Copyright

by

Robert Henry Bell, Jr.

2005

The Dissertation Committee for Robert Henry Bell, Jr. Certifies that this is the approved version of the following dissertation:

Automatic Workload Synthesis for Early Design Studies and Performance Model Validation

Committee:

Lizy K. John, Supervisor

Earl E. Swartzlander, Jr.

Douglas C. Burger

Adnan Aziz

Lieven Eeckhout

Automatic Workload Synthesis for Early Design Studies and Performance Model Validation

by

Robert Henry Bell, Jr., B.A.; M.S.E.E.

Dissertation

Presented to the Faculty of the Graduate School of
The University of Texas at Austin
in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

The University of Texas at Austin
December 2005

Dedication

To my wife, Susan Carol Honn

And my parents, Robert H. Bell and Joyce W. Bell

Acknowledgements

This work would not have been possible without the support of many people.

I would like to thank my advisor, Dr. Lizy Kurian John, for her advice, support, wisdom, and guidance. Dr. John had a profound influence on both the overall direction of this research and the specific content of this dissertation. Her unfailing passion for the subject matter and sound advice in the face of sometimes difficult issues always pointed in the correct direction. Her research in the complex fields of computer architecture and computer performance analysis continues to inspire researchers and developers in both academia and industry.

I would like to thank my graduate committee for their advice and friendship over the years. Many thanks are extended to Earl Swartzlander, who co-authored my first paper as a graduate student at the University of Texas at Austin; Adnan Aziz, whose class on logic synthesis inspired many analogous thoughts on the automatic synthesis of workloads; Lieven Eeckhout, for our collaboration on statistical simulation that launched this work, and his many friendly and helpful comments over the years. Special thanks go to Doug Burger, whose charm, wisdom, intellect and complete mastery of computer design are truly an inspiration.

I would also like to thank the many characters that I interacted with in the Laboratory on Computer Architecture at the University of Texas at Austin, especially Dr.

Tao Li, Dr. Madhavi Valluri, Dr. Ravi Bhargava, and Dr. Juan Rubio. Their research ideas, their knowledge in many diverse disciplines, willingness to listen and comment in many talks and discussions, and friendly help were much appreciated.

I would like to thank the faculty, staff, and administration of the University of Texas at Austin for providing the venue and resources for a world-class graduate education in computer architecture and performance analysis. Many thanks go to IBM, the IBM Systems and Technology Division, and the Advanced Learning Assistance Program at IBM for providing funding for this work. Thanks to Doug Balser, Tom Albers, Sam Thomas of IBM, and a special thanks to Jeff Stuecheli. I would like to thank Dr. Ann Marie Maynard of the IBM Austin Center for Advanced Study for her energy and enthusiasm in enhancing the interaction between IBM and academia. Thanks to Dr. James H. Aylor and Dr. Jim Jokl of the University of Virginia, who guided me through my M.S.E.E. and inspired continuing studies in engineering.

Many thanks go to my family. I would like to thank my father, Robert H. Bell, who inspired my work in the sciences and engineering, and whose ever-cheerful demeanor, friendly and helpful company, intellectual curiosity, and unfailing support are great comforts. Thanks also to my mother, Joyce W. Bell, whose support, kindness, extensive scientific education, and quest for knowledge are most important to me.

Finally, to my dear wife and friend, Susan C. Honn, a special thanks for enduring the long hours, late nights, and lost weekends that inevitably result from such an endeavor; and for her enthusiasm, loving support, and patience over the years - and in three states - I will be eternally thankful.

Robert H. Bell, Jr.

The University of Texas at Austin

December 2005

Automatic Workload Synthesis for Early Design Studies and Performance Model Validation

Publication No._____________

Robert Henry Bell, Jr., Ph. D.
The University of Texas at Austin, 2005

Supervisor: Lizy Kurian John

Computer designers rely on simulation systems to assess the performance of their designs before the design is transferred to silicon and manufactured. Simulators are used in early design studies to obtain projections of performance and power over a large space of potential designs. Modern simulation systems can be four orders of magnitude slower than native hardware execution. At the same time, the numbers of applications and their dynamic instruction counts have expanded dramatically. In addition, simulation systems need to be validated against cycle-accurate models to ensure accurate performance projections. In prior work, long running applications are used for early design studies while hand-coded microbenchmarks are used for performance model validation.

One proposed solution for early design studies is statistical simulation, in which statistics from the workload characterization of an executing application are used to create a synthetic instruction trace that is executed on a fast performance simulator. In prior work, workload statistics are collected as average behaviors based on instruction types. In the present research, statistics are collected at the granularity of the basic block. This improves the simulation accuracy of individual instructions.

The basic block statistics form a statistical flow graph that provides a reduced representation of the application. The synthetic trace generated from a traversal of the flow graph is combined with memory access models, branching models and novel program synthesis techniques to automatically create executable code that is useful for performance model validation. Runtimes for the synthetic versions of the SPEC CPU, STREAM, TPC-C and Java applications are orders of magnitude faster than the runtimes of the original applications with performance and power dissipation correlating to within 2.4% and 6.4%, respectively, on average.

The synthetic codes are portable to a variety of platforms, permitting validations between diverse models and hardware. Synthetic workload characteristics can easily be modified to model different or future workloads. The use of statistics abstracts proprietary code, encouraging code sharing between industry and academia. The significantly reduced execution times consolidate the traditionally disparate workloads used for early design studies and model validation.

Table of Contents

Table of Contents ix

List of Tables xiii

List of Figures xv

Chapter 1: Introduction 1
- 1.1 Early Design Studies and Application Simulation 3
- 1.2 Performance and Power Model Validation 5
- 1.3 Simulation Strategies 5
- 1.4 The Problems and Proposed Solutions 7
- 1.5 Thesis Statement 9
- 1.6 Contributions 9
- 1.7 Organization 12

Chapter 2: Workload Modeling and Statistical Simulation 13
- 2.1 Performance Simulation Strategies and Statistical Simulation 13
- 2.2 Overview of Statistical Simulation in HLS 18
- 2.3 Simulation Results 21
  - 2.3.1 Experimental Setup and Benchmarks 21
  - 2.3.2 The HLS Graph Structure 22
  - 2.3.3 The HLS Processor Model 24
  - 2.3.4 Issues in the Experimental Setup of HLS 25
  - 2.3.5 Challenges Modeling the STREAM Loops 28
- 2.4 Improving Processor and Workload Modeling in HLS 29
  - 2.4.1 Improving the Processor Model 29
  - 2.4.2 Improvements to Workload Modeling 31
    - 2.4.2.1 Basic Block Modeling Granularity 31
    - 2.4.2.2 Basic Block Maps 34
    - 2.4.2.3 Basic Block Maps for Strong Phases 38
- 2.5 Implementation Costs 39

2.6 Summary 41

Chapter 3: Automatic Workload Synthesis...43

3.1 Introduction to Performance Model Validation 43

3.2 Synthesis of Representative Workloads...46

3.3 Synthesis Approach 50

3.3.1 Workload Characterization 51

3.3.2 Graph Analysis...52

3.3.2.1 Instruction Miss Rate and I-cache Model 54

3.3.2.2 Instruction Dependences and Instruction Compatibility.55

3.3.2.3 Loop Counters and Program Termination 57

3.3.2.4 Memory Access Model 57

3.3.2.5 Branch Predictability Model 60

3.3.3 Register Assignment 61

3.3.4 Code Generation 62

3.4 Evaluation of Synthetic Testcase Performance.......................................64

3.4.1 Methodology 64

3.4.2 Evaluation of Synthetic Workload Characteristics 65

3.4.3 Evaluation of Design Changes..69

3.4.4 TPC-C Study 75

3.5 Drawbacks and Discussion 75

3.6 Early Synthesis and Related Workload Synthesis Research...................77

3.7 Summary 80

Chapter 4: Quantifying the Errors in Workload Characteristics Due to the Workload Synthesis Process 82

4.1 Introduction to Errors in Synthetic Workloads.......................................82

4.2 Sources of Error in Workload Synthesis...84

4.2.1 Sources of Error in Workload Characterization..........................85

4.3.2 Sources of Error in Graph Analysis 86

4.3.2.1 Instruction Miss Rate and I-cache Model 86

4.3.2.2 Instruction Dependences 87

4.3.2.3 Loop Counters and Program Termination 88

4.3.2.4 Memory Access Model 88

4.3.2.5 Branching Model 89

4.3.3 Sources of Error in Register Assignment...................................90

4.2.4 Sources of Error in Code Generation...91

4.3 The Flexibility of Statistical Simulation 92

4.4 Simulation Results 96

4.4.1 Experimental Setup and Benchmarks 96

4.4.2 Sensitivities to Changes in Workload Characteristics in Statistical Simulation 97

4.4.3 Sensitivities to Changes in Workload Characteristics from Testcase Synthesis 102

4.5 Summary..105

Chapter 5: Efficient Power Analysis using the Synthetic Workloads 107

5.1 Introduction to Power Dissipation Studies 107

5.2 Synthetic Testcases and Power Dissipation..109

5.3 Power Simulation Results...111

5.3.1 Experimental Setup and Benchmarks 112

5.3.2 Base Power Dissipation Results..112

5.3.3 Analysis of Design Changes 116

5.4 Summary..121

Chapter 6: Performance Model Validation Case Study for the IBM POWER5 Chip 122

6.1 Introduction to the POWER5 Chip 122

6.2 IBM PowerPC Synthesis and Model Validation 124

6.2.1 The POWER5 M1 Performance Model...................................124

6.2.2 PowerPC Performance Model Validation................................125

6.2.2.1 Validation using RTL Simulation................................125

6.2.2.2 Validation using a Hardware Emulator.........................128

6.3 Synthesis for PowerPC 128

6.3.1 Workload Characterization 128
6.3.2 Graph Analysis..129
6.3.2.1 Instruction Miss Rate and I-cache Model 130
6.3.2.2 Instruction Dependences and Compatibility.................130
6.3.2.3 Loop Counters and Program Termination 132
6.3.2.4 Memory Access Model 132
6.3.2.5 Branch Predictability Model 137
6.3.3 Register Assignment 139
6.3.4 Code Generation 139
6.4 POWER5 Synthesis Results 140
6.4.1 Experimental Setup...141
6.4.2 Synthesis Results 141
6.4.3 Design Change Case Study: Rename Registers and Data Prefetching 145
6.5 Performance Model Validation Results 146
6.5.1 RTL Validation 146
6.5.2 Hardware Emulation 149
6.6 Summary 150

Chapter 7: Conclusions and Future Work..151
7.1 Conclusions...152
7.2 Future Work 156

Bibliography 159

Vita 169

List of Tables

Table 1.1: Examples of Modern Benchmarks and Benchmark Suites....................4

Table 2.1: Machine Configuration for Pisa and Alpha Simulations......................21

Table 2.2: CPI Regression Analysis for the SPEC 95 Benchmark Suites 27

Table 2.3: Single-Precision STREAM Loops...27

Table 2.4: Benchmark Information for Implementation Cost Analysis 40

Table 2.5: Implementation Costs 41

Table 3.1: Glossary of Graph Analysis Terms and Reference to Figure 3.3 54

Table 3.2: Dependence Compatibilities for Alpha and Pisa Synthetic Testcases..55

Table 3.3: Synthetic Testcase Properties 56

Table 3.4: L1 and L2 Hit Rates as a Function of Stride (in 4B increments) 58

Table 3.5: Percent Error by Metric, Synthetics versus Applications.....................66

Table 3.6: Average Synthetic IPC Error and Relative Error by Benchmark Suite.... 72

Table 3.7: Percent IPC Error and Relative Error by Design Change.....................73

Table 4.1: Major Sources of Error in Synthetic Workloads 84

Table 4.2: Prominent Workload Characteristics and Pearson Correlation Coefficients for Factor Values versus Simulation Results 93

Table 4.3: Workload Characteristic Margins (at 3% from Base) 101

Table 4.4: Example Workload Characteristic Synthesis Changes (gcc).............102

Table 4.5: Average Percent Error Differences for Dispatch Window/LSQ and Width Studies...105

Table 5.1: Average Power Prediction Error (%), Synthetics vs. Benchmarks 113

Table 5.2: Correlation Coefficients of Power Dissipation versus IPC 114

Table 5.3: Average Absolute and Relative IPC and Power Dissipation Error 116
Table 5.4: Correlation Coefficients of Power vs. IPC for Design Changes and Quality of Assessing Power Dissipation Changes (Class).................119
Table 6.1: Graph Analysis Thresholds and Typical Tolerance...........................130
Table 6.2: PowerPC Dependence Compatibility Chart 131
Table 6.3: L1 and L2 Hit Rate versus Stride (PowerPC)...................................133
Table 6.4: L1 and L2 Hit Rates for Reset Instruction Number (Congruence Class Walks)...135
Table 6.5: Synthetic Testcase Properties for the POWER5 Chip.......................136
Table 6.6: Synthetic Testcase Memory Access and Branching Factors for the POWER5 Chip 138
Table 6.7: Default Simulation Configuration for the POWER5 Chip................141

List of Figures

Figure 1.1: IPC Prediction Error in Statistical Simulation by Benchmark Suite.....6

Figure 2.1: Overview of Statistical Simulation: Profiling, Synthetic Trace Generation, and Trace-Driven Simulation..18

Figure 2.2: Effect of Graph Connectivity in HLS on SPEC INT 95 Benchmarks..22

Figure 2.3: Effect of Changes in Backward Jump Fraction for *gcc*.......................23

Figure 2.4: Effect of Changes in Backward or Forward Jump Distance for *gcc*...23

Figure 2.5: Error in HLS by Benchmark as Experimental Setup Changes............25

Figure 2.6: Error in HLS by Benchmark Suite as Experimental Setup Changes ..26

Figure 2.7: Disassembled SAXPY Loop in the *Pisa* Language 28

Figure 2.8: Improved Error in HLS by Benchmark as Modeling Changes 30

Figure 2.9: Improved Error in HLS by Benchmark Suite as Modeling Changes..30

Figure 2.10: Improved Error in HLS by Benchmark as Modeling Changes with Basic Block Maps 36

Figure 2.11: Improved Error in HLS by Benchmark Suite as Modeling Changes with Basic Block Maps 36

Figure 2.12: IPC for HLS++ by Benchmark versus SimpleScalar for the SPEC 2000 in the Alpha Language 37

Figure 2.13: Versions of HLS Executing Two-Phase Benchmarks.......................38

Figure 2.14: HLS versus HLS++ for the SPEC 95 Benchmarks...........................38

Figure 2.15: HLS versus HLS++ by Benchmark for the SPEC 95........................39

Figure 3.1: Proprietary Code Sharing using Workload Synthesis.........................49

Figure 3.2: Overview of Workload Synthesis Methodology................................50

Figure 3.3: Step-by-Step Illustration of Synthesis 51

Figure 3.4: Flow Diagram of Graph Analysis Phase of Synthesis 53

Figure 3.5: Actual vs. Synthetic IPC 65

Figure 3.6: Instruction Frequencies 65

Figure 3.7: Basic Block Sizes 66

Figure 3.8: I-cache Miss Rates 66

Figure 3.9: Branch Predictability 67

Figure 3.10: L1 D-cache Miss Rates 67

Figure 3.11: L2 Cache Miss Rates 68

Figure 3.12: Average Dependence Distances per Instruction Type and Operand . 68

Figure 3.13: Dispatch Window Occupancies per Instruction Type 69

Figure 3.14: Dispatch Window Size 32 69

Figure 3.15: Dispatch Window Size 64 70

Figure 3.16: IPC Error per Dispatch Window 70

Figure 3.17: Delta IPC as Dispatch Window Increases from 16 to 32 70

Figure 3.18: Delta IPC as Dispatch Window Increases from 16 to 64 70

Figure 3.19: Delta IPC as L1 Data Latency Increases from 1 to 8 71

Figure 3.20: Delta IPC as Issue Width Increases from 1 to 4 71

Figure 4.1: L1 D-cache Hit Rate Factor for gzip 95

Figure 4.2: L1 I-cache Hit Rate Factor 97

Figure 4.3: L1 I-cache Hit Rate Factor (expanded) 97

Figure 4.4: L1 D-cache Hit Rate Factor 98

Figure 4.5: L1 D-cache Hit Rate Factor (expanded) 98

Figure 4.6: L2 Cache Hit Rate Factor 99

Figure 4.7: Branch Predictability Factor 100

Figure 4.8: Branch Predictability Factor (expanded)..100

Figure 4.9: Basic Block Changes..101

Figure 4.10: Dependence Distance Factor...101

Figure 4.11: Changes Due to Synthesis in Statistical Simulation........................103

Figure 4.12: Design Changes (No Synthetic Parameters)...................................104

Figure 4.13: Design Changes (Synthetic Parameters)...104

Figure 5.1: Power Dissipation per Cycle...113

Figure 5.2: Power per Cycle vs. IPC for Synthetics..113

Figure 5.3: Power per Instruction vs. IPC for Synthetics....................................115

Figure 5.4: Power Dissipation per Instruction...115

Figure 6.1: RTL Validation Methodology using Synthetic Testcases.................126

Figure 6.2: Flow Diagram of Graph Analysis and Thresholds for PowerPC Synthesis..129

Figure 6.3: IPC for Synthetics Normalized to Benchmarks................................142

Figure 6.4: Average Instruction Frequencies...142

Figure 6.5: Average Basic Block Sizes..143

Figure 6.6: I-cache Miss Rates...143

Figure 6.7: Branch Predictability...143

Figure 6.8: L1 D-cache Miss Rate...143

Figure 6.9: Average Dependence Distances..144

Figure 6.10: Normalized Error per Instruction and Cumulative Error for 10K Instructions (gcc)...146

Figure 6.11: Fractions of Instructions with Errors (gcc).....................................148

Figure 6.12: Average Error per Class for All Instructions or Instructions with Errors..148

Figure 6.13: Numbers of Instructions with Errors by 25-Cycle Bucket (gcc).....148

Figure 6.14: Normalized IPC for M1 versus AWAN for Synthetic Testcases....150

Chapter 1: Introduction

For many years, simulation tools have been used to ease the work of computer processor design. Among other tasks, simulation tools assess the accuracy and performance of processor designs before they are manufactured. A processor simulator applies a program and input dataset to a processor model and simulates the operation of the processor model.

Processor simulators range from instruction-level (also called functional simulators) to register-transfer-level (RTL) simulators. Instruction-level simulators simulate the functionality of the input instructions of a program without regard to how each instruction is implemented in the hardware. The processor model may be very simple or non-existent, but the input program and dataset are correctly simulated. RTL simulators simulate a model of a processor that typically possesses enough detail to be manufactured. Microarchitectural-level simulators, also known as performance simulators, operate on performance models that contain more microarchitectural detail than those exercised by instruction-level simulators but less detail than those exercised by RTL simulators.

Performance simulators are used to assess the performance of a design with respect to runtime, or to an aggregate performance metric for a fixed binary, such as instructions per cycle (IPC) or its inverse, cycles per instruction (CPI). An execution-driven performance simulator may execute a complete program binary along with an input dataset on a performance model, while a trace-driven performance simulator may execute an address trace containing only instruction addresses and partial information for each instruction [51]. An address trace usually specifies the sequence of executed instructions in the order that they complete as a program executes dynamically in a machine.

The *workload* for a performance simulator is usually the program or collection of programs that the designer wishes to use to stress the performance model. For example, the workload of an execution-driven simulator is a program binary and input dataset. The workloads that are used to assess processor performance are generally known as *benchmarks*. A workload is *synthetic* if it is not a fully functional user program or application but has specific execution characteristics in common with actual programs. A synthetic workload may be hand-coded or automatically generated; it may be a program binary or a trace. The process of automatically creating a synthetic workload is referred to as workload or program *synthesis*.

This dissertation is mainly concerned with improving two related tasks that utilize performance simulators: 1) early design studies and 2) performance and power model validations. The first task is concerned with the early evaluation of performance and power on a performance model for many potential designs in a large design space. The workloads used for early design studies are usually longer-running binaries or traces. The second task is concerned with deciding how accurate the performance model is with respect to an RTL model or hardware in later design stages. Because of the slow execution speed of RTL models, the workloads are usually short hand-coded programs, known as microbenchmarks or testcases. The benchmarks of interest to the designer, whether synthetic or not, have much to do with the performance simulation methodology that is used to carry out these two tasks.

This chapter gives an overview of the problems inherent in carrying out early design studies and performance model validations. The statistical simulation approach has been proposed as a solution for early design studies, but it is hampered by inaccurate traces that are not portable to multiple platforms. The need for complementing synthetic traces with a workload synthesis capability is presented. One side effect of this solution is that the traditionally disparate workloads used for early design studies and model validation are consolidated into the same workload, enabling design studies throughout the design process with performance models validated using the simulation workloads.

1.1 EARLY DESIGN STUDIES AND APPLICATION SIMULATION

It has long been recognized that many potential design points in a large design space need to be examined throughout the design process in order to guide and balance the performance of the overall design as components are added to the design. Thousands of changes in the microarchitecture of the machine may need to be investigated over the course of development. Each evaluation requires the execution of a benchmark and input dataset in a performance simulator. Unfortunately, many performance simulators are four or more orders of magnitude slower than native hardware.

An additional problem concerns the number of benchmark and dataset pairs that must be simulated in these design studies. Machines are too complicated to rely on short testcases or a small set of testcases to assess a design change. Simple codes are *unrepresentative* of real application performance; that is, they do not exercise the machine in the same way as actual codes. Representative codes evoke the same sequences of machine states as the original application [43], resulting in similar workload characteristics such as instruction mix, cache miss rates, etc. A code may be partially representative of an application for some range of instructions or for particular workload characteristics.

The early synthetic benchmarks such as Whetstone [22] and Dhrystone [103] were developed to represent the instruction mix of real programs, but they were easily manipulated by unfair compiler optimizations and became unrepresentative over time.

With synthetic workloads unable to evolve sufficiently, researchers have relied on the runtimes of real applications to assess computer performance. Recently there has been an explosion of applications in use as benchmarks. A partial list is given in Table 1.1.

Table 1.1: Examples of Modern Benchmarks and Benchmark Suites

Application Class	**Example Benchmark Suites**
General, Scientific and Engineering	SPEC CPU 89/92/95/2000/2006, STREAM, Livermore Loops
Parallel Processing	Perfect Club, NAS loops, SPLASH, Hint
Transactions	TPC-A/B/C/D/W
Java	SPECjbb, SPECjvm
Servers	SFS/LADDIS, SPECweb, AIM, Server Bench, NetBench
Multimedia	MiBench, MediaBench, MediaStones
Graphic	SPEC GPC, WinBench
Personal Computing	WinStone, PCBench, SYSmarks
Miscellaneous	Network processing, embedded processors, mobile computing, telecommunications, bioinformatics…

Trends such as the consolidation of multi-media, telecommunications and computing technologies on-chip [110] and the emergence of new application classes, like network processing [111] and bioinformatics [112], and languages, like C++ and Java, have fueled the benchmark explosion. Ideally, a designer would assess the performance of design changes using all benchmarks that are similar to any applications an end-user might execute.

Unfortunately, the length of the benchmarks, in terms of the total dynamic instruction count, leads to long runtimes and prohibits the simulation of all benchmarks. For example, the benchmarks in the popular SPEC CPU 2000 suite [89] (also called SPEC 2000 in this work) have dynamic instruction counts greater than one billion instructions [37][108] and some have several hundreds of billions of instructions. Execution times even on the fastest simulators can amount to days for a single design choice on one benchmark [83][108][28][58].

1.2 PERFORMANCE AND POWER MODEL VALIDATION

Performance model validation seeks to verify that a performance model is accurate with respect to a cycle-accurate functional model or hardware [13][11]. Inaccurate performance models can lead to incorrect performance projections and design decisions in the later design studies. Validation of the simulation accuracy of a performance model is necessary at various points in the design process to minimize decision errors. To reduce performance simulator runtime, validation is usually not concerned with machine details that do not contribute significantly to performance such as manufacturing test structures, elements of the datapath, and specific circuit details.

The validation problem is limited by the speed of the machine model used for cycle-accurate verification that is being compared to the faster performance simulation. Just as for early design studies, it is difficult to decide which benchmarks should be used for model validation, and full simulation of applications on cycle-accurate models is even worse in terms of runtime than performance simulators.

The problem of validating power models in a microarchitectural simulator has been approached in the same way as performance model validation. Hand-coded tests or deductive physical analysis can be used for simple power dissipation validation tests [15], but validating applications again leads to long runtimes.

1.3 SIMULATION STRATEGIES

Researchers have responded to the long runtimes of modern benchmarks with various simulation strategies. Analytical models [68] and reduced input datasets [50] have given way to more accurate sampling techniques [83][108] that can reduce overall runtimes, but the executions still amount to tens of millions of instructions.

A technique called statistical simulation [71][28] can further reduce the number of executed instructions used to simulate a workload. In statistical simulation, workload characteristics are profiled during full benchmark or phase execution, and the resulting statistics are used to generate a synthetic trace which is then simulated on a performance model. The simulation typically converges to a result in less than one million instructions. This property would make synthetic traces very useful as the workloads for both design studies and model validation.

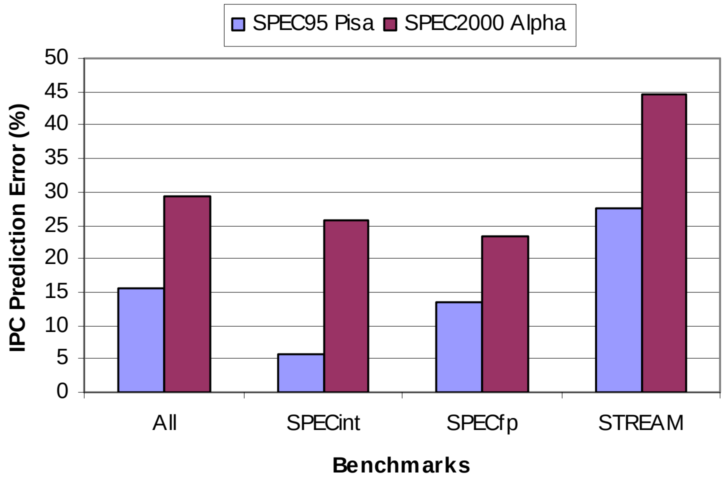


Figure 1.1: IPC Prediction Error in Statistical Simulation by Benchmark Suite

Unfortunately, statistical simulation systems (prior to the present work) can be inaccurate. Studies have shown that accurate design studies can be carried out when IPC prediction error is smaller than 10% [27]. Figure 1.1 shows the error in IPC versus cycle-accurate simulation for the HLS statistical simulation system [71] executing the SPEC CPU 95 (SPEC 95) Pisa (MIPS) binaries and the SPEC 2000 Alpha binaries. HLS performs well on the SPEC INT 95 suite, as expected from the results of Oskin *et al.* [71], but not as well on the SPEC FP 95 and Pisa STREAM suites, and it is remarkably poor on all of the Alpha binaries, showing an average error of 27.6%.

In addition to its inaccuracy, the synthetic traces from statistical simulation can not be executed on execution-driven simulators, RTL models, hardware emulators, or hardware itself, which would be useful for a variety of validation studies.

1.4 THE PROBLEMS AND PROPOSED SOLUTIONS

In summary, the growing number of applications and the long runtimes of those applications have caused concern among researchers in the simulation of computer designs for design studies and model validation. This concern has led to the use of sampling techniques to reduce runtimes. Statistical simulation further reduces runtimes and could be useful for consolidation of the workloads for these two tasks, but the synthetic traces are inaccurate on a variety of application classes, and, even if made accurate, they can not be executed on the diverse platforms necessary for performance and power model validation.

This dissertation focuses on the underlying problem: how to create representative workloads for accurate design studies that are also useful for performance and power model validations. In proposing a solution, several questions are asked: can the accuracy of the synthetic traces of statistical simulation be improved such that they are useful for rapid early design studies on a variety of workloads? Can the improved accuracy of the simulation technology be harnessed to provide rapid execution on a variety of platforms for performance and power model validations? Can the very different workloads used for design studies and model validation be consolidated into a single workload, enabling design studies with validated performance models?

Similarly, there are primarily three sub-problems that this dissertation addresses:

1) The inaccuracy of the synthetic traces in statistical simulation hinders their usefulness as tools for design studies.

2) The synthetic traces are not executable on a variety of simulation and hardware platforms, as would be useful for design studies and performance and power model validations.
3) The long-running real-world workloads used for design studies are not the same as the simpler microbenchmarks and testcases used for model validation. This opens a gap between validation using simpler but unrepresentative testcases and design studies using workloads which have not been validated on a performance simulator.

This dissertation provides a conceptual framework to solve these problems. It proposes specific techniques that improve the workload modeling in statistical simulation and allow for more accurate trace simulation of a variety of workloads. This makes possible the improved accuracy of rapid microarchitectural simulations for design studies. It also proposes the use of the same improved workload modeling technology as a basis for the synthesis of representative but flexible workloads that converge rapidly to a result. Workloads are synthesized in a high-level language so that they can be compiled onto the diverse platforms useful for performance and power model validations without forfeiting the low-level execution characteristics of the original application. This makes feasible the performance and power model validations of workloads that better represent real applications rather than microbenchmarks or other hand-coded workloads. The same synthetic workloads used for representative early design studies now execute in few enough instructions to be used for model validation. This consolidates the design study and model validation workloads into flexible workloads that are useful for a variety of performance simulation tasks.

1.5 THESIS STATEMENT

Simulation times of modern applications can be extremely long, decreasing their effectiveness for design studies of performance and power, performance model validations, and power model validations. Improved statistical modeling of workloads by modeling at the granularity of the basic block increases simulation accuracy while keeping runtimes short for rapid design studies. The improved models provide a foundation for the synthesis of workloads that achieve rapid and accurate performance and power simulations and model validations, consolidating the workloads used for design studies and model validation.

1.6 CONTRIBUTIONS

This dissertation makes several contributions to processor modeling and simulation in statistical simulation, characterization of the performance effects of individual changes in workload characteristics, a conceptual framework for workload synthesis, the automatic synthesis of representative workloads, the analysis of processor power dissipation using synthetic workloads, and performance and power model validation methodologies. The following paragraphs summarize these contributions:

1) The workload characterization in classical statistical simulation compiles statistics from executing applications at the granularity of the instruction, meaning that statistics are collected about a particular instruction class regardless of the context of the sequence of instructions that led up to its execution. This dissertation improves the accuracy of the workload modeling in statistical simulation by compiling statistics at the granularity of the basic block, that is, it takes into account the sequence of basic blocks encountered during dynamic execution. This is called a basic block map.

2) Synthetic workloads are useful for design studies, performance model validation, and power model validation. Classical synthetic workloads are written at a high language level to be representative of both the static high-level language features and dynamic low-level instruction frequencies of a workload. The high-level synthetics have advantages with respect to portability but they suffer from obsolescence, that is, they lose representativeness as languages and applications evolve. This dissertation proposes that workloads synthesized at a high language-level but retaining low execution-level characteristics using assembly inlining can closely represent the behavior of applications yet still converge rapidly to a result. It proposes an automatic workload synthesis process that permits synthetic workloads to be quickly recreated as the applications or languages change.

3) Because they are written at a high language-level, the synthetic workloads are portable across a variety of execution-driven and performance simulators and hardware platforms for the same ISA. They can also be ported across instruction set architectures by translating one-to-one the inline assembly instructions to the new instruction set, assuming similar instructions exist in both ISAs.

4) Since the synthesis process is based on statistical information, altering the characteristics of the synthetic workload is easy. Individual changes to program characteristics can be isolated and studied independently. For example, dependence distances for integer instructions can be modified changing the dependences of the other instruction types. In addition, changes to the workload characteristics that are anticipated in future workloads can be easily incorporated. Continuing with the previous example, if studies of the

workloads of interest show that dependence distances for integer instructions change by a certain factor per compiler generation, that factor could be applied to the integer dependences at synthesis-time and instantiated in the synthetic. Then studies using this future workload could be undertaken. In addition, the statistical nature of the synthetic workload abstracts the behavior of the application, effectively hiding its underlying function. The datasets, function boundaries and variable names are all removed. This encourages the sharing of proprietary codes for computer architecture research between industry and academia.

5) The accuracy of a synthetic workload depends on the accurate characterization of the workload prior to simulation. However, there exists the possibility that small changes in workload characteristics do not significantly impact performance in simulation. This has implications for any workload synthesis process. This dissertation quantifies the effects of the changes in workload characteristics on performance due to the proposed workload synthesis methodology.
6) The proposed workload synthesis method attempts to represent application performance as closely as possible. This dissertation demonstrates that the synthetic testcases created strictly for representative performance also provide representative dynamic power dissipation.
7) Performance and power model validation efforts have been limited to microbenchmarks and random testcases because of the infeasibility of using long-running applications for validation. Short snippets from the applications give no guarantee of covering the machine responses of the complete application. The synthetic workloads created using the proposed techniques

are executable on multiple platforms and can be used for rapid performance and power model validation. They can also be used for design space exploration, and a stair-step design exploration approach is suggested.

8) Use of the workload characterization of statistical simulation consolidates the workloads for both early design studies and model validation in the same synthetic trace. This closes the gap between the workloads used for early design studies and those used for validation.

1.7 ORGANIZATION

Chapter 2 presents the proposed workload modeling improvements to statistical simulation. The concept of modeling and simulation at the granularity of the basic block is introduced. The cost of the modeling improvements is quantified in terms of the amount of data collection necessary for the improvements.

Chapter 3 describes the use of the improved workload modeling technology as input to the proposed representative workload synthesis system. The automatic workload synthesis methodology is presented.

Chapter 4 investigates the sensitivity of performance results to changes in workload characteristics in the context of the workload synthesis process. The effects of the major changes in workload characteristics due to the synthesis process are studied.

Chapter 5 investigates the use of the synthetic workloads for efficient power dissipation analysis.

Chapter 6 extends the workload synthesis techniques to the detailed performance models of real-world high-end processors. Performance model validation experiments on two platforms for an industrial chip, the POWER5 processor, are described.

Chapter 7 concludes the dissertation with a summary of the contributions of the dissertation and suggestions for future research opportunities.

Chapter 2: Workload Modeling and Statistical Simulation

Statistical simulation systems provide an efficient way to carry out early design studies for processors [4][27]. This chapter describes statistical simulation and investigates workload modeling improvements that result in more accurate simulations. The profiling step is modified to collect dynamic execution statistics at the granularity of the basic block instead of collecting average statistics at the granularity of each instruction type.

These improvements to workload modeling also serve as the foundation for the accurate workload synthesis system discussed in the rest of this dissertation.

2.1 PERFORMANCE SIMULATION STRATEGIES AND STATISTICAL SIMULATION

The increasing complexity of modern processors [46][1][40][82][85] drives detail into the simulators used to project the performance of the designs and to study design tradeoffs [27][57][56]. The complexity of the simulator depends on the level of accuracy required for the performance evaluation, which is related to the maturity of the design in the development process.

The complexity of the performance simulator is also determined by the kind of program being simulated. For example, simple bandwidth and latency tests may focus on the memory subsystem and may not need a detailed processor core model. In general, however, a performance engineer would like to assess the performance of the full design and quantify the impact of microarchitectural changes running benchmarks that are representative of user applications. Applications and their associated datasets are preferred because microbenchmarks and the synthetic programs like Whetstone [22] and Dhrystone [103] may not stress the processor in the same manner as actual applications

[36][104][23][7]; that is, simple microbenchmarks and synthetics may not represent the performance of real programs.

Ideally, the performance of applications would be investigated using a detailed RTL simulator. However, modern applications and benchmarks like the SPEC 95 and 2000 [89] exhibit dynamic instruction counts in the tens of billions [37][58], making RTL simulation of the complete programs impractical [83][108][58]. Researchers have turned to fast architectural simulators like SimpleScalar [16], but even so the runtime for a complete program can be on the order of hours to days [83][108][58]. The TPC benchmarks [96] are often used to evaluate database performance, but they are difficult to set up even in hardware and they also have long runtimes [32][35].

Runtimes can be reduced using reduced input datasets [50], but detailed studies indicate that accuracies using the smaller datasets are mixed [30][95]. Older codes like SPEC 89 [89], the debit-credit database benchmark [2], and the PERFECT club scientific applications [74][18] can be used to reduce runtimes, but they are less representative of modern applications.

Recent work has shown that there is often more than one benchmark in suites like the SPEC CPU that exercise machines in essentially the same way when executed [25][26][80][28][99][75]. That is, they cover similar workload characteristics [47] when executed on the machine. The workload characteristics can include IPC, instruction mix, instruction dependences, cache miss rates, branch predictability, dispatch window occupancies, average fetches per cyles, etc. In Saveedra-Barrera [80], the runtimes and workload characteristics of a variety of benchmarks are compared and shown to exhibit similar behavior. In Dujmovic and Dujmovic [26], the runtimes of the SPEC 95 benchmarks over multiple computer systems are compared and found to contain significant redundancy with respect to obtaining the same performance on multiple machines. In Eeckhout [28], principal component and clustering analyses point to significant similarities in workload characteristics among the SPEC benchmarks. In Vandierendonck and De Bosschere [99], 80% of the statistical variation in the SPEC 2000 benchmarks can be obtained using only four benchmarks, and 91% using nine benchmarks. In Phansalkar *et al.* [75] it is shown that many of the SPEC benchmarks exhibit similar microarchitecture-independent workload characteristics. These papers reach the same conclusions, namely, that there are many benchmarks that exhibit similar workload characteristics.

Even internal to the benchmarks themselves, similar workload characteristics exist. In Sherwood *et al.* [83], many applications exhibit *phase* behavior in which IPC and other workload characteristics repeat over the runtime of the application. In Wunderlich *et al.* [108], fewer than ten thousand samples of one thousand instructions each from the application execution can achieve CPIs within 3% error of that obtained executing the complete application. The conclusion from these works is that applications exhibit a small set of representative phases, and that trace samples of these executions are sufficient to examine the effect of a design choice for a particular set of workload characteristics.

In spite of the success of phase identification and trace sampling techniques, even the fastest trace sampling techniques using benchmarks such as SPEC can require hours to evaluate a single design choice with one sample and dataset pair on an execution-driven simulator [83][108][58]. Checkpointed sampling [105][98] trades off dynamic memory usage versus runtime and can reduce runtimes to minutes, but large numbers of design points still make investigations of large design spaces prohibitive. Researchers have also used trace-driven simulation to reduce runtimes [51]. However, traces for just a few seconds of hardware execution time can be prohibitively large, impact simulation times, and are not easily modified to study a range of workload spaces [28].

Researchers have responded to long runtimes with the development of simulation systems that model aspects of the workload or performance model statistically. Noonburg and Shen [68] present a framework that models the execution of a program on a particular architecture as a Markov chain. The state space is determined by the microarchitecture and the transition probabilities are determined by program execution. The approach is demonstrated for simple in-order machines. Modeling of superscalar, out-of-order machines would result in unmanageably complex Markov chains.

Statistical simulation systems model the workload and aspects of the machine performance model statistically [17][71][69][70][48][28]. Statistical simulation can reduce runtimes to seconds or minutes and dynamic instruction counts to under a million. Statistics that describe workload characteristics are gathered during dynamic execution using a profiling tool. The statistics are then used to create a synthetic trace. The trace is applied to a fast and flexible performance model. The profile collects statistics for both *microarchitecture-independent* characteristics, such as the instruction mix and inter-instruction dependence frequencies, and *microarchitecture-dependent* statistics, such as cache miss rates and branch predictabilities. The workload characteristics are collected at the granularity of individual instruction types - the *context* in which an instruction appears in the dynamic instruction stream is not considered. For example, statistics about integer instructions such as the dependence distance distributions are collected in aggregate for all integers encountered over the entire execution regardless of the sequence of instructions executed prior to any particular integer instruction.

The execution engine typically models the stages in a superscalar out-of-order execution machine including fetch, dispatch, issue, execution, and completion. Cache accesses are modeled statistically using the miss rates from the profile, and, likewise, branching behavior is modeled using the global branch predictability from the profile. Cache misses are modeled as additional latency prior to instruction completion. Specialized workload features such as load-hit-store address collisions or data (for value prediction studies) are modeled statistically. In Joshi *et al.* [48], the execution engine is modeled as a series of delays, and additional statistics facilitate the modeling of read and write buffers in a multiprocessor system. Since workload characteristics are determined from a statistical distribution, the simulation converges to a result much faster than standard performance simulations.

Statistical simulation systems that correlate well with execution-driven simulators have been shown to continue to exhibit good accuracy as microarchitecture changes are applied in design studies [28][29]. Studies have achieved average errors less than 5% on specific benchmark suites [71][28][29], but most studies examine only the integer SPEC 2000 benchmarks, not a variety of codes.

In this chapter, the correlation of a statistical simulation system, HLS [71], over a range of benchmarks is studied, from general-purpose applications to technical and scientific benchmarks, and streaming kernels. The inaccuracy of HLS is studied and the results are used to improve it. The workload model is improved by collecting information at the granularity of the basic block instead of at the instruction level, and more detail is added to the processor model. Modeling detail is incrementally added to the HLS framework to uncover the additional complexity necessary to improve HLS. The cost of the improvements in terms of additional storage requirements is quantified.

Also, a simple regression model indicates that CPI results for the SPEC INT 95, the benchmarks originally used to calibrate HLS, can yield to very simple modeling. The

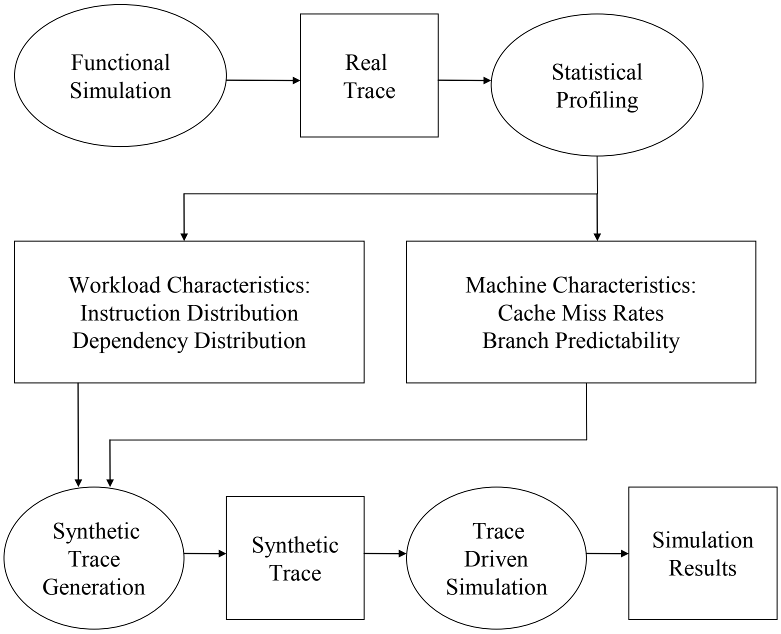

Functional Simulation
Real Trace
Statistical Profiling
Workload Characteristics: Instruction Distribution Dependency Distribution
Machine Characteristics: Cache Miss Rates Branch Predictability
Synthetic Trace Generation
Synthetic Trace
Trace Driven Simulation
Simulation Results

Figure 2.1: Overview of Statistical Simulation: Profiling, Synthetic Trace Generation, and Trace-Driven Simulation

analysis points to a larger problem for simulator developers: using a small set of benchmarks, datasets and simulated instructions to calibrate a simulation system.

In the next section, statistical simulation in HLS is described. Section 2.3 describes workload and processor modeling problems found in the HLS statistical simulation system. Section 2.4 investigates improvements to the modeling. The costs of the improvements are quantified in Section 2.5, followed by a summary of the findings.

2.2 OVERVIEW OF STATISTICAL SIMULATION IN HLS

Statistical simulation is carried out in three major steps: profiling the workload, creating a synthetic trace from the profile, and simulating the synthetic trace on a machine model [28]. These steps are shown in Figure 2.1 and are described in the paragraphs below in the context of the HLS statistical simulation system [71]. This section and the next describe only prior work implemented in HLS.

In the first step, a real trace from a functional simulation of the workload is fed into a profiler in which workload and machine characteristics are compiled. In HLS, machine-independent characteristics are profiled using a modified version of the *sim-fast* functional simulator from the SimpleScalar toolset [16]. An instruction mix distribution is computed that consists of the frequencies of five instruction types: integer, float, load, store and branches. Also computed are the average basic block size, the block size standard deviation, and a frequency distribution of the read-after-write dependence distances between instructions for each input of the five instruction types. Profiling does not consider instruction anti-dependences. The benchmarks are also executed for one billion cycles in *sim-outorder* [16], which provides an IPC to compare against the IPC obtained in statistical simulation. *Sim-outorder* also computes the machine characteristics used for statistical modeling of the locality structures: L1 I-cache and D-cache miss rates, the unified L2 cache rate, and the branch predictability. These characteristics could also be profiled in a fast cache simulator like *sim-cache* or a branch predictor like *sim-bpred*, both from the the SimpleScalar toolset.

In the second step, the profiled statistics are used to create a synthetic trace. HLS generates one hundred basic blocks using a normal random variable over the mean block size and standard deviation. A uniform random variable over the instruction mix distribution fills in the instructions of each basic block. For each randomly generated instruction, a uniform random variable over the dependence distance distribution generates a dependence for each instruction input. If a dependence points to a store or branch within the current basic block, another random trial chooses another dependence. If the dependence stretches beyond the limits of the current basic block, no change is made because the dynamic predecessor instruction is not known.

The basic blocks are connected into a graph structure. Each branch has both a taken pointer and a not-taken pointer to other basic blocks. The percentage of backward branches, set statically to 15% in the code, determines whether the taken pointer is a backward branch or a forward branch. For backward or forward branches, a normal random variable over either the mean backward or forward jump distances (set statically to ten and three in the code, respectively) determines the taken target. Later, during simulation, normal random variables over the overall branch predictability obtained from the *sim-outorder* run determine dynamically if the branch is actually taken or not, and the corresponding branch target pointer is followed. Note that there is no analysis to determine that simulation does not get stuck in a sub-graph of the full graph.

In the third step, the synthetic trace is simulated. After the machine statistics are processed and the basic blocks are configured, the instruction graph is traversed. As each instruction is encountered, it is simulated on a generalized superscalar execution model. Execution continues for ten thousand cycles and the IPC is averaged over twenty runs. The generalized model contains fetch, dispatch, execution, completion, and writeback stages. Fetches are buffered up to the fetch width of the machine. Instructions are dispatched to issue queues in front of the execution units and executed as their dependences are satisfied. Neither an issue width nor a commit width is specified in the processor model. In HLS, the procedure is to first calibrate the generalized processor model using a test workload and then execute a reference workload.

For loads, stores, and branches, the locality statistics determine the necessary delay before issue of dependent instructions. To provide comparison with the SimpleScalar *lsq*, loads and stores are serviced by a single queue. Parallel cache miss operations are provided through the two memory ports available to the load-store execution unit. As in SimpleScalar, stores execute in zero-time when they reach the tail of their issue queue and the execution unit is available.

2.3 SIMULATION RESULTS

This section describes the experimental setup and benchmarks used in the statistical simulation experiments, followed by an examination of HLS, which includes descriptions of several workload and processor modeling issues. This section describes only results using the original HLS system, except in section 2.3.4, two experimental set-up errors are found and the results with the fixes are given.

2.3.1 Experimental Setup and Benchmarks

The experimental procedure follows that in Oskin *et al.* [71][72]. SimpleScalar and the statistical simulation software are compiled for big-endian Pisa (MIPS) binaries on an IBM POWER3 p270. Using the parameters in Table 2.1 as in Oskin *et al.* [71], *sim-outorder* is executed on the SPEC 95 Pisa binaries for up to one billion instructions for the first reference input dataset. The modified *sim-fast* is executed on the input dataset for fifty billion instructions, to approximate complete program simulation.

Table 2.1: Machine Configuration for Pisa and Alpha Simulations

Feature	Pisa	Alpha
Instruction Size (bytes)	8 (effectively 4)	4
L1/L2 Line Size (bytes)	32/64	
Machine Width	4	
Dispatch Window/LSQ/IFQ	16/8/4	
Memory System	16K 4-way L1 D, 16K 1-way L1 I, 256K 4-way unified L2	
L1/L2/Memory Latency	1/6/34	
Functional Units	4 I-ALU, 1 I-MUL/DIV, 4 FP-ALU, 1 FP-MUL/DIV	
Branch Predictor	Bimodal 2K table, 3 cycle misspredict penalty	

In these experiments, the SPEC 95 integer benchmarks provide direct comparison with the original HLS results [71]. The SPEC 95 floating point benchmarks and single-precision versions of the STREAM and STREAM2 benchmarks [64] are added. Results are also given for Alpha versions of the SPEC 2000 and STREAM benchmarks. Unless noted, the following figures are for the SPEC 95 Pisa runs. The STREAM benchmarks are included because of the particular challenges they pose to statistical simulation systems, discussed in Section 2.3.5.

2.3.2 The HLS Graph Structure

First, the HLS front-end graph structure is examined. The percentages of backward branches, the backward branch jump distance, the forward branch jump distance, and the graph connections themselves are varied. Figure 2.2 shows the effect of varying the front-end graph connectivity. *Baseline* is the base HLS system running with the taken and not-taken branches connected as described in Section 3.2. *Random not-taken* is the base system with the not-taken target randomly selected from the configured basic blocks. *Single loop* is the base system with the taken and not-taken targets of each

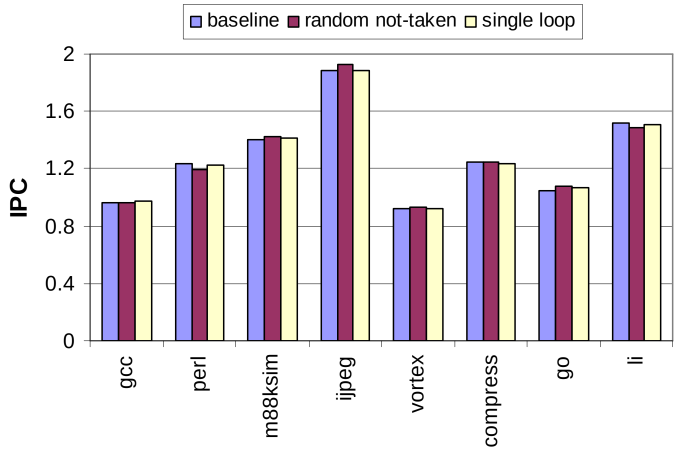


Figure 2.2: Effect of Graph Connectivity in HLS on SPEC INT 95 Benchmarks

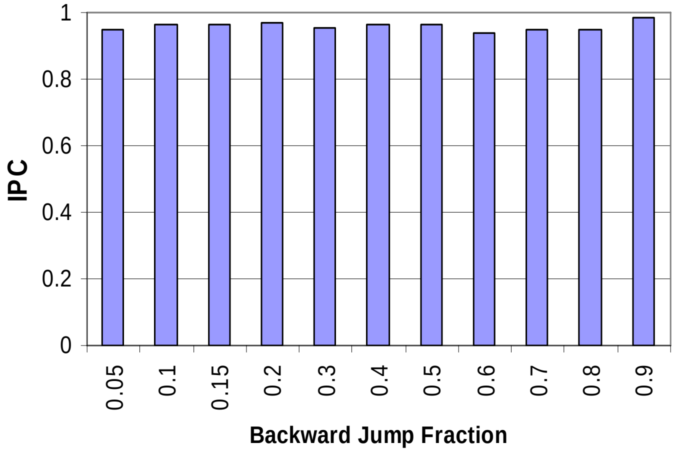


Figure 2.3: Effect of Changes in Backward Jump Fraction for *gcc*

basic block both pointing to the next basic block in the sequence of basic blocks, with the last basic block pointing back to the first. The maximum error versus the base system is 3.6% for *perl* using the random not-taken strategy. This is well below the average HLS correlation error versus SimpleScalar of 15.5% error shown in Figure 1.1.

Figure 2.3 shows the IPC for *gcc* as the fraction of backward jumps is varied. The hard-coded HLS default is 15% backward jumps, and the maximum error versus that default is 2.8%. Figure 2.4 shows IPC as the backward and forward jump distances are

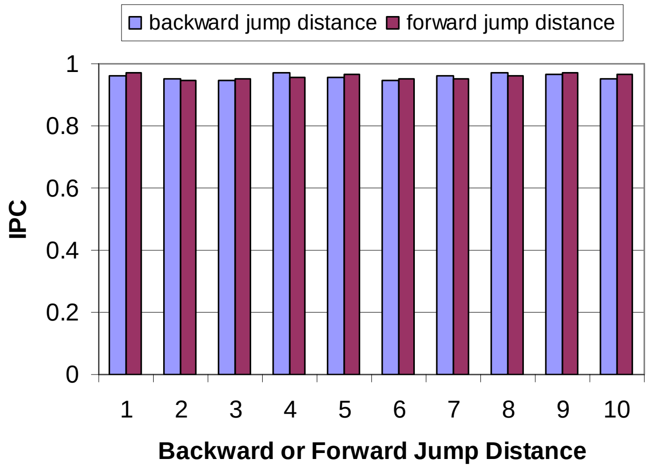


Figure 2.4: Effect of Changes in Backward or Forward Jump Distance for *gcc*

varied from their HLS defaults of ten and three, respectively. The maximum change versus either default is 2.0%.

From these figures, it is apparent that the graph connectivity in HLS has no effect on simulation performance. Intuitively, HLS models the workload at the granularity of the instruction. All instructions in all basic blocks in the graph are generated identically. The instruction type and dependences assigned to any instruction slot in any basic block in the graph is randomly selected from the global instruction mix distribution, so the instruction found at any slot on a jump is just as likely to be found at any other slot.

There is also a small probability that the random graph connectivity causes skewed results because the randomly selected taken targets can form a small loop of basic blocks, effectively pruning sub-graphs of the graph from the simulation. This is not a major problem for HLS, in which all blocks are statistically the same, but it has implications for the improvements to HLS described below, so the single loop strategy is employed for the remainder of this chapter.

2.3.3 The HLS Processor Model

In the generalized execution model of HLS, there is no issue-width concept. The issue of instructions to the issue queues is instead limited by the queue sizes and dispatch window and, ultimately, by the fetch window. There is also no specific completion width in HLS, so the instruction completion rate is also limited by the front-end fetch window. These omissions are conducive to obtaining quick convergence to an average result for well-behaved benchmarks, but they make it difficult to correlate the system to SimpleScalar for a variety of benchmarks, including STREAM.

2.3.4 Issues in the Experimental Setup of HLS

Figures 2.5 and 2.6 show the IPC prediction error [27] over all benchmarks as workload modeling issues are incrementally addressed. The *baseline* run gives the HLS results out-of-the-box with an average error of 15.5%. While SPEC INT 95 does well with only 5.8% error, as expected from Oskin *et al.* [71], SPEC FP 95 has twice the correlation error at 13.6%. The STREAM loop error is more than four times worse at 27.6%. Recalibrating the generalized HLS processor model did not achieve more accurate results.

In standard HLS, it may be recalled, measuring microarchitecture-independent characteristics is carried out on the complete benchmark using *sim-fast*, whereas microarchitecture-dependent locality metrics are obtained only for the first one billion instructions using *sim-outorder*. It stands to reason that workload information and locality information should be collected over the same instruction ranges. The *1B Instructions* run gives results with *sim-fast* executing the same one billion instructions as *sim-outorder*. Not all benchmarks improve, but the error in SPEC FP 95 drops by half to 6.8%. Overall

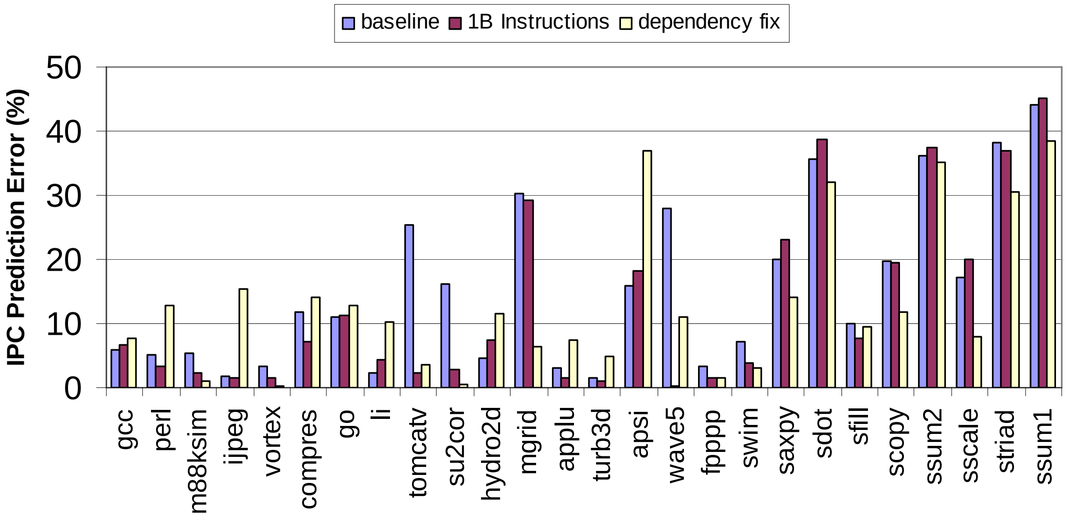


Figure 2.5: Error in HLS by Benchmark as Experimental Setup Changes

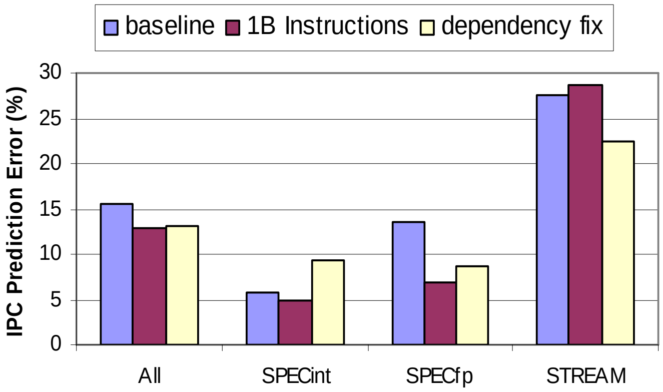


Figure 2.6: Error in HLS by Benchmark Suite as Experimental Setup Changes

error decreases to 13.1%. The results indicate that, as could be expected, the SPEC FP workload characteristics in the first one billion instructions are significantly different than those over the full execution. The difference is probably due to cache warmup effects.

The modified *sim-fast* makes no distinction between memory instructions that carry out auto-increment or auto-decrement on the address register after memory access and those that do not. The HLS *sim-fast* code always assumes the auto-modes are active. This causes the code to assume register dependences that do not actually exist between memory access instructions, and it makes codes with significant numbers of load and store address register dependences, including the STREAM loops, appear to run slower. The *sim-fast* code was modified to check the instruction operand for the condition and mark dependences accordingly, and the *dependence fix* bars in the figures give the results. The STREAM loops are improved, but the SPEC INT 95 error increases from 4.8% to 9.3%. This is most likely due to the original calibration of the generalized HLS processor model using SPEC INT 95 in the presence of the modeling error.

Table 2.2 shows a simple regression analysis over the locality features taken from *sim-outorder* runs: branch mispredictability, L1 I-cache and D-cache miss rates, and L2 miss rate. The *targeted CPI* is the particular CPI targeted in the analysis, either

SimpleScalar or the HLS result. The squared correlation coefficient, R^2, is a measure of the variability in the CPI that is predictable from the four features. The SPEC INT 95 benchmarks always achieve high correlation, while the analysis over all benchmarks or even over SPEC INT 95 together with SPEC FP 95 achieve lower correlation. This is an indication that a very simple processor model can potentially represent the CPI of the SPEC INT 95 by emphasizing the performance of the locality features, but it can not as easily do the same over all three suites.

Table 2.2: CPI Regression Analysis for the SPEC 95 Benchmark Suites

Benchmarks	**Targeted CPI**	**R^2**
SPEC INT	HLS	0.988
	SimpleScalar	0.970
SPEC INT and SPEC FP	HLS	0.972
	SimpleScalar	0.895
SPEC INT, SPEC FP and STREAM	HLS	0.757
	SimpleScalar	0.811

The remaining results in this chapter use the one billion instructions and dependence experimental setup fixes throughout.

Table 2.3: Single-Precision STREAM Loops

Benchmark	**Equation**	**Instructions per Loop**
saxpy	z[k] = z[k] + q * x[k]	10
sdot	q = q + z[k] * x[k]	9
sfill	z[k] = q	5
scopy	z[k] = x[k]	7
ssum2	q = q + x[k]	6
sscale	z[k] = q * x[k]	8
striad	z[k] = y[k] + q * x[k]	11
ssum1	z[k] = y[k] + x[k]	10

2.3.5 Challenges Modeling the STREAM Loops

The errors for STREAM in Figure 1.1 and Figure 2.6 point to additional workload modeling challenges in HLS. Table 2.3 shows single-precision versions of the STREAM benchmarks, including the kernel loop equation and the number of instructions in the kernel loop when compiled with *gcc* using *-O*. The STREAM loops are strongly phased, and in fact have only a single phase. The loops consist of one or a small number of tight iterations containing specific instruction sequences that are difficult for statistical simulation systems, including HLS, to model. Figure 2.7 shows one iteration of the *saxpy* loop in the Pisa language [16]. If the *mul.s* and *add.s* were switched in the random instruction generation process leaving the dependence relationships the same, the extra latency of the multi-cycle *mul.s* instruction is no longer hidden by the latency of the second *l.s*, leading to a generally longer execution time for the loop. A similar effect can be caused by changes in dependence relationships as the dependences are statistically generated from a distribution.

Shorter runs are also possible. The *mul.s* has a dependence on the previous *l.s*. If the *l.s* is switched with the one-cycle *add.s*, keeping dependences the same, the *mul.s* can dispatch much faster. While higher-order ILP distributions might work well for some loops, the results have been mixed and can actually lead to decrease in accuracy for

```
start:  addu $2, $3, $6
         l.s $f2, 0($2)
         mul.s $f2, $f4, $f2
         l.s $f0, 0($3)
         add.s $f2, $f2, $f0
         addiu $4, $4, 1
         slt $2, $5, $4
         s.s $f2, 0($3)
         addiu $3, $3, 4
         beq $2, $0, start
```

Figure 2.7: Disassembled *SAXPY* Loop in the *Pisa* Language

general-purpose programs [28].

The conclusion from this section is that, for strongly phased workloads like the STREAM loops, the sequence of instructions obtained from the compiler is critical to performance. The random instruction generation of HLS has little chance of capturing the performance of these loops, leading to the errors in Figure 1.1.

2.4 IMPROVING PROCESSOR AND WORKLOAD MODELING IN HLS

The last section the errors in workload and processor modeling in HLS including challenges the methodology has for the STREAM suite. This section presents the proposed improvements to the processor and workload models in HLS described in the last section to obtain more accurate simulation results. All improvements described here are new.

2.4.1 Improving the Processor Model

It is difficult to correlate the generalized HLS processor model to SimpleScalar for all benchmarks. For this reason, we augment HLS with a register-update-unit (RUU), an issue width and a completion width. The completion function in HLS is rewritten to be non-recurrent and called prior to execution, and the execution unit is rewritten to issue new instructions only after prior executing instructions have been serviced in the current cycle. Code is added to differentiate long and short running integer and floating point instructions.

First, the benchmarks are executed on the improved processor model using the same workload characteristics modeled in HLS. To get accurate results using the new processor model, we found experimentally that one thousand basic blocks must be generated instead of one hundred, and twenty thousand cycles must be simulated instead of ten thousand, so simulation time is about twice that of HLS. Similar increases in basic blocks and runtimes in HLS did not improve its results. The execution engine flow, delays, and parameters for both Pisa and Alpha ISAs are chosen to match those in the configuration in Table 2.1. The baseline system is validated by comparing *sim-outorder* traces obtained from executions of the STREAM loops to traces taken from HLS simulations.

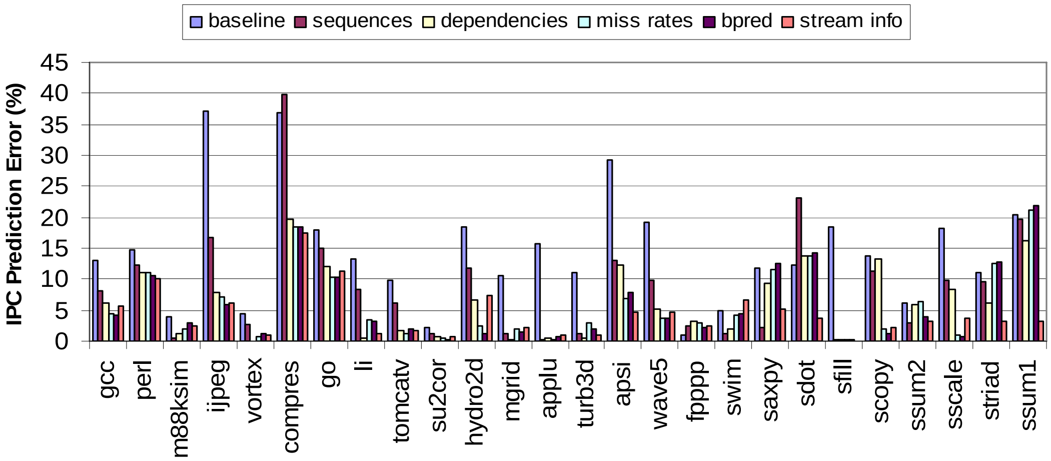


Figure 2.8: Improved Error in HLS by Benchmark as Modeling Changes

Figure 2.8 gives the results for the individual benchmarks, and Figure 2.9 shows the average results per benchmark suite. The *baseline* run gives the improved system

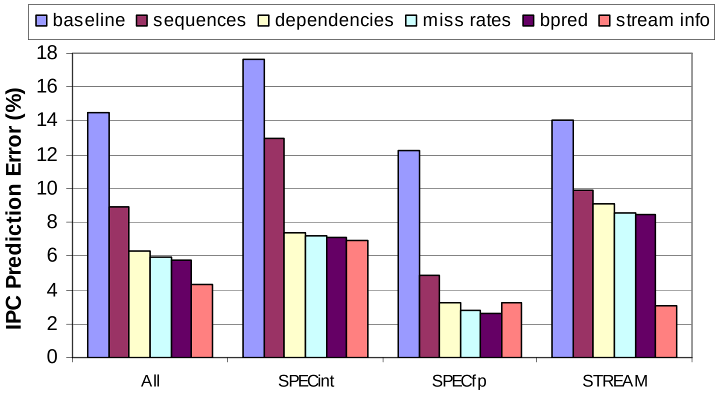


Figure 2.9: Improved Error in HLS by Benchmark Suite as Modeling Changes

results using the default SimpleScalar parameters and using the global instruction mix, dependence information, and load and store miss rates. There are errors greater than 25% for particular benchmarks, such as *ijpeg*, *compress* and *apsi*. The overall error of 14.4% is better than the 15.5% baseline error in HLS, but it is higher than the 13.1% error shown in Figure 2.6 for HLS with some improved workload modeling. In the following subsections, the rest of the bars are explained. The results are additive as each level of modeling is added.

2.4.2 Improvements to Workload Modeling

The workload model is also enhanced to reduce correlation errors. The analysis of the graph structure shows that modeling at the granularity of the instruction in HLS does not contribute to accuracy. In Nussbaum and Smith [69], the basic block size is the granule of simulation. However, this raises the possibility of aliasing among the basic block sizes, in which many blocks of the same size but very different instruction sequences and dependence relationships are combined.

2.4.2.1 Basic Block Modeling Granularity

Instead of risking reduced accuracy with block size aliasing, the workload is modeled at the granularity of the basic block itself. The dynamic frequencies of all basic blocks (including instruction sequences, and related workload characteristics such as dependences and locality statistics) are collected during profiling and used as a probability distribution function for building the sequence of basic blocks in the graph. The next basic block is determined using a random variable over the probability distribution function. To capture cache and branch predictor statistics for the basic blocks, *sim-cache* is augmented with the *sim-bpred* code of SimpleScalar.

For the *sequences* bars results of Figures 2.8 and 2.9, the basic block instruction sequences are used, but the dependences and locality statistics for each instruction in each basic block are still taken from the global statistics found for the entire benchmark. The overall correlation errors are reduced dramatically for the three classes of benchmarks. However, some benchmarks such as *compress* and *hydro2d*, and the STREAM loops, still show high correlation errors.

For the *dependences* results, dependence information local to each basic block is then included in simulation. In order to reduce the amount of information stored, the dependences are merged into the smallest dependence relationship found in any basic block with the same instruction sequence, as in Eeckhout *et al.* [27]. The average error is reduced significantly from 8.9% to 6.3%.

On investigation, it was found that the global miss rate calculations do not correspond to the miss rates from the viewpoint of the memory operations in a basic block. In the cache statistics, HLS pulls in the overall cache miss rate number from SimpleScalar, which includes writebacks to the L2. But for individual memory operations in a basic block, the part of the L2 miss rate due to writebacks should not be included in the miss rate that is compared to the miss rate local to a basic block. This is because the writebacks generally occur in parallel with the servicing of the miss so they do not contribute to the latency of the operation. This argues for either a global L2 miss rate calculation that does not include writebacks or the maintenance of miss rate information for each basic block. In addition, examination of the STREAM loops reveals that the miss rates for loads and stores are quite different. In *saxpy*, for example, both loads miss to the L1, but the store always hits. Because of these considerations, the L1 and L2 probabilistic miss rates for both loads and stores are maintained local to each basic block.

The *miss rates* results in the figures adds this information for simulation. All benchmarks improve, but a few of the STREAM loops still have errors greater than 10%. The problem is that the STREAM loops need information concerning how the load and store misses overlap and cause the phenomenon known as *delayed hits* [28]. In most cases load misses overlap, but the random cache miss variables often cause them to not overlap, leading to an underestimation of performance. Note that this is the reverse of the usual situation for statistical simulation in which critical paths are randomized to less critical paths, and performance is overestimated. An additional run, *bpred*, includes branch predictability local to each basic block. This helps a few benchmarks like *ijpeg* and *hydro2d*, but, as expected, the STREAM loops are unaffected.

One solution is to maintain statistics about how frequently memory operations overlap with each other and model the phenomenon statistically. This solves the delayed hits problem, but it does not provide a more general model of memory dependences. Instead, when the workload is characterized, one hundred L1 and L2 hit/miss indicators (i.e. if the memory operation was an L1 hit or miss or an L2 hit or miss) are maintained for the sequence of loads and stores in each basic block near the end of the one billion instruction simulation. Later, during statistical simulation, the stream indicators are used in order (without pairing them explicitly to particular memory operations) to determine the miss characteristics of the stream as the loads and stores are encountered. Obviously, this is an *ad hoc* and simplistic way to operate, since the stream hit/miss indicators are simply collected at the end of the run and are therefore not necessarily representative of the entire run. However, the technique may be useful given the trend to identify and simulate program phases [83] in which stream information may change little. Still, simulating one billion instructions without regard to phase behavior, the technique is expected to help only the STREAM loops, and to negatively affect the others.

The *stream info* bars in Figure 2.8 show the results. As expected, the STREAM loops improve significantly. However, only a small amount of accuracy is lost for the others. This indicates that there is only one or a small number of phases in the first one billion instructions for most benchmarks, at least with respect to the load and store stream behavior.

2.4.2.2 Basic Block Maps

In the previous simulations, the basic blocks were not associated with each other in any way since a random variable over the frequency distribution of the blocks is used to pick the next basic block to be simulated. At branch execution time, a random variable based on the global branch predictability is used simply to indicate that a branch misprediction occurred when the branch was dispatched, causing additional delay penalty before the next instruction can be fetched, but that is not linked to the successor block decision. All blocks are treated as if no phases exist in which one area of the graph is favored over another at different times.

By associating particular basic blocks with each other in specific time intervals, for example during a program phase, it is expected that better simulation accuracy can be obtained for multi-phase programs. One way to do that is to specify the phases, the basic blocks executing in those phases, and the relative frequencies of the basic block executions during those phases. These three things together constitute a *basic block map*.

Phase identification requires knowledge of when the relative frequencies of the basic blocks change. The identification of phases at a coarse granularity is most effectively carried out using an industrial-strength phase identification program such as SimPoint [83]. Alternatively, phase-like behavior can be identified dynamically during simulation by traversing a representation of the control flow graph of the program, called the *statistical flow graph* [27]. Since identification is carried out continuously as an artifact of the simulation, the possibility exists for detecting the fine-grained phases, called *micro-phases*, which are small shifts in relative block frequencies internal to a heavy-duty phase. The experiments below and the additional collaborative research in Eeckhout *et al.* [27] show that good accuracy can be achieved using this method.

The concept of basic block maps and the statistical flow graph is related to the work in Iyengar *et al.* [42][43]. That work introduces the notion of *qualified* basic blocks; that is, the basic blocks are qualified with their context, the particular sequence of basic blocks that led up to their execution. During dynamic execution of a trace, the workload characteristics of a basic block, such as branch predictability and cache behavior, are distinguished based on its execution context, i.e. what the preceding k dynamic basic blocks were. During creation of a representative trace, the workload characteristics for the current basic block are determined based on the characteristics recorded for the context. The context is similar to the statistical flow graph in Eeckhout *et al.* [27], but the statistical flow graph is reduced by modeling only the basic block instructions, dependences and branch predictability, not memory addresses and cache histories. The basic block map retains this statistical flow graph concept but aggregates the results for all prior basic blocks into one set of statistics (i.e. the context is all prior basic blocks) and augments the workload characteristics of each basic block with a history of the memory access behavior of loads and stores.

To implement this in HLS, each basic block is annotated with a list of pointers to its successor blocks along with the probabilities of accessing each successor (equivalent to $k = 0$ modeling in Eeckhout *et al.* [27]). By traversing the basic blocks as in the previous section, but using a random variable over the successor probabilities to pick the successor, the micro-phase behavior is uncovered. The total number of instances of a

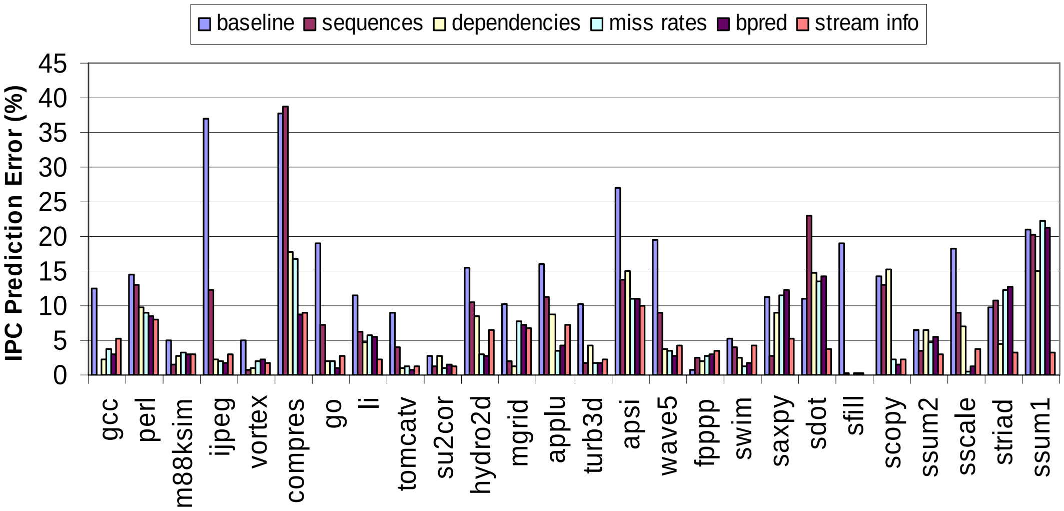


Figure 2.10: Improved Error in HLS by Benchmark as Modeling Changes with Basic Block Maps

basic block is limited to be proportional to its frequency in the original application [27]. The same strategies as before are simulated. This improved HLS system is called HLS++.

Figures 2.10 and 2.11 show the results. The overall error using all techniques is improved only a little from 4.35% to 4.11%, a 5.5% decrease. SPEC INT 95 is improved from 6.9% to 4.3%, or 38% on average. The STREAM loops are unchanged since they consist of a single phase, and there is no advantage in using basic block maps in that case.

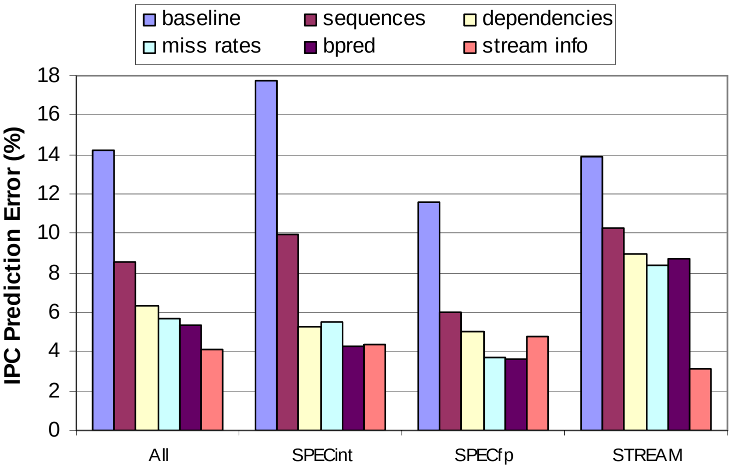


Figure 2.11: Improved Error in HLS by Benchmark Suite as Modeling Changes with Basic Block Maps

The SPEC FP 95 show an increase in error from 3.3% to 4.7%. Part of this is due to the negative effects of using stream information. The low overall improvement agrees with the results found in the last subsection, in which stream information - which should be phase dependent - causes few adverse effects. Coupled with increased variance from simulating only twenty thousand cycles, the result is not surprising. Improvements are also limited by errors in the graph structure, including the merge of dependences explained earlier.

In an additional experiment, the SPEC 2000 Alpha profiles are executed in the original HLS system with the default 100 basic blocks and 10K simulated cycles. An overall error of 29.2% is obtained, with SPEC INT at 25.7% error, SPEC FP at 23.4%, and STREAM at 44.5%. The error is higher than the original Pisa error because the Alpha codes contain more unique basic blocks than Pisa over one billion instructions (on average 746 unique basic blocks per benchmark versus 524 for Pisa) and the fact that HLS was calibrated for the SPEC INT 95 benchmarks [71]. Executing with 2500 basic blocks and 20K cycles does not help. However, the improved system is remarkably more

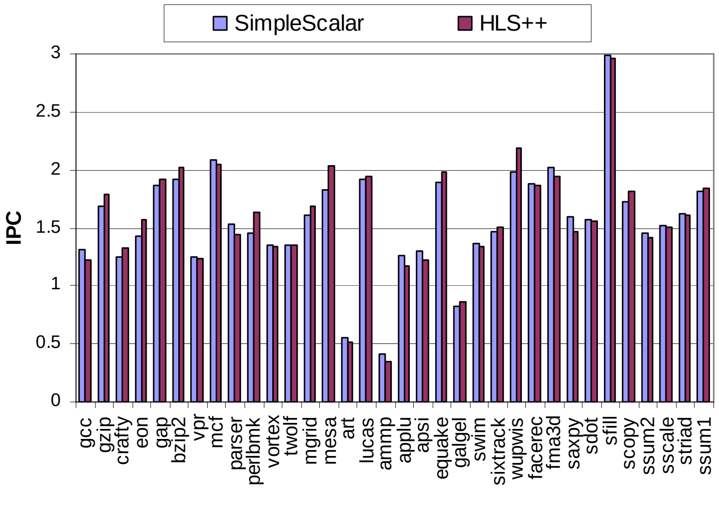


Figure 2.12: IPC for HLS++ by Benchmark versus SimpleScalar for the SPEC 2000 in the Alpha Language

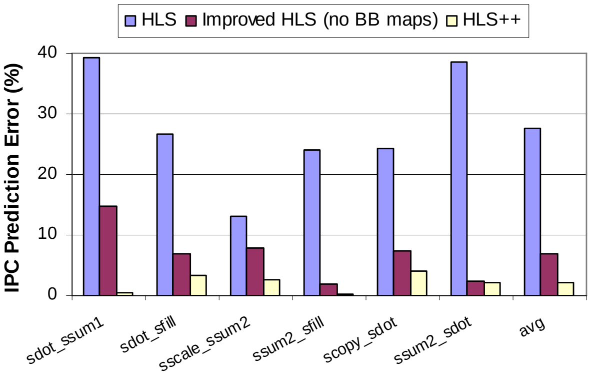


Figure 2.13: Versions of HLS Executing Two-Phase Benchmarks

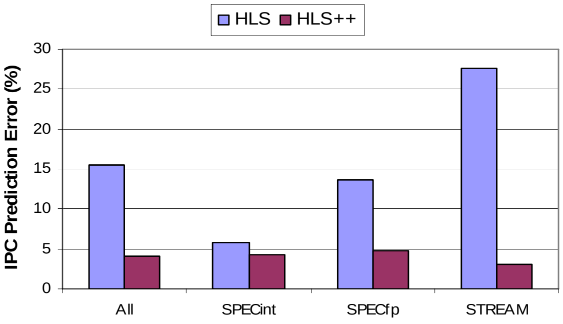


Figure 2.14: HLS versus HLS++ for the SPEC 95 Benchmarks

accurate. Figure 2.12 compares HLS++ simulation to the cycle accurate run for the SPEC 2000. Shown is an overall 4.7% error. The SPEC INT show an overall error of 4.8%, SPEC FP 5.7%, and STREAM 2.5%. This experiment demonstrates that the new HLS++ workload and processor modeling is robust across many benchmarks in both ISAs.

2.4.2.3 Basic Block Maps for Strong Phases

Basic block maps demonstrate larger improvements for programs with a number of strong phases. To demonstrate the effectiveness of the technique, several benchmarks are created using combinations of the STREAM loops. Figure 2.13 shows, for example, that a simple code created from the concatenation of *sdot* and *ssum1* has correlation errors of 39.4% and 14.8% in HLS and HLS++ without basic block maps, respectively. In HLS++ without basic block maps, given that 50% of the blocks are equivalent to *sdot* blocks, and 50% are equivalent to *ssum1* blocks, the resulting sequence of basic blocks is a jumble of both. The behavior of the resulting simulations tends to be pessimistic with long-latency L2 cache misses forming a critical chain in the dispatch window. When the basic block map technique is applied, the error shrinks to 0.4%.

Figure 2.14 and 2.15 compare HLS to the complete HLS++ system running with all optimizations including basic block maps. The improvements show a 4.1% average error, which is 3.78 times more accurate than the original HLS at 15.5% error.

2.5 IMPLEMENTATION COSTS

Table 2.4 shows the cost of the improvements in bytes as a function of the number of basic blocks (NBB), the average length of the basic blocks (LBB), the average number of loads and stores in the basic block (NLS), the average number of successors in the basic blocks (SBB), and the amount of stream data used (NSD). NSD is NLS x 100 = 4.71 x 100 = 471 in the runs. Table 2.5 shows the error reduction as the average reduction in correlation error as each technique augments the previous technique.

There are only five instruction types, so four bits are used to represent each. There are two dependences per instruction, each of which is limited to within 255; so two bytes of storage per instruction are needed. Both load and store miss rates for the L1 and L2 caches are maintained so four floats are needed. For basic block maps, the successor pointer and frequency are maintained in a 32-bit address and a float.

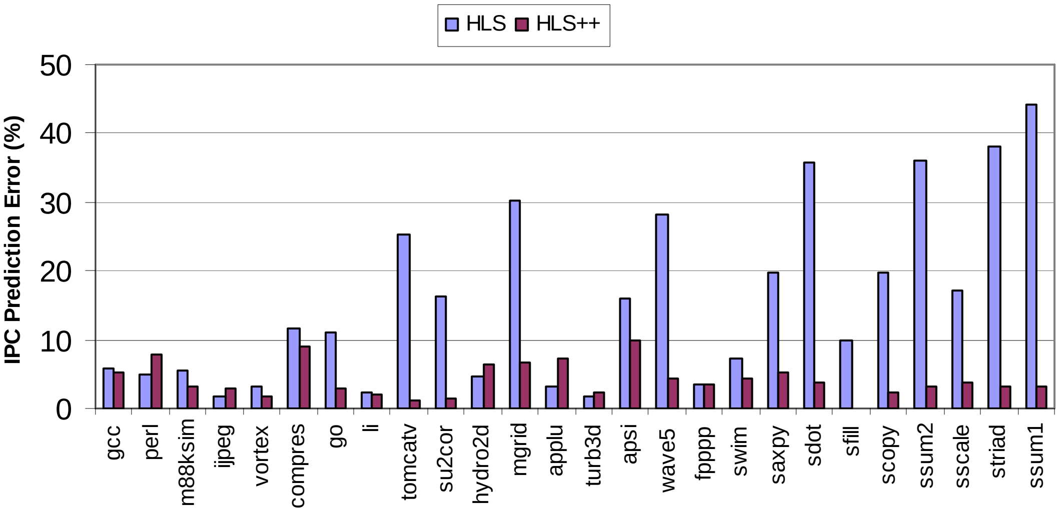


Figure 2.15: HLS versus HLS++ by Benchmark for the SPEC 95

Table 2.4: Benchmark Information for Implementation Cost Analysis

Name	**Number of Basic Blocks**	**Average Block Length**	**Average Ld/St per Block**	**Average Number of Successors**
gcc	2714	12.74	6.07	2.19
perl	575	9.39	4.93	1.82
m88ksim	398	10.90	4.7	1.86
ijpeg	661	13.09	6.03	1.76
vortex	1134	14.38	8.53	1.64
compress	151	8.30	3.4	1.94
go	1732	15.17	5.01	2.26
li	318	8.74	4.42	1.96
tomcatv	258	8.91	3.9	1.9
su2cor	406	9.58	3.84	1.76
hydro2d	646	11.91	3.99	1.81
mgrid	450	12.41	4.74	2.02
applu	552	25.24	8.21	1.87
turb3d	496	12.57	4.92	1.77
apsi	1010	17.94	8.45	1.65
wave5	507	9.89	3.96	1.86
fpppp	452	18.94	8.59	1.77
swim	419	12.44	4.66	1.91
saxpy	177	9.01	3.55	2.12
sdot	109	8.58	3.92	2.3
sfill	177	8.94	3.53	2.12
scopy	177	8.97	3.54	2.12
ssum2	109	8.50	3.89	2.3
sscale	177	8.98	3.54	2.12
striad	177	9.03	3.55	2.12
ssum1	177	9.02	3.55	2.12
Average	**524.4**	**10.7**	**4.71**	**1.89**

Clearly, including detailed stream data is inefficient on average compared to using the other techniques, but future work, including phase identification techniques, can seek to reduce the amount of data being collected.

The same calculations for the SPEC 2000 Alpha benchmarks show that the averages generally increase: NBB is 745.9, LBB is 13.9, NLS is 5.78, and SBB is 1.74. Using the same cost formulae in Table 2.5, the average overall cost increases to 161987 bytes per benchmark, but the error decreases from 29.2% to 4.7%, which is itself an 83.9% drop in error rate.

Table 2.5: Implementation Costs

Technique	**Cost Formula (Bytes)**	**Avg. Cost Per Benchmark (Bytes)**	**% Error Reduction**	**Cost Per % Error Reduction (Bytes)**	**~Storage per Block**
Cumulative Frequencies	NBB x 4	2098	42.7%	115	1 Float
Sequences	NBB x LBB x ½	2806			6 Bytes
Dependences	NBB x LBB x 2 x 1	11222	25.4%	442	22 Bytes
Miss Rates	NBB x 4 x 4	4195	6.5%	645	4 Floats
Branch Predictability	NBB x 4	2098	2.3%	912	1 Float
Stream Info	NBB x NSD x ¼	61701	25.0%	2468	118 Bytes
Basic Block Maps	NBB x SBB x 2 x 4	7929	5.3%	1496	4 Floats
Overall		**92049**	**73.5%**	**1252**	**186 Bytes**

2.6 SUMMARY

In this chapter, the concept of modeling the workload at the granularity of the basic block is presented. The new workload modeling is used to synthesize traces that help improve the accuracy for the SPEC 95 and STREAM Pisa benchmarks from 15.5% error to 4.1% error, and for the SPEC 2000 and STREAM Alpha benchmarks from 27.5% error to 4.7% error. In additional findings, the costs of the improvements in terms of increased storage requirements are quantified to less than 100K bytes and 200K bytes on average per benchmark for the SPEC 95 and SPEC 2000, respectively, to achieve the maximum error reduction. Runtime is approximately twice that of HLS. In addition, a simple regression analysis shows that the SPEC INT 95 workload is easily simulated using simplistic processor models. This result points to a major pitfall for simulator developers: reliance on a small set of benchmarks, datasets and simulated instructions to qualify a simulation system.

In the next chapter, the improved workload modeling becomes the basis for synthesizing high-level codes that are portable to diverse simulator and hardware platforms, making them useful for performance and power model validation. The traces synthesized here for accurate simulation are coupled with novel memory access and branch predictability modeling algorithms to synthesize C-code with inline assembly calls that represent the executing application.

Chapter 3: Automatic Workload Synthesis

This chapter describes a process for automatically creating synthetic workloads for early design studies and performance model validation [7][6][5]. The improved workload modeling from Chapter 2 is used to create a synthetic trace, which is coupled with memory access and branching models to synthesize representative testcases.

A distinction is made between *workload*, *benchmark*, and *testcase synthesis*. *Workload synthesis* is a generic phrase that is used to describe any automatic program synthesis capability. *Benchmark* synthesis usually refers to the synthesis of programs that represent the workload characteristics of applications that are normally used as benchmarks of computer performance. *Testcase* synthesis refers to the synthesis of workloads that represent most but not all of the workload characteristics of applications or benchmarks. Testcases are also called reduced or miniature benchmarks. In this chapter, testcases are synthesized from the SPEC and STREAM benchmarks. The synthetic testcases represent real workloads better than random, hand-coded or kernel testcases.

3.1 INTRODUCTION TO PERFORMANCE MODEL VALIDATION

Chapter 2 motivated and discussed the problem of trace synthesis in the context of improving statistical simulation for design studies. In this chapter, the synthesis process is extended further to the synthesis of code in a high-level programming language. Synthetic codes that are created in a programming language are useful for early design studies as in Chapter 2, but in addition they can be compiled and executed on various platforms, enabling performance model validation. The model validation problem is now described to motivate this next step in the synthesis methodology.

In the later phases of the pre-silicon design process, the validation of a performance model against a functional model or hardware is necessary at various times in order to minimize incorrect design decisions due to inaccurate performance models [13][11]. As functional models are improved, accurate performance models can pinpoint with increasing certainty the effects of particular design changes. This translates into higher confidence in late pre-silicon or second-pass silicon design performance. Once a hardware system exists, validation using internal performance counters is possible, but the process requires trial and error experimentation [73].

Prior validation efforts have focused on microbenchmarks or short tests of random instructions [14][91][11][66][65][64][67][56]. These tests are usually hand-written microbenchmarks that validate the basic processor pipeline latencies, including cycle counts of individual instructions, cache hit and miss latencies, pipeline issue latencies for back-to-back dependent operations, and pipeline bypassing. Black and Shen [11] describe tests of up to 100 randomly generated instructions, not enough to approximate many characteristics of applications. Desikan *et al.* [23] use microbenchmarks to validate an Alpha full system simulator to 2% error, but the validated simulator still gives errors from 18% to 40% on average when executing the SPEC 2000 benchmarks.

Ideally, SPEC and other applications would be used for performance model validation, but this is limited by their long runtimes on RTL simulators [13]. In Singhal *et al.* [84], only one billion simulated cycles *per month* are obtained. In Ludden *et al.* [56], farms of machines provide many cycles in parallel, but individual tests on a 175 million-transistor chip model execute orders of magnitude slower than the hardware emulator speeds of 2500 cycles per second. Sampling techniques such as SimPoint [83], SMARTS [108] and Luo *et al.* [58] can reduce application runtimes, making early design studies feasible, but it is still necessary to execute tens of millions of instructions. Statistical simulation creates representative synthetic traces with less than one million instructions [17][71][27], but traces are not useful for functional model validation.

There have been several efforts to synthesize representative codes. Sakamoto *et al.* combine a modified trace snippet with a memory image for execution on a specific machine and a logic simulator [81], but the method is machine-specific and there is no attempt to reduce the total number of simulated instructions. Intrinsic Checkpointing [79] inserts memory preload instructions into a compiled program to create a binary that represents a SimPoint phase, but there is no attempt to reduce the total number of executed instructions. In Hsieh and Pedram [38], assembly programs are generated that have the same power consumption signature as applications. However, all workload characteristics are modeled as microarchitecture-dependent characteristics, so the work is not useful for studies involving design trade-offs [28]. Wong and Morris [107] investigate synthesis for the LRU hit function to reduce simulation time, but no method of simultaneously incorporating other workload characteristics is developed. The research community recognizes the need for a general way to synthesize useful workloads [86].

In this chapter, the problem of synthesizing flexible workloads for early design studies and performance validation is discussed. An example synthesis system is described that uses the improved workload characterization of statistical simulation in combination with specific memory access and branching models [7][5]. Testcases are synthesized as C-code with low-level instructions instantiated as *asm* statements. When compiled and executed, the synthetic code reproduces the dynamic workload characteristics of an application, and yet it can be easily executed on a variety of performance and functional simulators, emulators, and hardware with significantly reduced runtimes.

The rest of this chapter is organized as follows. Section 3.2 presents properties important to the usefulness of the testcases and some of their benefits. Sections 3.3 and 3.4 give an overview of the synthesis concepts, approach and experimental results. Section 3.5 presents drawbacks and discussion. Section 3.6 presents related synthesis work, and the last section presents a summary.

3.2 SYNTHESIS OF REPRESENTATIVE WORKLOADS

Automatic workload synthesis is most useful if the synthesized workload has the following two properties:

1) The workload reproduces the machine execution characteristics or machine states [43] caused by the application upon which it is based.
2) The workload converges to a result much faster than the original application.

If the first property holds, the workload is said to be *representative* of the original application, at least over some range of instructions, characteristics, or states. Prior work usually focuses on one of the properties at the expense of the other, or on both properties but over a narrow range. The literature is reviewed from this perspective.

The hand-coded tests and automatic random tests in Black and Shen [11] converge quickly (property 2), but they provide limited or inefficient coverage of all the instruction interactions in a real application (property 1). The *reverse-tracer* system [81] achieves accurate absolute performance for a short trace (property 1), but no runtime speedup is obtained. In Hsieh and Pedram [38], both properties are achieved, but workload characteristics that are important to performance, like the instruction sequences and the dependence distances [4], are not maintained. Intrinsic Checkpointing [79] achieves both properties but does not reduce the number of instructions that must be executed to represent a SimPoint phase [83].

In practice, achieving both properties for the representative phases of an entire workload is difficult, but in most cases it is not necessary. For validation purposes, a reduced synthetic benchmark need represent only specific application features of interest, not all features. For early design studies, many prominent workload features must be represented, but absolute accuracy [27] need not be high as long as performance trends from design changes are visible, i.e. *relative* accuracy [27] is high. By synthesizing workloads that can be used for both purposes, the longer running traces or programs used for early design studies and the short microbenchmarks used for model validation into workloads that are in between in length and representativeness are consolidated.

A distinction is made between benchmark representativeness at a high *functional* level and representativeness at a low *execution* level. The most popular synthetic benchmarks were written in a high-level language to be representative of both the static high-level language features and dynamic low-level instruction frequencies of an application [104]. The fact that they were written at the same functional level as the original application had advantages: the code could be ported to multiple platforms, rewritten in different languages, and it would respond to compiler optimizations. None of these attributes, however, is relevant to the main purpose of the synthetic benchmark, which is property 1) above, i.e. to represent the machine response of the original workload. As soon as representative code is ported to another machine or language, or compiled with new compiler technology, even if the static high-level language characteristics are maintained, the code is most likely no longer representative of the low-level execution characteristics of the application undergoing the same transformation. A better outcome would be obtained by first transforming the application, executing it, then writing a new synthetic benchmark to represent the new workload characteristics of the application.

This dissertation proposes that low-level, execution-based representativeness is a more useful focus for the development of synthetic benchmarks. This representativeness can be achieved by synthesizing a testcase from a workload characterization of the low-level dynamic runtime characteristics of a compiled and executing application - the same workload characterization discussed in Chapter 2. The following key observation is made: the statistical flow graph [71][27][4] is a reduced representation of the control flow instructions of the application – a compact representative program. In addition, the synthetic trace from workload characterization converges to an accurate result in a fraction of the time of the original workload [17][71][27][4]. The representative trace is combined with novel algorithms for locality structure synthesis to automatically generate a simple but flexible testcase. The testcase is a C-code envelope surrounding a sequence of *asm* calls that reproduce the low-level behavior of the executing basic blocks. The testcase is easily retargeted for use on machines with similar ISAs. For example, transforming a C program with Alpha *asm* calls to PowerPC *asm* calls is straightforward given the similarity of the instruction sets, assuming the instructions used in the assembly calls are simple and have common operations in both ISAs.

Ideally, the synthetic testcases would be benchmark replacements, but the memory and branching models used to create them introduce errors (Section 3.3), making them a solution in the “middle” between microbenchmarks and applications. Many of the application characteristics are maintained, but there is much room for future work into more accurate models (Section 3.4).

The synthetic traces in statistical simulation have been shown to exhibit representative behavior when executed on program phases [27][4]. Likewise, a testcase to represent an entire program can be created by concatenating together testcases synthesized from each phase. In this work, testcase synthesis on a single phase is demonstrated.

The C-code envelope increases portability to a variety of execution-driven simulators, emulators and hardware. At synthesis-time, user parameters can modify workload characteristics to study predicted trends of future workloads. At runtime, parameters can switch between sections of code, changing the mix of program phases or modeling consolidated programs.

Since synthesis is based on low-level workload statistics, questions related to high-level programming style, language, or library routines that plagued the representativeness of the early hand-coded synthetic benchmarks such as Whetstone [22] and Dhrystone [103] are avoided. Synthesis using statistics rather than actual source also effectively hides the functional meaning of the code and data, and motivates increased code sharing between industry and academia. Many vendors hesitate to share their proprietary applications and data for research. This is particularly true in the database, embedded and systems software areas. Figure 3.1 shows the proposed path to code sharing. The vendor isolates the phases of the workload that are of interest, carries out synthesis on each, and sends the corresponding synthetic testcases to the researcher, who analyzes the workload characteristics of each.

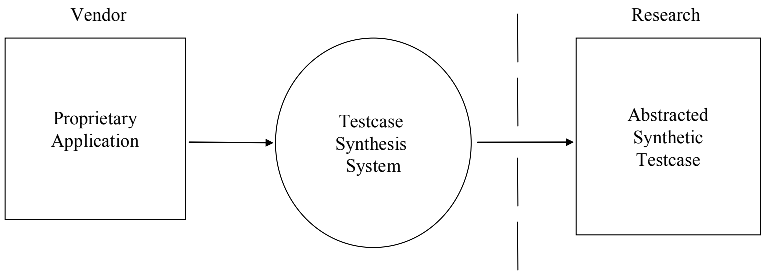


Figure 3.1: Proprietary Code Sharing using Workload Synthesis

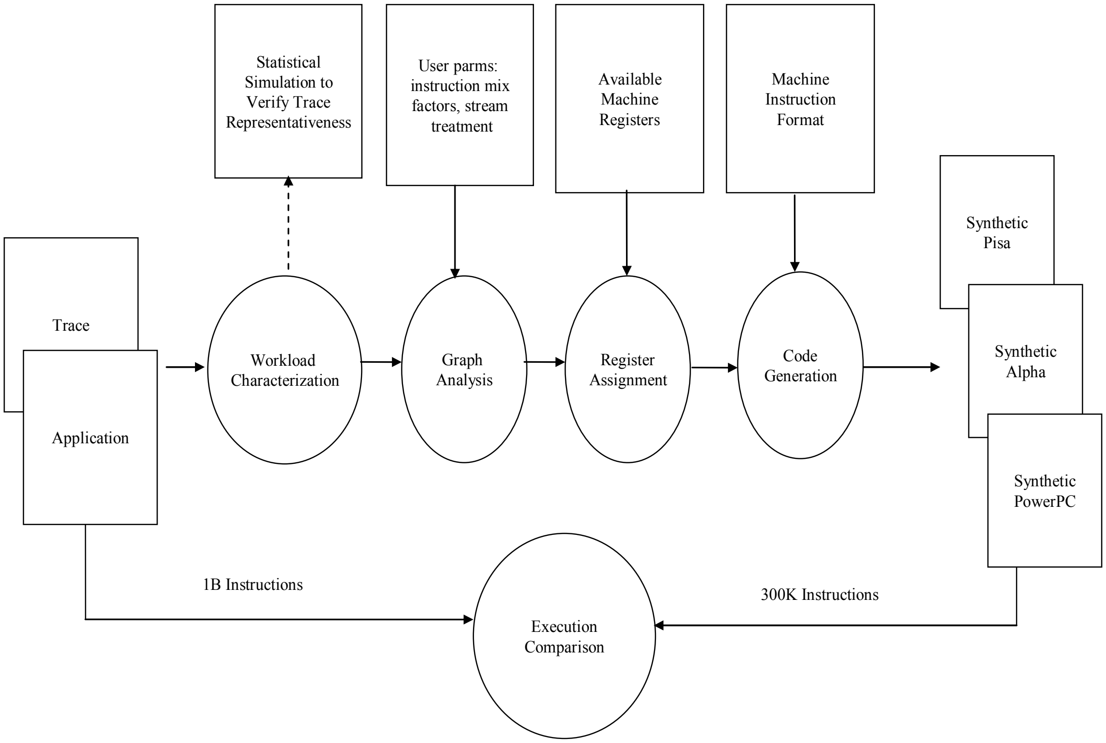
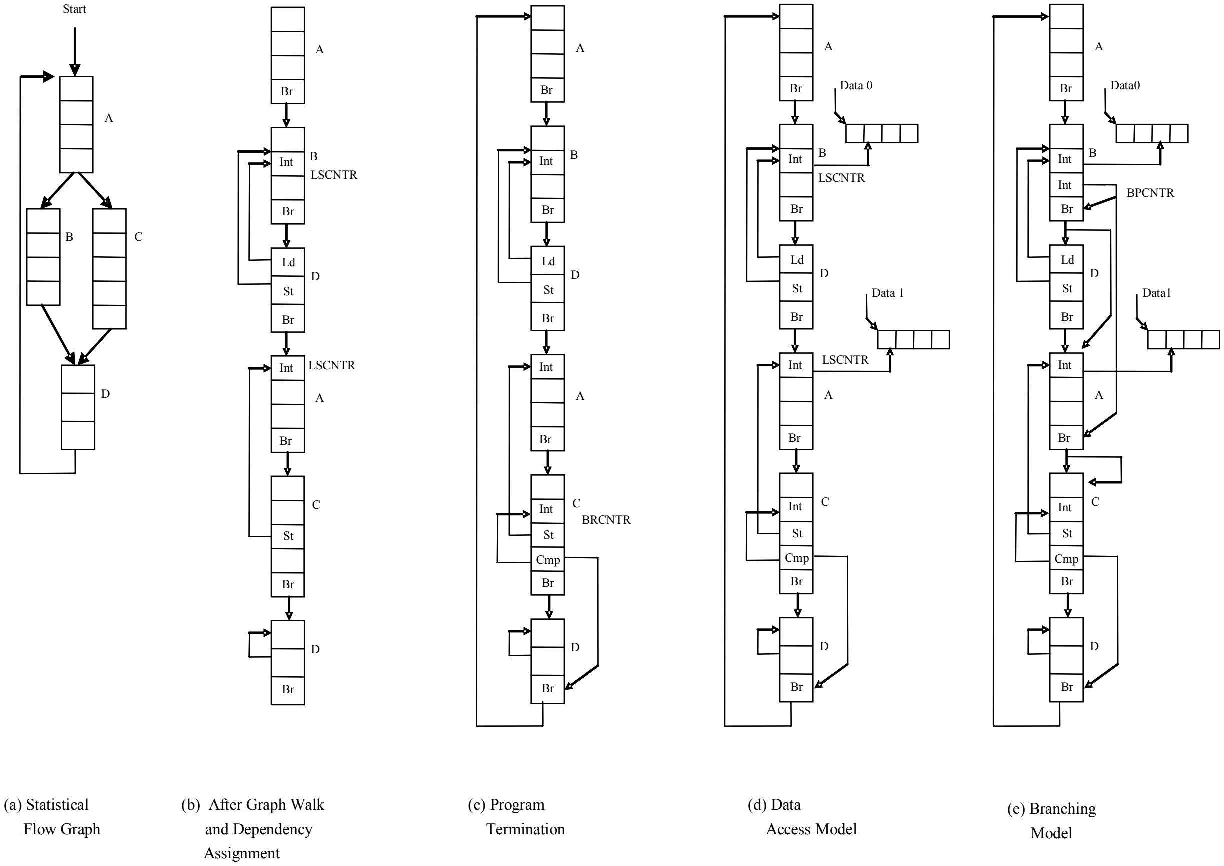

Statistical Simulation to Verify Trace Representativeness
User parms: instruction mix factors, stream treatment
Available Machine Registers
Machine Instruction Format
Trace
Application
Workload Characterization
Graph Analysis
Register Assignment
Code Generation
Synthetic Pisa
Synthetic Alpha
Synthetic PowerPC
1B Instructions
300K Instructions
Execution Comparison

Figure 3.2: Overview of Workload Synthesis Methodology

3.3 SYNTHESIS APPROACH

Figure 3.2 depicts the synthesis process at a high level. There are four major phases: *workload characterization; graph analysis; register assignment* and *code generation*. In this section, the synthesis process for the Pisa and Alpha code targets is presented. The synthesis process for the PowerPC target is described in detail in Chapter 6. Figure 3.3 gives a step-by-step illustration of the process, described below.

At a high level, the statistical flow graph from statistical simulation [27][4] is obtained, which is a reduced representation of the control flow instructions of the application. The graph is traversed, giving a representative synthetic trace. Algorithms are then applied to instantiate low-level instructions, specify branch behaviors and memory accesses, and generate code, yielding a simple but flexible program.



(a) Statistical Flow Graph (b) After Graph Walk and Dependency Assignment (c) Program Termination (d) Data Access Model (e) Branching Model

Figure 3.3: Step-by-Step Illustration of Workload Synthesis

3.3.1 Workload Characterization

The dynamic workload characteristics of the target program are profiled using a functional simulator, cache simulator and branch predictor simulator. This is the same profile used for statistical simulation as described in Chapter 2. The basic blocks, the instruction dependences, the branch predictabilities, and the L1 and L2 I-cache and D-cache miss rates are characterized at the granularity of the basic block. Instructions are abstracted into five classes plus sub-types: integer, floating-point (short or long execution times), load (integer or float), store (integer or float), and branch (on integer or float).

There is no separate input dataset for the synthetic testcases. The input dataset manifests itself in the final workload characteristics obtained from the execution profile. While separate testcases must be synthesized for each possible dataset, the automatic synthesis approach makes that feasible. The IPC and other execution characteristics of the original workload are tracked to compare to the synthetic result. The workload characterization process results in a statistical flow graph [27][4]. An example is given in Figure 3.3(a). Basic blocks A, B, C and D each has various probabilities of branching to one or more basic blocks.

3.3.2 Graph Analysis

The workload characterization from statistical simulation is used to build the pieces of the synthetic benchmark. The statistical flow graph is traversed using the branching probabilities for each basic block, and a linear chain of basic blocks is assembled, as illustrated in Figure 3.3(b). This chain will eventually be emitted directly as the central set of low-level operations in the synthetic benchmark. Figure 3.2 shows that a synthetic trace generated from the graph traversal can be executed in statistical simulation to verify the representativeness of the workload characterization.

An alternative to the linear chain would be to emit a statistical flow graph directly. However, the unsolved problem would then be how to ensure that the branching probabilities from one basic block to its successors would turn out the same as in the original workload. For example, for basic blocks with more than two high-frequency successors, multiple instances of the basic block would have to be synthesized in different parts of the synthetic benchmark. Investigation is needed to determine how to do that and still obtain not only the correct frequency of the basic block over the two instances but also the correct frequency of occurrence of its successors and their successors. By traversing the graph and generating a chain based on the branching probabilities, the probability of occurrence of each block and its successors is correct by construction. This assumes that enough basic blocks can be emitted in the synthetic workload to cover the high-frequency basic blocks and their successors. This turns out to be the case, empirically. For most workloads, the instruction mix is within 5% of the original instruction frequencies (see Table 3.5).

Figure 3.2 also shows that user parameters can influence the graph analysis phase. The parameters currently in use are given in Table 3.3 and are described in the paragraphs below.

Figure 3.4 represents the analysis carried out during a pass through synthesis plus execution of the resulting synthetic benchmark. User parameters adjust factors in the synthesis algorithms to meet particular thresholds or tolerances for the resulting synthetic workload characteristics. Those shown are typical and determined experimentally. A particular parameter is designed to affect only one characteristic, but sometimes characteristics previously analyzed are adversely affected and must be revisited.

Table 3.1 gives a short glossary of the graph analysis terms used in this section and their applicability to Figure 3.3.

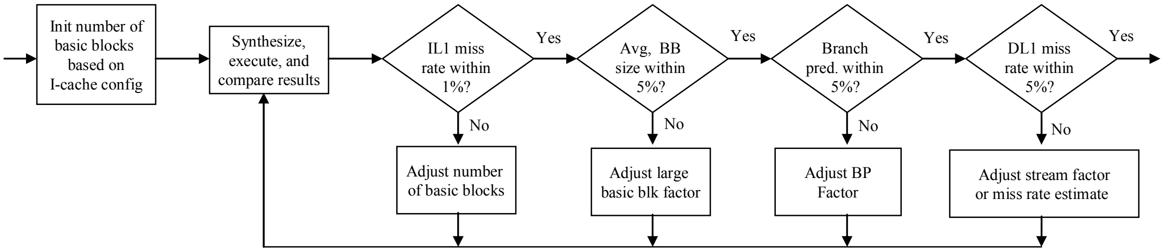

Init number of basic blocks based on I-cache config
Synthesize, execute, and compare results
IL1 miss rate within 1%?
Yes
No
Adjust number of basic blocks
Avg, BB size within 5%?
Yes
No
Adjust large basic blk factor
Branch pred. within 5%?
Yes
No
Adjust BP Factor
DL1 miss rate within 5%?
Yes
No
Adjust stream factor or miss rate estimate

Figure 3.4: Flow Diagram of Graph Analysis Phase of Synthesis

Table 3.1: Glossary of Graph Analysis Terms and Reference to Figure 3.3

Term	Description	Definition	Figure
IMR	I-cache Miss Rate	I-cache miss rate acronym.	3.3 (b)
LSCNTR	Ld-St Memory Access Counter	Integer instruction used to stride through data for memory access instructions.	3.3(b,d)
BRCNTR	Loop Counter	Integer instruction that counts down for the number of iterations to terminate.	3.3(c)
BPCNTR	Branch Prediction Counter	Integer instruction, either 0 or 1, inverts to cause branches to jump or not jump.	3.3(e)
BR	Branch Predictability	Overall branch predictability determined by configuring BPCNTRs.	3.3(e)

3.3.2.1 Instruction Miss Rate and I-cache Model

The number of basic blocks to be instantiated in the synthetic testcase is estimated based on a default I-cache size and configuration (16KB, 32B blocks, direct mapped, 4B instructions), and a workload that continuously rolls through the I-cache is assumed. Since there are eight instructions per cache block, there will be two misses for every eight instructions over 4096 in the workload. An initial estimate of the I-cache miss rate (IMR) is therefore:

$$IMR = (2 \cdot N)/(8 \cdot (4096 + N))$$

where N is the number of instructions over 4096. Solving for N, the initial estimate of the number of instructions above 4096 is:

$$N = (4 \cdot 4096 \cdot IMR)/(1 - 4 \cdot IMR)$$

This assumes the original IMR is less than 0.25. Starting with this estimate, the number of synthetic basic blocks is tuned to match the original IMR. Specific basic blocks are chosen from a traversal of the statistical flow graph, as in [27][4]. Usually a small number of synthesis iterations are necessary to match the IMR because of the effect of the branching model described below. The numbers of basic blocks and instructions synthesized for the Alpha versions of the SPEC 2000 and STREAM benchmarks are shown in Table 3.3, for the Pisa versions of the SPEC 95 and STREAM in Bell and John [5], and for the PowerPC versions in Chapter 6.

Obviously, synthesizing to a default I-cache size as a starting point is less than ideal because it means the I-cache behavior is dependent on the chosen I-cache size. The alternative would be to instantiate a statistical flow graph directly. Assuming the branch successor problems discussed earlier can be solved, the problem then becomes how to emit the flow graph such that the correct I-cache miss rate is obtained, while still getting a runtime speedup versus the original workload.

3.3.2.2 Instruction Dependences and Instruction Compatibility

For each basic block, the instruction input dependences are assigned, Figure 3.3(b). The starting dependence is exactly the dependent instruction chosen as an input during statistical simulation. The issue then becomes operand *compatibility*: if the dependence is not compatible with the input type of the dependent instruction, then another instruction must be chosen. The algorithm is to move forward and backward from the starting dependence through the list of instructions in sequence order until the dependence is compatible. The average number of moves per instruction input is shown in Table 3.3 for the SPEC 2000 and STREAM in column *dependence moves*, and is generally small. In the case of a store or branch that is operating on external data for

Table 3.2: Dependence Compatibilities for Alpha and Pisa Synthetic Testcases

Dependent Instruction	Inputs	Dependence Compatibility	Comment
Integer	0/1	Integer, Load-Integer	
Float	0/1	Float, Load-Float	
Load-Integer/Float	0	Integer	Memory access counter input
Store-Integer	0	Integer, Load-Integer	Data input
Store-Float	0	Float, Load-Float	Data input
Store-Integer/Float	1	Integer	Memory access counter input
Branch-Integer	0/1	Integer, Load-Integer	
Branch-Float	0/1	Float, Load-Float	

which no other instruction in the program is compatible, an additional variable of the correct data type is created.

Table 3.2 shows the compatibility of instructions for the Pisa and Alpha instruction sets. The *Inputs* column gives the assembly instruction inputs that are being tested for compatibility. For loads and stores, the memory access register must be an integer type. When found, it is attributed as a memory access counter (*LSCNTR*) for special processing during the code generation phase.

Table 3.3: Synthetic Testcase Properties

Name	**Number of Basic Blocks**	**Number of Instr-uctions**	**Stream Pools**	**Code Regis-ters**	**BP Factor**	**Stream Factor**	**Miss Rate Est. Factor**	**Loop Iterations**	**Depend-ency Moves**	**Actual Runtime (s)**	**Synthetic Runtime (s)**	**Runtime Ratio**
gcc	850	4585	9	8	1.15	1.07	1.00	51	0.943	6602.85	3.49	**1891.93**
gzip	408	4218	7	10	1.10	1.01	1.00	71	0.188	16695.06	3.88	**4302.85**
crafty	635	4896	9	8	1.15	1.00	1.00	54	0.363	6277.21	3.75	**1673.92**
eon	580	4394	9	8	1.15	1.00	1.00	50	1.209	67064.77	3.15	**21290.40**
gap	268	4193	9	8	1.15	1.00	1.00	62	0.477	5283.60	3.37	**1567.83**
bzip2	311	2515	9	8	1.15	1.00	1.00	109	0.147	10853.61	3.70	**2933.41**
vpr	550	4135	9	8	1.00	1.04	1.00	74	0.977	6470.38	4.32	**1497.77**
mcf	727	4189	9	8	1.05	1.00	1.00	61	0.374	18450.95	3.41	**5410.84**
parser	741	3949	9	8	1.10	1.05	1.00	71	0.567	6459.54	3.95	**1635.33**
perlbmk	606	4263	9	8	1.00	1.00	1.00	61	0.519	22269.29	3.36	**6627.76**
vortex	947	5006	9	8	1.10	1.00	1.00	47	0.466	5919.24	3.39	**1746.09**
twolf	739	4315	9	8	1.04	1.08	1.00	72	0.498	18976.74	4.26	**4454.63**
mgrid	30	3930	7	10	1.20	1.30	0.25	69	9.413	62918.27	3.81	**16513.98**
mesa	619	4292	9	8	1.05	1.00	1.00	64	0.867	56597.98	3.60	**15721.66**
art	450	3762	7	10	0.90	1.50	0.47	73	1.111	89628.10	5.74	**15614.65**
lucas	210	3359	7	10	1.00	1.00	1.00	164	0.691	14697.19	6.28	**2340.32**
ammp	715	4092	13	4	1.00	1.50	0.50	88	0.152	21799.12	9.34	**2333.95**
applu	19	3363	7	10	1.50	1.00	1.10	76	20.293	14149.28	4.00	**3537.32**
apsi	488	4379	7	10	1.00	1.00	0.30	64	7.195	61669.89	3.62	**18693.34**
equake	758	4328	7	8	1.00	1.00	1.00	65	1.468	17989.55	3.59	**5011.02**
galgel	273	3967	7	10	1.00	1.50	0.55	66	0.491	24391.40	4.80	**5081.54**
swim	131	3866	9	8	1.50	1.10	1.00	91	0.579	18347.04	4.97	**3691.56**
sixtrack	621	4173	9	8	1.00	1.02	1.00	84	1.810	21028.38	4.55	**4621.62**
wupwise	176	3656	6	6	1.05	1.03	1.00	224	0.926	18306.39	8.56	**2138.60**
facerec	176	3126	9	8	1.10	0.95	1.05	89	4.622	17156.95	3.79	**4526.90**
fma3d	869	4377	6	8	1.03	1.00	1.00	147	0.149	32235.77	6.59	**4891.62**
saxpy	1	10	2	12	1.00	1.00	1.00	30000	0.000	150.78	3.95	**38.17**
sdot	1	10	2	12	1.00	1.00	1.00	30000	0.000	417.74	3.75	**111.40**
sfill	1	6	1	12	1.00	1.00	1.00	70000	0.333	202.00	3.65	**55.34**
scopy	1	8	2	12	1.00	1.00	1.00	30000	0.000	38.24	3.07	**12.46**
ssum2	1	6	1	12	1.00	1.00	1.00	30000	0.143	53.27	2.26	**23.57**
sscale	1	8	2	12	1.00	1.00	1.00	30000	0.000	38.37	3.10	**12.38**
striad	1	12	3	12	1.00	1.00	1.00	30000	0.000	57.16	4.51	**12.67**
ssum1	1	10	3	12	1.00	1.00	1.00	30000	0.000	91.49	4.24	**21.58**

3.3.2.3 Loop Counters and Program Termination

When all instructions have compatible dependences, a search is made for an additional integer instruction that is attributed as the loop counter (*BRCNTR*) as in Figure 3.3(c). The branch in the last basic block in the program checks the BRCNTR result to determine when the program is complete. The number of executed loops, *loop iterations* in Table 3.3, is chosen to be large enough to assure IPC convergence. Conceptually, this means that the number of loops must be larger than the longest memory access stream pattern of any memory operation among the basic blocks. In practice, the number of loops does not have to be very large to characterize simple stream access patterns. Experiments have shown that the product of the loop iterations and the number of instructions must be around 300K to achieve low branch predictabilities and good stream convergence. The loop iterations are therefore approximately *300K/(number of instructions).* For most workloads this works out to *300K/4000* or 75 iterations. This can be tuned with a user parameter.

For the Alpha instruction target, an additional integer instruction is converted to a *cmple* to compare the BRCNTR to zero for the final branch test.

3.3.2.4 Memory Access Model

The LSCNTR instructions are assigned a stride based on the D-cache hit rate found for their corresponding load and store accesses during workload characterization, Figure 3.3(d). For the Pisa and Alpha targets, the memory accesses for data are modeled using the 16 simple stream access classes shown in Table 3.4.

The stride for a memory access is determined first by matching the L1 hit rate of the load or store fed by the LSCNTR, after which the L2 hit rate for the stream is predetermined. If the L1 hit rate is below 12.5%, the L2 hit rate is matched. The table was generated based on an idealized direct-mapped cache with an L1 line size of 32

Table 3.4: L1 and L2 Hit Rates as a Function of Stride (in 4B increments)

L1 Hit Rate	**L2 Hit Rate**	**Stride**
0.0000	0.000	16
0.0000	0.0625	15
0.0000	0.1250	14
0.0000	0.1875	13
0.0000	0.2500	12
0.0000	0.3125	11
0.0000	0.3750	10
0.0000	0.4375	9
0.0000	0.5000	8
0.1250	0.5000	7
0.2500	0.5000	6
0.3750	0.5000	5
0.5000	0.5000	4
0.6250	0.5000	3
0.7500	0.5000	2
0.8750	0.5000	1
1.0000	N/A	0

bytes, an L2 line size of 64 bytes, and 4 byte load accessess, and the corresponding stride is shown in 4 byte increments. An access will miss in the L1 every *stride/8* accesses if the stride is less than 8, while every other access to the L2 is a miss. Similarly, an access will miss in the L2 every *stride/16* accesses if the stride is greater than 8, and every access will miss in the L1. Therefore Table 3.4 is easily formulated:

$$L1_{HitRate} = \begin{cases} 1-(4\cdot stride/32), & stride < 8 \\ 0, & stride \geq 8 \end{cases}$$

$$L2_{HitRate} = \begin{cases} 1/2, & stride < 8 \\ 1-(4\cdot stride/64), & stride \geq 8 \end{cases}$$

By treating all memory accesses as streams and working from a base cache configuration, the memory access model is kept simple. This reduces the impact on the testcase instruction sequences and dependences, which have been shown to be critical for correlation with the original workload [4]. On the other hand, there can be a large error in stream behavior when an actual stream hit rate falls between the hit rates in two rows, and the simple model is responsible for correlation error when the cache hierarchy changes (see Section 3.5). Also, the table is idealized for loads; store writeback traffic is not considered. More complicated models might traverse cache congruence classes or pages (to model TLB misses), or move, add, or convert instructions to implement specific access functions. Adding a few instructions to implement a more complicated model will not impact most characteristics. There are many high-level models in the literature that can be investigated for possible implementation in the synthetics, for example [94][20] [52][88][24][31][61][75].

In some cases, additional manipulation of the streams was necessary in order to correlate the testcases because of the cumulative errors in stream selection. In Table 3.3, the *stream factor* multiplies the L1 hit rate taken from the table during each lookup, and if the result is greater than the original hit rate, the selected stream is chosen from the preceding row. This has the effect of reducing overall hit rates for the first load or store fed by an LSCNTR.

Because the dependence analysis may cause several memory access operations to use the same LSCNTR, the overall access rate at the granularity of the basic block may be significantly in error. During synthesis, the overall miss rate for the basic block is estimated as the number of LSCNTRs feeding the block divided by the total number of loads and stores in the block. The *miss rate estimate factor* in Table 3.3 multiplies the calculated miss rate estimate and causes the selected table row for the LSCNTR to change accordingly. Smaller factors increase the basic block miss rate while larger factors decrease it. Usually a small number of synthesis iterations are needed to find a combination of factors to model the overall access rates of the application.

3.3.2.5 Branch Predictability Model

A branch predictability model is superimposed onto the set of basic blocks that already represent the instruction mix, dependences and data access patterns of the original workload, Figure 3.3(e). To model branch predictability, the number of branches that will have taken-targets based on the global branch predictability, BR, of the original application (assumed greater than 50%) is calculated. An integer instruction (attributed as the BPCNTR) that is not used as a memory access counter or a loop counter is converted into an invert instruction operating on a particular register every time it is encountered. If the register is set, the branch jumps past the next basic block in the default loop. The *invert* mechanism causes a branch to have a predictability of 50% for 2-bit saturating counter predictors. The target *BR* is:

$$BR = (F \cdot N + (1 - F) \cdot N \cdot (0.5)) / N$$

where *(1 - F)* is the fraction of branches in the synthetic benchmark that are configured to use the invert mechanism, and *N* is the total number of synthesized branches. Solving for *(1 - F)*, the fraction of branches that must be configured is *2*(1 – BR)*. A uniform random variable over this fraction is used to pick which branches are configured.

The fraction *BR* is sometimes not sufficient to model the branch predictability because of variabilities in the mix of dynamic basic blocks used and the code size. To compensate, the *BP Factor* in Table 3.3 multiplies *BR* to increase or decrease the number of configured branches. Usually a small number of synthesis iterations are needed to tune this factor.

In an additional implementation, a branch jumps past a user-defined number of basic blocks instead of just one, but this did not result in improved branch predictability. In another implementation, a branch jumps past a user-defined number of instructions in the next basic block. This also did not improve predictability except for *mgrid* and *applu*, which have large average basic block sizes such that jumping past an entire basic block significantly changes the instruction mix. In those cases, the branch jumps past ten instructions of the next basic block.

During synthesis experiments, it was noticed that benchmarks with large average basic block sizes and therefore small numbers of basic blocks in the final synthetic code are prone to have a skewed basic block mix that favors shorter basic blocks. For *mgrid* and *applu*, during basic block selection, if a uniform random variable is greater than an additional factor, set to 0.5 and 0.9, respectively, then the successors of the previous block that are on average longer than 50 instructions are checked first to be included. When configuring branches, the BRCNTR, *cmple* (for Alpha) and BPCNTR instructions must not be skipped over by a taken branch, or loop iterations may not converge or the branch predictability may be thrown off. Code regions containing these attributed instructions are carefully avoided.

In practice, there are many synthetic benchmarks that more or less satisfy the metrics obtained from the workload characterization and overall application IPC. As mentioned above, the usual course of action is to iterate through synthesis a number of times until the metric deltas are as small as desired [6]. Sometimes the synthesis tolerances as in Table 3.4 cannot be satisfied using the factors, in which case the best-case synthesis result with respect to overall performance is retained [8]. Often this results in errors in the workload characteristics that cancel each other out, as described for particular benchmarks in Section 3.4. Usually a small number of synthesis iterations are needed to obtain reasonably small errors in characteristics and performance.

3.3.3 Register Assignment

All architected register usages in the synthetic testcase are assigned exactly during the register assignment phase. Most ISAs specify dedicated registers that should not be modified without saving and restoring. In practice, not all registers need to be used to achieve a good synthesis result. Various experiments showed that usually only 20 general-purpose registers divided between memory access stream counters and code use are necessary. For the benchmarks under study, the number of registers available for streams averages about 8 and for code use about 9 (*stream pools* and *code registers* in Table 3.3). Three additional registers are reserved for the BRCNTR, *cmple* (in Alpha), and BPCNTR functions.

Data access streams are pooled according to their stream access characteristics and a register is reserved for each class (*stream pools* in Table 3.3). All LSCNTRs in the same pool increment the same register. For applications with large numbers of stream pools, synthesis consolidates the least frequent pools together (using the most frequent LSCNTR stride among them) until the total number of registers is under the limit. A roughly even split between code registers and pool registers improves benchmark *quality*. High quality is defined as a high correspondence between the instructions in the compiled benchmark and the original synthetic C-code instructions. With too few or too many registers available for code use, the compiler may insert stack operations into the binary. The machine characteristics may not suffer from a few stack operations, but for this study the synthetic code is created without them. The available code registers are assigned to instruction outputs in a round-robin fashion.

3.3.4 Code Generation

The code generator of Figure 3.2 takes the representative instructions, the instruction attributes from graph analysis, and the register assignments and outputs a single module of C-code that contains calls to assembly-language instructions in the Pisa or Alpha languages. Each instruction in the representative trace maps one-to-one to a single volatile *asm* call in the C-code. The steps are detailed in the following paragraphs.

First, the C-code *main* header is emitted. Then variable declarations are emitted to link output registers to memory access variables for the stream pools, the loop counter variable (BRCNTR), the branching variable (BPCNTR), and the *cmple* variable (in Alpha). Pointers to the correct memory type for each stream pool are declared, and *malloc* calls for the stream data are generated with size based on the number of loop iterations. Each stream pool register is initialized to point to the head of its *malloced* data structure.

The loop counter register (BRCNTR) is initialized to the number of times the instructions will be executed, and the assignment is emitted. The instructions associated with the original flow graph traversal are then emitted as volatile calls to assembly language instructions. Each call is given an associated unique label. The data access counters (LSCNTRs) are emitted as *addiu* or *addl* instructions (for Pisa or Alpha, respectively) that add their associated stride to the current register value. The BRCNTR is emitted as an *add* of minus one to its register. Long latency floating-point operations are generated using *mul.s* or *muls* and short latency operations are generated using *add.s* or *adds.*. Loads use *lw*, *lwz,* or *l.s, lds,* depending on the type, and similarly for stores. Branches use the *beq* type, and can have either integer or float operands.

The basic blocks are analyzed and code is generated to print out any unconnected output registers depending on a switch value. The switch is never set, but the print statements guarantee that no code is eliminated during compilation. This mechanism, plus the use of *volatile* assembly calls, guarantees that no performance-sensitive code will be eliminated during compilation. Furthermore, the experiments show that none of the code is reordered by the *gcc* compiler using these mechanisms.

In a final synthesis step, code to free the *malloced* memory is emitted, and finally a C-code footer is emitted. Table 3.3 gives the synthesis information for the SPEC 2000 and STREAM Alpha codes as described in this section. The *runtime ratio* is the user runtime of the original benchmark for one billion instructions (1M for STREAM) divided by the user runtime of the synthetic testcase on various POWER3 (400MHz) and POWER4 (1.2 GHz) workstations. Variations in runtime reflect network traffic during the runs. Each pass through the synthesis process takes about three minutes on an IBM p270 (400 MHz). An average of about ten passes plus think-time were necessary to tune the synthesis parameters for each testcase.

3.4 EVALUATION OF SYNTHETIC TESTCASE PERFORMANCE

In this section, the performance results for the SPEC 2000 Alpha synthetic testcases are presented. The results for the SPEC 95 Pisa testcases are found in Bell and John [5], and synthesis and analysis for the SPEC 2000 PowerPC testcases can be found in Chapter 6. Aggregate performance metrics are examined, including workload characteristics as well as the machine responses to the synthetics.

3.4.1 Methodology

The system modified from HLS [71][72] as described in Section 3.2 is used. SimpleScalar Release 3.0 [16] was downloaded and *sim-cache* was modified to carry out the workload characterization. The SPEC 2000 Alpha binaries were executed in *sim-outorder* on the first reference dataset for the first one billion instructions, corresponding to a single program phase [4]. In addition, single-precision versions of the STREAM and STREAM2 benchmarks [64] with a ten million-loop limit were compiled on an Alpha machine. The default SimpleScalar configuration in Table 2.1 is used in order to compare results to the simulations of Section 3.2 and the results of Oskin *et al.* [71]. While the machine configuration is relatively small, in the experiments below and in Section 3.6, the window size and other machine parameters are varied significantly and good correlations are obtained. It is also a useful configuration for the study of smaller embedded, DSP or ASIC designs.

The code generator was built into HLS, and C-code was produced using the synthesis methods of Section 3.3. The synthetic testcases were compiled on an Alpha machine using *gcc* with optimization level *–O2* and executed to completion in SimpleScalar.

3.4.2 Evaluation of Synthetic Workload Characteristics

The following figures show results for both the original applications, *actual*, and the synthetic testcases, *synthetic*. Figure 3.5 shows the IPC for the benchmarks. The average error for the synthetic benchmarks is 2.4%, with a maximum error of 8.0% for *facerec*. The reasons for the errors are discussed in the context of the figures below and Table 3.5, which summarizes the average percent error and maximum error for each workload characteristic.

Figure 3.6 compares the average instruction percentages over all benchmarks for each class of instructions. The average prediction error for the synthetic testcases is 3.4% with a maximum of 7.7% for branches. Figure 3.7 shows that the basic block size varies per benchmark with an average error of 7.2% and a maximum of 21.1% for *mgrid*. The

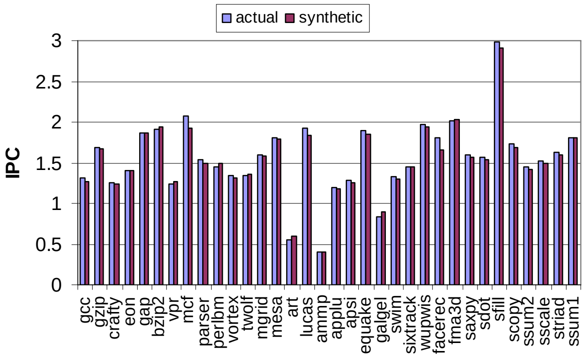


Figure 3.5: Actual vs. Synthetic IPC

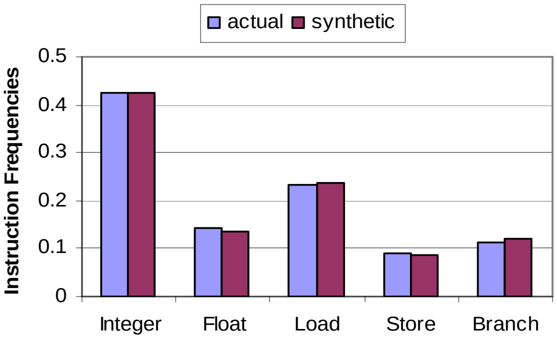


Figure 3.6: Instruction Frequencies

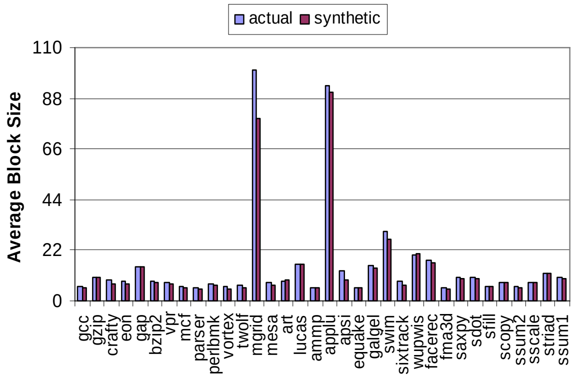


Figure 3.7: Basic Block Sizes

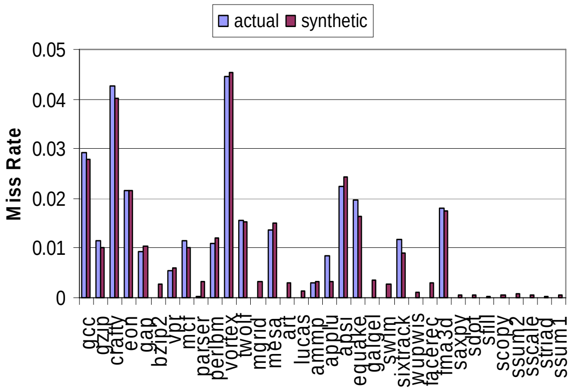


Figure 3.8: I-cache Miss Rates

errors are caused by variations in the fractions of specific basic block types in the synthetic benchmark with respect to the original workload, which is a direct consequence of selecting a limited number of basic blocks during synthesis. For example, *mgrid* is synthesized with a total of 30 basic blocks made up of only six different unique block types. *Applu* is synthesized with 19 basic blocks but 18 unique block types.

The I-cache miss rates are shown in Figure 3.8. They show an error of 8.6% for benchmarks with IMRs above 1%, with a maximum of 22.9% for *sixtrack*. The number of synthetic instructions, however, is within 2.8% of the expected number given the I-cache configuration calculation in Section 3.3. The errors are due to the process of choosing a small number of basic blocks with specific block sizes and implementing the

Table 3.5: Percent Error by Metric, Synthetics versus Applications

Metric	Avg. %Error	Max. %Error
IPC	2.4	8.0 (facerec)
Instruction Frequencies	3.4	7.7 (branches)
Dependence Distances	13.3	41.9 (integers)
Dispatch Occupancies	4.6	11.7 (floats)
Basic Block Sizes	7.2	21.1 (mgrid)
L1 I-cache Miss Rate (>1%)	8.6	22.9 (sixtrack)
L1 D-cache Miss Rate (>1%)	12.3	55.7 (mgrid)
L2 Cache Miss Rate (>15%)	18.4	61.2 (applu)
Branch Predictability	1.5	6.4 (art)

branching model. For miss rates close to zero, a number of instructions less than 4096 is used, up to the number needed to give an appropriate instruction mix for the testcase. For the STREAM loops, only one basic block is needed to meet both the *IMR* and the instruction mix requirements. For the synthetic testcases, there appears to be a small but non-zero IMR, versus an essentially zero miss rate for some of the applications. This is because the synthetic testcases are only executed for about 300K instructions, far fewer than necessary to achieve a very small I-cache miss rate. However, since the miss rates are small, their impact on IPC when coupled with the miss penalty is also small.

The average branch predictability error is 1.9%, shown in Figure 3.9. The largest error is shown for *art* at 6.4%, and *mgrid* has the third largest error at 4.9%. The L1 data cache miss rates are shown in Figure 3.10. For miss rates greater than 1%, the error is 12.3%. For these miss rates, the trends using the synthetic testcases clearly correspond with those of the original workloads. Again, there is more variation for smaller miss rates, but again the execution impact is also small.

In Figure 3.11, the unified L2 miss rates are shown. The large errors due to the simple streaming memory access model are often mitigated by small L1 miss rates. A good example is *gcc*, which has only a 2.6% L1 miss rate, and even the small L2 miss

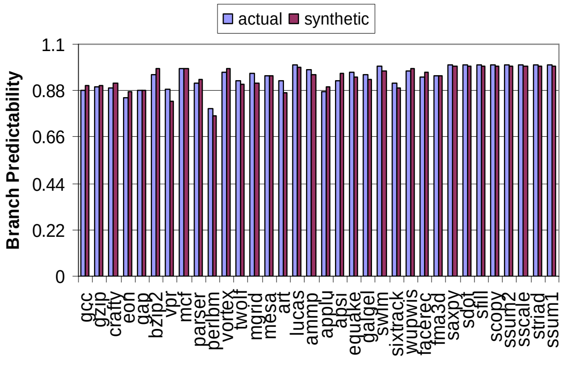


Figure 3.9: Branch Predictability

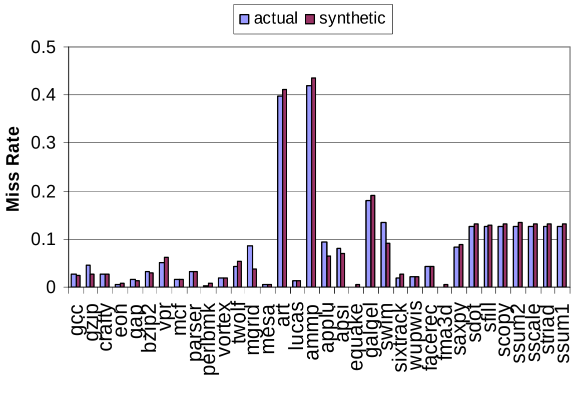


Figure 3.10: L1 D-cache Miss Rates

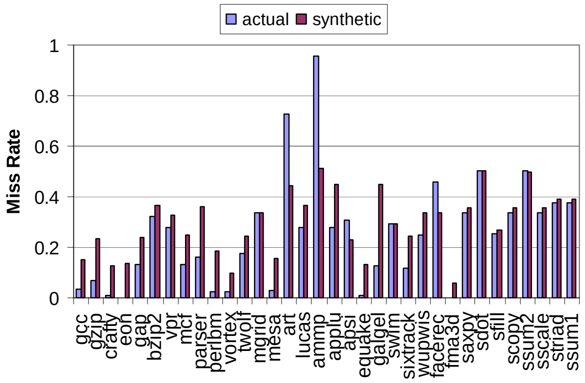


Figure 3.11: L2 Cache Miss Rates

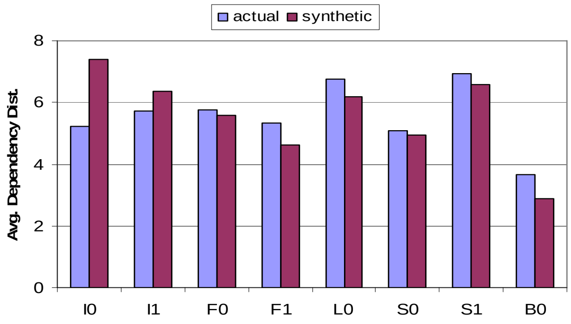


Figure 3.12: Average Dependence Distances per Instruction Type and Operand

rate will not impact IPC significantly. Even though *art* and *ammp* have large L1 miss rates, the smaller L2 miss rates are offset by relatively larger I-cache miss rates and smaller branch predictabilities. The main cause of these errors is the fact that the current memory access model focuses on matching the L1 hit rate, and the L2 hit rate is simply predetermined as a consequence. The large error for *ammp* is partially explained by the fact that the small data-footprint synthetic testcases have data-TLB miss rates near zero, while the actual *ammp* benchmark has a data-TLB miss rate closer to 13%. As a consequence, the synthetic version does not correlate well when the dispatch window is increased and tends to be optimistic.

Figure 3.12 shows the average dependence distances, with 13.3% error on average. The largest components of error are the integer dependences, caused by the conversion of many integer instructions to LSCNTRs, the memory access stride counters. A stride counter overrides the original function of the integer instruction and causes dependence relationships to change. Another source of error is the movement of dependences during the search for compatible dependences in the synthesis process. The movement is usually less than one position (Table 3.3), but *mgrid* and *applu*, the

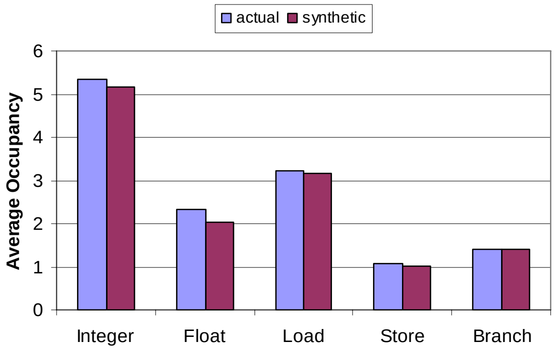


Figure 3.13: Dispatch Window Occupancies per Instruction Type

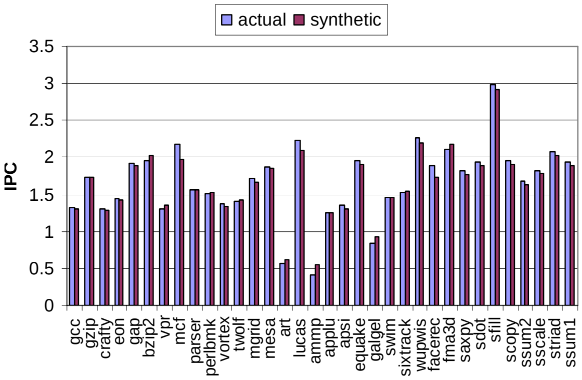


Figure 3.14: Dispatch Window Size 32

benchmarks with the largest average block sizes at 100.07 and 93.42, respectively, show significant movement. The branching model also contributes errors.

Despite the dependence distance errors, Figure 3.13 shows that the average dispatch window occupancies are similar to those of the original benchmarks with an average error of 4.6%.

3.4.3 Evaluation of Design Changes

Design changes using the same synthetic testcases are studied; that is, the same testcases used in the last section are re-executed after changing the machine parameters in SimpleScalar. Table 3.6 shows average results over all benchmarks for several design studies using Pisa and Alpha benchmarks. Several of these are investigated in more depth below with the help of figures that display results for all benchmarks. Some of the studies show significant change in IPC when the design change is applied; this issue is examined more carefully in Chapter 5.

Figures 3.14 and 3.15 show the absolute IPCs using a dispatch window of 32 and 64 and no change in LSQ with average errors of 3.0% and 3.1%, respectively. These numbers do not include *ammp*; as explained in the last section, *ammp* tends to be optimistic when the dispatch window changes because the small data footprint testcases

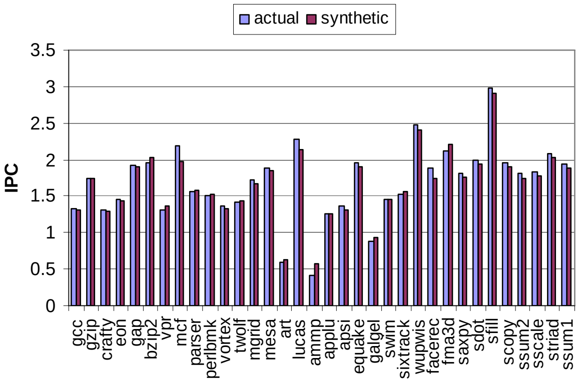


Figure 3.15: Dispatch Window Size 64

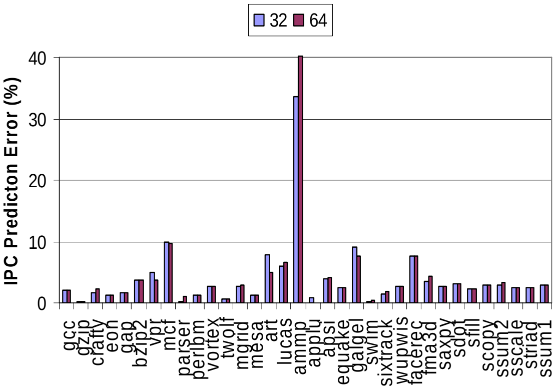


Figure 3.16: IPC Error per Dispatch Window

do not model data-TLB misses. Figure 3.16 graphs the IPC prediction errors for the dispatch windows. Most errors, except for *ammp,* are below 5%.

Figures 3.17 and 3.18 show the absolute change in IPC, *delta IPC*, as the same benchmarks and testcases are executed first with the default configuration (dispatch window of 16) and then with the dispatch window sizes changed to 32 and 64 respectively. The average relative errors are 1.3% and 1.5%, respectively. The graphs show that, when an application change is large with respect to the changes in the other applications, the synthetic testcase change is also large relative to the change in the other synthetic testcases. These IPC changes would be large enough to trigger additional studies using a detailed cycle-accurate simulator, including an analysis of *ammp*. Chip

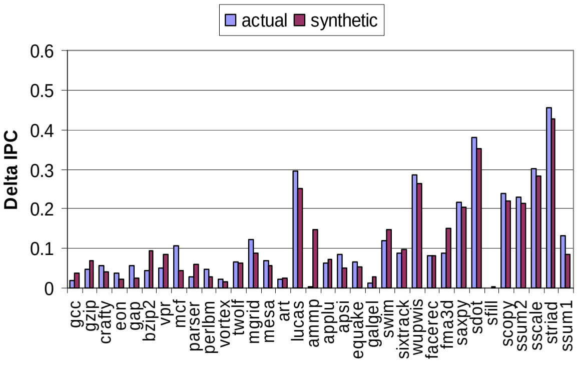


Figure 3.17: Delta IPC as Disp. Window Increases from 16 to 32

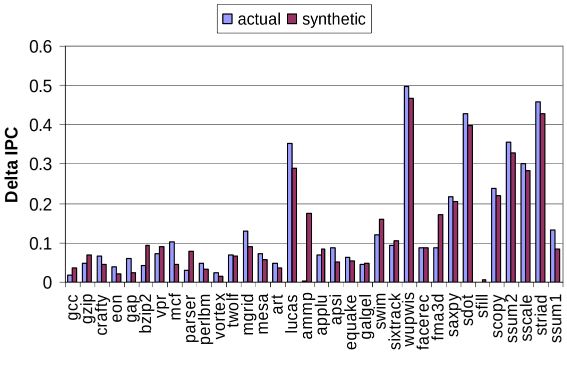


Figure 3.18: Delta IPC as Disp. Window Increases from 16 to 64

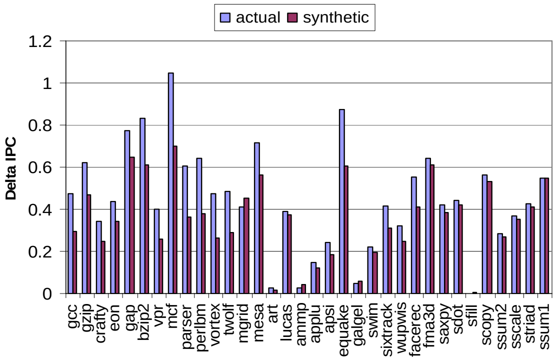


Figure 3.19: Delta IPC as L1 Data Latency Increases from 1 to 8

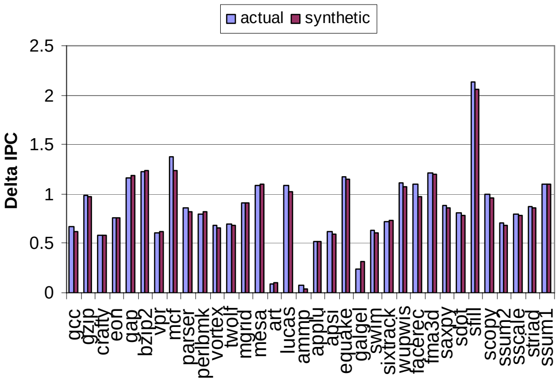


Figure 3.20: Delta IPC as Issue Width Increases from 1 to 4

designers are looking for cases in a large design space in which a design change may improve or worsen a design. In the case of the dispatch window studies, the results would trigger further cycle-accurate studies of *lucas, ammp, swim, wupwise, fma3d* and the STREAM benchmarks. Alternatively, the designers might be curious why the change did not help the SPEC INT testcases.

Figure 3.19 shows the delta IPC as the L1 D-cache latency is increased from 1 to 8. The average absolute IPC error is 9.5% and the relative error is 9.7%. The errors are high, but the larger changes in the actual benchmarks over 1B instructions are reflected in the synthetic testcases that run in seconds. However, it is apparent that the memory access model is less accurate for SPEC INT than for the other testcases. In fact, the average relative error for SPEC INT is 19.9% versus 4.2% for the others.

Figure 3.20 shows better results for the delta IPC as the issue width increases from 1 to 4. The average absolute error is 1.9%, and the relative error is 2.4%. Similar results for commit width changes, doubling the L1 D-cache (to 256 sets, 64B cache line, 8-way set associative), and doubling the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set associativity) are shown in Table 3.6. Results are reproduced from the technical report [5] for the SPEC 95 testcases synthesized the same way but targeting the Pisa ISA.

Table 3.7 lists results for the SPEC 2000 programs and gives results for additional design change studies to facilitate comparative analysis. A change in machine width implies that the decode width, issue width and commit width all change by the same amount from the base configuration in Table 3.5. When the caches are increased or decreased by a factor, the number of sets for the L1 I-cache, D-cache and L2 cache are increased or decreased by that factor. Likewise, when the bimodal branch predictor is multiplied by a factor, the table size is multiplied by that factor from the default size. As before, the *L1 D-cache 2x* and *L1 I-cache 2x* specify a doubling of the L1 D-cache (to 256 sets, 64B cache line, 8-way set associativity), and a doubling of the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set associativity). Also as before, the numbers here do not include *ammp*; as explained, *ammp* tends to be optimistic when the dispatch window changes because the small data footprint testcases do not model data-TLB misses.

Table 3.6: Average Synthetic IPC Error and Relative Error by Benchmark Suite

Design Change	**SPEC 95 (Perfect Branching)**		**SPEC 95 (Branching Model)**		**SPEC 2000 (Branching Model)**	
	%Error	**%Rel.Err.**	**%Error**	**%Rel Err.**	**%Error**	**%Rel Err.**
Dispatch Window 16, LSQ 8	3.9	n/a	2.4	n/a	2.4	n/a
Dispatch Window 32, LSQ 8	3.2	2.1	3.1	2.2	3.1	1.3
Dispatch Window 64, LSQ 8	3.1	2.7	3.3	2.4	3.0	1.5
Issue Width 1 (other widths 4)	2.6	4.1	2.1	2.2	1.9	2.3
Commit Width 1 (other widths 4)	2.8	4.2	3.2	3.9	2.8	2.7
Commit Width 8 (other widths 4)	3.7	1.1	2.6	1.4	2.4	0.2
L1 I-cache 2x (1024:64:2)	9.8	8.2	8.7	7.5	3.0	1.3
L1 D-cache 2x (256:64:8)	4.3	2.1	3.2	2.4	3.1	1.0
L1 D-cache Latency 8	8.9	6.9	11.1	10.4	9.5	9.7

Table 3.7: Percent IPC Error and Relative Error by Design Change

Design Change	Avg. %Error	Avg. %Rel. Err.
Dispatch Window 8, LSQ 4	2.8	2.4
Dispatch Window 32, LSQ 16	3.7	2.1
Dispatch Window 48, LSQ 24	4.9	3.8
Dispatch Window 64, LSQ 32	6.1	5.1
Dispatch Window 96, LSQ 48	8.3	7.5
Dispatch Window 128, LSQ 64	9.0	8.3
Machine Width 2	2.7	1.6
Machine Width 6	2.6	1.1
Machine Width 8	2.6	1.1
Machine Width 10	2.6	1.1
Issue Width 1	1.9	2.3
Issue Width 8	2.7	1.0
Commit Width 1	2.8	2.1
Commit Width 8	2.4	0.2
Instruction Fetch Queue 8	2.6	0.5
Instruction Fetch Queue 16	2.7	0.8
Instruction Fetch Queue 32	3.0	1.1
Caches 0.25x	20.1	19.4
Caches 0.5x	24.8	23.9
Caches 2x	4.1	3.3
Caches 4x	4.7	3.8
L1 I-cache 2x	3.0	1.3
L1 D-cache 2x	3.1	1.0
L1 D-cache Latency 8	9.5	9.7
BP Table 0.25x	2.5	1.1
BP Table 0.5x	2.3	0.3
BP Table 2x	2.3	0.3
BP Table 4x	2.3	0.4

Looking at the *Dispatch Window* rows with the LSQ size now changing, when the synthetics are executed on configurations close to the default configuration, the average IPC prediction error and average relative errors are below 5%. However, as the configuration becomes less similar to the configuration used to synthesize the benchmarks, the errors increase.

One conclusion is that the synthetics are most useful for design studies and validations closer to the synthesis configuration, and that testcases should be resynthesized when the configuration strays farther away - in this case, when the dispatch window rises to four times the default size. This suggests a *stair-step synthesis* approach for early design studies using the synthetics: as the design space is explored and a configuration choice changes by a multiple of two or four times its starting value, resynthesize from executions based on the new configuration and proceed with the exploration. This concept is left for future investigation.

The *Machine Width* results differ from the dispatch window results in that the errors are small regardless of the width change. For the dispatch studies, the absolute change in IPC from the default configuration for both synthetics and applications is greater than 17% for each case (for a dispatch window of 128 the change is over 56%). Likewise, when the width is reduced to 2, the absolute change in IPC is over 23%, which indicates that the low average prediction error and relative error are meaningful. But when the width increases to 6, 8 and 10, the change is never more than 2.3%, which is on the order of the IPC prediction errors of the synthetics versus the applications. However, the fetch queue size did not change from the default and it supplies too little ILP to stress the wider machine width. These configurations therefore cannot test the accuracy of the synthetics.

Similarly, the absolute IPC change from the default IPC for the *Issue Width 8* and *Commit Width 8* rows never gets greater than 1.4%, and likewise for the instruction fetch queue and branch predictability rows, it never gets greater than 1.6%. Simply changing the IFQ size, issue, or commit width without addressing the other pipeline bottlenecks does not improve performance, as expected. Other machine configurations may be needed to stress the branch predictor models.

The remaining studies yield changes in IPC significantly greater than the error of the synthetics versus the applications, except for the *Caches 0.25x* and *Caches 0.5x* studies. The synthetics underestimate performance when the cache is significantly reduced due to capacity misses among the synthetic data access streams.

3.4.4 TPC-C Study

In an additional study, a code generation target for the PowerPC ISA is implemented, and an internal IBM instruction trace for TPC-C (described in [12]) is fed through the synthesis system using the methods described in this chapter. The resulting testcase is then compiled and executed on a detailed performance simulator for the IBM POWER4 processor [93] and compared to results for the original trace that had a runtime more two orders of magnitude longer. A 6.4% IPC prediction error is obtained, demonstrating the flexibility and retargetability of the synthesis approach. Chapter 6 improves synthesis for the TPC-C testcase and other PowerPC testcases and describes advanced synthesis techniques for the POWER5 processor.

3.5 DRAWBACKS AND DISCUSSION

The main drawback of the approach is that the microarchitecture independent workload characteristics, and thus the synthetic workload characteristics, are dependent on the particular compiler technology used. It is argued that the use of low-level, post-compiler characteristics is necessary for representativeness. Since a synthesis pass is automatic, resynthesis based on workload characterization after the use of new compiler technology is simplified. It also avoids questions of high-level programming style, language, or library routines that plagued the representativeness of the early hand-coded synthetic benchmarks such as Whetstone [22] and Dhrystone [103].

One objection is that the synthetics are comprised of machine-specific assembly calls. However, use of low-level operations are a simple way to achieve true representativeness in a much shorter-running benchmark, and the *asm* calls are easily transformed to assembly instructions in other ISAs, assuming simple operations common to all ISAs are used for the synthetic instructions.

Another drawback is that only features specifically modeled among the workload characteristics appear in the synthetic benchmark. This will be addressed over time as researchers uncover additional features needed to correlate with execution-driven simulation or hardware [47], although the present state-of-the-art is quite good [27][4]. For certain experiments, researchers may want some workload characteristics be included and others not included. User parameters at synthesis time can specify which workload characteristics should or should not appear in the synthetics as necessary.

One consequence of the present method is that dataset information is assimilated into the final instruction sequence of the synthetic benchmark. For applications with multiple datasets, a family of synthetic benchmarks must be created. The automatic process makes doing so possible, but future research could seek to find the workload features related to changes in the dataset and model those changes as runtime parameters to the synthetic benchmark.

Ideally, the miniature programs would be benchmark replacements, but I-cache and D-cache memory access models and the branching models used in their creation introduce significant errors. This makes them a solution in the “middle” between micro-benchmarks and applications. However, as shown in Section 3.4, many of the characteristics of the original applications are maintained, and the synthesis approach provides a framework for the investigation of advanced cache access and branching models each independently of the other. As the design space is explored, if the cache and branch predictors should change significantly, the stair-step synthesis approach should be used. In that case, synthesis at the new design point is carried out, including perhaps using new synthesis methods that are appropriate to the caches or branch predictors being explored, and design space exploration can then continue.

The synthetics use a small number of instructions in order to satisfy the I-cache miss rate. This small number causes variations in workload characteristics, including basic block size, with corresponding changes in instruction mix, dependence relationships, and dispatch window occupancies. One solution is to instantiate additional basic blocks using *replication* [107]. Multiple sections of representative synthetic code could be synthesized and concatenated together into a single benchmark. Each section would satisfy the I-cache miss rate, but the number of basic blocks would increase substantially to more closely duplicate the instruction mix. Similarly, multiple sections of synthetic code, and possibly initialization code, could be concatenated together to recreate program phases [83]. Additionally, phases from multiple benchmarks could be consolidated together and configured at runtime through user parameters.

3.6 EARLY SYNTHESIS AND RELATED WORKLOAD SYNTHESIS RESEARCH

Decades ago, computer performance was quantified using synthetic workloads. The earliest metrics included single-instruction runtime, such as that of an ADD instruction [36], and the runtime of simple synthetic instruction mixes [33], but these were quickly outmoded as complex microarchitectural techniques to improve performance, including pipelining, the use of memory hierarchies, and the use of address translation mechanisms, made individual instruction execution times highly unpredictable.

The early synthetic benchmarks such as Whetstone [22] and Dhrystone [103] were developed to represent more contemporaneous program features. They were written by hand and proved difficult to maintain as languages evolved rapidly [55][45]. They were quickly made obsolete by new language paradigms such as object-oriented programming, concurrent programming (e.g. Ada), virtual machines and languages such as Java, the use of standard object code libraries and multi-language applications.

Inefficient coding styles, made possible by cheap memory, led to complex applications that were not easily represented by the early synthetics. A major problem was that compilers would “cheat” and eliminate code in the synthetics specifically intended to test machine performance but not contributing to a functional result [104]. Other problems included a lack of standards related to compiler parameters, input dataset use, and performance metrics [36]. The present work addresses these concerns by synthesizing low-level volatile *asm* calls that represent the original workload but cannot be pruned by compiler optimizations. The automatic synthesis process ensures that synthetics can be created quickly for new language paradigms.

Synthetic kernel programs such as the Lawrence Livermore Loops [62] and the Numerical Aerodynamic Simulation (NAS) kernels [55] represent inner loops of applications found in engineering and science. Kernels by themselves are not representative of any complete application and therefore are not used for general-purpose computer system evaluation. Other microbenchmarks, such as in the STREAM loops [64], LMBench [65], and Keating *et al.* [49] were developed to study the memory hierarchy or database functions, but again they do not represent real applications.

Several *ad-hoc* techniques to synthesize representative workloads were developed [107][90][106]. In Wong and Morris [107], a linear combination of microbenchmarks is found that, when combined in a process called *replication* and executed, duplicates the LRU hit function of the target benchmark. There is no clear way to incorporate other execution characteristics like instruction mix into the technique. In Sreenivasan and Kleinman [90], the joint probability distribution function of CPI over a set of chosen benchmark characteristics is used to combine microbenchmarks to create a synthetic workload. In Williams [106], database transactions are performed by a synthetic workload built from four kernel programs. The resulting synthetics are difficult to correlate with applications [55][90].

The primary focus in *program synthesis* is on mathematical theorem-provers or frameworks for synthesizing high-performance programs from formal specifications [59][60][77][9][3]. The workload characteristics of existing programs are not factors. In McCalpin and Smotherman [63], scientific programs are automatically generated for a particular machine from a concise specification of a mathematical algorithm. Similarly, in Whaley and Dongarra [102], a process for automatically tuning scientific code to improve performance is described. In both works the goal is to obtain the best performance given a particular cache configuration but not to match the performance of a particular workload. In Corno *et al.* [21], random instructions are generated to functionally verify processor designs without considering the performance of actual workloads.

Automatic methods have been developed to synthesize benchmark circuits that are similar to other circuits but sufficiently different to stress electronic design automation tools [100][54][76]. These works analyze graphs of nodes that represent physical circuits and generate other similar sets of nodes. Again, there is no attempt to match the performance characteristics of existing designs.

The well-organized microarchitectures of ASICs and DSPs lend themselves to automatic *code scheduling* from simple code specifications [19]. The goal is to schedule instructions to maximize performance with complete knowledge of the machine resources and pipeline structure, and no attempt is made to generate programs based on the workload characteristics of other programs

In Hsieh and Pedram [38], assembly programs are generated that have the same power consumption signature as applications. However, all workload characteristics are modeled as microarchitecture-dependent characteristics, so the work is not useful for studies involving design trade-offs [28]. In particular, the instruction sequences and dependence relationships of the synthetic programs are not representative of the original workload, in contrast to the present work. The cache access and branch predictor models in Hsieh and Pedram [38] are useful as high-level ideas or starting points, but the specific implementations in that work allow and rely on modifications to the workload features shown to be required for representative performance.

Sakamoto *et al.* [81] present a method to create a binary image of a trace and memory dump and execute those on a specific machine and a logic simulator, but the required binary image and fixup code are complicated and not easily portable to other systems and simulators. No attempt is made to create an abstract trace from statistics in order to reduce runtimes.

In SimSnap [92], programs are instrumented and executed on hardware, and checkpoints are emitted. The checkpoint is then used to initialize execution in a cycle-accurate simulator. At the end of simulation another checkpoint is emitted. That checkpoint can then be compared to the same checkpoint emitted by hardware, effectively validating the cycle-accurate simulator. Similarly, Intrinsic Checkpointing [79] inserts memory preload instructions into a program binary to represent a SimPoint phase. In neither method is there an attempt to reduce the total number of instructions that must be executed to represent the behavior of the original application.

3.7 SUMMARY

This chapter proposes the automatic synthesis of testcases from the dynamic workload characteristics of executing applications. The improved workload modeling and trace synthesis of Chapter 2 is combined with memory access and branch predictability models to synthesize testcases in the C language with inline assembly language calls to preserve representative workload characteristics. The method is used to synthesize testcases for the SPEC 2000 and STREAM Alpha benchmarks to IPCs within 2.4% on average of the IPCs of the target applications with similar average instruction mix, cache access characteristics, dispatch window occupancies, and dependence characteristics, while runtimes are often three orders of magnitude shorter.

Unlike prior synthesis efforts, the focus is on the low-level workload characteristics of the executing binary in order to create a workload that behaves like a real application executing on the machine. Multiple synthetic benchmarks are necessary if the application is executed on multiple machines, significantly different ISAs, or multiple datasets, but the automatic process minimizes the cost of creating new benchmarks and enables consolidation of multiple representative phases into a single small benchmark.

Other benefits include portability to various platforms and flexibility with respect to benchmark modification to isolate and study particular workload characteristics and to study future workloads. The synthesis technique abstracts the functionality of the original workload and thereby encourages code sharing between industry and academia. The significantly shorter-running testcases make feasible the consolidation of the workloads used for early design studies and performance model validation. The use of the stair-step synthesis approach is proposed for early design studies.

The next chapter continues the investigation of the synthetic workloads by detailing and quantifying the errors in the synthetics due to the synthesis process.

Chapter 4: Quantifying the Errors in Workload Characteristics Due to the Workload Synthesis Process

Chapter 2 presents improved workload modeling and trace synthesis for early design studies. Chapter 3 coupled the improved workload modeling with memory access and branching models to synthesize flexible workloads useful for design studies and performance model validation. This chapter continues the investigation of the synthetic workloads and synthesis process by quantifying the errors due to the synthesis process. While synthetic workloads have been shown to compare well against real workload executions with respect to performance metrics, the synthetic workload characteristics are often different from those of the original workload. The sources of error are described.

The results indicate that application of the major workload changes found in the synthetic testcases leads to performance errors that are within 1% on average of the base statistical simulation results. It is also shown that as design changes are investigated, application of the workload changes from the synthetics results in errors that remain small (less than 2% on average) relative to the base results. The conclusion is that small changes in workload characteristics, such as those produced in the synthesis process, have only a small impact on overall performance results.

In addition, no prior study has determined how exact workload characteristics must be to get good statistical simulation correlation. This chapter quantifies the impact of changes to workload characteristics in statistical simulation and shows that significant changes in workload characteristics often result in only small reductions in accuracy.

4.1 INTRODUCTION TO ERRORS IN SYNTHETIC WORKLOADS

As discussed in Chapter 2, statistical simulation has been proposed for rapid and accurate early design studies [4][27]. In statistical simulation, specific workload and machine characteristics are collected during dynamic execution of workload and processor model using a cycle-accurate simulator. Prior works attempt to model specific characteristics as accurately as possible in order to reduce IPC prediction error with respect to the original cycle-accurate simulation. No prior work studies the effect of using less accurate versions of the workload characteristics even though there is a trade-off between speed and the accuracy of profiling.

In Eeckhout *et al.* [29], the impact of changes in error is discussed and results are estimated using a mathematical extrapolation from accurate results. Extrapolations in the presence of small errors are shown to lead to correct design decisions. Also, errors in statistical simulation are shown to be small in general, indicating that statistical simulation is accurate enough for early design studies. The impact of changing the characteristics themselves is not studied.

In workload synthesis as proposed in this dissertation, the statistical flow graph is actualized as compilable source code. The synthesis process traverses the graph similar to synthetic trace generation, but the instructions in each basic block are instantiated as assembly calls inside a C-code wrapper. Memory models are instantiated from the cache and memory access statistics and branching models from the branch predictabilities. When the synthetics are executed, errors in dynamic execution characteristics versus the original application result because of the wrapper code and the use of these locality models. This chapter quantifies how performance is affected as the runtime characteristics change.

However, it is difficult to manipulate the synthetic testcases to study specific percentage changes in workload characteristics and to develop intuition about the impact of the errors due to the testcase synthesis process. In this work, statistical simulation is used to assess the impact of percentage changes in general and the changes found in the

Table 4.1: Major Sources of Error in Synthetic Workloads

Synthesis Phase	**Sources of Error in Synthetic**
Workload Characterization	Retention of basic blocks, instruction abstraction
Graph Analysis	I-cache miss rate model, number of basic blocks, compatible dependences, loop iterations, memory access model, pool elimination, branching model
Register Assignment	Stack instructions, register range, number of registers used, round-robin assignment, dependence models, stream pools
Code Generation	Header and footer overhead

synthetic workloads in particular. The workload characteristic changes due to the synthesis process are found to cause less than a 2% change in performance versus the original application. Other contributions of this chapter are:

i. The sources of error in the workload characteristics due to the synthesis process are listed and described.

ii. Statistical simulation is presented as a means for studying the impact of errors in workload characteristics on performance, and it is used to quantify the errors of specific errors, including the effects of changes in instruction dependences and mix.

iii. Statistical simulation is used to study the effects of the errors found in synthesis, and the impact on specific design changes is quantified as well.

In the next section, the errors in the synthesis process are identified. In Section 4.3, the challenges to quantifying performance changes using statistical simulation are quantified. In Section 4.4, the results of the studies are presented and the errors are quantified.

4.2 SOURCES OF ERROR IN WORKLOAD SYNTHESIS

In this section, some facts about workload synthesis are reviewed as a context for presenting the sources of the errors in the synthetics that result from the synthesis process. Section 4.4 quantifies the errors for specific workload features. Recall from Figure 3.2, the synthesis process has four major phases: *workload characterization; graph analysis; register assignment* and *code generation*. Each piece is reviewed at a high level and the errors in the synthesis process that result are presented. Table 4.1 lists the sources of error discussed in this section.

4.2.1 Sources of Error in Workload Characterization

The dynamic workload characteristics of the target program are profiled using a functional simulator, cache simulator and branch predictor simulator [71][4]. The characterization system currently takes input from fast functional simulation using SimpleScalar [27] or trace-driven simulation in an IBM proprietary performance simulator. The dynamic basic block instruction sequences are characterized, including the instruction dependences, the branch predictabilities, and the L1 and L2 I-cache and D-cache miss rates at the granularity of the basic block. Instructions are abstracted into five basic classes: integer, floating-point, load, store, and branch. Long and short execution times for integer and floating-point instructions are distinguished. The IPC of the original workload is tracked to compare to the synthetic result. The statistical flow graph [71][4] is assembled from the workload characterization.

Several sources of error can result in the final synthetic workload from this step. In practice, not all basic blocks are retained due to file space limitations. Probabilistically, very low frequency basic blocks will not be instantiated in a short synthetic testcase, so a typical system may only keep the basic blocks that account for 99% of the dynamic blocks seen during execution. Without the basic blocks themselves, their instruction sequences, dependence information, and machine characteristics will also not be represented in the synthetic workload. This adds a small but measurable error to the final workload characteristics of the synthetic workload.

Similarly, for the basic blocks that are retained, abstracting the instructions to five canonical types adds error. For example, all types of branches are abstracted to one conditional branch class. Specific opcodes and their associated number of operands have subtle effects in the machine when executing the original workload that are not exhibited by the synthetic workload.

4.3.2 Sources of Error in Graph Analysis

In this section, the steps of graph analysis are examined and errors related to each step are listed. The sources of error in the synthetic workload from graph analysis are numerous.

4.3.2.1 Instruction Miss Rate and I-cache Model

The workload characterization is used to build the pieces of the synthetic benchmark. The statistical flow graph is traversed using the branching probabilities for each basic block, and a linear chain of basic blocks is assembled. This chain will be emitted directly as the central operations of the synthetic workload. The number of instantiated basic blocks is equal to an estimate of how many blocks are needed to match the I-cache miss rate of the application given a default I-cache configuration (see Chapter 3). The number of synthetic basic blocks is then tuned to match the I-cache miss rate and instruction mix characteristics by iterating through synthesis a small number of times. For the particular I-cache size used in these experiments, anywhere from one to 1000 basic blocks may be necessary to meet the I-cache miss rate of a particular application, or approximately 4000 instructions.

With respect to errors versus the original workload, the final I-cache miss rate using these methods will not exactly match that of the original application even for the default cache configuration. For workloads like the SPEC CPU, the I-cache miss rates are non-zero but very small, and the major requirement for a synthetic code is that it fits in the I-cache. Even though errors are large for many benchmarks, the miss rates are so small in an absolute sense they usually do not impact performance significantly, as discussed in Chapter 3. Because of the low miss rates for the SPEC CPU, errors in the synthetic workload in which the I-cache miss rate decreases generally does not impact performance, but errors in which it increases can impact performance significantly.

If too few passes through the synthesis process occur, a non-optimal I-cache miss rate may result. Figure 3.4 shows one pass through synthesis. The parameter tuning process itself can perpetuate errors for certain characteristics since the focus is usually to reduce differences in those characteristics that cause the largest error.

4.3.2.2 Instruction Dependences

For each basic block, instruction input dependences are assigned. The starting dependence for each instruction is taken from the average found for the instruction during workload characterization. If the dependence is not compatible with the input operand type of the dependent instruction, then another instruction is chosen. The algorithm is to move forward and backward from the starting dependence through the list of instructions until the dependence is compatible, as explained in Chapter 3.

The algorithm as implemented may create synthetic dependences that would not exist in the actual application. In practice, the average number of moves away from the starting dependence per instruction input is small, usually fewer than one, but for particular benchmarks it can be larger. Looking at Table 3.3, *mgrid* and *applu* have the largest average number of moves per dependence assignment, at 9.4 and 20.3, respectively. Both of those workloads have very large average basic block sizes which results in small numbers of basic blocks being instantiated in order to properly match the I-cache miss rate. This limits the availability of the variety of dependences that the original workload exhibited. Also, the workload characterization records the shortest dependences among identical basic blocks with different dependence distances [27][4], so not only is the starting dependence sometimes in error, but movement is inevitable.

4.3.2.3 Loop Counters and Program Termination

When all instructions have compatible dependences, a search is made for an additional integer instruction that is attributed as the loop counter. The branch in the last basic block in the program checks the loop counter to determine when the program is complete. The number of executed loops is chosen to be large enough to assure IPC convergence given the memory accesses of the load and store instructions in the benchmark. In practice, the number of loops does not have to be very large to characterize simple stream access patterns. Experiments have shown that the product of the loop iterations and the number of instructions must be around 300K to achieve low branch predictabilities and good stream convergence. The loop iterations are therefore approximately *300K/4000* for most benchmarks.

Errors are introduced by limiting the iterations of the basic block in the synthetic to provide for fast convergence. Errors may be introduced for any of the spectrum of workload characteristics and may necessitate changes in synthesis parameters or additional parameters to adjust the synthesis algorithms to compensate.

4.3.2.4 Memory Access Model

The data access counter instructions are assigned a stride based on the D-cache hit rate found for the corresponding load and store accesses during workload characterization. The L1 and L2 hit rates are used to obtain the synthetic stride in a table. The problem with a stride-based model is that it is simplistic and does not support “random” accesses or frequent address interactions between separate operations. Errors are introduced into the cache hierarchy at all levels. Future work is needed to determine what the best modeling trade-off is between realistic, but complicated, access behavior and simple, stride-based models. Cache warm-up prior to synthetic execution may be necessary to replicate specific locality features such as capacity misses. Intrinsic Checkpointing [79] describes a memory initialization capability that could be integrated with the synthetics to potentially reduce cache misses.

Even with the stride-based model, there can be a large error in stream behavior when an actual stream hit rate falls between the hit rates in two rows of the stride table. In Section 3.4, the idealized load model is responsible for correlation error when the cache line sizes change from the default. The table does not consider cache size and associativity. Errors in the synthesized cache miss rates must then be compensated for by algorithm adjustments using parameters such as the *stream factor* and *miss rate estimate factor* in Table 3.3.

Another source of error is the consolidation of streams pools necessary to limit the number of data access registers that at being used. Ideally each load and store would have its own access stride, but this is not possible with a limited number of registers in use and the memory access model proposed here.

4.3.2.5 Branching Model

A branch predictability model is superimposed onto the set of basic blocks that already represent the instruction mix, dependences and data access patterns of the original workload. A number of branches in the trace are configured to branch past the next basic block or a number of instructions based on the global branch predictability of the original application. The mechanism causes a configured branch to have a fixed predictability for predictors that use 2-bit saturating counters.

The branching model is fairly accurate, giving 1.9% average error in Chapter 3, but it tracks overall branch predictability, not specific predictabilities local to basic blocks. Also, Table 3.3 shows large adjustments to the *BP factor* for a variety of benchmarks, including *mgrid* and *applu* that have large average basic block sizes.

The branching model only accounts for branch history tables that use 2-bit saturating counters, such as bimodal predictors. It is not designed for predictors that use branch shift register histories or global history, but it still seems to do well on the tournament predictor used in the POWER5 processor in Chapter 6. The branching model is usually not an issue for model validation, but it is an important consideration if the branch predictor configuration changes in design studies. Also, when the testcases synthesized for one machine are executed on another with a different branch predictor strategy, there may be a change in performance only explained by the synthetic branching model.

4.3.3 Sources of Error in Register Assignment

All architected register usages in the synthetic benchmark are assigned exactly during the register assignment phase. Only 20 or so general-purpose registers divided between data access counters and code use are necessary, and 30 floating point registers are used. These numbers were determined experimentally. With too few or too many registers available for code use, the compiler may insert stack operations into the binary. The machine characteristics may not suffer from a few stack operations, but for this study they are excluded. The available code registers are assigned to instruction outputs in a round-robin fashion.

Data access streams are pooled according to their stream access strides and a register is reserved for each class. All data access counters in the same pool increment the same register. The algorithm is to merge the least frequent two pools together using the most restrictive (highest miss rate) stride from the two pools until the total number of registers is under the register use limit.

No attempt is made to use the register ranges and numbers of registers in the original application. Subtle register usage patterns implemented by the compiler will not be exhibited in the synthetic. Also, only read-after-write (RAW) dependences are accounted for, as in statistical simulation [4][27], so the behavior of the resulting synthetic will not exhibit other types of dependence hazards.

The algorithm for merging stream pools is somewhat *ad hoc* and tends to disturb the access behaviors of the least frequent streams. These errors might necessitate additional adjustments using stream access parameters.

4.2.4 Sources of Error in Code Generation

The code generator takes the representative instructions and the attributes from graph analysis and register assignment and outputs a single module of C-code that contains calls to assembly-language instructions in the target language. Each instruction in the representative trace maps one-to-one to a single *asm* call in the C-code. Ordinary C-code is emitted for functions not related to the trace, as, for example, to instantiate and initialize data structures and variables. Emitted are the C-code *main* header and variable declarations to link output registers to data access variables for the stream pools, the loop counter variable, and the branching variable. Pointers to the correct memory type for each stream pool are declared, and *Malloc* calls for the stream data are generated with size based on the number of loop iterations. Each stream pool register is initialized to point to the head of its *malloced* data structure. The loop counter register initialization is emitted, and the instructions associated with the original graph traversal are emitted as volatile calls to assembly language instructions. The data access counters are emitted as integer additions of its output register value to the associated stride for the stream. The loop counter is emitted as an integer subtraction of one to its output register value. The basic blocks are analyzed and code is generated to print out unconnected output registers depending on a switch value. The switch is never set, but the print statements and volatile asm calls guarantee that no code is eliminated during compilation. Furthermore, using the *gcc* compiler, instructions are not reordered. Code to free the *malloced* memory is generated, and, finally, a C-code footer is emitted.

Emitting *asm* calls for low-level operations creates errors related to the abstraction of the instruction mix mentioned earlier. The instruction sequences instantiated in the assembly calls are designed to reproduce the performance impact of the original application, so it would be best for the C-code header and footer to not contribute error to the performance result. However, the number of iterations through the sequences of instructions is kept small to reduce runtimes. The header and footer account for about five thousand instructions, but this can add up to almost 2% of a synthetic that runs for only 300K instructions.

4.3 THE FLEXIBILITY OF STATISTICAL SIMULATION

As in the last section, there are many aspects of the testcase synthesis process that can lead to errors in the workload characteristics. The effects of these errors are assessed in general and in particular for the errors observed for a specific synthesis system. For the general case, each error is investigated independently of other errors so that a qualitative view or intuition about how specific error levels affect performance can be obtained.

It is difficult to quantify the effect of changes in workload characteristics independently of each other using synthetics or other source codes. The source code can be changed, compiled, and executed on a system, but it is difficult to know a priori what changes to make to only affect one characteristic. For example, it is a challenge to make a change to source code to get only a 1% change in L2 hit rate. After some work, a set of appropriate changes may be isolated to match the L2 hit rate, but the likelihood is that the same changes will also impact other observed workload characteristics. The memory access changes that affect the L2 hit rate will also affect the L1 hit rate, L3 hit rate, and main memory access patterns. The changes to source code would also affect such things as instruction mix and dependences since the change in L2 hit rate implies that different code areas are being exercised. The situation is even more complicated for operand dependence and branch predictability investigations.

Table 4.2 lists the workload characteristics that are investigated in this study. Characteristics are chosen that have a large impact on the synthesis result. Additional characteristics from Table 4.1 could also be modeled and studied. Statistical simulation provides a means to investigate changes in the workload characteristics of Table 4.2 independently of each other. In statistical simulation, the microarchitecture-independent and microarchitecture-dependent workload characteristics are captured from an executing application using profiling tools. The statistical profile is then input to a trace-driven performance simulator which contains a model that is always less detailed than a cycle-accurate RTL model.

The statistical profiling tools currently available can be divided into two

Table 4.2: Prominent Workload Characteristics and Pearson Correlation Coefficients for Factor Values versus Simulation Results

Class	Factor	Correlation Coefficient
Microarchitecture-Independent	Instruction Mix (Integer, Float, Load, Store)	(0.988, 0.998, 0.997, 0.999)
	Dependence Distances (I0, I1, F0, F1, L0, S0, S1, B0, B1)	(0.979, 0.970, 0.984, 0.981, 0.974, 0.977, 0.913, 0.980, 0.999)
	Number of Basic Blocks	n/a
Microarchitecture-Dependent	L1 I-cache Hit Rate	0.971
	L1 D-cache Hit Rate	0.978
	L2 Hit Rate	0.978
	Branch Predictability	0.990

categories: those that collect statistics as an aggregate without regard to the basic blocks [17][71][28], and those that collect statistics at the granularity of the basic block [4][27]. Profiling at the granularity of the basic block increases storage requirements for the profile, but accuracy is also significantly increased. There are major differences between Eeckhout *et al.* [27] and Bell *et al.* [4]. In the former, the $k=1$ flow graph maintains statistics for any basic block based on any possible prior block. The statistics are quite detailed and generally map one-to-one with the instruction classes implemented in the SimpleScalar microarchitecture. In Bell *et al.* [4], the basic block map maintains aggregate statistics for all prior basic blocks and the statistics are quite simplified. The instructions, for example, are abstracted to the five generic types described in Section 4.2. Also, the machine model is simplified and generalized compared to SimpleScalar and must be calibrated to the machine-under-study [71][4]. Both systems obtain average IPC prediction errors significantly less than 10%.

Table 4.2 shows seven classes of workload characteristics that are investigated using these statistical simulation capabilities. User parameters are implemented that specify factors to be multiplied with corresponding characteristics as basic blocks are selected during the traversal of the flow graph.

For the instruction mix metrics, random variables over the user parameter factors determine a new instruction mix. For example, if the integer factor is 1.03, then the floating point, load, and store instruction fractions are each converted to integer instructions by an amount equal to 1% of the integer fraction, leading to a total increase in integer instructions by 3%. Note that the branch instruction fraction is not modified because of the difficulty of removing branches and thereby joining basic blocks (and increasing the average basic block size) in the current simulator implementation, but this could be done in future work. To implement combinations of mix changes, as a non-branch instruction is encountered, the factors for the other three instruction types are applied in the following order based on a random variable over each factor: integer, floating point, load, and store. For example, if the current instruction is an integer, a random variable over the floating-point factor is applied. If the result is not less than the factor, then the same process is carried out for the load factor. If that test also fails, then the same is done for the store factor. Note that this can result in fractions of instructions that are slightly different from the specified fractions since, for example, the store factor is only checked if the previous two tests fail. In an ideal case in which instruction type frequencies are roughly equal, the round-robin application of the factors would give a roughly equal number of changes for all types.

The dependence distances in each basic block are multiplied by the dependence parameter factor and rounded to the nearest integer. Separate factors for all inputs of each instruction type are not modeled, just one global factor multiplied by all. The number of basic blocks is simply applied as specified in the parameter. The L1 and L2 hit rates for each basic block are multiplied by their corresponding parameter factors. Likewise, the branch predictability factor multiplies the branch predictability of each basic block.

Figure 4.1 shows the percentage change in L1 D-cache hit rate as the factor

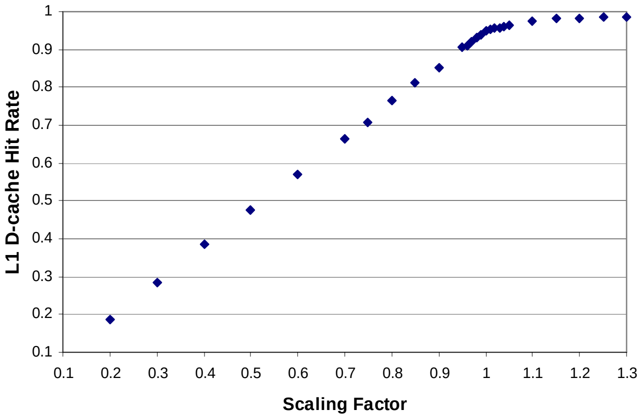


Figure 4.1: L1 D-cache Hit Rate Factor for *gzip*

changes for *gzip* in the statistical simulation system described in the next section. It is apparent that the resulting percentage follows the factor change. The roll-off for a factor larger than 1.0 is due to the presence of some very low hit rates in some basic blocks that continue to show small improvement even for large factors. The Pearson correlation coefficient for the figure is shown in Table 4.2.

Table 4.2 also gives the Pearson correlation coefficients for the other factors as they are applied for *gzip*, except that the floating point instruction mix and the zeroth and first operand (F0 and F1) dependence distance coefficients are for *wupwise* since *gzip* has few floating point instructions. The dependence distance coefficients are given for all operands of each instruction type in the order shown in the second column (e.g. I0 is integer operand 0 and has correlation coefficient equal to 0.979). Most are higher than 0.97, which includes the effect of any roll-off due to low hit rates or saturating to a hit rate of 1.0 for large factors.

In Eeckout *et al.* [27], statistical simulation IPC prediction errors less than 10% are shown to be useful for early design studies. In this paper, the range of change in a workload characteristic that keeps error within 3% of the base system error is quantified; this gives a good cushion for accurate design studies in spite of the change.

4.4 SIMULATION RESULTS

In this section, statistical simulation is used to study the effects of changes in workload characteristics.

4.4.1 Experimental Setup and Benchmarks

The experimental setup and benchmarks are the same as in the statistical simulation experiments in Chapter 2. The SPEC 2000 and STREAM Alpha binaries are again simulated on the machine configuration in Table 3.5. For each benchmark, a

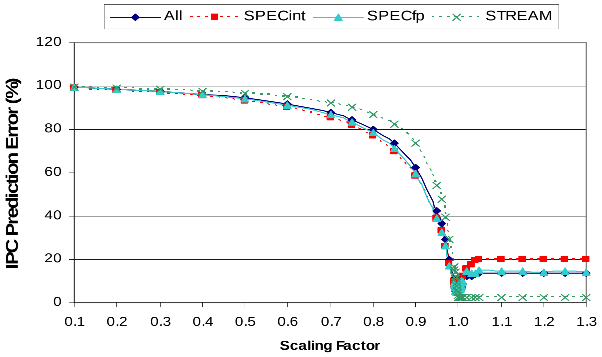


Figure 4.2: L1 I-cache Hit Rate Factor

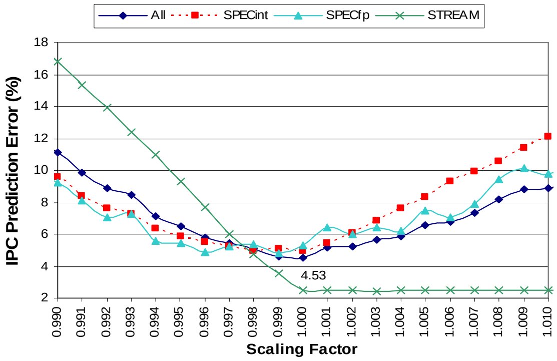


Figure 4.3: L1 I-cache Hit Rate Factor (expanded)

synthetic trace of 2500 basic blocks is generated and simulated for 20k cycles. The base results are given in Figure 2.11 and show an average IPC prediction error of 4.7% with a maximum error of 15.2% for *ammp*. As mentioned in Chapter 3, a large TLB miss rate coupled with large L1 and L2 D-cache miss rates make *ammp* a challenge for the more generalized machine model .

4.4.2 Sensitivities to Changes in Workload Characteristics in Statistical Simulation

Incremental percentage changes to the workload characteristics are examined in Table 4.2. For each characteristic, the factor value versus the error in IPC versus SimpleScalar for all benchmarks is shown and it is also broken down by suite: SPEC INT, SPEC FP and STREAM. As mentioned, the factors generally multiply the value of the corresponding characteristic. For example, an L1 D-cache factor of 0.5 reduces the L1 D-cache hit rate to 50% of its base value. A factor equal to 1.0 always gives the base system result.

Figure 4.2 shows the sensitivity to I-cache hit rate for the benchmarks. The x-axis shows the factor value and the y-axis gives the IPC prediction error [27] as the factor is applied. It is apparent that the benchmarks are very sensitive to smaller I-cache hit rates,

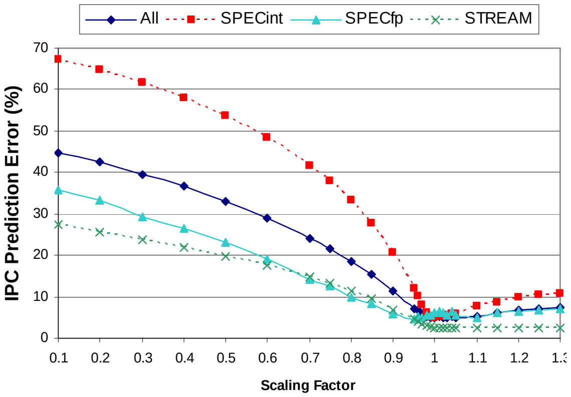


Figure 4.4: L1 D-cache Hit Rate Factor

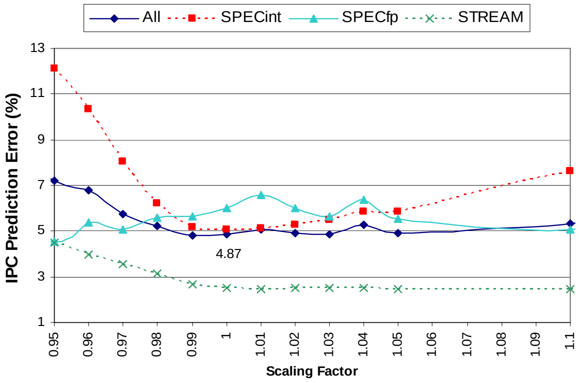


Figure 4.5: L1 D-cache Hit Rate Factor (expanded)

but less so to larger hit rates. This confirms that the SPEC and STREAM hit rates are already extremely high for the base system.

To understand the range in which errors are less than a 3% change from the base system, Figure 4.3 shows the sensitivity over a narrower range, from 0.99 to 1.01. The average IPC prediction error is 4.5% for the base system, and increases to 7.1% for a factor of 0.994, and 7.3% for a factor of 1.007. So there is a 1.3% *margin* or range for the factor within which the average error change is less than 3% from the base. For average errors less than 10%, the margin would start at 0.991 and end well above 1.01, giving more than a 2% margin. Figures 4.2 and 4.3 show that the STREAM benchmarks are the most sensitive to low I-cache hit rates. Performance in many of the STREAM is highly dependent on the simultaneity of hits and outstanding misses for two loads [4].

Figures 4.4 and 4.5 give the sensitivity to L1 D-cache hit rate changes. Here the margin is larger than 15%, ranging from 0.95 well up past 1.10. The figures show the highest sensitivity for SPEC INT. In the base run, the hit rates for SPEC INT are generally higher than those for SPEC FP and STREAM, and as a consequence SPEC INT has generally higher sensitivity to change than the others.

Figure 4.6 shows the sensitivity to L2 hit rate. There is a wide margin of 60% from 0.70 to more than 1.30. The STREAM benchmarks are the most sensitive, but the

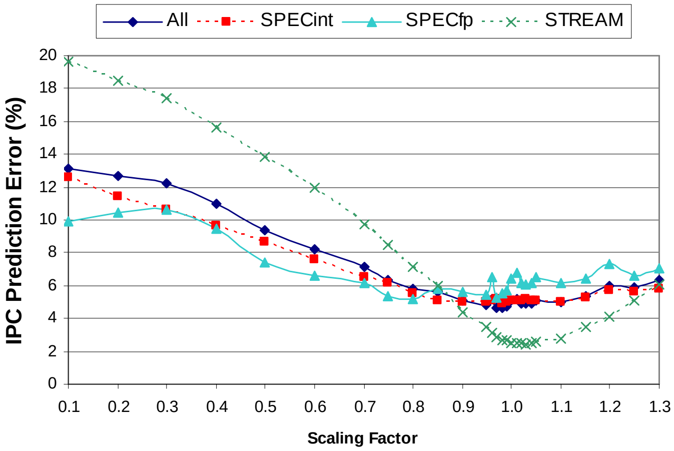


Figure 4.6: L2 Cache Hit Rate Factor

start of their margin only narrows to 0.80. The intuition behind these results is that generally the L1 I-cache and D-cache miss rates are low, which reduces sensitivity to L2 hit rate. But L1 miss rates for STREAM are higher, so it is more sensitive. These results confirm the observations in Bell and John [7] for the synthetic testcases. Note that the margin would almost certainly be reduced for L2 caches with higher latencies, but here a balanced design is assumed in which the lower memory latency compensates for no L3.

Obviously, even though sensitivities are low, one would not want to run early design studies either in statistical simulation or using synthetics containing such large changes to the L2 hit rate. One reason is that the standard deviation of the average errors in these studies is large, up to 4% around the mean for the base configuration. Therefore the results in this chapter specify trends that aid our intuition about the errors due to the synthesis process, but the effect for individual benchmarks may be larger or smaller and is usually not known a priori to simulation. Also, studies of cache design changes are obviously precluded at the limits of the margin, but statistical simulation is not currently used to study any design changes in the cache hierarchy [29]. The results argue that statistical simulation could be augmented with somewhat inexact memory access models

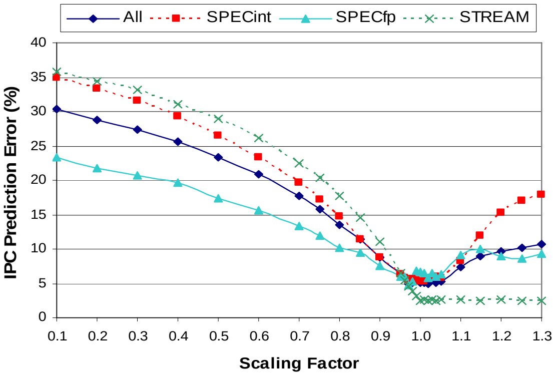


Figure 4.7: Branch Predictability Factor

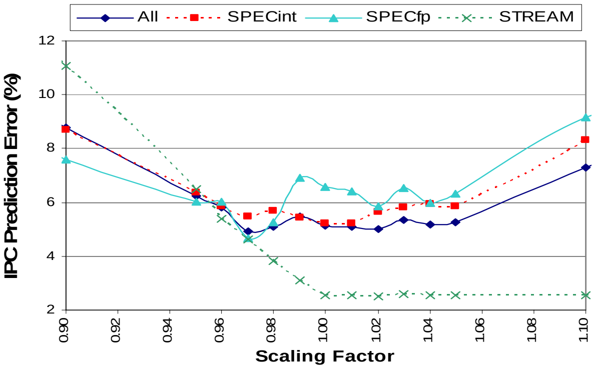


Figure 4.8: Branch Predictability Factor (expanded)

and still achieve reasonable accuracies against the machine responses of the original applications. The models would then be useful for synthesizing testcases for portability to various platforms. Statistical simulation may be the best platform to prove the validity of potential models before implementing them in the synthetic testcases.

Figures 4.7 and 4.8 show the sensitivity to branch predictability. The margin is over 16%, ranging from 0.94 to above 1.10. Similar to the L1 I-cache results, the benchmarks are much more sensitive to smaller factors because of already high branch predictabilities.

Figure 4.9 shows the effect of changing the number of basic blocks implemented in the statistical flow graph. It is apparent that quite representative behavior can be obtained using more than 500 basic blocks in the graph. Fewer than 200 blocks can increase errors. As mentioned, this leads to errror in some of the synthetic testcases which have very long average basic block sizes (e.g. *mgrid* and *applu* with average sizes of 100 and 93 instructions, respectively) and therefore can only use a small number of blocks to satisfy their I-cache miss rates (30 and 19, respectively) [7]. These errors are compensated for by larger parameter changes to tune the synthesis algorithms. The two hundred basic block limit only applies to the machine configurations and workloads

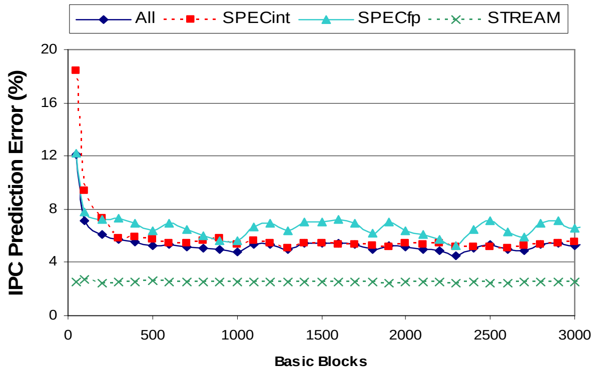


Figure 4.9: Basic Block Changes

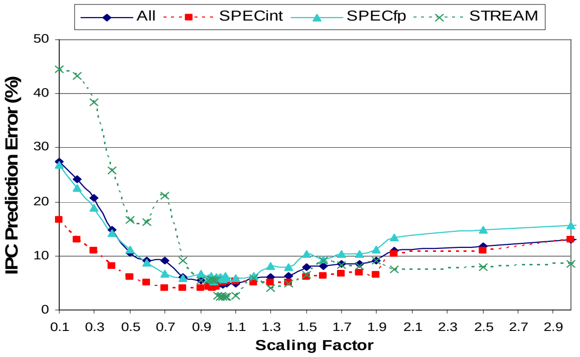


Figure 4.10: Dependence Distance Factor

studied here. Larger designs and different phases of instructions may need more or less basic blocks to represent the specific machine and instruction interactions in it.

Figure 4.10 shows the sensitivity to dependence distance changes. A margin from 0.8 to 1.50 is observed. As shown in Table 4.3, similar studies were carried out for the integer, load, store, and float instruction mixes. The figures show that some changes are better tolerated for specific classes of benchmarks. For example, the SPEC INT are very tolerant to large changes in floating point instructions since their base percentage of those is very low.

It is interesting that large changes in the instruction mix and dependence distances can be independently tolerated. This suggests that these microarchitecture-independent

Table 4.3: Workload Characteristic Margins (at 3% from Base)

Sensitivity	Margin Start/End	Margin (%)
L1 I-cache Hit Rate	0.994, 1.007	1.3%
L1 D-cache Hit Rate	0.95, 1.10	15%
L2 Hit Rate	0.70, 1.30	60%
Branch Predictability	0.94, 1.10	16%
Number of Basic Blocks	200, 3000	n/a
Dependence Distance	0.80, 1.50	70%
Integer Instructions	0.80, 1.30	50%
Float Instructions	0.85, 1.30	45%
Load Instructions	0.80, 1.30	50%
Store Instructions	0.10, 1.30	120%

characteristics are less critical to performance than many microarchitecure-dependent characteristics. The empirical results suggest that the margins can be exploited in simulation, since there is a likely profiling effort versus accuracy tradeoff involved. As noted, the margins may not easily be determined a priori to simulation and, also, combinations of changes may multiply their effects and cause larger errors; therefore margin should not be traded for extremely simple modeling effort. The gist of these studies is that small changes are not likely to cause disruption to the machine responses of a synthetic workload and that this observation is germane to testcase synthesis.

4.4.3 Sensitivities to Changes in Workload Characteristics from Testcase Synthesis

In this section, the sensitivity of statistical simulation to the changes in workload characteristics exhibited by the synthetic testcases created in Chapter 3 is investigated. From the workload characteristics obtained for each benchmark, the factor corresponding to the percentage change versus the base statistical simulation system is obtained. The factors for each benchmark are then applied singly or all simultaneously to the statistical simulation system, and the average IPC prediction errors for the benchmark suites are plotted. The goal is to confirm one of the results from Chapter 3, namely, that the changes in workload characteristics due to the synthesis process have a minimal impact

Table 4.4: Example Workload Characteristic Synthesis Changes (*gcc*)

Sensitivity	Actual Value	Synthetic Value	Factor
L1 I-cache Hit Rate	0.971	0.972	1.001
L1 D-cache Hit Rate	0.974	0.975	1.001
L2 Hit Rate	0.964	0.849	0.880
Branch Predictability	0.880	0.906	1.030
Number of Basic Blocks	n/a	850	850
Dependence Distance (average dep)	5.765	5.263	0.913
Integer Instructions	0.394	0.385	0.977
Float Instructions	0.000	0.000	1.000
Load Instructions	0.337	0.348	1.033
Store Instructions	0.112	0.083	0.741

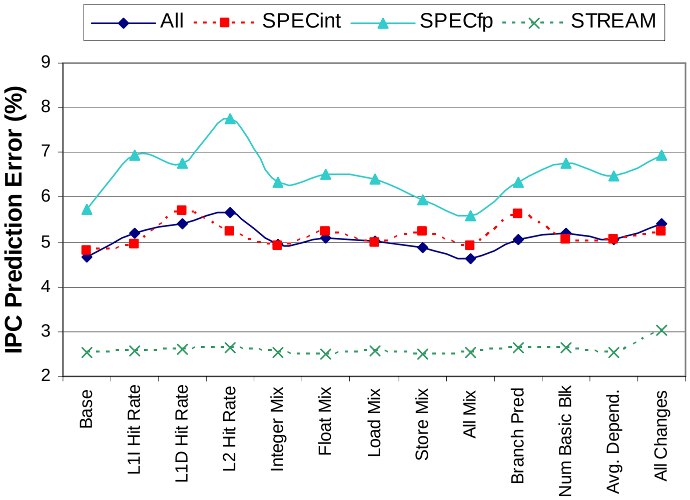


Figure 4.11: Changes Due to Synthesis in Statistical Simulation

on performance. In so doing, additional evidence is obtained that the synthetics are useful for early design and model validation studies.

Table 4.4 gives an example of the workload characteristics obtained from synthesis of *gcc*. The factors that are used in statistical simulation are the ratio of the synthetic value and the actual benchmark value. The number of basic blocks is not applicable to the original benchmark execution since its execution did not use a synthetic trace. Since there is only one dependence distance factor in the statistical simulation system, it is calculated from the ratio of the sum of the average dependence distances for the instruction type weighted by the frequency of the type. For example, the two integer input dependence distances are averaged and weighted by the frequency of integers in the benchmark. Then the result obtained for the five instruction types are summed.

Figure 4.11 shows the results. The base system results are given, followed by the ten workload characteristics applied singly, and then all changes applied simultaneously. Also included is the case of all instruction mix changes applied together. One sees that

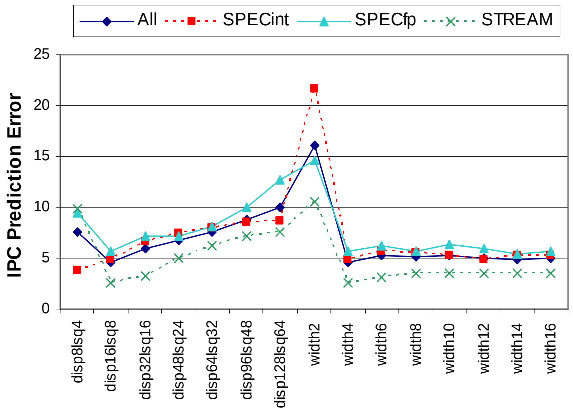


Figure 4.12: Design Changes (No Synthetic Parameters)

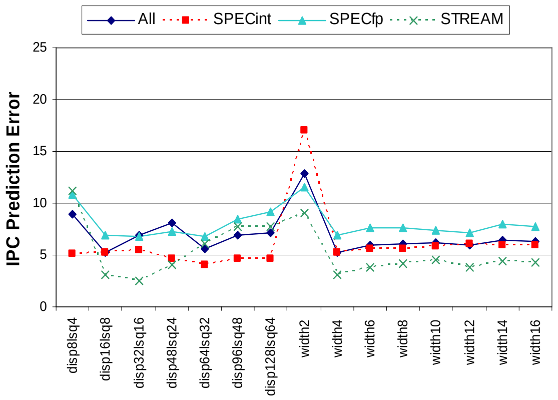


Figure 4.13: Design Changes (Synthetic Parameters)

the largest errors are exhibited when the L2 hit rates are applied, as expected from Chapter 3. The final error of 5.4% is only 0.75% different from the base error of 4.65%.

Note that the errors offset each other, since the final average error is less than the largest errors. This confirms the analysis in Chapter 3 in which large errors in characteristics are often mitigated by more significant offsetting errors in other characteristics. If the errors had been both large and significant, the effects might not have been seen in the synthetic behavior but would be more likely exposed in this study of individual error factors. The only case in which a significant factor (such as for L1 D-cache hit rate) was found is *ammp*, which is excluded from study due to its TLB miss rate, as previously mentioned. Still, the existence of large offsetting errors is a cause for concern and must be ruled out for any new synthetic.

The final IPC is close to the base statistical simulation IPC, but it is somewhat removed from the 2.4% average error for the synthetic testcases themselves. This difference is due to the additional sources of error in Table 4.1 that are not modeled in the statistical simulation runs. Examples include the use of loop iterations, the register usage errors, and the header and footer errors.

Also investigated are the error differences using the synthetic testcase factors when applying design changes. The *dispatch window/LSQ* size is varied from 8/4 to 128/64, and the *machine width* is varied from 2 to 16 (keeping fetch width fixed at 16). Figure 4.12 shows the results for the base system, and Figure 4.13 shows the results with the synthetic testcase factors applied. The figures are very similar. Table 4.5 gives the error differences averaged over the two classes of changes. No average error is greater than 2.5%. As usual, *ammp* is not included in the dispatch window runs.

Note that the *width2* base case has a remarkably larger error than the other cases, and in general the errors versus SimpleScalar become larger than those in Eeckhout *et al.* [27] as the dispatch window becomes larger. This is due to the simplicity of the HLS++ workload profile and machine model, including the instruction abstraction and generalized model. The machine was calibrated to model a dispatch window of 16 and LSQ size of 8. When the window becomes smaller, the machine model gives generally higher performance than the SimpleScalar model because subtle instruction interactions are not modeled.

4.5 SUMMARY

This chapter continues the investigation of the synthetic workloads from Chapter 3 by quantifying the errors in them due to the synthesis process. The results indicate that application of the major workload changes found in the synthetic testcases leads to performance errors that are within 1% on average of the base statistical simulation results. It is also shown that as design changes are investigated, application of the

Table 4.5: Average Percent Error Differences for Dispatch Window/LSQ and Width Studies

Study	All	SPEC INT	SPEC FP	STREAM
Dispatch/LSQ	1.6%	2.5%	1.4%	0.6%
Width	1.3%	1.0%	1.8%	0.8%

workload changes from the synthetics results in errors that remain small (less than 2% on average) relative to the base results. The conclusion is that small changes in workload characteristics, such as those produced in the synthesis process, have only a small impact on overall performance results.

It is also found that errors in various workload characteristics offset each other when executed together, so it is important to investigate the acceptability of the magnitudes of the errors in individual workload characteristics. The statistical simulation methodology described here provides a means for isolating and quantifying the individual errors.

The next chapter continues the investigation of the synthetic workloads with a study of power dissipation. The power dissipation of the synthetic workloads is compared to the power dissipation of the longer running workloads from which they are synthesized.

Chapter 5: Efficient Power Analysis using the Synthetic Workloads

Chapters 2 and 3 leveraged improved workload modeling and synthesis techniques into more representative synthetic traces and workloads that are useful for rapid early design studies and model validations with respect to performance. This chapter investigates the synthetic workloads as a means for early power dissipation studies and power model validation [8].

Power dissipation has recently become an important parameter in the design process. Just as for performance studies, assessing power using performance simulators is problematic given the long runtimes of real applications and is even exacerbated by the addition of counters to track events that contribute to power dissipation. In this chapter, it is shown that the synthetic workloads can rapidly and accurately assess the dynamic power dissipation of real programs. The synthetic testcases from Chapter 3 can predict the total power per cycle to within 6.8% error on average of the power dissipation of the original workloads, with a maximum of 15% error, and total power per instruction to within 4.4% error. In addition, for many design changes for which IPC and power change significantly, the synthetic testcases show small errors, many less than 5%. It is also shown that simulated power dissipation for both applications and synthetics correlates well with the IPCs of the real programs, often giving a correlation coefficient greater than 0.9.

5.1 INTRODUCTION TO POWER DISSIPATION STUDIES

Power dissipation has recently become an important consideration in the design of processors [34]. As frequencies have passed into the multiple gigahertz range and the number of transistors integrated onto a single chip has surpassed 100 million [93][85], the maximum power dissipation for high-end processors has surpassed 100 watts [93][97].

The increases in power dissipation are of significant impact to chip reliability and mobile system battery life [10]. In an effort to study and alleviate increases in on-chip power dissipation early in the design process at the same time that design tradeoffs are being studied, researchers have integrated microarchitectural power estimators into processor simulation systems [15][109][78][54].

Architectural level simulators can obtain accurate results in assessing the dynamic power dissipation of an executing workload. However, the long runtimes for the latest benchmarks such as SPEC 2000 make full program simulation impractical [15][109]. Statistical simulation creates representative synthetic traces of less than one million instructions and has successfully analyzed power-performance trade-offs in a trace-driven simulation system [109][27], but, as described in Chapter 3, traces are not very portable to many modern design platforms including execution-driven simulators, RTL models, hardware emulators, and hardware itself. That is important because execution-driven simulators are useful for assessing the power dissipation of more accurate simulation systems including operating system effects [54]. RTL models and hardware emulators are useful for performance model validation [7], and performance monitor counters in hardware facilitate rapid power dissipation studies [10].

The synthetic testcases from Chapter 3 execute orders of magnitude faster than the original workloads while retaining good accuracy. Prior work shows that IPC has a good correlation to average power dissipation [54][10]. Even though power is not considered in the synthesis process, the synthetic benchmarks display good IPC correlation with actual programs, so it is natural to expect that they can also be used to speed up power dissipation analysis for the applications with low errors.

In this chapter, the dynamic power dissipation characteristics of the synthetic testcases are described. The synthetics are executed on the Wattch simulation framework [15]. The specific contributions of this chapter are:

i) It is shown that the synthetic testcases generated strictly for performance purposes are also useful for analysis of dynamic power dissipation, giving reasonable errors while executing orders of magnitude faster.

ii) The synthetics are also shown to be useful for the relative power analysis of design changes.

iii) The results of the IPC and power dissipation design changes are classified to facilitate analysis.

iv) Prior results that demonstrate a good correlation between IPC and dynamic power dissipation are confirmed, and the results are extended to design change correlations.

In the next section, qualitative reasons are given for why power dissipation is expected to correlate using the synthetic testcases, and the benefits for power model validation are presented. In Section 5.3, the quantitative results of the power analyses are shown.

5.2 SYNTHETIC TESTCASES AND POWER DISSIPATION

Several studies have shown that the synthetic traces in statistical simulation exhibit power dissipation similar to cycle-accurate simulations [27][109]. The synthetic traces contain specific basic block sequences that represent the major components of the performance of the workload. The number of instructions is reduced because instruction sequences that do not contribute to performance are not included. Since the number of instructions is reduced, the total energy for execution of the synthetic trace cannot be compared to that of the original application.

Synthetic traces [4][27] provide accurate power analysis because the basic block sequences that provide accurate performance results must also exercise the machine such that the power dissipation is accurate. As found in Chapter 3, the synthetic traces exhibit dynamic workload features similar to those of the original applications, including instruction mix, number and type of operands, instruction-level parallelism, dependence distances, and memory access and branching behavior. The workload similarities imply that reorder buffer occupancies, pipeline throughput, cache hierarchy access and miss rates, pipeline stalls, and branch predictabilities will be similar. To a large extent, these machine features determine the dynamic power dissipation of the system [15][97][10]. As an example, the abstracted instruction types used for synthesis use one, two or three (for PowerPC) operands as in the original workload, so the proper power dissipation with respect to register port access is obtained.

The synthetic testcases described in Chapter 3 have dynamic workload characteristics similar to those of the synthetic traces, except that the locality models are less accurate. Memory accesses are modeled as strides through uninitialized data structures in order to match miss rates for a default cache configuration. The overall miss rates of the original application are obtained, but particular miss rates at the granularity of individual loads and stores may be quite different from those of the original since integer stride values only generate particular miss rate quanta, rather than a continuous spectrum of miss rates [6]. Microarchitecture-independent memory access models would seek to model more closely the original workload access patterns.

The branching models are also not accurate at the granularity of individual branches. The overall application predictability is matched by configuring a subset of branches to jump past the next basic block 50% of the time. The synthetic testcases would benefit from an exact analysis and configuration of particular branches in the workload. Other anomalies in the synthetic testcases include the retargeting of integer instructions for data structure access and testcase looping, register usage, and the other sources of error described in Chapter 4.

All of these inaccuracies imply that instruction dependences, memory accesses and branch behavior are different from the original workload, that the machine responses to the workload will be correspondingly different, and that the power dissipation results, in turn, will contain inaccuracies. However, Chapter 4 shows that the performance errors are relatively small or, if large, less relevant to the performance of the machine, so it is expected that the power dissipation errors will be correspondingly low, as shown in the next section.

One of the benefits of the synthetic testcases is their portability to multiple platforms [7]. Combined with overall runtimes that are two or three orders of magnitude faster than those of the original applications, the synthetic testcases are ideal for performance model validations using combinations of detailed execution-driven simulators, system simulators, RTL model simulators, hardware emulation systems, and hardware itself [7]. Likewise, the synthetic testcases are useful for *power model validation*. Simulators that assess dynamic power can be validated against slow RTL and circuit simulators with greater assurance that the validated simulator will give more accurate performance and power results for longer runs.

The next section compares the absolute and relative accuracy of the power dissipation of the synthetic testcases to those of longer running programs.

5.3 POWER SIMULATION RESULTS

In this section, the power dissipation results using the synthetic testcases from Chapter 3 are given.

5.3.1 Experimental Setup and Benchmarks

The SPEC 2000 and STREAM Alpha benchmarks and testcases described in Chapter 3 are used. As before, *sim-cache* from SimpleScalar release 3.0 is modified to carry out the workload characterization. To *sim-outorder* are added the event counters, the *power.h* includes, and *cacti* code from Wattch [15]. Wattch models the power dissipation of circuits and structures for a 0.35 micro, 600 MHz machine. Results are presented for an aggressive clock gating design with 10% leakage power [15].

As in Chapter 3, the Alpha binaries were executed in *sim-outorder* on the first reference dataset for the first billion instructions using the configuration in Table 3.5. While the machine configuration is relatively small, in the experiments below the window size and other machine parameters are varied significantly and good power dissipation correlations as IPC increases are still obtained. Another consideration is that this machine configuration is appropriate for use with the original Wattch model [34] [15][109]. It is also still a useful power model for smaller embedded or ASIC designs. The synthetic benchmarks were executed to completion in Wattch on an IBM p270 (400 MHz).

5.3.2 Base Power Dissipation Results

Figure 5.1 shows the power dissipation per cycle in Watts for the actual programs and the synthetics, which are uniformly lower. The average error is 6.8%, with a maximum error of 15% for *mcf*. The significant errors for dispatch window, register file, I-cache, D-cache, result bus, and clock are all low for the synthetics; the reason appears to be the uniformly lower window occupancy errors, with an average of 4.1%, shown in Figure 3.13. The SPEC INT synthetics exhibit larger average errors than the SPEC FP and STREAM, at 9.9% and 5.1% respectively, as shown in Table 5.1. Individual machine components generally show larger errors for SPEC INT, especially for the L1 I-cache and

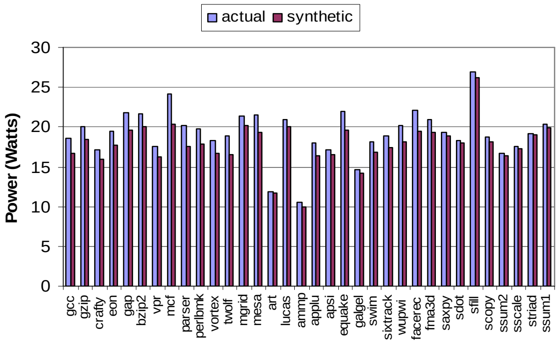


Figure 5.1: Power Dissipation per Cycle

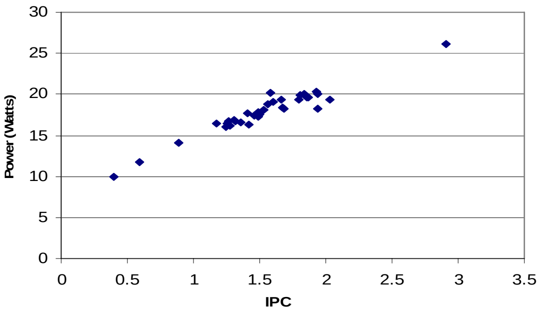


Figure 5.2: Power per Cycle vs. IPC for Synthetics

D-cache, the result bus, and the clock power. As mentioned in Section 5.2, the explanation for the cache errors is that the more complicated memory access behavior in SPEC INT is less likely to be well modeled by the simple synthetic streams, resulting in additional error in power dissipation in the I-cache and D-cache. This occurs in spite of the fact that IPC errors for synthetics with L1 cache miss rates above 1% are generally small [7] because the small miss rates have little impact on performance or are offset by errors in other parts of the memory subsystem [7]. However, these

Table 5.1: Average Power Prediction Error (%), Synthetics versus Benchmarks

Metric/Structure	% Error	%Error SPEC INT	%Error SPEC FP/STREAM	Max. %Error
Power per Cycle	6.8	9.9	5.1	15.0 (mcf)
Power per Instruction	4.4	5.4	3.9	11.2 (twolf)
Rename	2.4	2.8	2.1	7.4 (mcf)
Branch Predictor	3.7	4.3	3.3	15.8 (apsi)
Dispatch Window	5.3	7.1	4.3	12.5 (wupw)
LSQ	2.8	2.5	3.0	15.2 (applu)
Register File	6.4	4.5	7.4	22.1 (wupw)
L1 I-cache	6.0	11.0	3.2	14.1 (mcf)
L1 D-cache	5.8	7.3	4.9	18.5 (applu)
L2 cache	2.3	2.6	2.2	11.8 (applu)
ALU	1.8	1.7	1.8	5.2 (facerec)
Result Bus	6.6	10.9	4.3	16.1 (gcc)
Global Clock	12.4	17.7	9.5	27.0 (mcf)
Fetch	3.4	6.0	2.1	9.7 (mcf)
Dispatch Logic	2.4	2.8	2.1	7.4 (mcf)
Issue Selection Logic	3.6	4.9	2.9	8.2 (mcf)

results indicate that the SPEC INT could especially benefit from more accurate microarchitecture-independent memory access models.

The synthetics also have relatively low clock power dissipation versus the applications. A partial explanation is the uniformly lower dispatch window occupancies for the synthetics, which exhibit an average decrease in occupancy error of 4.1% [7]. Additional errors for the SPEC INT again point to the memory access model as a major contributor to the overall clock and result bus power dissipation errors. The large errors in many features for *mcf* mirror its relatively large IPC error, 7.4%.

Figure 5.2 is a scatter plot of the IPC versus the power per cycle for the synthetics. The correlation is quite good, with a Pearson correlation coefficient of 0.96. Table 5.2 gives correlation coefficients for the power dissipation metrics and the various machine features for both the actual programs and synthetics. The synthetic correlation coefficients generally follow those of the actual programs.

The results of Table 5.2 generally confirm the findings in prior microarchitectural

Table 5.2: Correlation Coefficients of Power Dissipation versus IPC

Metric/Structure	**Actual**	**Synthetic**
Power per Cycle	0.94	0.96
Power per Instruction	-.84	-.84
Rename	0.99	0.99
Branch Predictor	0.65	0.62
Dispatch Window	0.96	0.97
LSQ	0.30	0.22
Register File	0.77	0.75
L1 I-cache	0.91	0.96
L1 D-cache	0.43	0.46
L2 cache	0.016	-0.033
ALU	0.83	0.81
Result Bus	0.90	0.94
Global Clock	0.91	0.95
Fetch	0.88	0.92
Dispatch Logic	0.99	0.99
Issue Selection Logic	0.90	0.90

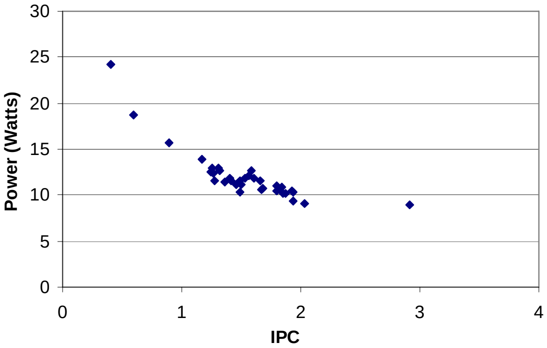


Figure 5.3: Power per Instruction vs. IPC for Synthetics

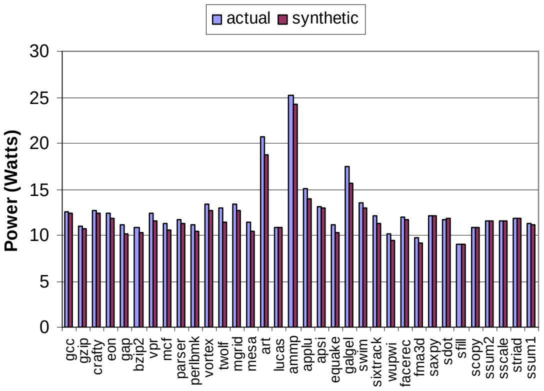


Figure 5.4: Power Dissipation per Instruction

power simulations [54][10]. For many features the correlation is greater than 0.90. However, there are some important exceptions. The LSQ, L1 D-cache, and L2 D-cache power dissipations appear not to be correlated with IPC, at least over the instructions executed. Also, the branch predictor, register file, and ALU power dissipations are only weakly correlated. IPC can generate a good rough estimate of overall power dissipation, especially for the pipeline structures, but additional analysis is necessary to estimate the power dissipation for many machine structures, especially the data cache hierarchy.

Figure 5.3 shows that the power dissipation per instruction decreases as IPC increases. The power dissipation of unused structures is 10% of maximum in the conditional clocking scheme used in Wattch, and as IPC increases the unused structure overhead is amortized over more instructions per cycle, but it never reaches zero. In Table 5.3, a negative 0.84 correlation for both actual programs and synthetics is found because the curve is non-linear, and the minimum power appears to be asymptotic to a line below 9 Watts as IPC increases in Figure 5.3. This trend would extrapolate to a maximum power of about 36 Watts for this four-issue machine.

Figure 5.4 breaks down the per instruction power dissipation by benchmark. The average error from Table 5.1 is 4.4%, with a maximum error of 11.3% for *twolf*. At 5.4%, the error in power per instruction is more evenly distributed for SPEC INT than is power per cycle, while SPEC FP and STREAM give 3.9% error.

5.3.3 Analysis of Design Changes

The power dissipation for design changes using the same synthetic workloads is now studied; that is, the testcases described in the last section are re-executed with changes to the machine configurations in Wattch, and the results are compared to executions of the actual programs on the same configurations.

Table 5.3 gives information for the absolute and relative IPC prediction errors [27] and the absolute and relative power dissipation prediction errors when executing

Table 5.3: Average Absolute and Relative IPC and Power Dissipation Error

	IPC					**Power Dissipation**						
Design Change	%Error	Max %Err	Rel. %Err	Max Rel. %Err	%Change	%Error	Max %Err	Rel. %Err	Max Rel. %Err	%Change	Max Avg. Structure %Err(strc)	Max %Err Structure (synth)
Disp 8 LSQ 4	2.8	16.1	2.3	7.5	23.8	4.9	10.9	2.5	6.6	16.8	9.2(clk)	21.2 (gap)
Disp 32 LSQ 16	3.7	9.9	2.2	9.4	16.1	8.5	18.5	1.8	8.8	18.5	14.3(clk)	28.9 (mcf)
Disp 48 LSQ 24	5.0	12.3	3.9	14.3	24.6	9.2	17.9	2.9	10.9	31.7	15.4(clk)	32.4(applu)
Disp 64 LSQ 32	6.1	18.1	5.2	20.8	31.0	9.5	18.4	3.6	13.7	41.5	16.0(clk)	36.6(applu)
Disp 96 LSQ 48	8.3	29.3	7.6	32.2	43.2	9.9	22.4	4.8	18.9	63.4	17.1(clk)	41.4(applu)
Disp 128 LSQ 64	9.1	34.0	8.5	37.0	52.0	10.0	23.3	5.2	22.7	82.8	17.7(clk)	43.4(applu)
Issue Width 1	1.5	5.3	1.9	6.5	53.6	1.6	3.2	5.9	14.9	38.2	5.8(rbus)	16.9(fma3d)
Issue Width 8	3.0	9.4	1.6	9.2	7.0	6.3	13.1	1.0	7.3	11.3	11.4(clk)	26.7(wupw)
Commit Width 8	2.9	9.1	1.4	8.9	7.0	5.4	11.9	2.3	11.4	42.5	9.9(clk)	24.3(wupw)
Machine Width 2	2.8	7.6	2.0	6.5	23.5	5.4	10.9	1.7	5.6	17.6	10.4(regf)	26.8(wupw)
Machine Width 6	3.2	9.8	1.0	7.6	5.6	6.9	15.6	0.68	4.4	9.23	12.5(clk)	16.9(mcf)
Machine Width 8	3.2	8.8	1.1	7.6	6.0	6.6	13.7	0.79	6.2	14.5	11.9(clk)	24.4(wupw)
Machine Width 10	3.1	9.2	1.5	9.1	6.5	6.0	12.5	1.2	8.1	20.1	11.1(clk)	24.7(wupw)
Machine Width 12	3.1	9.3	1.5	9.1	6.6	5.6	11.7	1.6	9.4	25.7	10.5(clk)	25.0(wupw)
Machine Width 14	3.1	9.4	1.5	9.2	6.6	5.2	11.6	2.0	10.8	30.2	9.8(clk)	25.3(wupw)
Machine Width 16	3.2	9.4	1.5	9.2	6.7	4.7	11.2	2.6	12.1	34.1	10.0(Icac)	26.7(wupw)
IFQ 8	3.0	9.3	1.2	7.7	5.4	4.6	10.8	2.6	11.5	33.2	8.9(clk)	25.5(wupw)
IFQ 16	3.2	9.4	1.5	9.2	6.7	4.8	11.2	2.5	12.1	34.2	10.0(regf)	26.7(wupw)
IFQ 32	3.3	9.0	1.3	8.9	7.4	5.0	11.3	2.2	11.8	34.9	9.7(Icac)	26.7(wupw)
Caches 0.25x	19.4	49.8	18.8	48.8	15.2	14.1	34.8	9.2	28.5	22.8	18.6(rbus)	45.0(wupw)
Caches 0.50x	23.9	46.7	23.1	47.0	7.0	17.6	33.2	11.9	26.7	13.2	23.0(wind)	45.5(fma3d)
Caches 2.0x	4.2	16.0	3.2	18.8	4.3	6.5	15.6	1.6	7.6	18.7	12.7(clk)	28.1(mcf)
Caches 4.0x	4.9	17.8	3.9	12.6	6.2	9.2	19.0	3.1	8.4	40.5	15.8(clk)	32.1(twolf)
L1 I-cache 2.0x	3.0	10.7	1.3	5.9	7.0	8.1	18.6	2.0	6.9	36.5	14.5(clk)	35.1(sfill)
L1 D-cache 2.0x	3.2	14.3	1.2	12.8	2.2	7.5	16.6	1.1	4.1	30.9	13.3(clk)	31.0(mcf)
L1 D-cache Lat 8	9.7	38.0	9.9	34.8	22.0	2.6	7.2	6.4	21.3	13.4	6.7(regf)	19.4(perlb)
BPred 0.25x	2.9	8.6	1.3	6.0	0.63	6.4	14.3	0.67	2.7	0.40	12.0(clk)	25.9(mcf)
BPred 0.50x	2.5	8.6	0.60	3.0	0.18	6.7	15.4	0.31	1.4	0.22	12.2(clk)	26.7(mcf)
BPred 2.0x	2.6	8.3	0.32	1.8	0.15	6.9	15.5	0.18	0.88	0.20	12.4(clk)	26.9(mcf)
BPred 4.0X	2.5	8.0	0.40	2.0	0.22	6.9	15.2	0.21	0.87	0.52	12.4(clk)	26.7(mcf)

various design changes. Included here are the IPC data from Chapter 3 to ease discussion. The only differences with those results are for the studies of machine width, issue and commit width and IFQ size. For each of those, the other parameters related to machine width are held at 16, instead of 4 in Chapters 2 and 3, to give larger changes in IPC as the parameter changes.

In the table, *Disp* is the dispatch window size. A change in *Machine Width* implies that the decode width, issue width and commit width all change by the same amount from the base configuration in Table 3.5. When the caches are increased or decreased by a factor, the number of sets for the L1 I-cache, D-cache and L2 cache are increased or decreased by that factor. Likewise, when the bimodal branch predictor is multiplied by a factor, the table size is multiplied by that factor from the default size. The *L1 D-cache 2x* and *L1 I-cache 2x* specify a doubling of the L1 D-cache (to 256 sets, 64B cache line, 8-way set associativity), and a doubling of the L1 I-cache configuration (to 1024 sets, 64B cache line, 2-way set associativity). The numbers here do not include *ammp* or *galgel*. Those benchmarks tend to be optimistic when the dispatch window changes because, as mentioned in Chapter 3, the small data footprint benchmarks do not model data-TLB misses, and the actual programs have large TLB miss rates, over 13% each.

A commit width change from 4 to 1 was also studied, but the data for some actual programs is inconsistent, so it is not used. Speculation is that there is a power modeling error when the commit width is reduced to one. Evidence that the modeling is incorrect is given in Table 5.4, where most correlation coefficients for IPC and power dissipation are above 0.90, except when the commit width equals one. The commit width equal one results are not examined further in this paper.

The other correlation coefficients in Table 5.4 indicate that the power dissipation achieved by the synthetics for a design change follows fairly well the IPC change for that design change. Table 5.3 shows that the absolute and relative IPC errors and power dissipation errors *(%Error* and *Rel. %Error*) due to a design change for the synthetics versus the actual programs is often below 5% or 10%, except for the two cases in which the caches are reduced in size. The synthetics underestimate performance when the cache is significantly reduced due to capacity misses among the synthetic data access streams [6].

There are four classes of results in Table 5.3 that shed light on the quality of the power dissipation analysis using the synthetics and provide a starting point for discussion. One is concerned about whether the synthetic testcases properly indicate power dissipation changes when the IPC changes significantly:

Class 1: The change in IPC and the change in power dissipation (*%Change* in Table 5.3) are greater than two times (or more) their absolute or relative errors for the design change.

Class 2: The change in IPC is greater than two times its absolute or relative error, but the change in power dissipation is not.

Class 3: The change in power dissipation is greater than two times its absolute or relative error, but the change in IPC is not.

Class 4: Neither the change in IPC nor the change in power dissipation is greater than two times its error.

The classes for each design change are given in Table 5.4. The choice of threshold equal to *2x* is *ad hoc*, but it gives a good cushion between the average errors and the average change that is being indicated. Evidence that that is a good metric is given by the max absolute and relative error columns (*Max %Err* and *Max Rel. %Err*) for the design

Table 5.4: Correlation Coefficients of Power vs. IPC for Design Changes and Quality of Assessing Power Dissipation Changes (Class)

Design Change	Actual	Synthetic	Class
Disp 8 LSQ 4	0.95	0.97	1
Disp 32 LSQ 16	0.92	0.95	1
Disp 48 LSQ 24	0.91	0.94	1
Disp 64 LSQ 32	0.90	0.93	1
Disp 96 LSQ 48	0.92	0.93	1
Disp 128 LSQ 64	0.93	0.94	1
Issue Width 1	0.87	0.91	1
Issue Width 8	0.95	0.97	1
Commit Width 1*	-0.27	0.74	N/A
Commit width 8	0.95	0.95	1
Machine Width 2	0.92	0.94	1 (borderline 3)
Machine Width 6	0.95	0.97	1 (borderline 3)
Machine Width 8	0.95	0.97	1
Machine Width 10	0.95	0.97	1
Machine Width 12	0.95	0.96	1
Machine Width 14	0.95	0.96	1
Machine Width 16	0.95	0.95	1
IFQ 8	0.95	0.95	1 (borderline 3)
IFQ 16	0.95	0.95	1 (borderline 3)
IFQ 32	0.95	0.95	1 (borderline 3)
Caches 0.25x	0.98	0.99	4
Caches 0.50x	0.96	0.99	4
Caches 2.0x	0.92	0.96	3
Caches 4.0x	0.89	0.95	3
L1 I-cache 2.0x	0.94	0.98	1
L1 D-cache 2.0x	0.82	0.89	3
L1 D-cache Lat 8	0.97	0.97	1
BPred 0.25x	0.94	0.96	4
BPred 0.50x	0.94	0.96	4
BPred 2.0x	0.94	0.96	4
BPred 4.0X	0.94	0.96	4

changes. Generally the max errors are less than or not too far removed from the *%Change*. The *%Change* is the minimum change for either the actual or synthetic workload. For all cases the *%Change* for either is close to that of the other.

Most of the design changes are class 1 (or borderline class 3) and none of the design changes are class 2, which indicates that significant power dissipation changes can be assessed when the IPC changes significantly. The class 3 and borderline class 3 design changes indicate that the correct power dissipation changes are reflected even though the IPC changes may not be significant. In these cases, the correlation coefficients between IPC and power dissipation are lower.

The class 4 design changes indicate that the changes in IPC and power were not much different from the errors in the synthetics themselves. As mentioned, the reduced cache sizes put pressure on the memory access streams used in the synthetics, causing large percent errors. For the branch predictor studies, the percent errors are small, but the increase or decrease in the bimodal predictor table causes little change in performance (*%Change*), whether for the actual or synthetic workloads. A different predictor configuration is needed to assess the quality of predictor design changes for the synthetics.

Table 5.3 also gives the maximum average percent power dissipation error from among all the structures listed in Table 5.2 for each design change (*Max Avg. Structure %Err*). Similar to the absolute structure errors in the last section, the most prominent error is the global clock (*clk*), but the errors average only 12.5%, and for the strong class 1 design changes they are well below the average change in power dissipation.

The table also shows the maximum error found among all the synthetics for any particular structure for the design change (*Max %Err Structure Synthetic*). This is usually the global clock structure. Since these are much larger than the average structure errors (*Max Avg. Structure %Err*), these particular points are outliers.

The results indicate that the synthetics for *mcf*, *applu*, and *wupwise* have predicted power dissipations that are more variable than those of many others when the design changes. Chapter 3 shows that *mcf* has a relatively low IPC compared to the original workload which translates to lower power dissipation. It also has large I-cache power underestimation of 14% and clock power underestimation of 27%. *Applu* has an underestimated I-cache miss rate of 32% and a clock power underestimation of 19%. All three workloads have the largest register file power underestimations, the largest being *wupwise* at 22%. Other benchmarks have errors as evidenced in Figure 5.1, but these large errors set these three up to be outliers in the power studies. Ultimately the errors in power estimation trace back to the inaccuracies in the synthesis process described in Chapter 4.

5.4 SUMMARY

This chapter shows that synthetic workloads from Chapter 3 can rapidly and accurately assess the power dissipation of real programs. Synthetic versions of the SPEC 2000 and STREAM benchmarks predict the total power per cycle to within 6.8% error on average, with a maximum of 15% error, and total power per instruction to within 4.4% error. Since the testcases execute orders of magnitude fewer instructions while maintaining accuracy, performance and power model validations using more realistic tests are feasible.

In addition, for many design changes for which IPC and power change significantly, the synthetic workloads show small errors, many less than 5%. Also, simulated power dissipation for both applications and synthetics correlates well with the IPCs of real programs, often giving a correlation coefficient greater than 0.9. This confirms prior results that demonstrate a good correlation between IPC and power dissipation for simulated processors and hardware performance counters, and it verifies that the synthetic testcases produce similar results.

The next chapter extends workload synthesis to the PowerPC instruction set architecture. A case study of performance model validation for the POWER5 chip is presented.

Chapter 6: Performance Model Validation Case Study for the IBM POWER5 Chip

The previous chapters propose synthetic workloads for early design studies and model validation. The SimpleScalar performance simulator was used for the investigations, and the machine configuration was chosen for direct comparison of results to prior work [71][15]. While correlation was often good as the machine configuration was changed, the base configuration is relatively small, at least with respect to the dispatch window. The base dispatch window was set to 16, versus windows of more than 100 for modern processors [85]. Also, SimpleScalar does not model an L3, so the memory latency of the base configuration is shortened to compensate. The configuration is still a useful model for smaller machines, embedded designs and ASICs. However, the configuration begs the question of how synthesis does for modern designs.

In this chapter, the synthesis methodology from Chapter 3 is augmented to support the PowerPC instruction set architecture and the POWER5 chip. Additional memory access models and synthesis parameters are added to help create representative PowerPC versions of the SPEC 2000, STREAM, TPC-C and Java benchmarks. Specific examples of performance model validation and analysis are presented. An average IPC error of 2.4% versus the original benchmarks is obtained.

6.1 INTRODUCTION TO THE POWER5 CHIP

Modern high-performance microprocessors are quite complex. For example, the POWER4 and POWER5 chips are dual-core PowerPC microprocessors used in IBM high-end server systems [93][85]. The POWER4 chip is built from 1.5 million lines of VHDL and 174 million transistors [56], and the POWER5 chip contains 276-million transistors [85].

This complexity poses challenges to performance modeling, model simulation, and performance model validation efforts. As complexity increases, the gap in accuracy can grow quickly, so validation is needed more frequently. While the relative error of design changes based on inaccurate performance models is often similar to the relative error using accurate models, subtle instruction interactions in the POWER4 and POWER5 chips necessitate very accurate performance models.

To achieve accurate performance projections, the performance models must be written at a very detailed level, which reduces the efficiency of the model simulation. The detail also exacerbates the problem of performance model validation, which seeks to execute codes and compare results between performance models and hardware or functional models built from hardware descriptions of the machine. As described in Chapter 3, the current state-of-the-art is to use simple hand-coded bandwidth and latency testcases, but these are not comprehensive for most processors, especially processors as complex as the POWER5 chip. Applications and benchmark suites such as SPEC CPU are more difficult to set up or take too long to execute on detailed performance models.

In this chapter, the synthesis effort of Chapter 3 is broadened to support high-performance PowerPC processors such as the POWER4 and POWER5 chips. The specific contributions of this chapter are the following:

i) The synthesis system is extended to include the PowerPC ISA target.

ii) Current synthesis techniques are extended and new techniques are presented that are necessitated by the features of the POWER5 chip, including new memory access models.

iii) Two performance model validation approaches with validation results are presented using the POWER5 synthetics.

The rest of this chapter is organized as follows. Section 6.2 gives a quick overview of synthesis and model validation for IBM PowerPC processors. Section 6.3 describes the extensions to synthesis to support the PowerPC ISA. Section 6.4 presents experimental synthesis results for the POWER5 processor. Section 6.5 presents validation results for a follow-on PowerPC processor using the POWER5 testcases. The last section presents a summary.

6.2 IBM POWERPC SYNTHESIS AND MODEL VALIDATION

Just as in Chapter 3, representative testcase synthesis is achieved based on the workload characterization and statistical flow graph of statistical simulation [27][4]. A fast profiler is incorporated into the IBM performance tools, and synthesis is carried out using the resulting profile. The microarchitecture-independent workload characteristics are combined with memory access and branching models to build a representative synthetic tsetcase.

In the case of the IBM PowerPC processors, executions of a cycle-accurate model built directly from the functional VHDL hardware description language model [101][56] can be compared against the detailed M1 performance model [44][41] used for performance projections.

6.2.1 The POWER5 M1 Performance Model

The IBM PowerPC performance modeling environment is trace-driven to reduce modeling and simulation overhead [66][51][41]. Traces are collected from executions on real machines. The M1 performance model implements a detailed, cycle-accurate core. Coupled with the M1 is a realistic model of the L2, L3 and memory [41].

The M1 captures the functional details of the processor. The POWER5 chip features out-of-order instruction execution, two fixed point and two floating point units, 120 general purpose rename registers, 120 floating point rename registers, complex pipeline management, 32-entry load and store reorder queues, on-board L2 controller and 1.9-MB cache, L3 cache controller, 36-MB off-chip victim L3, memory controller, SMP fabric bus interface, dynamic power management, and support for simultaneous multithreading [85]. Table 6.7 gives additional configuration parameters.

6.2.2 PowerPC Performance Model Validation

The POWER5 M1 performance model was validated by hand to within 1% for particular microbenchmarks [41]. However, executing more realistic code traces produces larger errors for the M1, as was found in the literature [23]. The M1 often needs to be validated against codes that are more representative of actual programs. However, it is a difficult and time consuming proposition to validate a detailed model on realistic programs. Automatic testcase synthesis seeks to automate the synthesis process and synthesize high level codes that can target various platforms. In this work, two platforms for validating the performance model are targeted: cycle-accurate RTL model simulation and simulation on a hardware emulator.

6.2.2.1 Validation using RTL Simulation

Figure 6.1 shows the RTL validation methodology. The synthetic testcase is compiled and converted to execute on a VHDL model using the standard IBM functional verification simulation methodology [101][56]. The converted code is then executed to completion on the VHDL model. The converted testcase is also unrolled into a trace using a PowerPC instruction interpreter [66][51][12]. Only completed instructions are maintained in the trace. The trace is then executed to completion on the M1 performance model.

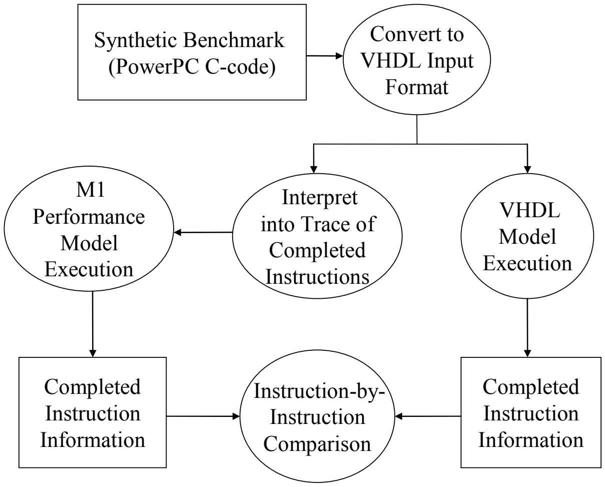

Synthetic Benchmark (PowerPC C-code)
Convert to VHDL Input Format
M1 Performance Model Execution
Interpret into Trace of Completed Instructions
VHDL Model Execution
Completed Instruction Information
Instruction-by-Instruction Comparison
Completed Instruction Information

Figure 6.1: RTL Validation Methodology using Synthetic Testcases

Both VHDL and performance model executions generate information at the level of the instruction, including the cycle the instruction completes (write back register result), address and opcode. The cycles at which an instruction completes in both the VHDL and performance models are not identical in an absolute sense because of how cycles are maintained and counted in the two simulators. However, the performance model is of sufficient detail such that the completion cycle of an instruction minus the completion cycle of the previous instruction, that is, the cycle difference between any two completing instructions, should be equivalent.

The analysis relies on the fact that the methodology generates instructions for both models from a compilation of a single piece of code. Each and every instruction is executed on both the VHDL and M1 models and completes in the same order in both. Instructions may complete in different completion buffers in the same cycle, but should complete in the same cycle.

The *instantaneous error*, *E(i),* for each *i* instruction is defined in terms of the difference in cycles between the completion time of the current instruction and the completion time of the previous instruction in both VHDL and M1 models. The instantaneous error for the ith instruction is defined as:

$$E(i) = D_R(i) - D_P(i)$$

where

$$D_R(i) = C_R(i) - C_R(i-1)$$

and likewise

$$D_P(i) = C_P(i) - C_P(i-1)$$

In these equations, $C_R(i)$ is the completion cycle of the ith instruction when executing the VHDL (RTL) model, and $C_P(i)$ is the completion cycle of the ith instruction when executing the M1 (Performance) model. The intuition behind the *E* calculation is that an instruction is using the same machine resources and completing in the same order, putatively in the same cycle, in both models, so differences between sequential instructions should be identical. A difference indicates that resource allocations or execution delays are not modeled similarly. Note that instruction dependences and resource usages using the same workload in both models will limit the instantaneous error and push it toward zero; there can be no large accumulation of error between any two related instructions. However, the completion time of an instruction that is modeled properly may be underestimated if an older instruction in program order fails to complete on time and the younger instruction is not dependent on the older but on a prior instruction and has been ready to complete for some time. It is therefore advisable to first analyze the modeling errors of instructions with larger instantaneous errors.

The instantaneous errors can be categorized to narrow down the search for microarchitectural bugs in which a hardware feature was not implemented properly, and also to find modeling, abstraction, and specification errors [11] in the performance model. Section 6.5 gives an example model validation analysis.

6.2.2.2 Validation using a Hardware Emulator

The compiled synthetic testcases can also be input to an RTL model executing on the AWAN hardware accelerator [56]. AWAN is composed of programmable gate arrays and can be configured to emulate a VHDL model. The AWAN array supports very large chips. The entire POWER4 chip, for example, can be emulated at a rate of more than 2500 cycles per second [56]. The cycle counts for a run are obtained from AWAN registers and can be compared to the M1 performance model cycle counts. Detailed execution information can also be obtained. In Section 6.5, validation results using AWAN are presented.

6.3 SYNTHESIS FOR POWERPC

Following Chapter 3, the four major phases of synthesis are described, but particular attention is paid to additions for the PowerPC ISA and high-performance chips like the POWER5. Recall that the four phases are *workload characterization*, *graph analysis*, *register assignment* and *code generation*.

6.3.1 Workload Characterization

The workload characterization for the PowerPC ISA is similar to the characterization for the Alpha and Pisa ISAs in Chapter 3. The M1 was augmented with the same workload characterization code used for the other ISAs. The dynamic execution of a benchmark trace in the M1 produces a set of workload characteristics. Instructions are still abstracted into five classes plus sub-classes, except that the floating-point instructions have three sub-classes: short execution times, such as for a floating-point move or add; long execution times, as for a floating-point multiply or divide; and extremely long execution times, only for the PowerPC *fma* operations. As usual, the

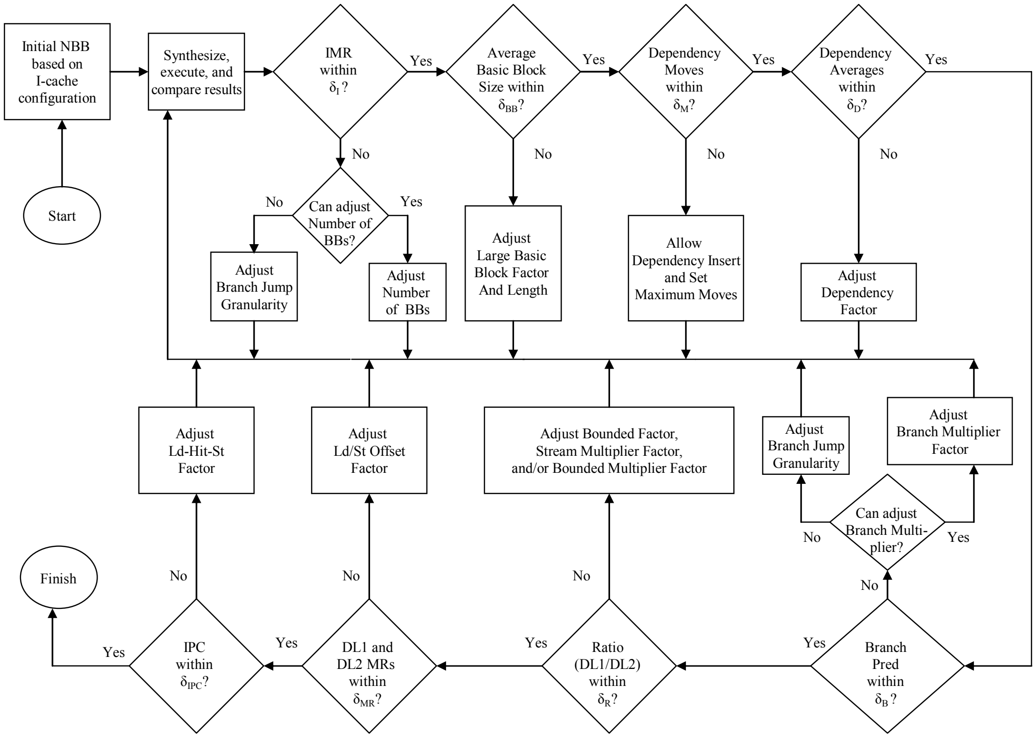

Initial NBB based on I-cache configuration
Start
Synthesize, execute, and compare results
IMR within δ_I?
Yes
No
Can adjust Number of BBs?
No
Yes
Adjust Branch Jump Granularity
Adjust Number of BBs
Average Basic Block Size within δ_{BB}?
Yes
No
Adjust Large Basic Block Factor And Length
Dependency Moves within δ_M?
Yes
No
Allow Dependency Insert and Set Maximum Moves
Dependency Averages within δ_D?
Yes
No
Adjust Dependency Factor
Adjust Ld-Hit-St Factor
Adjust Ld/St Offset Factor
Adjust Bounded Factor, Stream Multiplier Factor, and/or Bounded Multiplier Factor
Adjust Branch Jump Granularity
Adjust Branch Multiplier Factor
Can adjust Branch Multi-plier?
No
Yes
No
Finish
No
No
No
Yes
IPC within δ_{IPC}?
Yes
DL1 and DL2 MRs within δ_{MR}?
Yes
Ratio (DL1/DL2) within δ_R?
Yes
Branch Pred within δ_B?

Figure 6.2: Flow Diagram of Graph Analysis and Thresholds for PowerPC Synthesis

workload characterization process results in a statistical flow graph which is traversed to produce a set of synthetic basic blocks [27][4].

6.3.2 Graph Analysis

The graph analysis phase has some significant additions. Figure 6.2 shows a flow diagram of graph analysis. The diagram represents one pass through synthesis, at each stage of which a workload characteristic is checked against a threshold. The thresholds are defined in Table 6.1 and typical tolerances for them, determined experimentally, are given. If a characteristic is above a threshold after a synthesis pass, an adjustment is made to a parameter, or *factor*, that putatively changes a synthesis algorithm to meet the threshold, and synthesis is repeated.

In addition to the usual synthesis information, the factors in Table 6.6 and additional factors are described in the following sections, as well as the final properties of the synthetic testcases in Table 6.5.

6.3.2.1 Instruction Miss Rate and I-cache Model

As in Chapter 3, the number of basic blocks to be instantiated in the synthetic testcase is estimated based on a default I-cache size and configuration. The number of synthetic basic blocks is then tuned to match the original I-cache miss-rate (*IMR)*. In the PowerPC experiments, it was noticed that the IMR can change after the branches are configured in the branching model (Section 6.3.2.5). To compensate, the *branch jump granularity* is adjusted to change the number of basic blocks or instructions that a configured branch jumps.

Usually a small number of synthesis iterations are necessary to match the IMR. The numbers of basic blocks and instructions synthesized for the PowerPC versions of the SPEC 2000 and STREAM benchmarks are shown in Table 6.5.

6.3.2.2 Instruction Dependences and Compatibility

All instruction input dependences are assigned. The starting dependence is exactly the dependent instruction chosen as an input during statistical simulation. If the dependence is not compatible with the input type of the dependent instruction, then

Table 6.1: Graph Analysis Thresholds and Typical Tolerance

Threshold	**Tolerance Definition**	**Typical Value**
δ_I	I-cache Miss Rate	1%
δ_{BB}	Basic Block Size	3%
δ_M	Dependence Moves	10
δ_D	Dependence Average	5%
δ_B	Branch Predictability	5%
δ_R	L1/L2 Miss Rate Ratio	5%
δ_{MR}	L1 or L2 Miss Rate	5%
δ_{IPC}	IPC	5%

another instruction must be chosen. The algorithm is to move forward and backward from the starting dependence through the list of instructions in sequence order until the dependence is compatible.

In the PowerPC experiments, some benchmarks have very high average numbers of moves away from the starting dependence. To compensate, after 25 moves, a compatible instruction is inserted into the basic block near the starting dependence. The total number of inserts for each benchmark is shown in the *dependence inserts* column of Table 6.5. The highest numbers are associated with *mgrid* and *applu*, which have the largest average basic blocks sizes, at 125 and 115 respectively. Inserts are also used for some codes with short average basic block sizes, like *gcc*, due to long sequences of branches without sufficient intervening instructions for proper dependences. Using the inserts, the average number of moves per instruction input, shown in column *dependence moves*, is reduced. In the case of a store or branch that is operating on external data for which no other instruction in the synthetic instructions is compatible, an additional variable of the correct data type is created.

Table 6.2 shows the compatibility of instructions for the PowerPC instruction set. The columns are the same as in Table 3.3. The PowerPC *fma* operations have three input operands. In some cases, the *dependence factor* in Table 6.6 is used to multiply the synthetic dependences to more closely match the overall averages in the original

Table 6.2: PowerPC Dependence Compatibility Chart

Dependent Instruction	Inputs	Dependence Compatibility	Comment
Integer	0/1	Integer, Load-Integer	
Float	0/1/2	Float, Load-Float	3 Inputs for *fma*
Load-Integer/Float	0	Integer	Memory access counter input
Store-Integer	0	Integer, Load-Integer	Data input
Store-Float	0	Float, Load-Float	Data input
Store-Integer/Float	1	Integer	Memory access counter input
Branch	0/1	Integer, Load-Integer	Condition registers

application. The dependence adjustment for *mgrid* was necessary due to its large average basic block size and, therefore, small number of synthetic basic blocks. This factor was not needed for the Pisa and Alpha syntheses. Note that branch operands specify condition code registers, not indirect jumps to register values. This causes errors in the synthetic branch dependences when the branching model is implemented (see Sections 6.3.2.5 and Figure 6.9) but there is minimal performance impact.

6.3.2.3 Loop Counters and Program Termination

As in Chapter 3, a search is made for an additional integer instruction that is attributed as the loop counter (*BRCNTR*). The branch in the last basic block in the program checks the BRCNTR result to determine when the program is complete. The number of executed loops, *loop iterations* in Table 6.5, is chosen to be large enough to assure IPC convergence. As for the other ISAs, PowerPC experiments show that if the product of the loop iterations and the number of instructions is around 300K instructions, low branch predictabilities and good stream convergence are achieved. An *mtspr* instruction initializes the internal count register to the loop iterations, and the final branch checks for zero in the count register and decrements it.

6.3.2.4 Memory Access Model

As for the other ISAs, the LSCNTR instructions are assigned a stride based on the D-cache hit rate found for their corresponding load and store accesses during workload characterization. The memory accesses for data are modeled using the stream access classes shown in Table 6.3. The stride assignment for a memory access is determined first by matching the L1 hit rate of the load or store fed by the LSCNTR, after which the L2 hit rate for the stream is predetermined. The first two rows are only useful for stores in store-through POWER5 machines. Stores in the L2 gather such that a simple traversal through memory results in a 50% L2 hit rate. If the L1 hit rate is below 3.17%, the L2 hit rate is matched.

The *L1 Hit Rate* for both loads and (non-zero hit rate) stores is based on the line size of 128 bytes in the POWER4/5 chip:

$$L1_{HitRate} = 1 - (stride\ / 128) \cdot 4$$

where the stride is given in increments of 4 bytes.

By treating all memory accesses as streams and working from a base cache configuration, the memory access model is kept simple. This reduces additional impact on the testcase instruction sequences and dependences, which have been shown to be

Table 6.3: L1 and L2 Hit Rate versus Stride (PowerPC)

L1 Hit Rate	**L2 Hit Rate**	**Stride**
0.0000	1.00	0
0.0000	0.50	1
0.0000	0.00	32
0.0313	0.00	31
0.0625	0.00	30
0.0942	0.00	29
0.1250	0.00	28
0.1563	0.00	27
0.1875	0.00	26
0.2188	0.00	25
0.2500	0.00	24
0.2813	0.00	23
0.3125	0.00	22
0.3438	0.00	21
0.3750	0.00	20
0.4063	0.00	19
0.4380	0.00	18
0.4688	0.00	17
0.5000	0.00	16
0.5313	0.00	15
0.5625	0.00	14
0.5938	0.00	13
0.6250	0.00	12
0.6563	0.00	11
0.6875	0.00	10
0.7188	0.00	9
0.7500	0.00	8
0.7813	0.00	7
0.8125	0.00	6
0.8438	0.00	5
0.8750	0.00	4
0.9063	0.00	3
0.9375	0.00	2
0.9688	0.00	1
1.0000	n/a	0

important for correlation with the original workload [4]. On the other hand, there can be a large error in stream behavior for two reasons. An actual L1 hit rate may fall between the hit rates in two rows, but for the PowerPC configuration this maximizes to only about 3% error. A larger error is associated with the lack of distinguishing L2 hit rate quanta. Since the L1 and L2 line sizes are the same in the POWER5 machine, it is difficult to get positive L2 hit rates with simple stride models.

Consequently, traversals through particular cache congruence classes are implemented. They are called *bounded streams* to differentiate them from streams that continually increment through memory, i.e. unbounded streams. The implementation makes use of the default 4-way set associative L1 and 10-way set associative L2 in the machines under study. The difference in associativity means that traversals through a class will hit in the L2 but miss in the L1 to the extent that the entire class is traversed. If the L2 hit rate is greater than the L1 hit rate multiplied by the *bounded factor* in Table 6.6, then the stream in the basic block is changed from a simple stride stream (*stream pools* in Table 6.5) to a congruence class traversal (*bounded stream pools* in Table 6.5).

To achieve the particular L1 and L2 hit rates in a row of Table 6.4, the *instruction reset* column gives the total number of 8K accesses that are necessary before starting over and repeating the same access sequence in the congruence class. In equation form:

$$L1_{HitRate} = \begin{cases} 1.0, & 1 \le reset < 4 \\ (8 - reset)/reset, & 4 \le reset \le 8 \\ 0, & 8 < reset \le 20 \end{cases}$$

$$L2_{HitRate} = \begin{cases} 1.0, & 0 \le reset < 11 \\ (20 - reset)/reset, & 11 \le reset \le 20 \end{cases}$$

Note that, for studies of cache size design changes, congruence class traversals essentially clamp the hit rates to a particular level, since the rates will not change unless the associativity changes. The effect of the use of this factor is to adjust the ratio of the L1 and L2 hit rates to more closely match that of the original application. This is also shown in Figure 6.2.

In some cases, additional manipulation of the streams was necessary to correlate the testcases because of the cumulative errors in stream selection. In Table 6.6, the stream factor multiplies the moving average of the L1 hit rate taken from the table during each lookup, and if the result is greater than the original hit rate by (N·10%), the selected stream is chosen from the preceding (N+1)st row. This has the effect of reducing overall hit rates for the first load or store fed by an LSCNTR. Similarly for the bounded streams, the *bounded stream factor* in Table 6.6 multiplies the L1 hit rate.

For the Pisa and Alpha syntheses, the *miss rate estimate factor* in Table 3.3 was implemented to estimate and modify the basic block miss rate [5][7]. This was not needed for the PowerPC synthesis, but additional related factors were added. The *load-store-offset* factor in Table 6.6 changes the address offset of loads and stores to a value from one to 8K based on a uniform random variable. Interestingly, the factor value usually has a proportional effect on cache miss rates *and* IPC because of the random access but fewer load-stores address collisions. The *load-hit-store factor* changes the

Table 6.4: L1 and L2 Hit Rates for Reset Instruction Number (Congruence Class Walks)

L1 Hit Rate	**L2 Hit Rate**	**Instruction Reset**
1.0000	1.0000	4
0.6000	1.0000	5
0.3333	1.0000	6
0.1429	1.0000	7
0.0000	1.0000	8
0.0000	0.8182	11
0.0000	0.6667	12
0.0000	0.5385	13
0.0000	0.4286	14
0.0000	0.3333	15
0.0000	0.2500	16
0.0000	0.1765	17
0.0000	0.1111	18
0.0000	0.0526	19
0.0000	0.0000	20

number of stores that have the same word address offset as loads. The factor value has an inversely proportional effect on IPC. Both factors are shown as knobs in the flow diagram of Figure 6.2. An additional simple way to increase both L1 and L2 misses is implemented by configuring a fraction of non-bounded streams to stride by a fraction of a 4KB page. *Mcf* configures three streams to walk with stride equal to a page, and *art* and *java* configure one stream to traverse 0.8 and 0.6 of a page, respectively.

Ideally, these last few factor values would be based on a characterization of the workload, but load-hit-stores, load-offsets and page walks were not analyzed in the

Table 6.5: Synthetic Testcase Properties for the POWER5 chip

Name	**Number of Basic Blks**	**Number of Instructions**	**Loop Iterations**	**Stream Pools**	**Bounded Stream Pools**	**Code Registers**	**Dependence Moves**	**Dependence Inserts**	**Runtime Ratio**
gcc	750	2524	80	5	3	12	6.093	60	**437.58**
gzip	840	3683	119	4	4	12	0.465	1	**481.1**
crafty	360	3699	56	5	3	12	0.797	4	**718.11**
eon	330	3879	41	3	5	12	3.113	40	**1181.51**
gap	510	3940	65	4	4	12	0.33	0	**954**
bzip2	300	1859	144	5	5	10	0.418	0	**562.62**
vpr	400	2855	121	7	3	10	0.648	13	**1051.48**
mcf	800	3561	71	7	3	10	0.649	0	**495.19**
parser	795	4013	54	8	2	10	0.833	0	**1113.85**
perlbmk	600	3834	55	9	1	10	1.95	0	**998.11**
vortex	500	2417	90	2	8	10	0.889	0	**994.29**
twolf	540	3952	71	3	7	10	0.596	1	**1190.29**
mgrid	30	4008	65	8	2	10	1.632	255	**1050.13**
mesa	400	3362	81	5	3	12	1.32	23	**1123**
art	200	4213	46	6	2	12	1.4	228	**902.18**
lucas	80	2367	141	3	5	12	1.915	0	**872.08**
ammp	200	1700	160	4	4	12	6.608	0	**749.18**
applu	30	3851	63	6	2	12	1.204	272	**378.77**
apsi	200	3208	70	8	0	12	4.585	0	**345.54**
equake	50	2459	71	7	1	12	9.499	0	**700.57**
galgel	120	3868	53	6	2	12	11.225	0	**583.31**
swim	70	3468	71	4	4	12	1.769	85	**1079.23**
sixtrack	150	2624	144	5	2	12	1.12	0	**494.55**
wupwise	200	2756	69	5	3	12	11.095	0	**1258.9**
facerec	200	2530	113	5	3	12	3.982	0	**616.75**
fma3d	150	3596	49	6	2	12	5.594	0	**445.68**
saxpy	1	8	33334	2	0	12	0	0	**28.27**
sdot	1	6	50001	2	0	12	0.125	0	**71.85**
sfill	1	3	100001	1	0	12	6.25	0	**22.47**
scopy	1	6	50001	2	0	12	0	0	**61.75**
ssum2	1	4	100001	1	0	12	0.2	0	**18.67**
sscale	1	7	50001	2	0	12	0	0	**23.23**
striad	1	9	33334	3	0	12	0	0	**27.16**
ssum1	1	9	33334	3	0	12	0	0	**26.97**
tpc-c	4500	23102	12	2	8	10	0.571	0	**447.77**
java	4750	23391	14	3	7	10	0.416	0	**447.59**

current version of the code and are left as future work. As discussed in Chapter 3, more complicated models might move, add, or convert instruction types to implement more realistic access functions. There are also many access models in the literature that can be investigated as future work. Usually a small number of synthesis iterations are needed to find a combination of factors to model the overall access rates of the application, although more were needed for the PowerPC syntheses than for Pisa and Alpha.

6.3.2.5 Branch Predictability Model

The same branch predictability model from Chapter 3 is used. An integer instruction (the *BPCNTR*) that is not used as a memory access counter or a loop counter is converted into an *invert* instruction (*nor.*) operating on a particular register every time it is encountered. When the condition code is set by the inversion, the configured branch checking it jumps past the next basic block in the default loop. The *invert* mechanism was designed to yield a predictability of 50% for 2-bit saturating counter predictors. The POWER5 branch predictor uses combined tables of single bit predictors [85], but the invert mechanism still gives good correlation. To compensate for the resulting errors and variabilities in the mix of synthetic basic blocks and code size, the *BP Factor* in Table 6.6 multiplies the calculated branch predictability to increase or decrease the number of configured branches. Usually a small number of synthesis iterations are needed to tune this factor.

In an additional implementation, the *branch jump granularity* is adjusted such that a branch jumps past a user-defined number of basic blocks instead of just one, but as for the Alpha and Pisa syntheses, this did not result in improved branch predictability. In another implementation, the branch jumps past a user-defined number of instructions in the next basic block. Unlike for the Pisa and Alpha syntheses, this was not needed for *mgrid* and *applu* because their POWER5 versions have very high branch predictabilities, but it was effective in tuning the branch predictability for several SPEC INT benchmarks such as *eon* and *twolf*, which have relatively low branch predictabilities. In those cases, the branch jumps past one instruction of the next basic block.

The capability to skew the average length of the basic block by choosing sized successors as in Chapter 3 was not needed for *mgrid* and *applu*, but it was used for slight tunings of the average block sizes of various benchmarks. In Table 6.6, as the *basic block size factor* is reduced from unity, the block size is skewed toward the *basic block length* value.

Table 6.6: Synthetic Testcase Memory Access and Branching Factors for the POWER5 Chip

Name	**Dependence Factor**	**Bounded Factor**	**Stream Factor**	**Bounded Stream Factor**	**Load-Hit-Store Factor**	**Load-Store Offset Factor**	**BP Factor**	**Basic Block Size Factor**	**Basic Block Length**
gcc	1	1	1	1	0.58	0.24	1.01	0.95	4
gzip	1	1	1.2	1.2	1	1	0.65	1	-
crafty	1	1	0.9	0.9	0.15	0.97	1	0.8	10
eon	1	1	0.9	0.9	0.1	1	0.8	0.9	10
gap	1	1	1	1	0.28	0.98	1.03	1	-
bzip2	1	1	1	1	0.92	0.965	0.5	0.96	6
vpr	1	1.05	0.75	0.7	1	1	0.75	1	-
mcf	1	1	1	1	1	1	0.9	1	-
parser	1	1	1	1	1	1	1.02	0.9	5
perlbmk	1	1.05	1	1	0.85	0.998	1.05	0.95	5
vortex	1.5	0.01	1.1	0.1	0.01	1	1	0.9	5
twolf	1	1	1.05	1.1	1	1	0.8	1	-
mgrid	3.0	0.75	0.8	0.8	0.1	0.92	1	1	-
mesa	0.9	1	0.96	0.95	1	0.96	0.95	1	-
art	1	1.5	1.5	1.5	1	0.01	1	1	-
lucas	1	0.1	1	1	1	0.83	1	0.9	20
ammp	1	0.9	1.35	1	1	0.89	1	1	-
applu	1	1	1	1	1	0.7	1	1	-
apsi	1	1.05	1	1	0.53	0.93	1	1	-
equake	1	0.98	1.1	1.1	0.33	0.88	1	1	-
galgel	1	1.1	1	1.1	0.22	0.74	1	1	-
swim	1	1	1.05	1.2	0.98	0.99	1	1	-
sixtrack	1.5	1.1	0.9	0.9	0.08	0.78	1.03	1	-
wupwise	1	1	1	1	0.29	0.98	1.03	0.95	10
facerec	1	1	1	1	1	0.85	1	1	-
fma3d	1	1	1	1.02	0.25	1	0.93	0.98	20
saxpy	1	1	1	1	1	1	1	1	-
sdot	1	1	1	1	1	1	1	1	-
sfill	1	1	1	1	1	1	1	1	-
scopy	1	1	1	1	1	1	1	1	-
ssum2	1	1	1	1	1	1	1	1	-
sscale	1	1	1	1	1	1	1	1	-
striad	1	1	1	1	1	1	1	1	-
ssum1	1	1	1	1	1	1	1	1	-
tpc-c	2.0	1	1	1	0.3	1	1	0.93	5
java	1	1	1	1	1	1	1.05	0.95	5

6.3.3 Register Assignment

All architected register usages in the synthetic testcase are assigned exactly during the register assignment phase. As for the Alpha and Pisa experiments in Chapter 3, only 20 general-purpose registers divided between memory access stream counters and code use are necessary. For the PowerPC codes under study, the number of registers available for streams averages about 8 and for code use about 12 (*stream pools* + *bounded stream pools,* and *code registers* in Table 6.5). Two additional registers are reserved for the BRCNTR and BPCNTR functions.

Memory access streams are pooled according to their stream access characteristics and a register is reserved for each class (*stream pools* and *bounded stream pools* in Table 6.5). All LSCNTRs in the same pool increment the same register, so new stream data are accessed similarly whether there are a lot of LSCNTRs in the pool and few loop iterations or few in the pool but many iterations. For applications with large numbers of stream pools, synthesis consolidates the least frequent pools together until the total number of registers is under the limit. In the *Pisa* and *Alpha* studies, pools are greedily consolidated by iteratively combining the two least frequent pools until the limit is reached. For the PowerPC codes, the top most frequent pools are never watered down with less frequent pools; the last pool under the limit consolidates all less frequent pools. In all cases, the consolidated pools use the pool stride or reset value that minimizes the hit rate. The *stream pools* and *bounded stream pools* are consolidated separately.

6.3.4 Code Generation

As in Chapter 3, the code generator takes the representative instructions, the instruction attributes from graph analysis, and the register assignments and emits a single module of C-code that contains calls to assembly-language instructions in the PowerPC language. As before, the C-code *main* header is emitted, then variable declarations, stream pool pointer declarations, *malloc* calls for the stream data, and memory access pointer initializations, loop counter (BRCNTR) initialization, the volatile asm instructions themselves, and a C-code footer.

For the PowerPC ISA, the data access counters (LSCNTRs) are emitted as *addi* instructions that add their associated stride to the current register value. The BRCNTR is emitted as an *add* of minus one to its register. Long latency floating-point operations are generated using *fmul* and short latency operations are generated using *fadd*. Loads use *lwz* or *lfs*, depending on the type, and *stw* or *stfs* for stores. Branches use the *bc* with operands set to check the same condition register set by the BPCNTR. The basic blocks are analyzed and code is generated to print out unconnected output registers depending on a switch value. The switch is never set, but the print statements and volatile *asm* calls guarantee that no code is eliminated or reordered during compilation.

Tables 6.5 and 6.6 give the synthesis information for the PowerPC SPEC 2000 and STREAM codes as described in this section. The *runtime ratio* is the user runtime of the original benchmark for up to one hundred million instructions divided by the user runtime of the synthetic testcase on various POWER3 and POWER4 workstations. Each pass through the synthesis process takes less than five minutes on an IBM p270 (400 MHz). The results show a two or three order of magnitude speedup using the synthetics.

6.4 POWER5 SYNTHESIS RESULTS

In this section, results are presented for the synthetic POWER5 testcases obtained using the methods in the last section.

Table 6.7: Default Simulation Configuration for the POWER5 Chip

Instruction Size (bytes)	4
L1/L2 Line Size (bytes)	128/128
Machine Width	8
Dispatch Window;LSQ;IFQ	120 GPRs, 120 FPRs;32 LD, 32ST;64
Memory System	32KB 4-way L1 D, 64KB 2-way L1 I, 1.9M 10-way L2, 36MB 12-way L3
Functional Units	2 Fixed Point Units, 2 Floating Point Units
Branch Predictor	Combined 16K Tables, 12 cycle misspredict penalty

6.4.1 Experimental Setup

The profiling system from Chapter 3 is again used. The POWER5 M1 performance model described in Section 6.2 is augmented with profiling code to carry out the workload characterization. The 100M instruction SPEC 2000 PowerPC traces used in Jacobson *et al.* [44] and Hur and Lin [41] and described in Borkenhagen *et al.* [12] are executed on the augmented M1. Also added is an internal DB2 instruction trace of TPC-C [12][39] and a 100M instruction trace for SPECjbb (*java*) [89]. In addition, single-precision versions of the STREAM and STREAM2 benchmarks [64] with a one million-loop limit are compiled on a PowerPC machine. The default POWER5 configuration in Table 6.7 is used along with other parameters in Sinharoy *et al.* [85].

A code generator for the PowerPC target is built into the synthesis system, and C-code is synthesized. The synthetic testcases are compiled on a PowerPC machine using *gcc* with optimization level *–O2* and executed to completion in the M1.

6.4.2 Synthesis Results

The following figures show results either for the synthetics normalized to the original application results or for both the original applications, *actual*, and the synthetic testcases, *synthetic*. Figure 6.3 shows the normalized IPC for the testcases. The average IPC prediction error [27] for the synthetic testcases is 2.4%, with a maximum error of

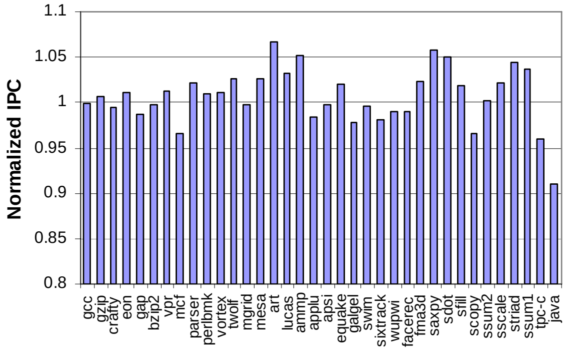


Figure 6.3: IPC for Synthetics Normalized to Benchmarks

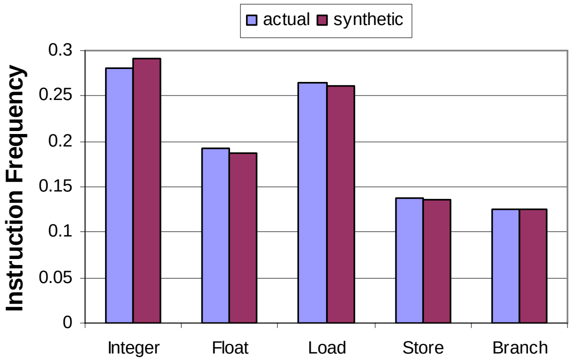


Figure 6.4: Average Instruction Frequencies

9.0% for *java*. The other commercial workload, *tpc-c*, gives 4.0% error. The reasons for the errors are discussed in the context of the figures below.

Figure 6.4 compares the average instruction percentages over all benchmarks for each class of instructions. The average prediction error for the synthetic testcases is 1.8% with a maximum of 3.4% for integers. Figure 6.5 shows that the basic block size varies per benchmark with an average error of 5.2% and a maximum of 18.0% for *apsi*. The largest absolute errors by far are for *mgrid* and *applu*. The errors are caused by variations in the fractions of specific basic block types in the synthetic benchmark with respect to the original workload, which is a consequence of selecting a limited number of basic blocks during synthesis. For example, *mgrid* is synthesized with a total of 30 basic blocks made up of eight unique block types. The top 90% of basic block frequencies in the synthetic *mgrid* differ by 27.5% on average from the basic block frequencies of the original workload. This is in contrast to testcases with large numbers of basic blocks such as *gcc*, which differ by only 3.5% for the top 90% of blocks.

The POWER5 I-cache miss rates normalized to the maximum miss rate are all accounted for in Figure 6.6, but they are not very interesting because most are less than 1%, and much less than the *tpc-c* and *java* miss rates. The low miss rates are due to the

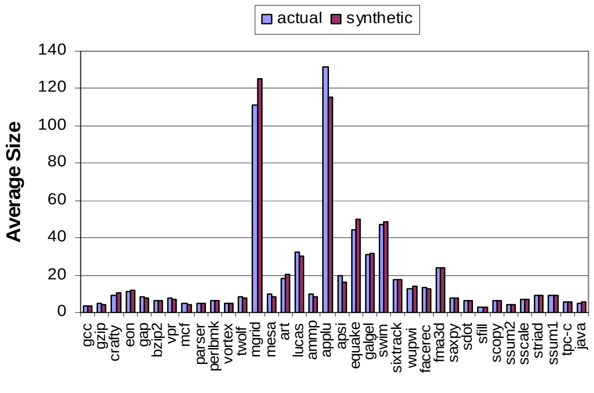


Figure 6.5: Average Basic Block Sizes

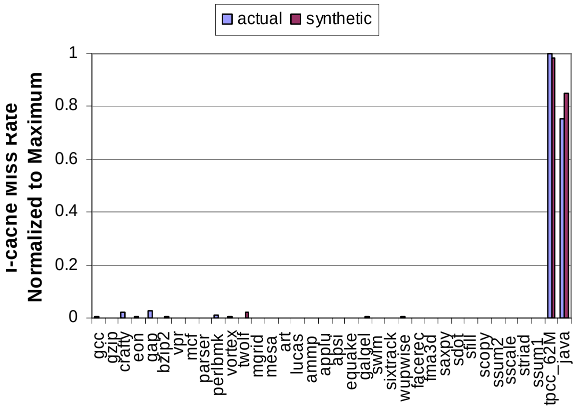


Figure 6.6: I-cache Miss Rates

effectiveness of instruction prefetching in the POWER5 chip [85][93]. The results also support the common wisdom that the SPEC do not sufficiently challenge modern I-cache design points. The synthetic benchmarks do well on the commercial workloads but still average 7% error. These errors could probably be reduced by carrying out more synthesis passes. In general, the synthetics have larger I-cache miss rates than the applications because they are executed for fewer instructions [7]. However, since the miss rates are small, their impact on IPC when coupled with the miss penalty is also small.

The average branch predictability error is 1.1%, shown normalized in Figure 6.7. The largest errors are is *bzip2* at 5.4% and *equake* at 4.7%. The L1 data cache miss rates are shown normalized in Figure 6.8. The average error is 4.3% with a maximum error of

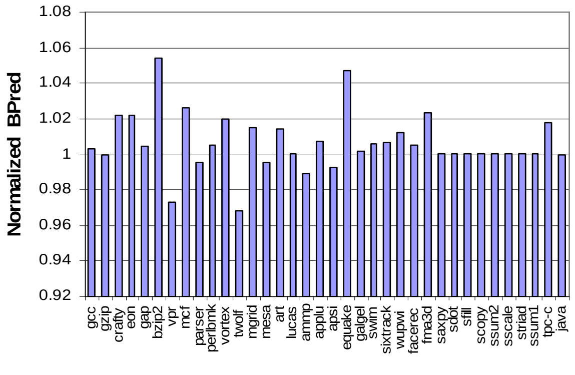


Figure 6.7: Branch Predictability

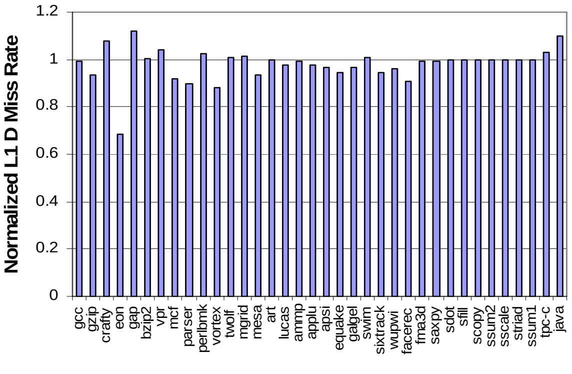


Figure 6.8: L1 D-cache Miss Rate

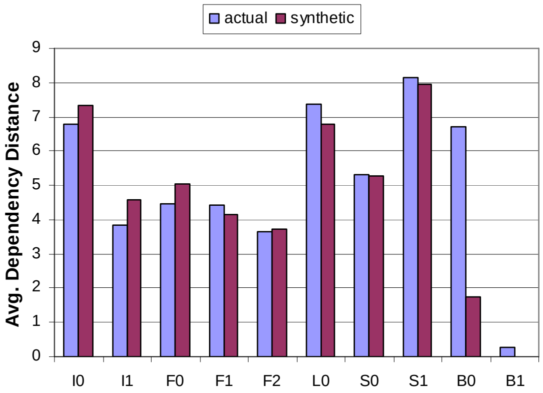


Figure 6.9: Average Dependence Distances

31% for eon. But *eon* has a very small miss rate, as do the other benchmarks with errors much greater than 4%, so again the execution impact of their errors is also small. Looking at the raw miss rates, the trends using the synthetic testcases clearly correspond with those of the original workloads. This can not be seen in the normalized results.

For the unified L2 miss rates, the average error is 8.3% for those benchmarks with miss rates higher than 3%. For the others, errors can be large. Large errors due to the simple streaming memory access model are often mitigated by the small magnitude of the L1 and L2 miss rates [7]. But problematic situations occur when a large L2 miss rate error is offset by an L1 miss rate that is smaller than that of the original application. Still, for larger L2 miss rates, the trends in miss rates using the synthetics clearly correspond to those of the original applications. The L3 miss rates are also generally very small and often not represented well by the synthetics. As mentioned, research into more accurate memory access models is needed to represent all levels of the memory hierarchy.

Figure 6.9 shows the average dependence distances for the input operands over all benchmarks. It shows a 7.5% error on average for the non-branch dependence distances. For the branch dependence errors, the M1 profiler classifies certain PowerPC trap and interrupt operations as unconditional jumps to the next basic block, i.e. they define the end of a basic block, and their dependences are not modeled. Also, the profiler records particular condition registers as the dependences for conditional branches, while synthesis uses one specific condition register to implement the branching model; the performance impact is small compared to the branch predictability itself.

The integer dependence errors are caused by the conversion of many integer instructions to LSCNTRs, the memory access stride counters. A stride counter overrides the original function of the integer instruction and causes dependence relationships to change. Another source of error is the movement of dependences during the search for compatible dependences in the synthesis process. The movement is usually less than one position (Table 6.5), but several benchmarks show significant movement. The dependence insertion technique reduces errors for the dependent instruction, but the inserted instruction may itself contain dependence errors.

6.4.3 Design Change Case Study: Rename Registers and Data Prefetching

A set of design changes using the same synthetic codes is now presented; that is, the testcases described in the last section are applied along with changes to the machine parameters in the M1 model and re-executed. In this study, pipeline resources are increased by more than *2x*. These are the number of rename registers available to the mapper for GPRs, FPRS, condition registers, link/count registers, FPSCRs, and XERs [85][93]. This effectively increases the dispatch window of the machine. Data prefetching is also enabled and execution is in “real” mode, i.e. with translation disabled. With these enhancements the average increase in IPC for the applications for the full 100M instructions is 56.7%.

The synthetics are executed using the same configuration. The average change in IPC for the synthetics is 53.5%. The raw IPC comparison again clearly shows that the trends in the IPC changes for all synthetics follow those of the original applications. The average absolute IPC prediction error is 13.9%, much less than half of the IPC change percentage, an important consideration when deciding if the testcase properly evaluates the design change [8]. The average relative error [27] for the change is 13.3%. The synthetic IPC change is generally lower than that of the application. This effect is explained by the use of bounded streams in PowerPC synthesis, which clamps particular synthetic stream miss rates to the base levels, as explained Section 6.3.

6.5 PERFORMANCE MODEL VALIDATION RESULTS

The synthetic testcases are now used to validate the performance model of a POWER5 follow-on processor. This is an actual case study carried out using an early version of a performance simulator for a new processor.

6.5.1 RTL Validation

Several of the same traces used in Section 6.4 for the POWER5 are executed on the VHDL and M1 models of the new PowerPC processor using the process described in Section 6.2.2. Figure 6.10 illustrates a detailed analysis that was carried out using the

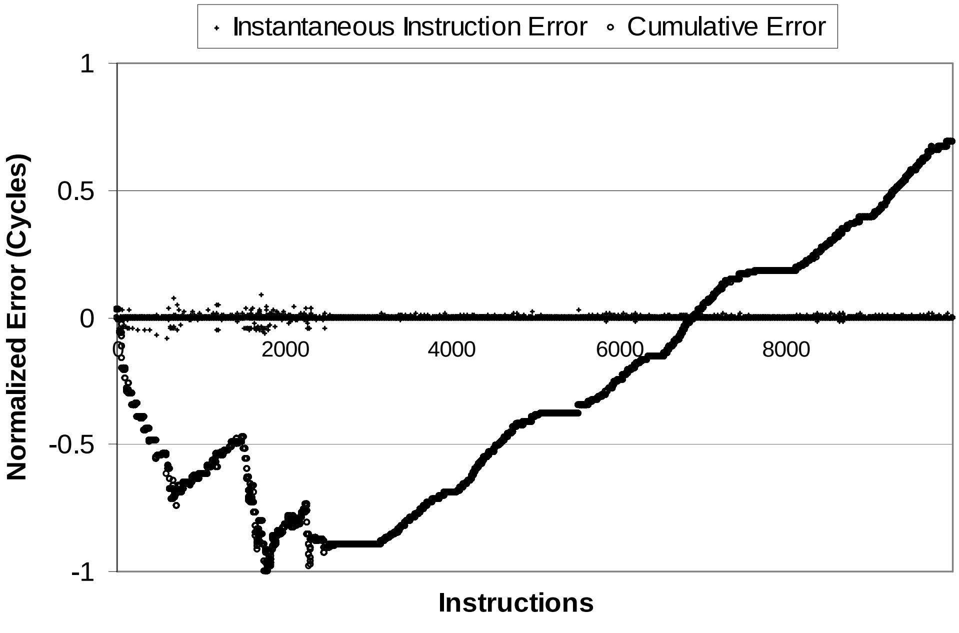


Figure 6.10: Normalized Error per Instruction and Cumulative Error for 10K Instructions (*gcc*)

RTL validation methodology. Information for each instruction in the execution of the first 10K instructions of the synthetic *gcc* is plotted, normalized to the maximum cumulative error.

The instantaneous errors are plotted as individual points either above or below the x-axis. If above the axis, the error is positive, meaning that the execution of the instruction in the VHDL model took longer than execution in the M1 performance model. It is difficult to see the small errors that are close to zero, but many errors are zero even near the end of execution. This indicates that many instructions are modeled correctly. The focus of examination is on the instructions that exhibit errors.

Ideally, the performance model would execute at the same rate or slower than the RTL model, so that designers do not project overestimates of performance for their designs. The cumulative sum of the instantaneous errors is plotted in Figure 6.10. It is clear that the M1 is providing overly-optimistic projections for *gcc* after only 10K instructions. The slope of the cumulative error later in the testcase indicates the direction of the performance model projections where positive is worse than flat or negative. The error is more erratic at the beginning of the execution because the early instructions are associated with the header and initialization C-code, not the body of the testcase, and they are not repeated.

The cumulative error plot can also be seen to indicate repeating sequences of behavior, or phases, at various scales in the synthetic testcases. The phases are related to specific code areas, and their identification can lead to rapid performance model fixes. Viewed at a particular scale, a phase starts after 2800 instructions and ends at about 4000. Another phase starts there and ends at 5200, and then the phases repeat. Both phases together are about as long as the body of the synthetic testcase. The shape of the curve indicates a steady, repeating set of instruction errors.

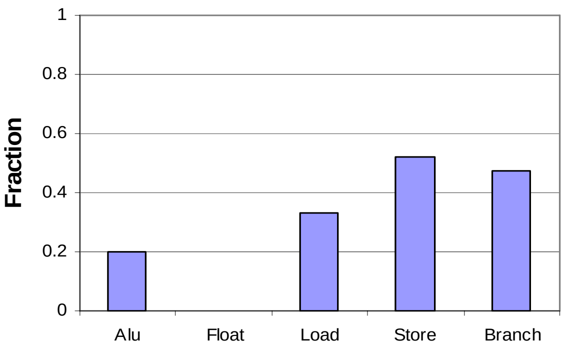


Figure 6.11: Fractions of Instructions with Errors (gcc)

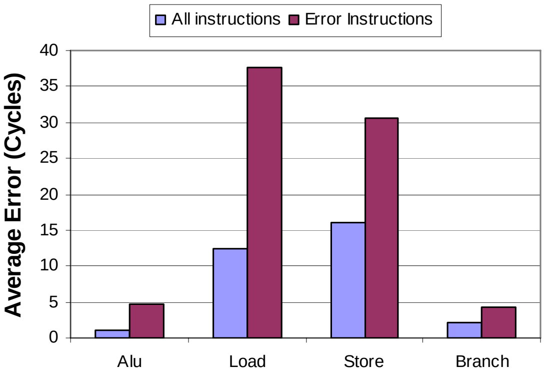


Figure 6.12: Average Error per Class for All Instructions or Instructions with Errors

To pinpoint the differences between VHDL and M1 model execution for *gcc*, the errors are analyzed by instruction class as in Figure 6.11. All classes show a large percentage of errors (*gcc* has no floating point instructions), but Figure 6.12 shows that average load and store errors, whether calculated over all instructions or just instructions with errors, have the largest impact on performance.

Figure 6.13 breaks down the fraction of instructions with errors into buckets of

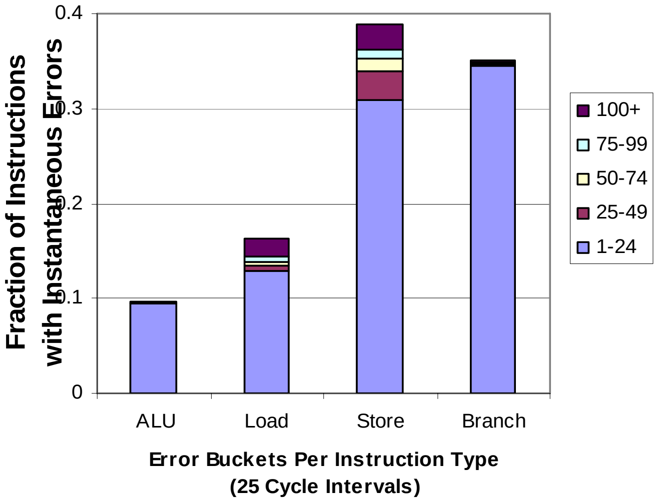


Figure 6.13: Fraction of Instructions with Errors by 25-Cycle Bucket (*gcc*)

errors that are multiples of 25 cycles. The vast majority of ALU and Branch operations with errors have errors that are less than 25 cycles, while 12.0% of loads and 6.8% of stores with errors have errors that are higher than 100 cycles, deep into the memory hierarchy.

An additional issue relates to how error is assigned for instructions that follow branches. Branches complete even if the branch prediction is incorrect, and the next instruction potentially experiences a larger completion delay because the mispredicted prior branch causes the pipe to flush. If the instantaneous error for the instruction following the branch is large, the problem may be the modeling of the branch predictor. For *gcc*, 9.3% of the loads and stores have errors greater than ten cycles and occur after a branch.

Regardless of the accuracy of the memory access models used to create the synthetic streams, the results using the models indicate that loads and stores are very likely to be modeled incorrectly in the performance model. This is valuable information to feed back to the performance modeling team. Note that these errors in the performance model were found prior to manufacturing the design in silicon.

6.5.2 Hardware Emulation

The same synthetic testcases are input to an AWAN hardware emulator executing the PowerPC VHDL models and to the M1 models used in the last section. In this case the runtime to collect the data is many times faster than VHDL simulation [56]. Figure 6.14 shows the M1 IPC normalized to the AWAN results for some of the testcases. The average error is 12.7%. Most of the errors are within 20%, but there are several outliers, including *bzip2* and *galgel*. The hardware emulator provides more rapid VHDL simulation to speed model validation investigations.

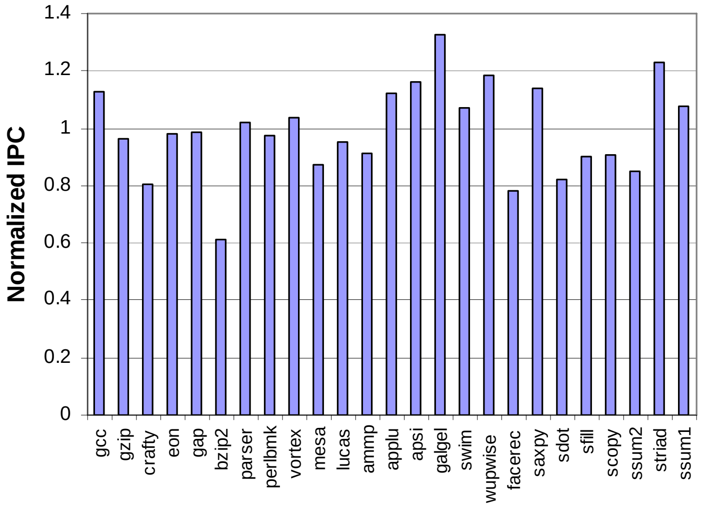


Figure 6.14: Normalized IPC for M1 versus AWAN VHDL for Synthetic Testcases

6.6 SUMMARY

This chapter extends the synthesis techniques of Chapter 3 to support the PowerPC ISA and presents a case study of the POWER5 processor. Representative versions of the SPEC 2000, STREAM, TPC-C and Java benchmarks are synthesized, compiled and executed, and an average IPC within 2.4% of the average IPC of the original benchmarks is obtained. The synthetic testcases often execute two orders of magnitude faster than the original applications, typically in less than 300K instructions. The synthetic workloads are then used in example performance model validations to compare the results of an IBM proprietary performance model to those of a VHDL model using RTL simulation and execution on the IBM AWAN hardware emulator.

Chapter 7: Conclusions and Future Work

This dissertation discusses the growing number of applications and the long runtimes of those applications that has caused concern about the simulation of computer designs for design studies and performance and power model validations. For design studies, researchers have responded with various methods to reduce runtimes, including analytical modeling, reduced input datasets, sampling techniques and statistical simulation. Statistical simulation converges the most rapidly to a result, but prior work shows that it can be inaccurate. This dissertation improves the accuracy of trace synthesis in statistical simulation by modeling workload characteristics at the granularity of the basic block.

In spite of the runtime reductions achieved for performance simulation, no similar evolution in simulation capability has occurred for performance and power model validation. The improved synthetic traces of statistical simulation can not be used because traces can not be executed on the variety of platforms necessary for model validation. Short snippets of benchmark instructions can be transformed for use on simulators or hardware, but since no attempt is made to analyze a larger simulation phase and reduce the number of instructions intelligently, it is unlikely that validation over the snippets can cover the behavior of the benchmark in the machine. The state-of-the-art in model validation has been to use hand-coded testcases and microbenchmarks to study individual components of the latency and bandwidth in cycle-accurate and performance model simulations. This dissertation proposes a new methodology for automatically synthesizing more representative workloads from the improved workload characterization of statistical simulation together with locality models and workload synthesis methods.

The core technology in representative workload synthesis is conceptually easy to grasp. The workload characterization generates a flow graph that includes the most frequent basic blocks found in the executing workload, a list of the successors of each basic block and a list of the probabilities that each successor is branched to. More complicated flow graphs can keep lists of sequences of basic blocks to further improve accuracy. A synthetic trace is generated by traversing the flow graph; the trace forms the spine of a representative workload synthesis.

This dissertation advocates the application of this core technology to both the early design and model validation problems. By holding to this philosophy, the simulation methodology used for design studies of performance and power dissipation is consolidated with that used for performance and power model validations. The representative trace forms a single starting point for all performance simulation tasks. From that base, workloads can be synthesized to reduce runtime, enhance simulation flexibility, and improve portability to diverse platforms. The full potential of synthetic workloads requires future research into synthetic memory access and branching models and their integration into the workloads.

Additional conclusions and future work are discussed in the next sections.

7.1 CONCLUSIONS

This dissertation makes the following specific contributions to computer engineering research related to workload synthesis, performance model simulation and model validation:

1) The accuracy of the workload characterization in statistical simulation is improved by compiling statistics at the granularity of the basic block rather than the granularity of the instruction. Specifically, instruction sequences, dependences, miss rates, branch predictability and cache access history are all

modeled at the basic block level and contribute to improvements in accuracy for a variety of workloads. In addition, basic block maps add successor information at the basic block level, which provides more accurate modeling of micro-phases, or small shifts in relative block frequencies inside a larger number of instructions. The SPEC 95 benchmarks are improved from 15.5% IPC prediction error on average to 5.3% error, and the SPEC 2000 benchmarks are improved from 27.6% error to 4.7% error. The cost of the additional modeling is quantified to less than 100KB per benchmark in SPEC 95 and less than 200KB in SPEC 2000. In an additional collaboration with researchers at the University of Ghent, Belgium, the ideas presented here are implemented in the statistical flow graph, which maintains up to *k* lists of prior basic blocks [27]. In that study, errors in IPC on the SPEC INT 2000 are shown to be less than 7%, and relative errors are typically less than 3%, confirming the usefulness of modeling at the granularity of the basic block. These improvements facilitate more accurate simulations of designs and design change studies without large increases in runtimes.

2) This dissertation presents a method for synthesizing workloads into C-code with inline assembly calls that represent the low-level execution characteristics of the application. Specific workload synthesis approaches for the Pisa, Alpha and PowerPC instruction set architectures are presented. It is shown that average performance using the synthetic workloads matches the original applications within 2.4% for all three ISAs while runtimes are often three orders of magnitude shorter. Small relative errors are also obtained when the machine configuration changes. These results indicate that the synthetic workloads are useful for design studies and performance model validations.

An automatic process is presented that permits workloads to be quickly recreated as the applications or languages change. User parameters can tune the synthesis process to more closely match application workload characteristics. This work demonstrates that if synthesis starts with the locality models of the machine-under-study, good correlation can be obtained versus the original workload for many workload characteristics and for performance of design changes. Results generally become worse as the configuration under study moves farther away from the configuration used for synthesis. Therefore, the present techniques should not be used to study large changes in machine configuration. However, a stair-step synthesis approach to design studies is suggested in which resynthesis occurs when machine configuration parameters change significantly from their starting value.

3) By synthesizing the workloads in a high-level language, they are portable across multiple platforms including execution-driven and trace-driven performance simulators, functional RTL simulators, emulators and hardware. This variety makes them useful for both design studies and model validations. This dissertation demonstrates the same workload running on a trace-driven performance simulator, a RTL simulator, and an RTL hardware emulator.
4) Since the synthesis process is based on a statistical workload characterization, altering the characteristics of the synthetic workload is straightforward. Individual changes to program characteristics can be isolated and studied independently, and changes to the workload characteristics thatare anticipated for future workloads can be easily incorporated. The statistical nature of the workload characterization removes the dataset and abstracts the details and

behavior of the application, effectively hiding its underlying function. This encourages the sharing of proprietary codes between industry and academia.

5) The accuracy of the synthetic workloads depends on the accurate characterization of the original workload prior to synthesis. It is shown that relatively large margins exist wherein performance is not impacted as workload characteristics change. These results indicate that the relatively small changes in workload characteristics due to the workload synthesis process do not impact performance significantly.
6) This dissertation demonstrates that the synthetic workloads created strictly for representative performance also match the dynamic power dissipation per cycle of the applications within 6.8% error on average, and for many design changes the average error is less than 5%. The results also confirm prior correlation between IPC and power dissipation and extend the correlations to design changes. These strong correlations indicate that the synthetic testcases are useful for early design studies of power and power model validations.
7) This dissertation demonstrates specific performance model validation techniques. For example, synthetic testcases are executed in both a performance simulator and an RTL simulator and the instruction completion times are compared. The results give specific instructions, instruction types, and error buckets that can be examined in more detail to isolate significant problems in the performance model or RTL. The notion of Instantaneous Error is introduced. Validation using a hardware emulator is also presented.
8) This dissertation demonstrates that the same synthetic workloads that are useful for design studies are also useful for performance and power model validations. It shows that the workloads used for early design studies and

model validations are not only consolidated at an abstract level by the use of synthetic traces as the basis for execution and synthesis, but that the workloads can also be consolidated together more concretely as codes that can be compiled and executed on various simulation and hardware platforms. This process gives higher confidence in performance projections as the detailed RTL level models are available in the design process.

7.2 FUTURE WORK

There are several areas that could benefit from additional work.

1) The memory access models as formulated in this dissertation are based on the cache hit rates of loads and stores in individual basic blocks and are therefore microarchitecture-dependent. Those models show large errors versus the original applications, especially in response to design changes. Many models of memory access behavior exist in the literature and could be formulated for use in the synthetic testcases. Such models could make the synthetics more independent of any particular microarchitecture and therefore more amenable to design studies and other studies related to changes in the cache hierarchy. The problem with the stride-based model presented here is that it is simplistic and does not support "random" accesses or frequent address interactions between separate operations. Future work could determine what the best modeling trade-off is between realistic, but complicated, access behavior and simple, stride-based models. Cache warm-up prior to synthetic execution may be necessary to match specific locality features such as capacity misses. Intrinsic Checkpointing [79] provides a memory initialization capability that could be integrated into the methodology.

2) Similarly, the branching model described here is formulated to operate correctly only for 2-bit saturating counters in the branch history table with out global branch history or local branch history shift registers. While branch predictability errors are very low for the cases studied here, more microarchitecture-independent models could make the workloads more useful for studies involving changes in the branch predictor technology.
3) In the current synthesis approach, a synthetic trace is developed from a traversal of the statistical flow graph, and low-level instructions from the trace are instantiated in the testcase. Future work could seek methods to instantiate all or part of the flow graph, or a reformulated flow graph, directly into a synthetic benchmark, with the instructions necessary to branch correctly from one basic block to another. The major problem with that approach appears to be the retention of representative branching and dependence compatibilities.
4) The workload characterization used for statistical simulation and workload synthesis is obtained from a profile of a specific set of workload characteristics. While the present collection of characteristics gives good simulation results, many additional features of workloads could be collected to make the synthetic traces more realistic. Examples include detailed memory address stream features, such as the probability of a load address hitting a store address, the probability and extent of unaligned accesses, or address index and offset relationships. Likewise, more detailed dependence information could be modeled, including write-after-write and write-after-read anti-dependences.
5) Similarly, there are currently only five abstract instruction types with sub-types instantiated in the synthetics. Future work could expand these categories

to provide more representative execution, or minimize abstraction in favor of actual instructions taken from the source code.

6) In the current synthesis approach, the microarchitecture-independent workload characteristics, and therefore the synthetic workload characteristics, are dependent on the compiler technology used to compile the original application, as is the case in statistical simulation. The automatic process ensures that resynthesis based on new compiler technology operating on the original application is not difficult, and it is argued that the low-level instruction behavior is necessary to achieve representativeness. However, future work might find combinations of synthesis techniques and compiler technology to instantiate part or all of the functional workload as higher-level code without significantly impacting representativeness, thereby enabling optimizations using new compiler technology.
7) Synthesis could be extended to create representative multiprocessor workloads. The major issues are related to characterization and modeling of coherency and locking mechanisms. A multiprocessor or full system simulator would likely be needed to carry out these studies.
8) In the current approach, workload characteristics from both user code and operating system code are profiled together. Future work using full system simulation could separate these instruction streams and create separate sections in the synthetic to represent each. The interaction between the OS and user contexts could be modeled.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S.W. Keckler and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," *Proceedings of the International Symposium on Computer Architecture*, June 2000, pp. 248-259.

[2] Anonymous, “A Measure of Transaction Processing Power,” *Datamation*, Vol. 31, No. 7, 1985, pp. 112-118.

[3] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam and P. Sdayappan, “A High-Level Approach to Synthesis of High Performance Codes for Quantum Chemistry,” *Proceedings of the ACM/IEEE Conference on Supercomputing*, November 2002, pp. 1-10.

[4] R. H. Bell, Jr., L. Eeckhout, L. K. John and K. De Bosschere, “Deconstructing and Improving Statistical Simulation in HLS,” *Proceedings of the Workshop on Debunking, Duplicating, and Deconstructing*, June 20, 2004, pp. 2-12.

[5] R. H. Bell, Jr. and L. K. John, *Experiments in Automatic Benchmark Synthesis*, Technical Report TR-040817-01, Laboratory for Computer Architecture, Department of Electrical and Computer Engineering, University of Texas at Austin, August 17, 2004.

[6] R. H. Bell, Jr. and L. K. John, “The Case for Automatic Synthesis of Miniature Benchmarks,” *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 4, 2005, pp. 88-97.

[7] R. H. Bell, Jr. and L. K. John, “Improved Automatic Testcase Synthesis for Performance Model Validation,” *Proceedings of the International Conference on Supercomputing*, June 20, 2005, pp. 111-120.

[8] R. H. Bell, Jr. and L. K. John, “Efficient Power Analysis using Synthetic Testcases,” *Proceedings of the IEEE International Symposium on Workload Characterization*, October 7, 2005, pp. 110-118.

[9] J. Bilmes, K. Asanovic, C.-W. Chin and J. Demmel, “Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology,” *Proceedings of the International Conference on Supercomputing*, June 1997, pp. 340-347.

[10] W. L. Bircher, M. Valluri, J. Law and L. K. John, “Runtime Identification of Microprocessor Energy Saving Opportunities,” *Proceedings of the International Symposium on Low Power Electronics and Design*, August 2005, pp. 275-280.

[11] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," *IEEE Computer Magazine*, May 1998, pp. 59-65.

[12] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla and S. R. Kunkel, "A Multithreaded PowerPC Processor for Commercial Servers," *IBM J. Res. and Dev.*, Vol. 44, 2000, pp. 885-898.

[13] P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," *IEEE Computer Magazine,* May 1998, pp. 41-49.

[14] P. Bose, "Architectural Timing Verification and Test for Super-Scalar Processors," *Proceedings of the International Symposium on Fault-Tolerant Computing*, June 1994, pp. 256-265.

[15] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *Proceedings of the International Symposium on Computer Architecture*, June 2000.

[16] D. C. Burger and T. M. Austin, *The SimpleScalar Toolset Version 2.0*, Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[17] R. Carl and J. E. Smith, "Modeling Superscalar Processors Via Statistical Simulation," *Workshop on Performance Analysis and Its Impact on Design*, June 27, 1998.

[18] Z. Cvetanovic, E. G. Freedman and C. Nofsinger, "Perfect Benchmarks Decomposition and Performance on VAX Multiprocessors," *Proceedings of the Conference on Supercomputing*, 1990, pp. 455-464.

[19] W. K. Cheng and Y. L. Lin, "Code Generation for a DSP Processor," *Proceedings of the Seventh International Symposium on High Level Synthesis*, May 1994, pp. 82-87.

[20] T. Conte and W. Hwu, "Benchmark Characterization for Experimental System Evaluation," *Proceedings of the Hawaii International Conference on System Science*, 1990, pp. 6-18.

[21] F. Corno, E. Sanchez, M. S. Reorda and G. Squillero, "Automatic Test Program Generation: A Case Study," I*EEE Design & Test of Computers*, March-April 2004, pp. 102-109.

[22] H. J. Curnow and B.A. Wichman, "A Synthetic Benchmark," *Computer Journal*, Vol. 19, No. 1, February 1976, pp. 43-49.

[23] R. Desikan, D. Burger and S. Keckler, “Measuring Experimental Error in Microprocessor Simulation,” *Proceedings of the International Symposium on Computer Architecture*, 2001, pp. 266-277.

[24] C. Ding and Y. Zhong, “Predicting Whole Program Locality Through Reuse-Distance Analysis,” *Proceedings of the Conference on Programming Languages, Design and Implementation*, 2003.

[25] J. J. Dujmovic, “Evaluation and Design of Benchmark Suites,” K. Bagchi, G. Zobrist and K. Trivedi, editors, *State-of-the-Art in Performance Modeling and Simulations: Theory, Techniques, and Tutorials*, Chapter 12, Gordon and Breach Publishers, 1996.

[26] J. J. Dujmovic and I. Dujmovic, “Evolution and Evaluation of SPEC Benchmarks,” *ACM Sigmetrics Performance Evaluation Review*, Vol. 26, No. 3, December 1998, pp. 2-9.

[27] L. Eeckhout, R. H. Bell, Jr., B. Stougie, L. K. John and K. De Bosschere, “Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies,” *Proceedings of the International Symposium on Computer Architecture*, June 2004, pp. 350-361.

[28] L. Eeckhout, *Accurate Statistical Workload Modeling*, Ph.D. Thesis, Ghent University, 2003.

[29] L. Eeckhout and K. De Bosschere, “How Accurate Should Early Design Stage Power/Performance Tools Be? A Case Study with Statistical Simulation,” *The Journal of Systems and Software*, Vol. 73, No. 1, 2004, pp. 45-62.

[30] L. Eeckhout, H. Vandierendonck and K. De Bosschere, “Designing Computer Architecture Research Workloads,” *IEEE Computer Magazine*, Vol. 36, No. 2, February 2003, pp. 65-71.

[31] C. Fang, S. Carr, S. Onder and Z. Wang, “Instruction Based Memory Distance Analysis and Its Application,” *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, September 2005, pp. 27-37.

[32] D. F. Garcia and J. Garcia, “TPC-W E-Commerce Benchmark Evaluation,” *IEEE Computer Magazine*, February 2003, pp. 42-48.

[33] J. C. Gibson, *The Gibson Mix*, IBM Technical Report 00.2043, IBM Systems Development Division, Poughkeepsie, N.Y., 1970.

[34] M. K. Gowan, C. Polychronopoulos and G. Stamoulis, “Power Considerations in the Design of the Alpha 21264 Microprocessor,” Proceedings of the Design Automation Conference, 1998, pp. 726-731.

[35] Richard Hankins, Trung Diep, Murali Anavaram, Brian Hirano, Harald Eri, Hubert Nueckel and John P. Shen, "Scaling and Characterizing Database Workloads: Bridging the Gap between Research and Practice," Proceedings of the International Symposium on Microarchitecture, December 2003, pp. 151-164.

[36] J. Hennessy and D. Patterson, *Computer Architecture, A Quantitative Approach*, 2nd Edition, San Francisco: Morgan-Kaufman, 1996.

[37] J. L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millenium," IEEE Computer Magazine, Vol. 33, No. 7, July 2000, pp. 28-35.

[38] C. T. Hsieh and M. Pedram, "Microprocessor power estimation using profile-driven program synthesis," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, Vol. 17, 1998, pp. 1080-1089.

[39] W. W. Hsu, A. J. Smith and H. C. Young, "Characteristics of Production Database Workloads and the TPC Benchmarks," *IBM Systems Journal,* Vol. 40, 2001, pp. 781-802.

[40] J. Huk, S. W. Keckler and D. Burger, "Exploring the Design Space of Future CMPs," *Proceedings of the Conference on Parallel Architectures and Compilation Techniques*, October 2001, pp. 199-210.

[41] I. Hur and C. Lin, "Adaptive History-Based Memory Schedulers," *Proceedings of the International Symposium on Microarchitecture*, 2004.

[42] V. S. Iyengar, L. H. Trevillyan and P. Bose, "Representative Traces for Processor Models with Infinite Cache," *Proceedings of the Symposium on High Performance Computer Architecture*, February 1996, pp. 62-73.

[43] V. S. Iyengar and L. H. Trevillyan, *Evaluation and Generation of Reduced Traces for Benchmarks,* IBM Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center, October 1996.

[44] H. Jacobson, P. Bose, Z. Hu, A. Buyuktosunoglu, V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, B. Sinharoy and J. Tendler, "Stretching the Limits of Clock-Gating Efficiency in Server-Class Processors," *Proceedings of the International Symposium on High-Performance Computer Architecture*, February 2005.

[45] R. Jain, *The Art of Computer Systems Performance Analysis*, New York: John Wiley & Sons, 1991.

[46] M. Johnson, *Superscalar Microprocessor Design*, Englewood Cliffs: P T R Prentice Hall, New Jersy, 1991.

[47] L. K. John, P. Vasudevan and J. Sabarinathan "Workload Characterization: Motivation, Goals, and Methodology," *Proceedings of the Workshop in Workload Characterization*, November 29, 1998, pp. 3-14.

[48] C. P. Joshi, A. Kumar and M. Balakrishnan, "A New Performance Evaluation Approach for System Level Design Space Exploration," *Proceedings of the International Symposium on System Synthesis*, October 2002, pp. 180-185.

[49] K. Keaton and D. A. Patterson, "Towards a Simplified Database Workload for Computer Architecture Evaluations," *Proceedings of the Workshop on Workload Characterization*, October 1999, pp. 115-124.

[50] A.J. KleinOsowski and D.J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *Computer Architecture Letters*, Vol. 1, June 2002, pp. 22-30.

[51] S. R. Kunkel, R. J. Eickemeyer, M. H. Lipasti, T. J. Mullins, B. O'Krafka, H. Rosenberg, S. P. VanderWiel, P. L. Vitale and L. D. Whitley, "A Performance Methodology for Commercial Servers," *IBM J. Res. and Develop.*, Vol. 44, 2000, pp. 851-872.

[52] T. Lafage and A. Seznec, "Choosing Representative Slices of Program Execution for Microarchitecture Simulations," *Proceedings of the Workshop on Workload Characterization,* September 16, 2000, pp. 102-110.

[53] C. Lee and M. Potkonjak, "A Quantitative Approach to Development and Validation of Synthetic Benchmarks for Behavioral Synthesis," *Proceedings of the International Conference on Computer Aided Design*, November 1998, pp. 347-350.

[54] T. Li and L. John, "Run-Time Modeling and Estimation of Operating System Power Consumption," *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, June 10-14, 2003, pp. 160-171.

[55] D. J. Lilja, *Measuring Computer Performance*, Cambridge: Cambridge University Press, 2000.

[56] J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. M Chu,. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor and B. Wile, "Functional Verification of the Power4 Microprocessor and the Power4 Multiprocessor Systems," *IBM J. Res. and Dev.*, Vol. 46, 2002, 53-76.

[57] V. T. Lund, Preface to the *IBM J. of Res. and Dev.*, Vol. 46, 2002, pp. 3-4.

[58] Y. Luo, A. Joshi, A. Phansalkar, L. K. John and J. Ghosh, “Analyzing and Improving Clustering Based Sampling for Microprocessor Simulation,” *Symposium on Computer Architecture and High-Performance Computing*, October 2005.

[59] Z. Manna and R. Waldinger, “Toward Automatic Program Synthesis,” *Communications of the ACM*, Vol. 14, No. 3, March 1971, pp. 151-165.

[60] Z. Manna and R. Waldinger, “A Deductive Approach to Program Synthesis,” *ACM Transactions on Programming Languages and Systems*, Vol. 2, 1980, pp. 90-121.

[61] G. Marin and J. Mellor-Crummey, “Cross-Architecture Performance Predictions for Scientific Applications using Parameterized Models,” *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, Vol. 32, pp. 2-13.

[62] F.H. McMahon, *Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range*, Lawrence Livermore National Laboratories, Livermore, California, 1986.

[63] J. D. McCalpin and M. Smotherman, “Automatic Benchmark Generation for Cache Optimization of Matrix Operations,” *Proceedings of the 33rd Annual Southeast Conference*, March 1995, pp. 195-204.

[64] J. D. McCalpin, “Memory Bandwidth and Machine Balance in Current High Performance Computers,” *IEEE Technical Committee on Computer Architecture Newsletter*, December 1995, pp. 19-25.

[65] L. McVoy, “lmbench: Portable Tools for Performance Analysis,” *Proceedings of the USENIX Technical Conference*, January 22-26, 1996, pp. 279-294.

[66] M. Moudgill, J. D. Wellman and J. H. Moreno, “Environment for PowerPC Microarchitecture Exploration,” *IEEE Micro Magazine*, May-June 1999, pp. 15-25.

[67] A. Nanda and L. M. Ni, “Benchmark Workload Generation and Performance Characterization of Multiprocessors,” *Proceedings of the IEEE Supercomputing Conference*, November 1992, pp. 20-29.

[68] D. B. Noonburg and J. P. Shen, “Theoretical Modeling of Superscalar Processor Performance,” *Proceedings of the International Symposium on High Performance Computer Architecture*, February 1997, pp. 298-309.

[69] S. Nussbaum and J. E. Smith, "Modelling Superscalar Processors Via Statistical Simulation," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001, pp. 15-24.

[70] S. Nussbaum and J.E. Smith, "Statistical Simulation of symmetric Multiprocessor Systems," *Proceedings of the 35th Annual Simulation Symposium*, April 2002, pp. 89-97.

[71] M. Oskin, F. T. Chong and M. Farrens, "HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design," Proceedings of the International Symposium on Computer Architecture, June 2000, pp. 71-82.

[72] http://www.cs.washington.edu/homes/oskin/tools.html

[73] D. A. Penry, D. I. August and M. Vachharajani, "Rapid development of a Flexible Validated Processor Model," *Proceedings of the Workshop on Modeling, Benchmarking and Simulation*, June 4, 2005, pp. 21-30.

[74] M. Berry, *The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers*, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champagne, CSRD report #896, May 1989.

[75] A. Phansalkar, A. Joshi, L. Eeckhout and L. K. John, "Measuring Program Similarity: Experiments with the SPEC CPU Benchmark Suites," *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, March 2005, pp. 10-20.

[76] Joachim Pistorius, Edmee Legai and Michel Minoux, "Generation of Very Large Circuits to Benchmark the Partitioning of FPGA's," *Proceedings of the International Symposium on Physical Design,* April 1999, pp. 67-73.

[77] A. Pnueli and R. Rosner, "On the Synthesis of a Reactive Module," *Proceedings of the Symposium on Principles of Programming Languages*, 1989, pp. 179-190.

[78] R. Rao, M. Oskin, F. T. Chong, "HLSpower: Hybrid Statistical Modeling of the Superscalar Power-Performance Design Space," *International Conference on High Performance Computing*, December 2002, pp. 620-629.

[79] J. Ringenberg, C. Pelosi, D. Oehmke and T. Mudge, "Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification," *Proceedings of the International Symposium on Performance and Simulation Systems*, March 2005, pp. 78-88.

[80] R. H. Saavedra-Barrera, *CPU Performance Evaluation and Execution Time Prediction Using Narrow Spectrum Benchmarking*, Ph. D. Thesis, UC Berkeley, Technical Report No. UCB/CSD 92/684, Feb. 1992.

[81] M. Sakamoto, L. Brisson, A. Katsuno, A. Inoue and Y. Kimura, "Reverse Tracer: A Software Tool for Generating Realistic Performance Test Programs," *Proceedings*

of the Symposium on High-Performance Computer Architecture, February 2002, pp. 81-91.

[82] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler and C. R. Moore, "Exploiting ILP, TLP, DLP with Polymorphous TRIPS Architecture," *Proceedings of the International Symposium on Computer Architecture*, June 2003, pp. 422-433.

[83] T. Sherwood, E. Perleman, H. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proceedings of the Conference on Architected Support for Programming Languages and Operating Systems*, October 2002, pp. 45-57.

[84] R. Singhal, K. S. Venkatraman, E. Cohn, J. G. Holm, D. Koufaty, M.-J. Lin, M. Madhav, M. Mattwandel, N. Nidhi, J. Pearce and M. Seshadri, "Performance Analysis and Validation of the Intel Pentium4 Processor on 90nm Technology," *Intel Tech. J.*, Vol. 8, No. 1, February 2004, pp. 33-42.

[85] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer and J. B. Joyner, "POWER5 System Microarchitecture," IBM J. Res. & Dev., Vol. 49, 2005, pp. 505-521.

[86] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja and V. S. Pai, "Challenges in Computer Architecture Evaluation," *IEEE Computer Magazine*, August 2003, pp. 30-36.

[87] J. E. Smith and G. S. Sohi, "The Microarchitecture of Superscalar Processors," Proceedings of the IEEE, Vol. 83, 1995, pp. 1609-1624.

[88] E. S. Sorenson and J. K. Flanagan, "Evaluating Synthetic Trace Models using Locality Surfaces," *Proceedings of the Workshop on Workload Characterization*, November 2002, pp. 23-33.

[89] http://www.spec.org

[90] K. Sreenivasan and A.J. Kleinman, "On the Construction of a Representative Synthetic Workload," *Communications of the ACM*, March 1974, pp.127-133.

[91] S. Surya, P. Bose and J. A. Abraham, "Architectural Performance Verification: PowerPC Processors," *Proceedings of the International Conference on Computer Design*, 1999, pp. 344-347.

[92] P. K. Szwed, D. Marques, R. M. Buels, S. A. McKee, and M. Schulz, "SimSnap: Fast-Forwarding via Native Execution and Application-Level Checkpointing," *Proceedings of the Workshop on Interaction Between Compilers and Computer Architecture*, February 2004.

[93] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le and B. Sinharoy, "POWER4 System Microarchitecture," *IBM J. of Res. and Dev.*, Vol. 46, 2002, pp. 5-25.

[94] D. Thiebaut, "On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio," *IEEE Transactions on Computers*, Vol. 38, 1989, pp. 1012-1026.

[95] R. Todi, "SPEClite: Using Representative Samples to Reduce SPEC CPU2000 Workload," *Proceedings of the Workshop on Workload Characterization*, December 2001, pp. 15-23.

[96] http://www.tpc.org

[97] M. Valluri and L. John, "Is Compiling for Performance == Compiling for Power?" *Proceedings of the Workshop on the Interaction Between Compilers and Computer Architectures*, 2001.

[98] M. Van Biesbrouck, L. Eeckhout and B. Calder, "Efficient Sampling Startup for Sampled Processor Simulation," *International Conference on High Performance Embedded Processors and Compilers*, November 2005.

[99] H. Vandierendonck and K. De Bosschere, "Many Benchmarks Stress the Same Bottlenecks," *Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 14 2004, pp. 57-64.

[100] P. Verplaetse, J. Van Campenhout and D. Stroobandt, "On Synthetic Benchmark Generation Methods," *Proceedings of the International Symposium on Circuits and Systems*, May 28, 2001, pp. 213-216.

[101] D. W. Victor, J. M. Ludden, R. D. Peterson, B. S. Nelson, W. K. Sharp, J. K. Hsu, B.-L. Chu, M. L. Behm, R. M. Gott, A. D. Romonosky and S. R. Farago, "Functional Verification of the POWER5 Microprocessor and POWER5 Multiprocessor Systems," *IBM J. Res. and Dev.*, Vol. 49, 2005, pp. 541-553.

[102] R. C. Whaley and J. J. Dongarra, "Automatically Tuned Linear Algebra Software," *Proceedings of the International Conference on Supercomputing*, November 1998, pp. 38-38.

[103] R. P. Weicker, "Dhrystone: A Synthetic Systems Programming Benchmark," *Communications of the ACM*, Vol. 27, 1984, pp. 1013-1030.

[104] R. P. Weicker, "An Overview of Common Benchmarks," *IEEE Computer Magazine*, December 1995, pp. 65-75.

[105] T. F. Wenisch, R. E. Wunderlich, B. Falsafi and J. C. Hoe, "TurboSMARTS: Accurate Microarchitecture Simulation in Seconds," poster session in the

International Conference on Measurement and Modeling of Simulation Systems, June 2005.

[106] J. N. Williams, “The Construction and Use of a General Purpose Synthetic Program for an Interactive Benchmark for on Demand Paged Systems,” *Communications of the ACM*, 1976, pp. 459-465.

[107] W. S. Wong and R. J. T. Morris, "Benchmark Synthesis Using the LRU Cache Hit Function," *IEEE Transactions on Computers*, Vol. 37, 1988, pp. 637-645.

[108] R. E. Wunderlich, T. F. Wenisch, B. Falsafi and J. C. Hoe, “SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling,” *Proceedings of the International Symposium on Computer Architecture*, June 2003, pp. 84-95.

[109] L. Eeckhout and K. De Bosschere, “Early Design Phase Power/Performance Modeleing Through Statistical Simulation,” *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, November 2001, pp. 10-17.

[110] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer and D. Shippy, “Introduction to the Cell Multiprocessor,” *IBM J. Res. and Dev.*, Vol. 49, 2005, pp. 589-604.

[111] B. K. Lee and L. K. John, “Implications of Executing Compression and Encryption Applications on General Purpose Processors,” *IEEE Transactions on Computers*, Vol. 54, 2005, pp. 917-922.

[112] C.-B. Cho, A. V. Chande, U. Li and T. Li, “Workload Characteristics of Biometrics Applications on Pentium 4 Microarchitecture,” *Proceedings of the International Symposium on Workload Characterization*, October 2005, pp.76-86.

Vita

Robert Henry Bell, Jr. was born in Richmond, Virginia, on November 1, 1964, the son of Robert H. Bell and Joyce W. Bell. He received his high school diploma from St. Christopher's School in Richmond in 1983, followed by the Bachelor of Arts degree in Mathematics from the University of Virginia in 1987. Following two years working as a programmer-analyst specializing in statistics at Information Management Services, Inc., in Rockville, Maryland, he completed the degree of Master of Science in Electrical Engineering with a major in Computer Engineering at the University of Virginia in January, 1992. He joined the International Business Machines Corporation in Essex Junction, Vermont and transferred to IBM Austin in 1996. He entered the Ph.D. program at the University of Texas at Austin in August, 1999, while working full time at IBM. He is employed by the IBM Systems and Technology Division in Austin, specializing in computer design and performance analysis.

Permanent Address: 9304 Le Conte Cove
Austin, Texas 78749

This dissertation was typed by the author.