section, a better memory system should lead to a smaller
cpi, and a more scalable memory system should have a smaller cpi increase, or no cpi increase at all, as problem size scales up. The procedure PROG REG of SAS is used for the scalability comparison. The response variable is cpi. PROG REG generates table 5.15 for the scalability comparison of HEAT over problem size range [50,100].

In Table 5.15, "INTAC" stands for INTerACtion effect. At the 0.0001 level (see last column of Table 5.15), the hypothesis of zero effect has been rejected, so an interaction effect exists. The parameter estimate of "INTAC" is 0.0795, which means that the term \( \beta_{c,m} \) is positive (see equations in Section 4.2.3.3) and the performance difference of the two machines decreases with problem size. PowerChallenge is more scalable than Origin2000 over the range of problem sizes. This reduction in difference is very reasonable. When problem size increases into main memory, the advantage of having a larger L2 cache fades away. The performances of the two machines, therefore, become closer.

Different codes have different memory access/computing ratios and have different memory reference patterns. Some codes have good locality and some do not. Some memory reference patterns can take advantage of the underlying memory support, some cannot. These factors and others give codes different scalabilities on different memory systems. While the resulting table is not shown, HYDRO has an INTAC probability level of 0.0111 indicating interaction effects exist for HYDRO. Unlike HEAT, HYDRO's parameter estimate is -.050885 which means that the performance difference increases with problem size. Origin 2000 has a better scalability than PowerChallenge for HYDRO. The scalability improvement may be due to Origin2000's larger L2 cache or hardware support in handling cache misses or faster memory access.
Table 5.16 Scalability comparison of Sweep

| Variable | DF | Parameter Estimate | Standard Error | T for Ho: Parameter=0 | Prob>|T| |
|----------|----|--------------------|----------------|------------------------|----------|
| INTERCEP | 1  | 1.613494           | 0.02647227     | 60.95                  | 0.0001   |
| CODE     | 1  | 0.049352           | 0.00966631     | 5.106                  | 0.0003   |
| MEMORY   | 1  | -0.390073          | 0.02647227     | -14.735                | 0.0001   |
| INTAC    | 1  | 0.012463           | 0.00966631     | 1.289                  | 0.2216   |

time. The results for codes SWEEP, DSWEEP and HYDROT are different. Our null hypothesis stands. The more advanced memory system of Origin2000 does not improve the performance difference of these three codes when problem sizes scale up. The relative performances over the two machines remain unchanged.

Table 5.16 lists results generated by PROG REG for scalability analysis of SWEEP. From Table 5.16, the probability level of interaction effect is 0.2216. Therefore, $\beta_{c,m} = 0$ and SWEEP has the same scalability on the two machines. For DSWEEP and HYDROT, the probability level of interaction effect is 0.3002 and 0.2799 respectively.

5.6.4 Evaluation of Memory Components

The memory systems of the SGI machines consist of four primary components: L1 cache, L2 cache, outstanding cache misses, and main memory. In the level four evaluation we examine the role of the four components in scalability variation. The same regression method used in the scalability study is used here. We use SAS procedure PROC REG to evaluate the relative performance of L1 and L2 cache independently. The response variable is the cache-hit ratio of L1 and L2 accordingly. The cache-hit ratios of L1 and L2 are independent of each other and can be used as
independent variables. Outstanding cache misses cannot be measured. However, based on the scalability comparison given in the previous section, its role in performance variation can be estimated when the variations of the L1 and L2 hit ratios are known. Tables 15.17 and 15.18 show the analysis table for L1 and L2 hit ratio variation of HEAT. We can see from Table 5.17 that the probability level of "INTAC" is 0.3156 > 0.05. The null hypothesis is true for the L1 hit ratio of HEAT. HEAT has a constant L1 hit ratio difference over the two machines. By Table 5.18, code-machine interaction effect exists ($\alpha = 0.001 < 0.05$) and the effect is negative ($\beta_{c,m} = -0.05011 < 0$). In practice we prefer a smaller cpi and a larger hit ratio. The negative effect means that the L2 hit ratio difference of HEAT on Origin 2000 goes down relative to PowerChallenge, when problem size goes up. As we know from a previous section,

Recall that the underlying SGI PowerChallenge and Origin2000 machine have the same CPU and the same L1 cache. It is no surprise that the relative L1 hit ratio does not change for all five codes. HEAT has demonstrated how the regression method can be used repeatedly for different components of a memory system. Table 5.19 is the L2 hit ratio analysis table for HYDRO. As given in Table 5.19, the null hypothesis of interaction is accepted. The hit ratio differences of HYDRO remain the same for the SGI machines when problem size scales up. As analyzed previously, HYDRO on the Origin2000 has a better scalability than HYDRO on PowerChallenge. This scalability increase is not due to the larger L2 cache of Origin2000 as shown by the cache hit ratios across machines. It is due to the outstanding cache miss ability and faster main memory access time supported by Origin2000.

We have applied the four-level evaluation proposed to analyze the performance of two ASCI machines and five benchmarks. In the level one evaluation we have found that both code and machine effects exist. Performance varies with codes and machines. Continued from the first level evaluation, in level two evaluation, the codes and

| Variable | DF | Estimate | Standard Error | Parameter=0 | Prob>|T|
|----------|----|----------|----------------|-------------|-------|
| INTERCEP | 1  | 0.911569 | 0.00944229     | 96.541      | 0.0001|
| CODE     | 1  | -0.011458| 0.00211136     | -5.427      | 0.0001|
| MEMORY   | 1  | 0.046284 | 0.00944229     | 4.902       | 0.0001|
| INTAC    | 1  | 0.003901 | 0.00211136     | 1.847       | 0.0771|
machines have been classified into four and two groups respectively based on their
performance. This classification shows that, while the codes have a wide distribution in
performance due to their inherited memory reference/computation patterns, the
Origin2000 definitely outperforms PowerChallenge on all the codes. It is interesting to
note, that, despite the fact that all the codes had a better performance on Origin2000, by
level three evaluation these codes have different relative performance variations over
the two machines when problem size scales up. When problem size becomes large, the
performance difference of HEAT on these two machines becomes smaller and the
performance difference of HYDRO on these two machines becomes larger; while the
differences of the other three codes remain unchanged. Obtaining the variation in
relative performance is important for benchmarking and other performance
comparisons. For instance, the scalability analysis shows that the relative performance
of HEAT and HYDRO are more likely to vary with problem size than the other three
codes. A more detailed evaluation, the level four evaluation, has found the causes of the
scalability difference over the codes. In addition to a larger L2 cache capacity, the four
outstanding cache misses and the faster main memory access supported by Origin2000
have played an important role in performance improvement. This is especially true for
HYDRO and SWEEP.

5.7 **Hybrid Model Analysis**

We now apply the hybrid method to draw conclusions regarding our codes. We should
note that some of the statistical steps involved can be performed by simple inspection at
times. For simple cases this can be effective, but generally simple inspection will not
allow quantification of the statistical variance among observations. For this reason, we
utilize statistical methods in our results. Inspection should certainly be used whenever
the confidence of conclusions is high. Here we provide the general conclusions obtained
via these methods, such as whether or not a hypothesis is rejected. The observations
used in our experiments include various measurements for the codes mentioned at
varying problem sizes. All codes were measured on both machines using the same
compiled executable to avoid differences and with the following problem size
constraints: HEAT [50,100], HYDRO [50,300], SWEEP [50,200], and HYDROT
[50,300].

5.7.1 Level 1 Results

For the first hypothesis, "overall effect does not exist," we use level one of the original
statistical model. A straightforward two-factor factorial experiment shows that in fact
the hypothesis is rejected. This indicates further study is warranted and so, we continue
with the next 3 hypotheses. Using cpi\textsubscript{pipeline} as the dependent variable, the two-factor
factorial experiment is performed over all codes and machines to determine whether or
not code effect exists. Since identical executables are used over the two machines, no
variations are observed for cpi\textsubscript{pipeline} values over the measured codes. This is expected
as the case study was prepared to focus on memory hierarchy differences. Thus the
hypothesis holds, and no further study of cpi\textsubscript{pipeline} is warranted for these code-machine
combinations.

Next, we wish to test the hypothesis "machine effect does not exist". We
perform the two-factor factorial experiment using cpi\textsubscript{memory}. The results show variations
for the performance of cpi\textsubscript{memory} across the two machines. This will require further
analysis in level two of the hybrid model. Not rejecting the hypothesis would have
indicated that our codes perform similarly across machines. The third hypothesis asks whether "machine-code interaction exists". In fact, performing the two-factor factorial experiment, shows that machine-code interaction is present since we reject the hypothesis. This will have to be addressed in level two of the hybrid model as well.

5.7.2 Level 2 Results

Now that we have addressed each of the hypothesis warranted by rejection of the "overall effect" hypothesis, we must further analyze anomalies uncovered (i.e. each rejected hypothesis). We have identified code effect existence in level 1. It is necessary to analyze the $m_0$ term of Equations 4.4 and 4.5. Statistical results and general inspection show strong variations with problem size in HYDRO on the Origin 2000. Less fluctuations, although significant occur for the same code on the PowerChallenge. This indicates that unpredictable variations are present in the memory performance for HYDRO. As problem size scales, the $m_0$ term fluctuates indicating memory accesses do not achieve a steady state to allow performance prediction for larger problem sizes. Performing the somewhat costly linear fitting required by the empirical model supports the conclusions as shown in Figures 5.15 and 5.16. In these figures, problem size represents the y-axis and calculated $m_0$ values have been plotted. The scalability of HYDRO is in question since the rate at which latency overlap contributes to performance fluctuates.

On the other hand HEAT, HYDROT, and SWEEP show indications of predictability on the PowerChallenge. Statistical analysis of $m_0$ for problem sizes achieving some indication of steady state (greater than 50 for these codes - necessary to compensate for cold misses, counter accuracy, etc.) reveals little variance in $m_0$. For
problem sizes $[50,100]$, $[75,300]$, and $[50,200]$ respectively, $m_0$ is close to constant indicating the percentage of contribution to overlapped performance is steady. This is indicative of a code that both scales well and is somewhat predictable in nature over these machines. For these same codes on the Origin 2000, larger problem sizes are necessary to achieve little variance in $m_0$. Respectively, this occurs at sizes of $[75,100]$, $[100,300]$, and $[100,200]$. The shift this time is due to the cache size difference on the Origin 2000. It takes larger problem sizes to achieve the steady state of memory behavior with respect to the latency tolerating features previously mentioned. For both machines, these three codes exhibit predictable behavior and generally good scalability.

For two codes, HEAT and HYDROT, the fluctuations in the differences between $m_0$ values are minimal. This can be confirmed visually by simply subtracting

![m0 Values on O2K (m0)](image)

**Figure 5.15 m0 values calculated on the Origin 2000**
the \( m_0 \) values between the Origin and PowerChallenge. Such results indicate scaling between machines for these two codes over these two machines is somewhat predictable as well. HYDRO and SWEEP show larger amounts of variance for differences in \( m_0 \) values conversely. The scalability across the two machines for these codes should be analyzed further.

\[ \text{m0 Values on PC (m0')} \]

![Graph showing \( m_0 \) values](image)

**Figure 5.16** \( m_0 \) values calculated on the PowerChallenge

Finally, we must address the rejected hypothesis of machine-code interaction. Identifying this characteristic is suitable for analysis by level 2 of the original statistical method since it is not clear whether the memory architecture influence is the sole contributor to such performance variance. The statistical method refined for individual code performance, shows that the variance is caused by performance variations in 2
codes. Further investigation reveals that these two codes are statistically the same, allowing us to discount this rejected hypothesis.

We have shown that the hybrid approach provides a useful analysis technique for performance evaluation of scientific codes. The technique provides insight previously not available to the stand-alone statistical method and the empirical memory model. Results indicate that 3 of the 4 codes measured show promising signs of scaled predictability. We further show that scaled performance of latency overlap is good for these same three codes.

5.8 Summary

In this chapter we have provided a large amount of information that is gathered as a result of the robust methods presented in Chapter 4. We discussed the implementation of the PTERA tool for automated cross-platform performance measurements using performance monitors. We showed the usefulness of the empirical memory model and analytical instruction-level model by analyzing several ASCI scientific applications. Particularly, we found utilization of the latency hiding techniques of the PowerChallenge and Origin 2000 is generally good, but varies from code to code. The Intel-based Xeon does not provide the same level of performance gain from its latency hiding techniques for the codes measured. At the instruction level, memory instructions tend to be the overall bottleneck architecturally speaking except for codes that generally have good memory access (and high overlap, namely HYDRO and HYDRO-T). Based on our previous workload characterizations and these results, an architecture with a dynamically mutable functional unit may be able to provide better bandwidth to these codes if switching penalty and frequency trade-offs are minimal. In fact, as shown in
this chapter, viable architectures of this type can gain as much as 14% improvement over normal implementations.

The statistical method allowed us to further analyze the memory hierarchy differences of the Origin 2000 and PowerChallenge. We observed that some codes with diverse applications perform similarly in statistic performance across architectures. Further analysis allowed us to isolate the performance differences between architecture-code combinations. We showed that the scaled memory performance, despite the improvements available in the newer Origin 2000 architecture, is sometimes worse for certain codes counter-intuitively on these two architectures. At times, no performance was gained for the scaled codes. Still at other times, scaled performance gain actually decreases. With the isolation of cache miss performance, we were able to determine what the primary contributors to performance differences were. The hybrid method shows the ability to combine a statistical method with an empirical method to provide streamlined performance analysis techniques.
Chapter 6
Conclusions

6.1 Overall Summary

We have presented a large amount of material in this document. We began with a historical discussion of architecture and modeling techniques contrasting our methods. Our techniques are novel in their ability to analyze the ILP of codes and architectures using performance monitors. We discussed the testbeds consisting of primarily MIPS R10000 architectures and the Intel-based Xeon architecture. ASCI and SPEC codes were used consistently as proof of concept for the presented methods. The empirical memory model was presented to provide a method for quantifying the overlapped memory performance of superscalar architectures. The instruction-level model provides bottleneck analysis at the pipeline level of a microprocessor. The statistical model isolated performance variations to identify the primary architectural enhancements that the measured codes take advantage of. The hybrid model showed that we could incorporate the properties of the empirical memory model to statistically analyze the projected scalability and predictive nature of the code measured.

We presented the PTERA performance tool for added automation of result gathering. The instruction-level model was extended and verified using the SynBAD portion of the PTERA tool to develop synthetically built assemble directives for validation and experimentation. Data flow dependences were defined and quantified with the help of PTERA and the instruction-level model.

Results from the instruction-level model and the clustering characterization of the SPEC codes led to the design and simulator implementation of variants of a mutable
functional unit. **Superior performance** was achieved for the RS-MFU scheme which added the complexity of another reservation station while decreasing the number of entries in other reservation stations to better utilize die area. Performance gains between 8% and 14% were achieved for integer intensive applications and floating point intensive applications did not see a decrease in performance despite mutation penalties.

### 6.2 Scientific Contributions

Our thesis was to provide new methods for performance analysis utilizing the power of performance monitors. We have shown viable empirical, analytical, and statistical techniques can be developed for use by architectural designers and code developers. The PTERA tool can automate the process for data collection while providing extended abilities useful to performance tuning and architecture analysis. We contribute new techniques for performance analysis that are both useful and easy-to-use. Particularly, the conclusions both separately and together can provide methods for analyzing sets of codes or benchmarks generally speaking. It is our hope the methods will be used to analyze certain architectural differences and to group like codes by performance. We also believe, as shown here, that general conclusions for single codes can allow both analysis and prediction of performance across machines.

Our contributions are varied. We not only present the methods and their theoretical and practical validation, but we show the application of such methods in practice to analyze scientific codes. We provide a step-by-step discussion of the background material necessary to understand our methods along with the workload characterization techniques and results developed to provide a basis for our models.
These novel techniques allow us to develop unique models for performance analysis based on the general characteristics of our codes. Our techniques are successful in providing further insight into the measured codes. An added benefit of our presented methods is the implication that certain architectural techniques may affect performance dramatically. This is affirmed by our workload characterization and simulator implementation. Our techniques are generally applicable to other architectures since we utilize the common problem set for all measurements. The user must simply understand the underlying architecture in order to interpret results.

6.3 Future Work

We have discussed the novel nature of our work. These are preliminary techniques attempting to utilize the strengths inherent to performance monitors. Obvious future work involves the extension of many of the models discussed. Simple extensions such as eliminating certain constraints in the models themselves must be tackled first. The instruction-level model in particular has need for improvements of this type. While extensions of the empirical model have been achieved, work can be done to minimize the necessarily expensive curve fitting process of the current method.

The statistical and empirical methods should be extended as well to include elements of multi-processing performance. This is a complicated endeavor since it is difficult to define instruction-level parallelism in a multi-processor environment. Further characterization of codes for single processors must be studied as well, but these methods must be platform independent.

The PTERA tool implementation can be extended as well. It is our desire to complete implementation for a CISC architecture and another RISC architecture,
namely the Compaq EV6. The EV6 provides sampling at the pipeline level, giving further opportunities for modeling and validation that need study. The SynBAD model should be expanded in functionality to model loops more proficiently than the current version. Such control would provide a tool of great use in library and compiler development allowing direct measurement of architectural performance. This allows direct confirmation of advertised performance while providing the detailed measurements necessary to analyze a particular memory hierarchy implementation.

Such techniques have implications in compiler development as well. It is our intention to use the PTERA tool (and SynBAD) to begin detailed studies of optimization techniques in an effort to quantify performance variations between optimization methods. Using the SynBAD functionality along with our overall technique of model development, we will attempt to model the interaction between performance optimization techniques, generally speaking, for certain scientific codes.

Lastly, and certainly not least, we must continue studies of the mutable functional unit performance and its possible impact on DSP and FPGA architectures. Instruction-stream prediction, implementation in high-level synthesis, impact of future architectural trends, and implementation of the by-pass network are areas of future focus for this architecture.
Appendix

Performance Monitor Survey

This appendix contains the results of a feasibility study. The study's goal was to ascertain which current commodity processors would supply the appropriate performance counters when compared to the derived cross-platform subset discussed in this text. The left column contains the proposed subset, the right gives the appropriate counter for the processor under examination. Some results require minor formulas as shown. Results are based on published details of architecture and discussions with vendor experts when possible.

<table>
<thead>
<tr>
<th>CPU</th>
<th>MIPS R10K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>CYCLES</td>
</tr>
<tr>
<td>Instructions</td>
<td>GRAD_INSTR</td>
</tr>
<tr>
<td>Loads</td>
<td>ISSUED_LOADS</td>
</tr>
<tr>
<td>Stores</td>
<td>GRAD_STORES</td>
</tr>
<tr>
<td>L1 Misses</td>
<td>PRIMARY_DCACHE_MISSES</td>
</tr>
<tr>
<td>L2 Misses</td>
<td>SECONDARY_DCACHE_MISSES</td>
</tr>
<tr>
<td>Fl pt instructions</td>
<td>GRAD_FL_PT_OPS</td>
</tr>
<tr>
<td>Branches</td>
<td>DECODED_BRANCHES</td>
</tr>
<tr>
<td>Branch misses</td>
<td>MISPREDICTED_BRANCHES</td>
</tr>
<tr>
<td>TLB misses</td>
<td>TLB_MISSES</td>
</tr>
<tr>
<td>Icache misses</td>
<td>PRIMARY_ICACHE_MISSES</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel PPRO/PENTIUM II</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------</td>
</tr>
<tr>
<td>Cycles</td>
<td>CPU_CLK_UNHALTED</td>
</tr>
<tr>
<td>Instructions</td>
<td>INST RETIRED</td>
</tr>
<tr>
<td>Loads</td>
<td>DATA_MEM_REFS</td>
</tr>
<tr>
<td>Stores</td>
<td>DATA_MEM_REFS</td>
</tr>
<tr>
<td>L1 Misses</td>
<td>DCU_LINES_IN</td>
</tr>
<tr>
<td>L2 Misses</td>
<td>L2_LINES_IN</td>
</tr>
<tr>
<td>F1 pt instructions</td>
<td>FLOPS</td>
</tr>
<tr>
<td>Branches</td>
<td>BR_INST_DECODED</td>
</tr>
<tr>
<td>Branch misses</td>
<td>BR_MISS_PRED_RETIRED</td>
</tr>
<tr>
<td>TLB misses</td>
<td>ITLB_MISS</td>
</tr>
<tr>
<td>Icache misses</td>
<td>IFU_IFETCH_MISS</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Compaq Alpha EV4-6</th>
</tr>
</thead>
</table>
| Cycles              | CYCLES"
| Instructions        | ISSUES"
| Loads               | LOAD_INSTRUCTIONS"
| Stores              | STORE_INSTRUCTIONS"
| L1 Misses           | DCACHE_MISSES"
| L2 Misses           | SCACHE_MISS"
| F1 pt instructions  | FP_INSTRUCTIONS"
| Branches            | BRANCHES"
| Branch misses       | BRANCH_MISPREDICTS"
| TLB misses          | ITB_MISS"
| Icache misses       | ICACHE_MISSES"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>CPU</th>
<th>SUN UltraSPARC Ili</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>CYCLE_CNT</td>
</tr>
<tr>
<td>Instructions</td>
<td>INSTR_CNT</td>
</tr>
<tr>
<td>Loads</td>
<td>DC_RD</td>
</tr>
<tr>
<td>Stores</td>
<td>DC_WR</td>
</tr>
<tr>
<td>L1 Misses</td>
<td>EC_REF</td>
</tr>
<tr>
<td>L2 Misses</td>
<td>EC_REF - EC_HIT</td>
</tr>
<tr>
<td>Fl pt instructions</td>
<td></td>
</tr>
<tr>
<td>Branches</td>
<td></td>
</tr>
<tr>
<td>Branch misses</td>
<td></td>
</tr>
<tr>
<td>TLB misses</td>
<td></td>
</tr>
<tr>
<td>Icache misses</td>
<td>IC_REF - IC_HIT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CPU</th>
<th>Cyrix 6x86MX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
</tr>
<tr>
<td>Instructions</td>
<td>INSTRUCTIONS DECODED</td>
</tr>
<tr>
<td>Loads</td>
<td>DATA_READS</td>
</tr>
<tr>
<td>Stores</td>
<td>DATA_WRITES</td>
</tr>
<tr>
<td>L1 Misses</td>
<td>DATA_READ OR WRITE MISSES</td>
</tr>
<tr>
<td>L2 Misses</td>
<td></td>
</tr>
<tr>
<td>Fl pt instructions</td>
<td>FLOATING POINT OPERATIONS</td>
</tr>
<tr>
<td>Branches</td>
<td>BRANCHES</td>
</tr>
<tr>
<td>Branch misses</td>
<td>BRANCHES - BTB HITS</td>
</tr>
<tr>
<td>TLB misses</td>
<td>L1 TLB_MISS</td>
</tr>
<tr>
<td>Icache misses</td>
<td>CODE CACHE MISSES</td>
</tr>
<tr>
<td>CPU</td>
<td>HP PA-RISC (PA 8000 V Class)</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Cycles</td>
<td>CPU TIME</td>
</tr>
<tr>
<td>Instructions</td>
<td>INSTRUCTIONS</td>
</tr>
<tr>
<td>Loads</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Stores</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>L1 Misses</td>
<td>CACHE MISSES?</td>
</tr>
<tr>
<td>L2 Misses</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Fl pt instructions</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Branches</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Branch misses</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>TLB misses</td>
<td>TLB MISSES</td>
</tr>
<tr>
<td>Icache misses</td>
<td>-----------------------------</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Vita

Kirk W. Cameron was born in Toms River, New Jersey, in 1970. He moved with his family to Port Charlotte, Florida, during high school. He attended the University of Florida where he earned a bachelor of science degree in mathematics in 1994. While in Gainesville, he met his future wife Melissa, and the two moved to Baton Rouge, Louisiana, to pursue graduate studies. He is currently funded as a graduate research assistant at Los Alamos National Laboratory in New Mexico. Professionally, Kirk is the author and co-author of over 10 refereed papers and technical reports at this time. Publications included portions of book chapters in Workload Characterization: Methodologies and Case Studies published by Computer Society Press, and Lecture Notes in Computer Science 1615 published by Springer-Verlag. Kirk has presented tutorials at two premier computer architecture conferences, the 6th International Symposium on High Performance Computer Architecture and the 26th International Symposium on Computer Architecture. He has also published in the refereed proceedings of the Society for Industrial and Applied Mathematics on Parallel Processing, the 2nd International Symposium for High Performance Computing, the 1st Workshop on Software Performance, the 13th International Parallel Processing Symposium and Supercomputing 1999 to name a few. He has been awarded five competitive conference grants to attend symposiums throughout the United States, Europe, and Japan. He was also a key collaborator and co-author of a $300,000 three-year funded grant through the Los Alamos National Laboratory. Kirk has accepted a position as an assistant professor at the Florida Institute of Technology beginning in August of 2000.
DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate:  Kirk W. Cameron

Major Field:  Computer Science

Title of Dissertation:  Empirical and Statistical Application Modeling Using On-Chip Performance Monitors

Approved:

[Signatures]

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

[Signatures]

Date of Examination:

June 29, 2000