The Dissertation Committee for Ajay Manohar Joshi certifies that this is the approved version of the following dissertation:

Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation

Committee:

Lizy K. John, Supervisor

Lieven Eeckhout

Joydeep Ghosh

Stephen W. Keckler

Michael Orshansky

Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation

by

Ajay Manohar Joshi, B.E.; M.S.

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 2007

Dedication

To my family and teachers

Acknowledgements

I would like to thank my advisor, Dr. Lizy John, for her guidance, support, and advice. She has motivated and encouraged me to always strive to do better. Her willingness and availability at all times to discuss ideas, answer questions, and provide feedback has deeply touched me. I am grateful to her for the flexibility and freedom that she gave me throughout my PhD study.

I would also like to thank (in alphabetical order) Dr. Lieven Eeckhout, Dr. Joydeep Ghosh, Dr. Michael Orshansky, and Dr. Stephen Keckler for serving on my dissertation committee and providing invaluable comments and feedback.

I would like to thank Dr. Robert H. Bell Jr. for jump starting my research with long discussions and emails about synthetic benchmarks and bringing me up to speed with the simulation tools and framework.

I had the privilege to collaborate with Dr. Lieven Eeckhout from Ghent University, Belgium. He provided invaluable guidance and has had a profound impact on shaping this dissertation work.

I found great friends and collaborators in Dr. Joshua Yi and Dr. Aashish Phansalkar. Our philosophical discussions about research, graduate school, and life gave the much needed support through several years of graduate school.

I am thankful for the opportunities I had to co-author papers with Dr. Yue Luo. His discipline, dedication, and attention to details have played a vital role in shaping my research methodologies.

I am grateful to Alan MacKay for providing me with the opportunity to intern at International Business Machines Corp. (IBM) and serving as my technical contact and mentor during the IBM PhD Fellowship.

I would also like to thank IBM, National Science Foundation, and Intel for generously funding my research work. Special thanks to The University of Texas, Austin, for the support in the form of Graduate Teaching Assistantship and the David Bruton Jr. PhD Fellowship.

I would like to profusely thank the CART and HPS group at The University of Texas, Austin, for providing me with access to the Alpha servers.

Amy Levin, Melanie Gulick, Deborah Prather, and Shirley Watson were always very prompt and cheerful in helping out with any administrative issues and questions.

I would like to thank other members of the Laboratory for Computer Architecture, Dr. Shiwen Hu, Dr. Madhavi Valluri, Dr. Tao Li, and Dr. Juan Rubio for their words of wisdom and sharing their research experiences, and Lloyd Bircher, Dimitris Kaseridis, Ciji Isen, Jian Chen, Karthik Ganesan, Deepak Pauwar, Arun Nair, Nidhi Nayyar, and Jeff Stuecheli for attending my practice talks and providing invaluable constructive feedback.

I am thankful to David Williamson for providing me with the opportunity to work at ARM Inc. and always taking deep interest in my research work. Working at ARM gave me the hands on experience and exposure in performance modeling, evaluation, and benchmarking.

I am thankful to my sister, Kirti Bhave, and my brother-in-law, Manoj Bhave, for taking interest in my research work, encouraging me, and celebrating all my small and big achievements.

I would also like to thank my parents-in-law for their patience and encouragement through the trying years of graduate life.

I am deeply indebted to my parents for their love, nurturing and support. They always put my interests ahead of theirs and provided me with the opportunity to seek whatever I wanted. My father is a stellar example of rising to the top from adverse conditions and has instilled in me the importance of higher education. My mother always strived for my all round development and provided me with all the opportunities that she herself never had.

Completing a doctorate is as much of an emotional challenge as an intellectual one. I am eternally grateful to my wife, Aparajita, for her emotional support, love, and motivation through the ups and downs of graduate school. She took active interest in my research, helped me refine and shape ideas, proof-read all my papers, and gave feedback on all my presentations. This doctorate is as much hers as mine.

Constructing Adaptable and Scalable Synthetic Benchmarks for Microprocessor Performance Evaluation

Publication No._____

Ajay Manohar Joshi, PhD

The University of Texas at Austin, 2007

Supervisor: Lizy K. John

Benchmarks set standards for innovation in computer architecture research and industry product development. Consequently, it is of paramount importance that the benchmarks used in computer architecture research and development are representative of real-world applications. However, composing such representative workloads poses practical challenges to application analysis teams and benchmark developers - (1) Benchmarks that are standardized are open-source whereas applications of interest are typically proprietary, (2) Benchmarks are rigid, measure single-point performance, and only represent a sample of the application behavior space (3) Benchmark suites take several years to develop, but applications evolve at a faster rate, and (4) Benchmarks geared towards temperature and power characterization are difficult to develop and standardize. The objective of this dissertation is to develop an adaptive benchmark generation strategy to construct synthetic benchmarks to address these benchmarking challenges.

We propose an approach for automatically distilling key hardware-independent performance attributes of a proprietary workload and capture them into a miniature synthetic benchmark clone. The advantage of the benchmark clone is that it hides the functional meaning of the code, but exhibits similar performance and power characteristics as the target application across a wide range of microarchitecture configurations. Moreover, the dynamic instruction count of the synthetic benchmark clone is substantially shorter than the proprietary application, greatly reducing overall simulation time – for the SPEC CPU 2000 suite, the simulation time reduction is over five orders of magnitude compared to the entire benchmark execution.

We develop an adaptive benchmark generation strategy that trades off accuracy to provide the flexibility to easily alter program characteristics. The parameterization of workload metrics makes it possible to succinctly describe an application's behavior using a limited number of fundamental program characteristics. This provides the ability to alter workload characteristics and construct scalable benchmarks that allows researchers to explore a wider range of the application behavior space, conduct program behavior studies, and model emerging workloads.

The parameterized workload model is the foundation for automatically constructing power and temperature oriented synthetic workloads. We show that machine learning algorithms can be effectively used to search the application behavior space to automatically construct benchmarks for evaluating the power and temperature characteristics of a computer architecture design.

The need for a scientific approach to construct synthetic benchmarks, to complement application benchmarks, has long been recognized by the computer architecture research community, and this dissertation work is a significant step towards achieving that goal.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Estimating and comparing the performance of computer systems has always been a challenging task faced by computer architects and researchers. One of the classic and most popular techniques to measure the performance of a computer system is to characterize its behavior when executing a representative workload. Typically, the representative workload is a set of benchmark programs that is believed to be representative of typical applications that could be executed on the computer system. The use of benchmarks for quantitatively evaluating novel ideas, analyzing design alternatives, and identifying performance bottlenecks has become the mainstay in computer systems research and development. A wide range of programs, ranging from microbenchmarks, kernels, hand-coded synthetic benchmarks, to full-blown real-world applications, have been used for the performance evaluation of computer architectures.

Early synthetic benchmarks, such as Whetstone [Curnow and Wichman, 1976] and Dhrystone [Weicker, 1984], had the advantage of being able to consolidate application behaviors into one program. However, these benchmarks fell out of favor in the nineteen-eighties because they were hand coded, hence difficult to upgrade and maintain, and were easily subject to unfair optimizations. Smaller benchmark programs such as microbenchmarks and kernel codes have the advantage that they are relatively easier to develop, maintain, and use. However, they only reflect the performance of a very narrow set of applications and may not serve as a general benchmark against which the performance of real-world applications can be judged. At the other end of the spectrum, the use of real-world applications as benchmarks offers several advantages to architects, researchers, and customers. They increase the confidence of architects and researchers in making design tradeoffs and make it possible to customize microprocessor

design to specific applications. Also, the use of real-world applications for benchmarking greatly simplifies purchasing decisions for customers. As a result researchers began to heavily rely on applications with specific datasets to assess computer performance. Consequently, application programs have now become the dominant benchmarks.

Due to the prohibitive simulation time of application benchmarks there has been a revival of interest in the computer architecture community to develop synthetic workloads [Skadron *et al.,* 2003-1]. Researchers have expended some effort in developing techniques for automatically constructing synthetic workloads which can mimic the performance of longer-running real-world applications [Eeckhout and De Bosschere, 2000] [Oskin *et al.,* 2000] [Bell and John, 2005-3]. The central idea behind these proposed techniques is to measure workload attributes of a benchmark and model them into a synthetic workload. Due to the statistical nature of the synthetic workload, it rapidly converges to a steady-state result. Therefore, the key motivation for these techniques was to reduce the simulation time. The primary shortcoming of these techniques was that the generated workload was not representative across microarchitectures and had to be resynthesized in response to a microarchitectural change.

1.1 MOTIVATION

The motivation of this dissertation is to address the limitations of prevailing workload synthesis approaches, and improve their usefulness to address some acute challenges in benchmarking computer architectures. This will enable a much broader application of the synthetic benchmarks beyond reduction in simulation time. In this section we outline the benchmarking challenges that are the key motivation for this research work.

1.1.1 Proprietary Nature of Real World Applications

If it would be possible to make a real-world customer workload available to architects and designers, computer architecture design tradeoffs could be made with higher confidence. Moreover, if a real world application that a customer cares about was used to project the performance of a microprocessor, it would tremendously increase the customer's confidence when making purchasing decisions. However, many of the critical real world applications are proprietary and customers hesitate to share them with third party computer architects and designers.

1.1.2 Benchmarks Represent a Sample of the Performance Spectrum

The application programs that are being run on computer systems constantly evolve, and given the diversity of these application domains, benchmark programs only represent a sample of the performance spectrum. There may be several application characteristics for which standardized benchmarks do not (yet) exist. This makes it difficult to project the performance of such applications.

1.1.3 Benchmarks Measure Single-Point Performance

A benchmark typically measures the performance of a computer system for a set of workload characteristics. This may make it difficult to get statistical confidence in the evaluation. Typically, it is not easy to vary the benchmark characteristics to understand whether a performance anomaly is an artifact of the benchmark or a characteristic of the underlying system. Moreover, the rigid nature of benchmarks makes it difficult to isolate and study the effect of individual benchmark characteristics on performance.

1.1.4 Applications Evolve Faster than Benchmark Suites

Typically, architects and researchers use prevailing benchmarks to make processor design decisions. However, it is known that as applications evolve, benchmark

characteristics drift with time and an optimal design using benchmarks of today may not be optimal for applications of tomorrow. This problem has been aptly described as: "Designing tomorrow's microprocessors using today's benchmarks built from yesterday's programs" [Weicker, 1997] [Yi *et al.,* 2006-1]. Therefore, it is important for architects and researchers to analyze the effect of workload behavior drift on microprocessor performance. However, developing new benchmark suites and upgrading existing benchmark suites is extremely time-consuming and by consequence very costly. Therefore, it is not possible for the benchmark development process to keep pace with the rate at which new applications emerge.

1.1.5 Challenges in Studying Commercial Workload Performance

Commercial workloads, such as online transaction processing (OLTP) and decision support systems (DSS), which handle day-to-day business transactions form an important class of applications. However, these workloads are complex and have large hardware requirements for full-scale setup. As a result, it is difficult to study the performance of these workloads in simulation based research and during pre-silicon performance and power studies.

1.1.6 Prohibitive Simulation Time of Application Benchmarks

A key challenge in engineering benchmarks from real-world applications is to make them simulation friendly – a very large dynamic instruction count results in intractable simulation times even on today's fastest simulators running on today's fastest machines. Also, application benchmarks that need execution of several other software layers, *e.g.* operating system calls, can be simulated only on a complete system performance model. Therefore, it is usually impossible to execute these applications on Register-Transfer-Language (RTL) models.

1.1.7 Need for Power and Temperature Oriented Stress Benchmarks

Estimating the maximum power and thermal characteristics of a microarchitecture is essential for designing the power delivery system, packaging, cooling, and power/thermal management schemes for a microprocessor. Typical benchmark suites used in performance evaluation do not stress the microarchitecture to the limit, and the current practice in industry is to develop artificial benchmarks that are specifically written to generate maximum processor (component) activity. However, manually developing and tuning such synthetic benchmarks is extremely tedious, requires an intimate understanding of the microarchitecture, and is therefore very time-consuming.

1.2 OBJECTIVES

In this section we outline the key objectives of this dissertation research and highlight how they improve the usefulness of synthetic workloads for computer performance evaluation and benchmarking.

1.2.1 Evaluating the Efficacy of Statistical Workload Modeling

Statistical workload modeling has been proposed as an approach to reduce the time needed to generate quantitative performance estimates early in the design cycle. The key idea in statistical workload modeling is to model a workload's important performance characteristics in a synthetic trace, and execute the trace in a statistical simulator to obtain a performance estimate. Since the performance estimate quickly converges, the simulation speed of statistical simulation makes it an attractive technique to quickly explore a large design space.

Statistical workload modeling forms the foundation of synthetic workload generation. Therefore, one of the objectives of this dissertation is to develop a technique to quantify the representativeness of synthetic workloads. The use of a rigorous approach

to evaluate the efficacy of statistical workload modeling will increase the confidence of computer architects and researchers in the use of synthetic workloads for performance evaluation.

1.2.2 Techniques for Distilling the Essence of Applications into Benchmarks

Prior work [Bell and John, 2005-1] [Bell and John., 2005-2] shows that if a program property is modeled into synthetic workloads using a microarchitecture-dependent feature (*e.g.* generating a cache access pattern to match a target miss-rate) it yields high errors on configurations that are very different from the ones for which they were synthesized. In order to ensure that the generated synthetic workload is representative across a wide range of configurations we choose microarchitecture-independent workload attributes to capture the program property to be modeled. An objective of this research is to develop modeling approaches for incorporating locality and control flow predictability of programs into synthetic workloads. The approach used in these models is to use an inherent program attribute to quantify and abstract code properties related to spatial locality, temporal locality, and branch predictability. These attributes are then used to generate a trace or a benchmark with similar properties. If the feature faithfully captured the program property, the resulting performance metrics *e.g.* cache miss-rate and branch prediction rate will be similar to that of the original application program.

1.2.3 Miniature Workload Synthesis for Early Design Stage Studies

An objective of this research is to develop synthetic benchmarks that not only capture the essence of proprietary workloads but can also be simulated in a reasonable amount of time. The important distinction of this objective from prior research is that our goal is to maintain the representativeness of the synthetic workload across a wide range

of microarchitectures and still achieve the reduction in simulation time. Having synthetic benchmarks that are miniature will enable one to use longer-running benchmarks and applications, which are typically difficult to setup, for early design space performance and power studies. The miniaturization of synthetic benchmarks is also essential to make it possible to execute them on RTL models. The synthetic benchmarks can then be used to effectively narrow down a design space in a tractable amount of time.

1.2.4 Exploring the Feasibility of Parameterized Workload Modeling

A synthetic program that can be tuned to produce a variety of benchmark characteristics would be of great benefit to the computer architecture community. An objective of this research is to develop an adaptive benchmark generation strategy for constructing scalable synthetic benchmarks. Essentially one needs to generate a framework based on a parameterized workload model. The scalable benchmarks provide the flexibility to alter program characteristics and explore and wider range of the application behavior space.

1.2.5 Power and Temperature Oriented Synthetic Benchmarks

Although power, temperature, and energy have recently emerged as first class design constraints, the computer architecture community currently lacks benchmarks that are oriented towards temperature and power characterization of a design. Estimating the maximum power and thermal characteristics of a microarchitecture is essential for designing the power delivery system, packaging, cooling, and power/thermal management schemes for a high-performance microprocessor. An objective of this research is to develop synthetic workloads that are oriented towards evaluating microarchitecture-level power and temperature characteristics. Typical benchmark suites used in performance evaluation do not stress the microarchitecture to its limit, and the

current practice in industry is to develop artificial benchmarks that are specifically written to generate maximum processor (component) activity. However, manually developing and tuning such synthetic benchmarks is extremely tedious, requires an intimate understanding of the microarchitecture, and is therefore very time-consuming. Automatic construction of power and temperature oriented workloads will significantly reduce the time required for power and temperature characterization.

1.3 THESIS STATEMENT

A hardware-independent workload model and an adaptive benchmark generation strategy to construct representative miniature synthetic benchmarks, can be used to disseminate proprietary applications as benchmarks, construct scalable benchmarks to represent emerging workloads, model commercial workloads, and develop power and temperature oriented synthetic benchmarks.

1.4 CONTRIBUTIONS

This dissertation makes a contribution towards advancing state-of-the-art in developing representative synthetic benchmarks and broadening the applicability of synthetic workloads for microprocessor performance evaluation. The contributions from this dissertation have the following impact - (1) enable *computer designers and researchers* to use proprietary and longer-running application programs for power and performance evaluation, (2) help *end users and customers* to project the performance of a proprietary workload on a given microprocessor, (3) foster sharing of benchmarks between *industry and academia*, (4) provide a mechanism for *benchmark designers* to model emerging workloads, and (5) enable *computer architects and compiler designers* to conduct program behavior studies. The following is a summary of the specific contributions from this research work.

This dissertation uses a rigorous statistical approach to systematically evaluate the absolute and relative accuracy of synthetic workload modeling in design space exploration studies. The empirical results obtained from a thorough evaluation of synthetic workload modeling significantly increases the confidence of architects, designers, and researchers in the use of synthetic benchmarks for microprocessor performance evaluation.

An important contribution of this dissertation over prior work in synthetic benchmark generation [Bell and John, 2005-1] [Bell and John, 2005-2] is that we demonstrate that it is possible to capture the performance of a program using an abstract workload model that only uses hardware-independent workload attributes. Since the design of the synthetic workload is guided only by the application characteristics and is independent of any hardware specific features, the workload can be used across a wide range of microarchitectures. We show in our evaluation that the synthetic benchmark clone shows good correlation with the original application across a wide range of cache, branch predictor, and other microarchitecture configurations. This improves the representativeness of the synthetic workload and obviates the need to resynthesize the workload when the underlying microarchitecture is altered.

The ability to capture the essence of a workload only using hardware-independent workload attributes makes it possible to disseminate real-world applications as miniature benchmarks without compromising on the applications' proprietary nature. The advantage of the synthetic benchmark clone is that it provides code abstraction capability, *i.e.,* it hides the functional meaning of the code in the original application but exhibits similar performance characteristics as the real application. Source code abstraction prevents reverse engineering of proprietary code, which enables software developers to share synthetic benchmarks with third parties. The ability to automatically distill key

behavior characteristics of an application into benchmarks is an important contribution of this dissertation. Moreover, automated synthetic benchmark generation significantly reduces the effort of developing benchmarks, making it possible to upgrade the benchmarks more often.

One of the key contributions of this dissertation is our finding that it is possible to fully characterize a workload by only using a limited number of microarchitecture-independent program characteristics, and still maintain good accuracy. This finding enables adaptation of the benchmark generation strategy to tradeoff the representativeness of the synthetic workload in favor of the flexibility to alter program characteristics. Moreover, since these program characteristics are measured at a program level they can be measured more efficiently and are amenable to parameterization. We demonstrate that the parameterized workload synthesis approach is the foundation for constructing scalable benchmarks that can be used to model emerging workloads, and conduct program behavior studies.

This dissertation makes a contribution towards addressing an important industry problem of characterizing the maximum power dissipation and thermal characteristics of a microarchitecture. Typically, hand-coded synthetic streams of instructions have been used to generate maximum activity in a processor to estimate the maximum power dissipation. This dissertation proposes a framework that uses machine learning algorithms to automatically search the program behavior space to generate power and temperature stress benchmarks. We demonstrate that this framework is very effective in constructing stress benchmarks for measuring maximum sustainable power dissipation, maximum single-cycle power dissipation, and temperature hot spots. The automated approach to stress benchmark synthesis can eliminate the time-consuming complex task

of hand-coding a stress benchmark, and also increase the confidence in the quality of the stress benchmark.

1.5 ORGANIZATION

Chapter 2 reviews the prior art in constructing synthetic workloads, approaches used for workload modeling, workload design space exploration, and developing power and temperature oriented workloads.

Chapter 3 applies a statistically rigorous approach for systematically evaluating the representativeness of synthetic workload modeling approaches and quantifies its effectives in exploring a microprocessor design space.

Chapter 4 shows that it is possible to completely capture the essence of an application by only using a set of hardware-independent workload characteristics. It proposes algorithms for modeling the hardware-independent attributes into a synthetic workload.

Chapter 5 applies the workload synthesis approach to construct miniature synthetic clones for longer-running proprietary applications. It provides example results of miniature synthetic clones representative of general purpose, embedded, and scientific benchmarks. It demonstrates the usefulness and applicability of the miniature benchmarks clones for early design stage power and performance studies.

Chapter 6 develops an adaptive benchmark generation strategy for constructing scalable benchmarks that can be used to model emerging applications and conduct performance behavior studies.

Chapter 7 develops a methodology for automatically constructing power and temperature oriented workloads. It demonstrates the usefulness of such benchmarks in evaluating the power and temperature characteristics of a design.

Chapter 8 summarizes the key contributions and results from this dissertation and suggests directions for future research.

Chapter 2: Related Work

This chapter briefly summarizes prior research work in the area of statistical simulation, workload synthesis, workload characterization, power and temperature characterization of microprocessors, approaches for workload design space exploration, and application of statistical techniques in computer architecture research. In each of these areas we compare and contrast this dissertation research to prior work. We also highlight how this dissertation builds upon and addresses limitations of prior research to advance state-of-the-art in workload synthesis.

2.1 STATISTICAL SIMULATION

[Noonburg and Shen, 1997] [Carl and Smith, 1998] [Oskin *et al.*, 2000] introduced the idea of statistical simulation which forms the foundation of synthetic workload generation. The approach used in statistical simulation is to generate a short synthetic trace from a statistical profile of program attributes such as basic block size distribution, branch misprediction rate, data/instruction cache miss rate, instruction mix, dependency distances, *etc.,* and then simulate the synthetic trace using a statistical simulator. The primary objective of these techniques was to reduce simulation time during early design space exploration studies. [Eeckhout and De Bosschere, 2000] improved the accuracy of performance predictions in statistical simulation by measuring conditional distributions and incorporating memory dependencies using more detailed statistical profiles, and guaranteeing syntactical correctness of synthetic traces. [Nussbaum *et al.*, 2001] proposed correlating characteristics such as the instruction type, instruction dependencies, cache behavior, and branch behavior to the size of the basic block. They also compared the accuracy of several models for synthetic trace generation. [Eeckhout *et al.*, 2004-2] [Bell *et al.,* 2004] further improved the accuracy of statistical

simulation by proposing to profile the workload attributes at a basic block granularity and using the statistical flow graph (SFG) to capture the control flow behavior of the program. Recent improvements include more accurate memory data flow modeling for statistical simulation [Genbrugge *et al.*, 2006]. The important benefit of statistical simulation is that the synthetic trace is extremely short in comparison to real workload traces. Overall, the follow up research work in statistical simulation has focused on improving its accuracy by modeling program characteristics at a finer granularity. This has improved the absolute and relative accuracy of the statistical simulation technique, albeit at the cost of increased complexity and profiling cost.

Various studies have demonstrated that statistical simulation is capable of identifying a region of interest in the early stages of the microprocessor design cycle while considering both performance and power consumption [Eeckhout *et al.,*2004-1] [Eeckhout *et al.,* 2004-2] [Genbrugge *et al.*, 2006]. They show that the important application of statistical simulation is to cull a large design space in limited time in search for a region of interest. Although this previous work has shown that statistical simulation has good absolute and relative accuracy and is a viable tool for design space exploration, researchers and architects are reluctant to use statistical simulation due to questions related to the accuracy across diverse set of processor configurations, ability to stress processor bottlenecks, and the tradeoff between accuracy and complexity of statistical workload models.

[Eeckhout *et al.,* 2001] showed that using a combination of analytical and statistical modeling, it is possible to efficiently explore the workload and microprocessor design space. [Oskin *et al.,* 2000] [Nussbaum *et al.,* 2001] have also demonstrated the usefulness of statistical simulation for exploring the application behavior space. However, these techniques use a combination of microarchitecture-independent and

microarchitecture-dependent workload characteristics – limiting the application behavior space that can be explored. The other limitation of statistical simulation is that it generates synthetic traces rather than synthetic benchmarks; synthetic traces, unlike synthetic benchmarks, cannot be executed on execution-driven simulators, real hardware, and RTL models.

2.2 WORKLOAD SYNTHESIS

Early synthetic benchmarks, such as Whetstone [Curnow and Wichman, 1976] and Dhrystone [Weicker, 1984], were hand coded to consolidate application behaviors into one program. Several approaches [Ferrari, 1984] [Sreenivasan and Kleinman, 1974] have been proposed to construct a synthetic workload that is representative of a real workload under a multiprogramming system. In these techniques, the characteristics of the real workload are obtained from the system accounting data, and a synthetic set of jobs are constructed that places similar demands on the system resources. [Hsieh and Pedram, 1998] developed a technique to construct assembly programs that, when executed, exhibit the same power consumption signature as the original application. [Sorenson and Flanagan, 2002] evaluate various approaches to generating synthetic address traces using locality surfaces. [Wong and Morris, 1998] use the hit-ratio in fully associative caches as the main criteria for the design of synthetic workloads. They also use a process of replication and repetition for constructing programs to simulate a desired level of locality of a target application.

The work most closely related to this dissertation is the one proposed by [Bell and John, 2005-3]. They present a framework for the automatic synthesis of miniature benchmarks from actual application executables. The key idea of this technique is to capture the essential structure of a program using statistical simulation theory, and generate C-code with assembly instructions that accurately model the workload

attributes, similar to the framework proposed in this dissertation. The reduction in simulation time gained from the synthetic benchmarks and their ability to be executed on execution driven simulators and RTL models was applied to, power analysis in early design space studies [Bell and John, 2005-1], and validation of a performance model against the RTL [Bell and John, 2005-2] [Bell *et al.,* 2006]. This approach models memory access patterns and control-flow behavior to match a target metric *e.g.* cache miss-rate, branch prediction rate *etc.*, and hence the synthetic workloads also reflect machine properties rather than pure program characteristics. Consequently, the synthetic workloads generated from these models may yield large errors when cache and branch microarchitecture configurations are changed from the targeted configuration [Bell and John, 2005-2]. Therefore, in order to enable the portability of the generated synthetic workload across a wide range of microarchitectures, it is important to capture inherent program characteristics into the synthetic workload rather than generate a synthetic workload to match a target metric. This dissertation significantly improves the usefulness of this workload synthesis technique by developing hardware-independent models for capturing locality and control flow predictability of programs into synthetic workloads.

Approaches to generate synthetic workloads have been investigated for performance evaluation of I/O subsystems, file system, networks, and servers [Bodnarchuk and Bunt, 1991] [Barford and Crovella, 1998] [Ganger, 1995] [Kurmas *et al.,* 2003]. The central idea in these approaches is to model the workload attributes using a probability distribution such as Zipf's law, binomial distribution, *etc.,* and to use these distributions to generate a synthetic workload.

[Keeton and Patterson, 1998] [Shao *et al.,* 2005] studied the characteristics of commercial workloads and hand crafted scaled down microbenchmarks that are representative of commercial workloads. The approach proposed in this dissertation has

a similar objective but does so automatically. This significantly reduces the time required to construct a representative synthetic benchmark.

[Chen and Patterson, 1994] developed an approach to generate parameterized self-scaling I/O benchmarks that can dynamically adjust the workload characteristics according to the performance characteristic of the system being measured. Automatic test case synthesis for functional verification of microprocessors [Bose, 1998] has been proposed, and there has been prior work on hand crafting microbenchmarks for performance validation [Desikan *et al.,* 2001] [Bose and Abraham, 2000].

2.3 WORKLOAD CHARACTERIZATION

[John *et al.,* 1998] advocates the need for understanding the characteristics of workloads in order to design efficient computer architectures. This article provides an excellent survey on prior work in workload characterization and stresses the importance of developing architecture-independent workload metrics. Further more, the paper proposes an idea for developing an architecture-independent workload model and using it to generate benchmarks. This dissertation is a significant step towards achieving the goals and vision outlined in this article.

[Weicker, 1990] used characteristics such as statement distribution in programs, distribution of operand data types, and distribution of operations, to study the behavior of several stone-age benchmarks. [Saveedra and Smith, 1996] characterized FORTRAN applications in terms of number of various fundamental operations, and predicted their execution time. Source code level characterization has not gained popularity due to the difficulty in standardizing and comparing the characteristics across various programming languages.

There has been research on microarchitecture-independent locality and ILP metrics. For example, locality models researched in the past include working set models,

least recently used stack models, independent reference models, temporal density functions, spatial density functions, memory reuse distance, and locality space [Conte and Hwu, 1990] [Lafage and Seznec, 2000] [Spirn, 1972] [Sorenson and Flanagan, 2002] [John *et al.,* 1998] [Denning, 1968]. Generic measures of ILP based on the dependency distance in a program have been used by [Noonburg *et. al.,* 1997] and [Dubey *et. al*., 1994]. Microarchitecture-independent characteristics such as, true computations versus address computations, and overhead memory accesses versus true memory accesses have been proposed by several researchers [Hammerstrom and Davidson, 1997] [John *et al.,* 1995].

Microarchitecture-independent characteristics have also been used for measuring program similarity, benchmark subsetting, finding program phases, and performance prediction [Joshi *et al.,* 2006-3] [Phansalkar *et al.,* 2005] [Eeckhout *et al.,* 2005] [Hoste *et al.,* 2006-1] [Hoste and Eeckhout, 2006] [Sherwood *et al.,* 2002] [Lafage and Seznec, 2000] [Luo *et al.,* 2005].

2.4 OTHER APPROACHES TO REDUCE SIMULATION TIME

Statistical sampling techniques [Conte *et al.,* 1996] [Wunderlich *et al.,* 2003] have been proposed for reducing the cycle-accurate simulation time of a program. The central idea in these approaches is to use statistical sampling theory to reduce simulation time and provide a confidence interval for the estimated performance. The SimPoint project [Sherwood *et al.*, 2002] proposed basic block distribution analysis for finding program phases which are representative of the entire program. The SimPoint approach can be considered orthogonal to the approach proposed in this dissertation, because one can generate a synthetic benchmark clone for each phase of interest.

[Iyengar *et al.*, 1996] developed a concept of fully qualified basic blocks and applied it to generate representative traces for processor models with infinite cache. This work was later extended [Iyengar and Trevillyan, 1996] to generate address traces to match a particular cache miss-rate. [Ringenberg *et al.*, 2005] developed a technique, intrinsic checkpointing, a checkpoint implementation that loads the architectural state of a program by instrumenting the simulated binary rather than through explicit simulator support. This technique makes it possible to execute the simulation point(s) of a program on real hardware or an execution-driven simulator. The synthetic benchmark approach proposed in this dissertation has advantages over these simulation points, namely (i) hiding the functional meaning of the original application, and (ii) having even shorter dynamic instruction counts.

2.5 POWER AND TEMPERATURE CHARACTERIZATION OF MICROPROCESSORS

A lot of research work has been done in the VLSI research community to develop techniques for estimating the power dissipation of a CMOS circuit. The primary approach in these techniques is to use statistical approaches, heuristics, and to develop a test vector pattern that causes maximum switching activity in the circuit [Rajgopal, 1996] [Najm *et al.,* 1995] [Chou and Roy, 1996] [Tsui *et al.,* 1995] [Qui *et al.,* 1998] [Hsiao *et al.,* 2000] [Lim *et al.,* 2002]. Although an objective of this dissertation is the same as this prior work, there are two key differences compared to our work. Firstly, our technique aims at developing an assembly test program (compared to a test vector) that can be used for maximum power estimation at the microarchitecture level. Secondly, developing stress benchmarks provides insights into the interaction of workload attributes and power/thermal stress, which is not possible with a bit vector. [Vishwanathan *et al.,* 2000] [Gowan *et al.,* 1998] [Felter and Keller, 2004] refer to hand-crafted synthetic test cases developed in industry that have been used for estimating maximum power dissipation of

a microprocessor. In [Lee *et al.,* 2005], stress benchmarks have been developed to generate temperature gradients across microarchitecture units.

2.6 STATISTICAL AND MACHINE LEARNING TECHNIQUES IN COMPUTER PERFORMANCE EVALUATION

[Yi *et al.*, 2003] proposed to use the Plackett & Burman (P&B) design to choose processor parameters, to select a subset of benchmarks, and to analyze the effect of a processor enhancement. Also, [Yi *et al.*, 2005] [Yi *et al.,* 2006-3] used the P&B design as a characterization technique to compare simulation techniques and characterize the bottlenecks in SPEC CPU2000 benchmark programs.

Principal Component Analysis (PCA) and Clustering Techniques have been applied to measure the similarity between programs and find a subset of representative programs to reduce simulation time and in benchmark suite design [Phansalkar *et al.,* 2007-1] [Phansalkar *et al.,*2007-2] [Yi *et al.,* 2006-2] [Eeckhout *et al.,* 2005] [Eeckhout *et al.,* 2003-2] [Eeckhout *et al.,* 2002] [Joshi *et al.,* 2006-3]. [Sherwood *et al.,* 2002] also use clustering techniques to find phase behavior in a program.

[Hoste *et al.,* 2006-1] applied genetic learning algorithm to find weights for benchmarks to predict the performance of a customer application. [Eyerman *et al.,* 2006] used different machine learning algorithms to explore a microprocessor design space.

Chapter 3: Evaluating the Efficacy of Statistical Workload Modeling

Recent research has proposed statistical simulation as a technique for fast performance evaluation of superscalar microprocessors. Statistical workload modeling is the foundation for developing synthetic benchmarks proposed in this dissertation. The idea in statistical simulation is to measure a program's key performance characteristics, generate a synthetic trace with these characteristics, and simulate the synthetic trace. Due to the probabilistic nature of statistical simulation the performance estimate quickly converges to a solution, making it an attractive technique to efficiently cull a large microprocessor design space. Therefore, it is important to improve confidence in the use of synthetic workloads by evaluating the effectiveness of statistical workload modeling approaches and the trade-offs involves therein.

In this chapter, we evaluate the efficacy of statistical workload modeling approaches in exploring the design space. Specifically, we characterize the following aspects of statistical workload modeling: (i) fidelity of performance bottlenecks, with respect to cycle-accurate simulation of the program, (ii) ability to track design changes, and (iii) trade-off between accuracy and complexity in statistical workload models.

3.1 Introduction to Statistical Workload Modeling

In computer architecture, the simulation of benchmarks is a widely used technique for evaluating computer performance. Computer architects and researchers use microprocessor models to accurately make performance projections during the pre-silicon phase of the chip design process, and also to quantitatively evaluate microprocessor innovations. Unfortunately, when using a detailed cycle-accurate performance model, the simulation time may span several weeks or months. Further compounding this problem is the growing complexity of microarchitectures (*i.e.*, decreasing simulation

speed) and the increasing execution-times of modern benchmarks. Therefore, in order to meet the time-to-market requirements of a microprocessor, designers use different simulation models during the various stages of the design cycle. Although a detailed and highly accurate cycle-accurate simulator is necessary to evaluate specific design points later in the design cycle, earlier in the design cycle, a simulation technique that has a short development time and can quickly provide performance estimates with reasonable accuracy is desirable.

Statistical simulation [Oskin *et al.,* 2000] [Eeckhout and De Bosschere*,* 2000] [Nussbaum and Smith, 2001] has been proposed as an approach to reduce the time needed to generate quantitative performance estimates early in the design cycle. The basic idea in statistical simulation is to model a workload's important performance characteristics with a synthetic trace, and execute the trace in a statistical simulator to obtain a performance estimate. Since the performance estimate quickly converges, the simulation speed of statistical simulation makes it an attractive technique to quickly explore a large design space.

Although previous work has shown that statistical simulation has good absolute and relative accuracy and is a viable tool for design space exploration [Eeckhout *et al.,* 2004-1] [Eeckhout *et al.,* 2004-2] [Nussbaum and Smith, 2001], researchers and architects are reluctant to use statistical simulation due to questions such as: (i) What is the absolute and relative accuracy across a diverse set of processor configurations?, (ii) Does the synthetic trace stress the same bottlenecks as the original program to the same degree?, and (iii) What is the trade-off between simulation accuracy and the complexity of various statistical simulation models?

The remainder of this chapter is organized as follows: Section 3.2 presents a brief overview of statistical simulation and the framework we have used in this study. Section

3.3 describes the benchmarks used for the evaluation experiments. Section 3.4 presents the results from our evaluation of statistical simulation. Section 3.5 summarizes the key findings.

3.2 STATISTICAL SIMULATION FRAMEWORK

We developed an enhanced version of HLS++ [Bell and John, 2004] statistical simulation framework, called SS-HLS++, as our statistical simulation environment. It consists of three steps: 1) Profiling the benchmark program to measure a collection of its execution characteristics to create a *statistical profile*, 2) Using the statistical profile to generate a *synthetic trace*, and 3) Simulating the instructions in the synthetic trace on a trace-driven simulator to obtain a performance estimate. Figure 3.1 illustrates these steps.

In the first step, we characterize the benchmark by measuring its microarchitecture-independent and microarchitecture-dependent program characteristics. The former is measured by functional simulation of the program; examples include: instruction mix, basic block size, and the data dependency among instructions. Note that these characteristics are related only to the functional operation of the benchmark's instructions and are independent of the microarchitecture on which the program executes. On the other hand, the microarchitecture-dependent characteristics include statistics related to the locality and branch behavior of the program. Typically, these statistics include L1 I-cache and D-cache miss-rates, L2 cache miss-rates, instruction and data TLB miss-rates, and branch prediction accuracy. The complete set of microarchitecture-dependent and microarchitecture-independent characteristics form the statistical profile of the benchmark.

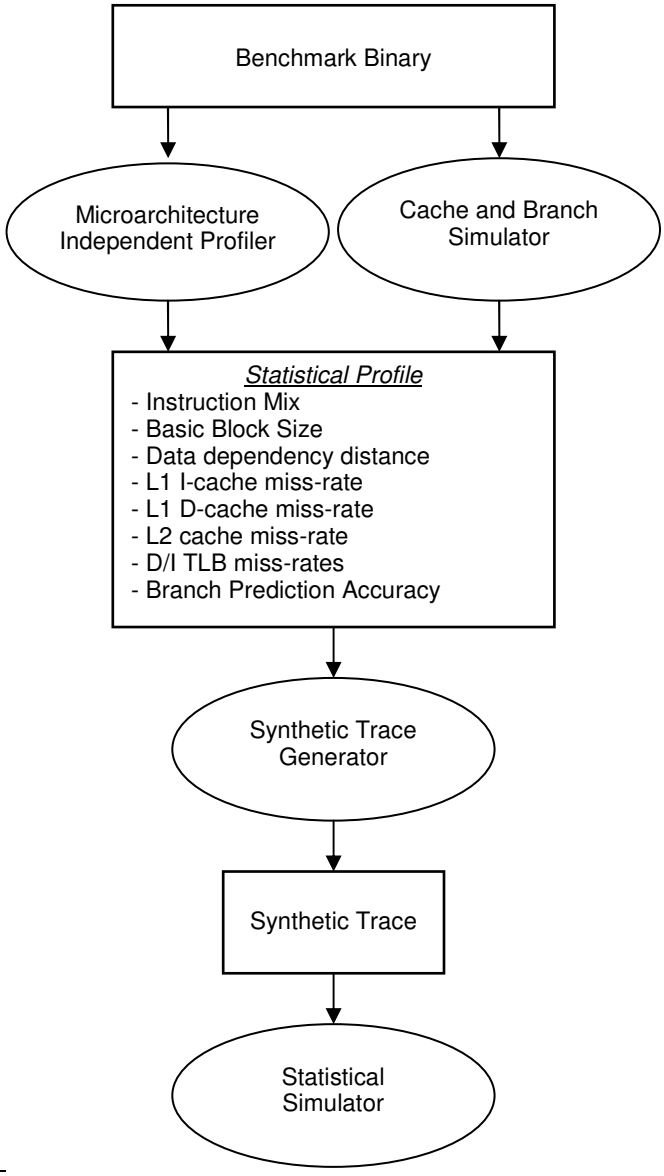


Figure 3.1: SS-HLS++ statistical simulation framework

After generating the statistical profile, the second step is to construct a synthetic trace with similar statistical properties as the original benchmark. The synthetic trace consists of a number of instructions contained in basic blocks that are linked together into a control flow graph, similar to conventional code. However, instead of actual arguments

and opcodes, each instruction in the synthetic trace is composed of a set of statistical parameters, such as: instruction type (integer add, floating-point divide, load, *etc.*), ITLB/L1/L2 I-cache hit probability, DTLB/L1/L2 D-cache hit probability (for load and store instructions), probability of branch misprediction (for branch instructions), and dynamic data dependency distance (to determine how far a consumer instruction is away from its producer). The values of the statistical parameters describing each instruction are assigned by using a random number generator following the distributions of the various workload characteristics in the statistical profile of the benchmark.

Finally, in the third step, the synthetic trace is executed on a trace-driven statistical simulator. The statistical simulator is similar to a trace-driven simulator of real program traces, except that the statistical simulator probabilistically models cache misses and branch mispredictions. During simulation, the misprediction probability that is assigned to the branch instruction is used to determine whether the branch is mispredicted, and if so, the pipeline is flushed when the mispredicted branch executes. Likewise, for every load instruction and instruction cache access, the simulator assigns a memory access time depending on whether it probabilistically hits or misses in the L1 and L2 cache.

Although, these statistical simulation models that have been recently proposed differ in the complexity of the model used to generate the synthetic trace, fundamentally, each model uses the same general framework described in Figure 3.1. They primarily differ in the granularity (basic block level, program level, *etc.*) at which they measure the workload characteristics in the statistical profile. For this study, we implemented the following four statistical simulation models:

HLS [Oskin *et al.,* 2000]**:** This is the simplest model where the workload characteristics (instruction mix, basic block size, cache miss-rates, branch misprediction

rate, and dependency distances) are averaged over the entire execution of a program. This model assumes that the workload characteristics are independent of each other and are normally distributed. A synthetic trace of 100 basic blocks is then generated from a normal distribution of these workload statistics and simulated on a general superscalar execution model until the results (Instructions-Per-Cycle) converge. Since the synthetic instructions are few in number and are probabilistically generated, the results converge very quickly.

HLS + BBSize: We implemented a slightly modified version of the model proposed in [Nussbaum and Smith, 2001]. In this model, other than the basic block size, all workloads characteristics are averaged over the entire execution of the program. However, for the basic block size, we maintain different distributions of the basic block size based on the history of recent branch outcomes.

Zeroth Order Control Flow Graph (CFG, k=0) [Eeckhout *et al.,* 2004] [Bell *et al.,* 2004]: In this modeling approach, we average the workload characteristics at the basic block granularity (instead of averaging them over the entire execution of the program). While building the statistical profile, we create a control flow graph of the program. This control flow graph stores the dynamic execution frequencies of each unique basic block along with the transition probabilities to its successor basic blocks. The workload characteristics (instruction mix, cache miss-rates *etc.*) are measured for each basic block. Since the statistical profile is now at the basic block level, the size of the profile for this model is considerably larger than for the first two. When generating a synthetic trace, we probabilistically navigate the control flow graph and generate synthetic instructions based on the workload characteristics that were measured for each basic block.

First Order Control Flow Graph (CFG, k=1) [Eeckhout *et al.,* 2004]: This is the state-of-the art modeling approach. This approach is the same as the one described in the *Zeroth Order Control Flow Graph* model described above, except that all workload characteristics are measured for each unique pair of predecessor and successor basic blocks in the control flow graph, instead of just for a unique single basic block. Gathering workload characteristics at this granularity improves the modeling accuracy in the synthetic trace because the performance of a basic block depends on the context (predecessor basic block) in which it was executed.

The *First Order Control Flow Graph* model is the state-of-the-art statistical simulation model, and we therefore use it in all the experiments. In Section 3.4.3, we compare the accuracy of the other three models described above against the accuracy of the *First Order Control Flow Graph* model.

3.3 BENCHMARKS

We used 9 benchmark programs and their reference input sets from the SPEC CPU 2000 benchmark suite to evaluate the statistical workload models. All benchmark programs were compiled using `SimpleScalar's` version of the `gcc` compiler, version 2.6.3, at optimization level `-O3`. Table 3.1 lists the programs, their input sets, and benchmark type. In order to compare the statistical simulation results for the configurations used in P&B design to the corresponding results from a cycle-accurate simulator, we had to run 44 cycle-accurate simulations of reference input sets for every benchmark program. To reduce this simulation time, we simulated the first one billion instructions only for each benchmark.

Table 3.1: SPEC CPU 2000 benchmarks and input sets used to evaluate statistical workload modeling.

Benchmark	Input Set	Type
175.vpr-Place	ref.net	Integer
175.vpr-Route	ref.arch.in	Floating-Point
176.gcc	166.i	Integer
179.art	-startx 110	Floating-Point
181.mcf	ref.in	Integer
183.equake	ref.in	Floating-Point
253.perlbmk	diffmail	Integer
255.vortex	lendian1	Integer
256.bzip2	ref.source	Integer

3.4 EVALUATING STATISTICAL SIMULATION

In this section we characterize and evaluate the accuracy of statistical simulation. The objective of our characterization is to analyze the efficacy of statistical simulation as a design space exploration tool by stressing it using a number of aggressive configurations. Using aggressive configurations affords us an opportunity to evaluate the accuracy of statistical simulation by systematically exposing a diverse set of processor performance bottlenecks.

Our evaluation consists of three parts: In Section 3.4.1 we evaluate the ability of statistical simulation to identify important processor performance bottlenecks. Specifically, we use a Plackett & Burman (P&B) design that uses a number of diverse configurations to evaluate the representativeness of the synthetic trace in terms of its performance bottlenecks. In Section 3.4.2, we measure the relative accuracy of statistical simulation by examining its ability to accurately track design changes across 44 aggressive processor configurations. Finally, in Section 3.4.3, we measure the absolute and relative accuracy of the four previously described statistical simulation models, and discuss the trade-offs between their complexity and level of accuracy.

3.4.1 Identifying Important Processor Bottlenecks

Due to their inherent characteristics, different benchmark programs stress different processor performance bottlenecks to different degrees. Since architects use benchmark programs to make quantitative evaluations of various points in the design space and propose processor enhancements to relieve specific performance bottlenecks, the synthetic trace used in statistical simulation should have the same key microprocessor performance bottlenecks that are present when simulating the benchmark on a cycle-accurate simulator. We quantify the representativeness of the synthetic trace by quantifying the difference between the bottlenecks stressed by the original workload and the synthetic trace.

For architects, the P&B design [Plackett and Burman, 1946] [Yi *et al.,* 2003] can determine which processor and memory parameters have the largest effect on performance (cycles-per-instruction) *i.e.*, identify the biggest performance bottlenecks. The P&B design is a very economical experimental design technique that varies N parameters simultaneously over approximately (N + 1) simulations. Based on the results of the P&B design, we assign a rank for each performance bottleneck based on its *P&B magnitude*. The P&B magnitude represents the significance of that bottleneck, or more specifically, the effect that the bottleneck has on the variability in the output value, *e.g.*, cycles-per-instruction (CPI). The bottleneck that has the largest impact on the CPI, *i.e.*, the microarchitectural parameter with the highest P&B magnitude, is the largest performance bottleneck in the processor core and memory subsystem. Based on their significance, we assign a rank to each bottleneck, *i.e.*, the most significant bottleneck has a rank of 1, while the least significant has a rank of N.

We evaluated 43 parameters in an out-of-order superscalar microprocessor related to the L1 I-cache, L1 D-cache, L2 cache, instruction and data TLB, branch predictor

configuration, integer execution units, and floating point execution units. To determine the P&B magnitude, and subsequently the rank, of each bottleneck, we use 44 very different processor configurations. The configurations represent the "envelope of the hypercube" of processor configurations and provide a stress test for statistical simulation by systematically exposing diverse performance bottlenecks. To characterize bottlenecks, the input parameter values were set to low and high values that were similar to those found in [Yi *et al.,* 2003]. To quantify the representativeness of the synthetic trace, we first vectorize the ranks (from statistical simulation and cycle-accurate simulation) and then compute the Euclidean distance between the pair of vectors. Smaller Euclidean distances indicate that the ranks from statistical simulation are very similar to those obtained by simulating the program with a cycle-accurate simulator. When the vectors of ranks are identical (*i.e.,* the significance of each bottleneck is the same for both statistical and cycle-accurate simulation), the Euclidean distance is 0. When the ranks are completely "out-of-phase" (*i.e.* <1, 2, 3 … 41, 42, 43> versus <43, 42, 41 … 3, 2, 1>), the Euclidean distance is at a maximum of 162.75. We normalize the Euclidean distance between each pair of vectors to this maximum, and then scale the distance to a 0 to 100 range.

Since the ranks for all bottlenecks are included in the Euclidean distance, insignificant bottlenecks may deceptively inflate the Euclidean distance. Additionally, one has to be careful when interpreting the results based only on the ranks of the parameters. It is possible that while the Euclidean distance is fairly high, their significance may be, in fact, quite similar. In such cases, seemingly large Euclidean distances are the result of quantization error due to using ranks. To avoid such a pitfall, we also separately present the normalized Euclidean distance for the most significant 3, 5, 10, and 20 parameters, in addition to all 43.

Figure 3.2 shows the normalized Euclidean distance for the 9 benchmarks. The results in this figure show that statistical simulation can identify the 10 most important bottlenecks for all programs with good accuracy (normalized Euclidean distance less than or equal to 15). For all 43 bottlenecks, the accuracy is very high for `179.art`, good for `176.gcc` and `183.equake`, moderate for `175.vpr-Place`, `175.vpr-Route`, and `253.perlbmk`, and poor for `181.mcf`, `255.vortex`, and `256.bzip2`.

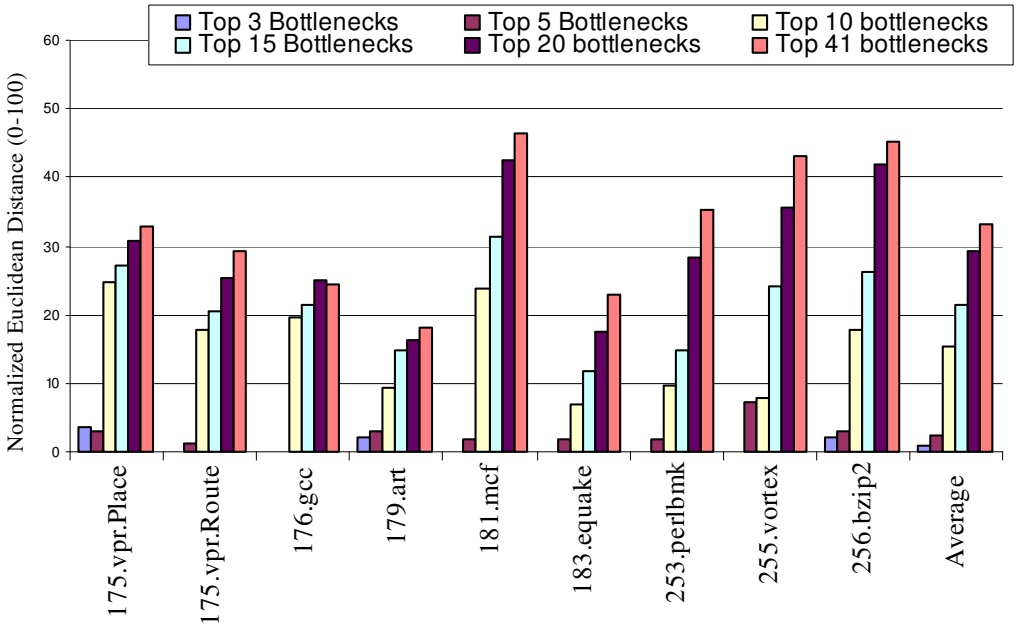


Figure 3.2: Normalized Euclidean distance (0 to 100) between the ranks of processor and memory performance bottlenecks estimated by statistical simulation and cycle-accurate simulation. Smaller Euclidean distances imply higher representativeness of synthetic trace.

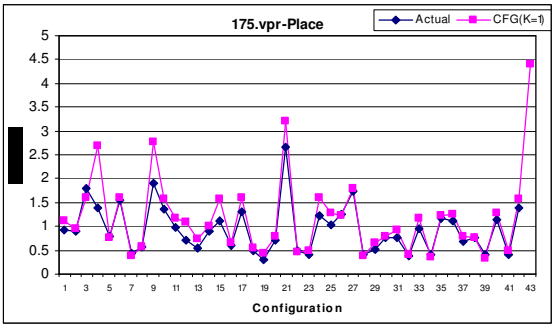
In order to understand the reasons for the difference in level of accuracy of statistical simulation for different programs, we analyzed the absolute values of the P&B magnitude. For `179.art` and `183.equake`, the absolute values of P&B magnitudes for the most important and the least important parameters ranges from 138 (L2 cache size) to 1 (the number of Return Address Stack entries) and 80 (L1 I-cache size) to 0.4 (I-

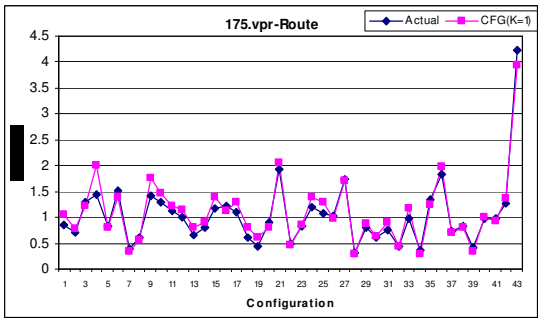
TLB associativity), respectively. Note that larger differences in the P&B magnitudes imply larger performance impacts for that bottleneck. Therefore, in benchmarks such as `179.art` and `183.equake`, the importance of the most and least significant parameter is very distinct. However, for `256.bzip2`, the difference in the significance of the bottlenecks is less distinct since the range of magnitudes is only 16. Since the importance of the most and least significant bottlenecks is not substantially different, incorrectly estimating the importance of bottlenecks that have relatively little impact on the CPI does not imply any additional inaccuracy on the part of statistical simulation.

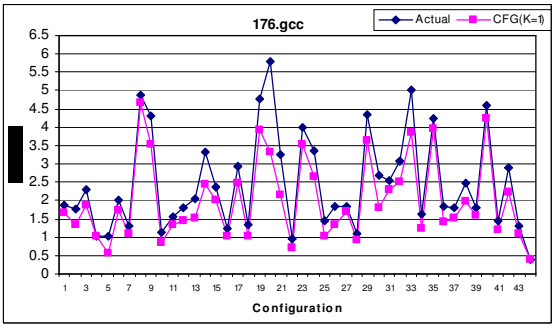
In any case, the primary goal of early design space studies is to identify a range of feasible design values for the most important performance bottlenecks. Since we observe that statistical simulation can do so, we conclude that statistical simulation is useful during early design space exploration. For programs such as `176.gcc`, `179.art`, and `183.equake`, since the synthetic trace is very representative for all 43 bottlenecks stressed by the original benchmark program, statistical simulation may be a valuable tool even beyond the earliest stages of the design space exploration studies.

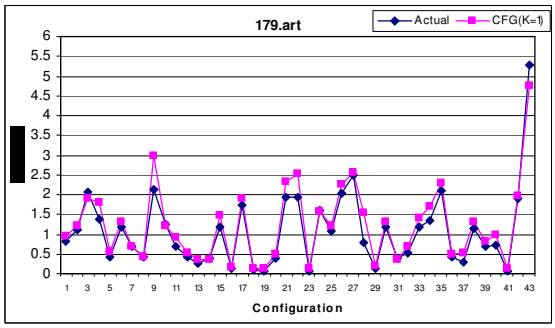
3.4.2 Tracking Design Changes

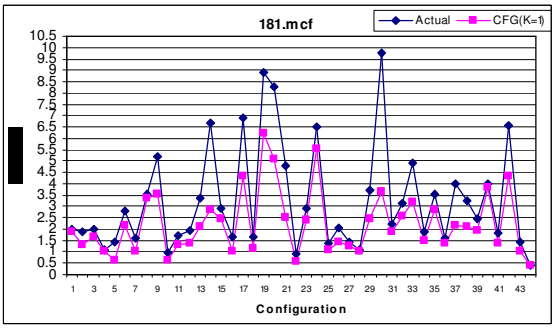
During early design space exploration, the ability of a simulation technique, *e.g.*, statistical simulation, to accurately predict a performance trend, is a very important feature. Or, in other words, the relative accuracy of statistical simulation is more important than its absolute accuracy. If a simulation technique exhibits good relative accuracy, it means that the technique will accurately track performance changes, and therefore can help to identify the interesting design points that need to be further analyzed using detailed simulation.

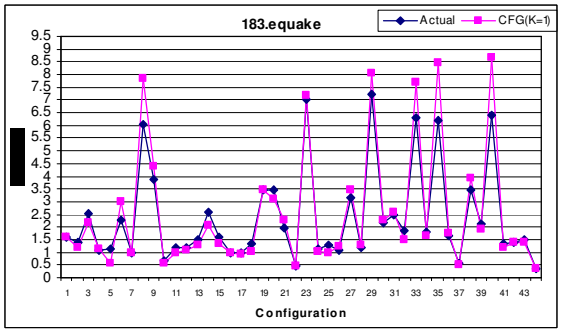

175.vpr-Place
Actual
CFG(K=1)
Configuration

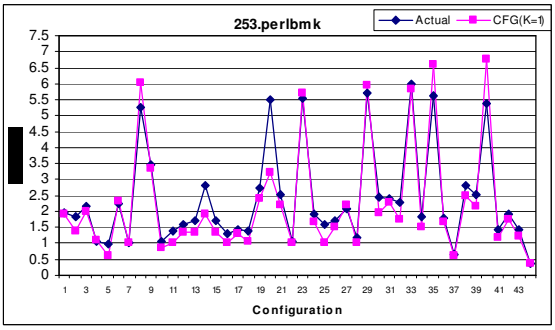

175.vpr-Route
Actual
CFG(K=1)
Configuration

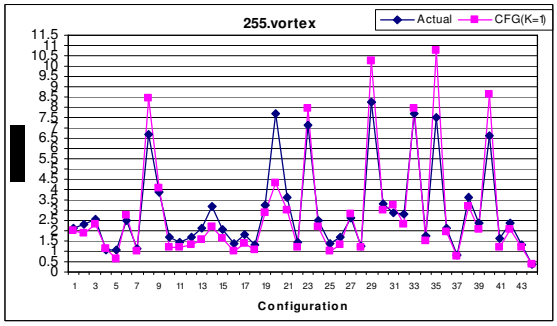

176.gcc
Actual
CFG(K=1)
Configuration


179.art
Actual
CFG(K=1)
Configuration


181.mcf
Actual
CFG(K=1)
Configuration


183.equake
Actual
CFG(K=1)
Configuration


253.perlbmk
Actual
CFG(K=1)
Configuration


255.vortex
Actual
CFG(K=1)
Configuration

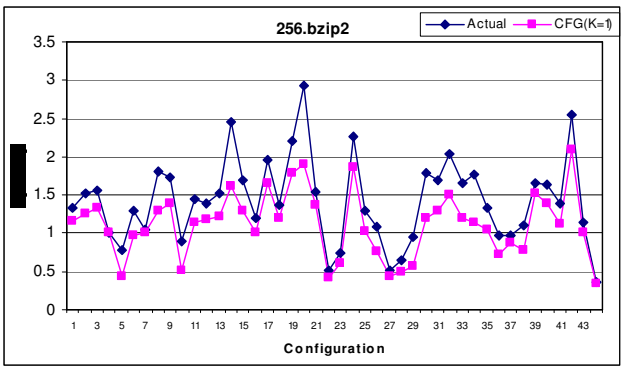


Figure 3.3: Actual and estimated speedup across 43 configurations for 9 SPEC CPU2000 benchmarks.

To evaluate the relative accuracy, we used the 44 P&B configurations that represent a wide range of processor configurations. It is important to note that while these processor configurations are not realistic, they enable us to evaluate whether statistical simulation is accurate enough to track the processor's performance across a wide range of configurations. The approach that we used to characterize the relative accuracy of statistical simulation was to examine the *correlation between the estimated and actual ranking of the configurations.* In particular**,** we measured the speedup in CPI obtained from statistical simulation and cycle-accurate simulation for 43 configurations relative to the 44^{th} configuration, and ranked the 43 processor configurations in descending order of their speedups. Figure 3.3 shows the individual speedups for each configuration for all benchmark programs. We observe that in general, across all programs, statistical simulation tracks both local and global speedup minima/maxima extremely well.

We now use Spearman's rank correlation coefficient to measure the relation between the ranks estimated by cycle-accurate and statistical simulation. The Spearman's rank correlation coefficient is calculated as:

$$R_S = 1 - 6 \sum d_i^2 / (n^3 - n) \quad \text{.....(3.1)}$$

where d_i is the difference between ranks estimated for i^{th} configuration and n is the total number of configurations. The value of R_S ranges from -1 to 1. A value of 1 for R_S indicates that statistical simulation correctly estimated the ranks for all configurations (highest relative accuracy), and a value of -1 means that the ranks estimated by statistical simulation are perfectly opposite to the ones estimated from cycle-accurate simulation (lowest relative accuracy).

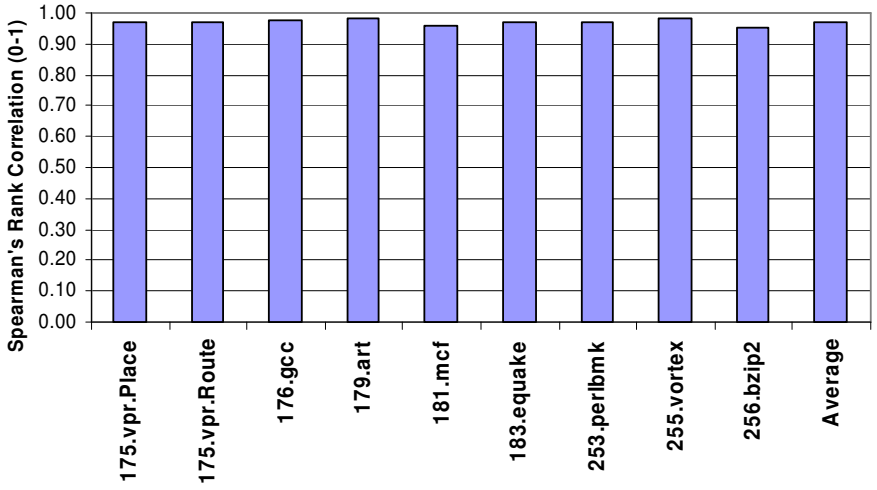


Figure 3.4: Relative Accuracy in terms of Spearman's correlation coefficient between actual and estimated speedups across 43 processor configurations

Figure 3.4 shows that the relative accuracy is very good for all programs (> 0.95). This suggests that for all programs, the ranks for the 43 configurations estimated by statistical simulation are very similar to the ranks estimated from cycle-accurate simulation.

From these results, we conclude that statistical simulation can be effectively used to narrow down a large design space to a few feasible design points. Subsequently, the

architect can use a more accurate simulation technique to further study these feasible design points.

3.4.3 Comparing the Accuracy of Statistical Simulation Models

Researchers have proposed a number of different statistical simulation models that mainly differ in the complexity of the model used to generate the synthetic trace. Fundamentally, each model uses the same general framework described in Figure 3.1 and is a refinement of the basic approach to statistical simulation.

Intuitively, increasing the degree-of-detail in the model should improve the representativeness of the synthetic trace and thus its absolute accuracy. However, what is not clear is how the additional modeling affects the relative accuracy, and whether there is a good trade-off between the model's complexity and its associated absolute and relative accuracy. In this section, we compare the following 4 modeling approaches, described in Section 3.2, namely: *HLS, HLS+BBSize, Zeroth Order Control Flow Graph (CFG, k=0),* and *First Order Control Flow Graph (CFG, k=1).*

We use the 44 P&B configurations to evaluate and compare the absolute error, relative accuracy, and the ability to identify important processor bottlenecks of the four models. The absolute error (AE) is computed as the percentage error in CPI between cycle-accurate simulation (CS) and statistical simulation (SS), which is:

$$AE = (| CPI_{CS} - CPI_{SS}|) * 100 / CPI_{CS} \quad \text{.....(3.2)}$$

To calculate the relative accuracy, we use the R_S measure of relative accuracy as described in equation (3.1). To measure the fidelity of the processor bottlenecks, we compute the Normalized Euclidean. Distance between the ranks of the bottlenecks from cycle-accurate simulation and statistical simulation for the most significant 5, 20, and all 43 bottlenecks.

Figure 3.5 shows that increasing the level-of-detail in the statistical simulation model improves the absolute accuracy for all benchmarks. For the simplest model, *HLS*, the AE is 36.8%; for the *First Order Control Flow Graph (CFG, k=1),* the most sophisticated model, the AE is 16.7%. Therefore, if the primary goal is high absolute accuracy, a computer architect should use as detailed a statistical simulation model as possible to generate the synthetic traces. It is very important to note that the average error of 16.7% for the state-of-the-art statistical simulation model is for the 44 aggressive, unrealistic configurations. (Note that from our experiments with using balanced (realistic) configurations, the average absolute error is 11% for the *First Order Control Flow Graph (CFG, k=1)* statistical simulation model, which is very similar to the level of accuracy in previously published work [Eeckhout *et al.,* 2004]).

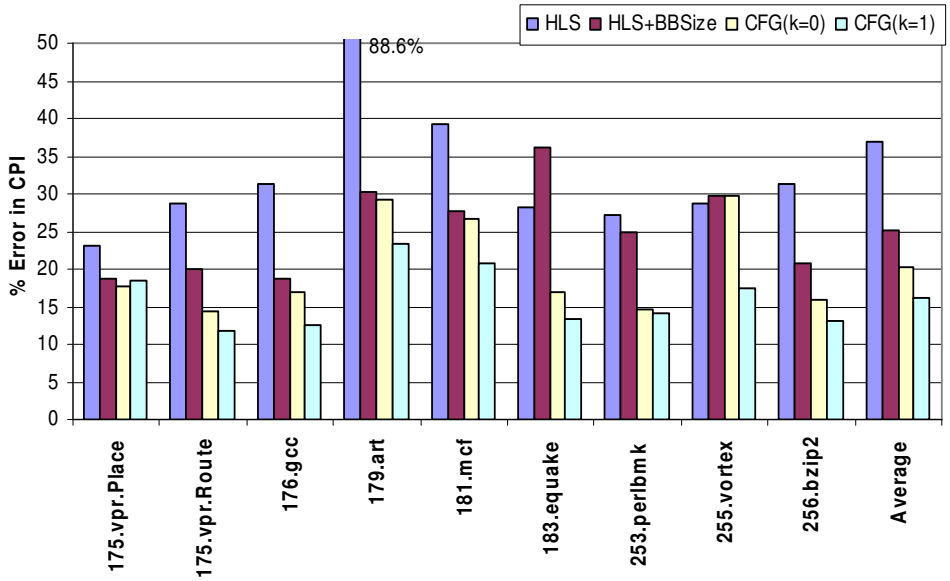


Figure 3.5: Comparison between absolute accuracy of 4 statistical simulation models on the 44 extreme processor configurations

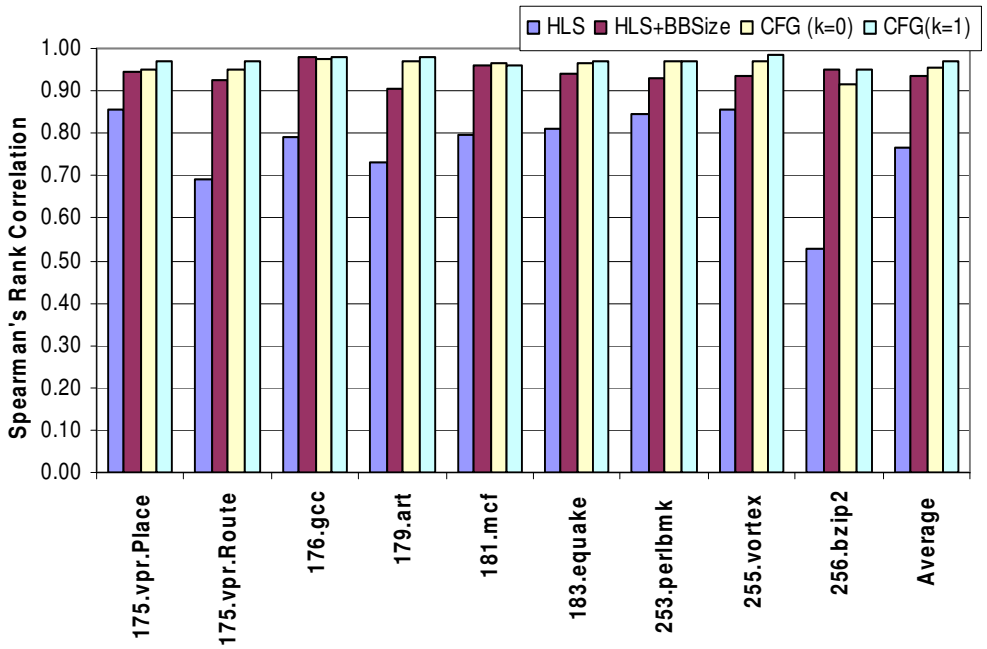
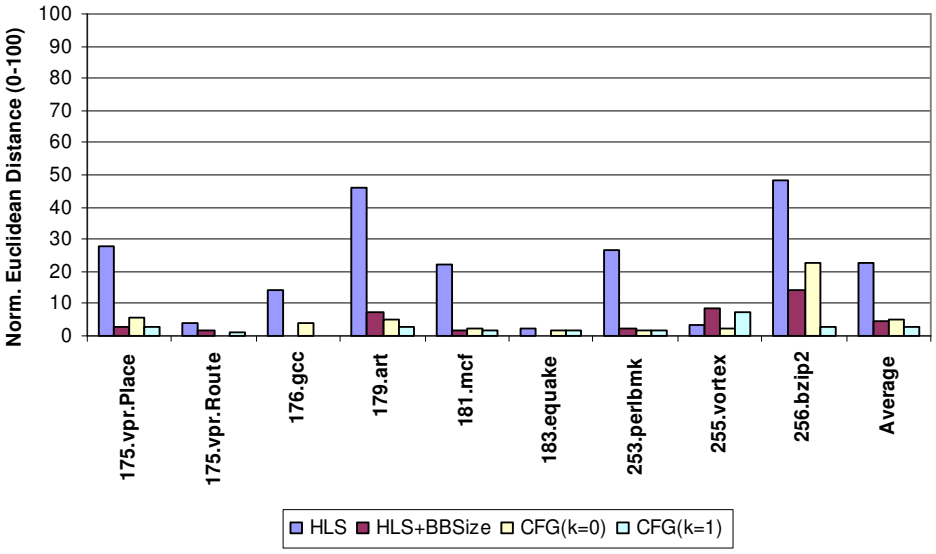


Figure 3.6: Relative accuracy based on the ability to rank 43 configurations in order of their speedup

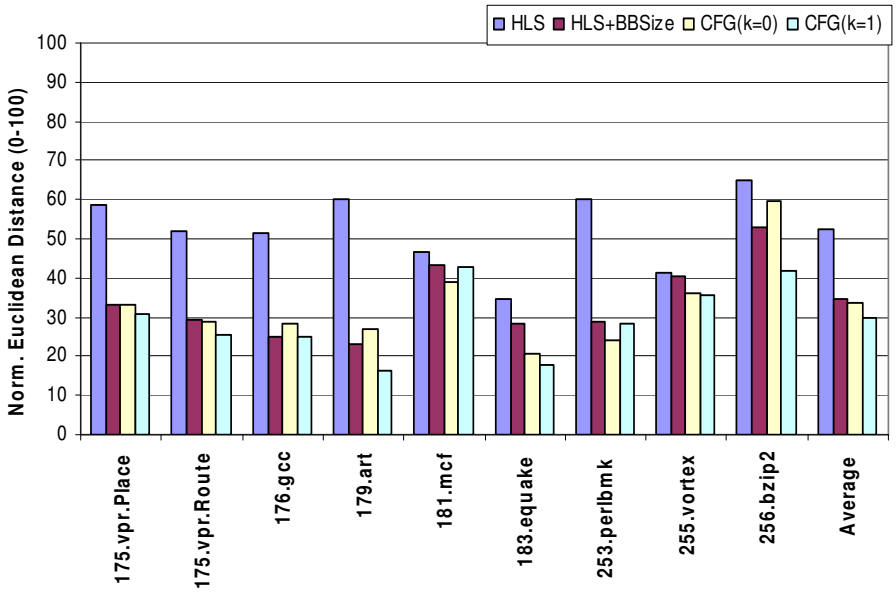
Figure 3.6 shows the relative accuracy of the 4 simulation models based on the ability of statistical simulation to rank 43 diverse processor configurations in order of their speedups (R_S). The figure shows that although there is a large improvement in relative accuracy between the *HLS* and *HLS+BBSize*, additional modeling yields only slight improvements in the relative accuracy.

Figure 3.7 shows the results of processor bottleneck characterization for the four statistical simulation models. The accuracy of the *HLS* model is good enough to identify only top 3 performance bottlenecks for all programs except `181.art` and `256.bzip2`. By increasing the complexity of the *HLS+BBSize* model allows statistical simulation to correctly identify the order of the Top 3, 10, 20, and all 43 bottlenecks. The two statistical simulation models, *Zeroth Order Control Flow Graph (CFG, k=0)* and *First*

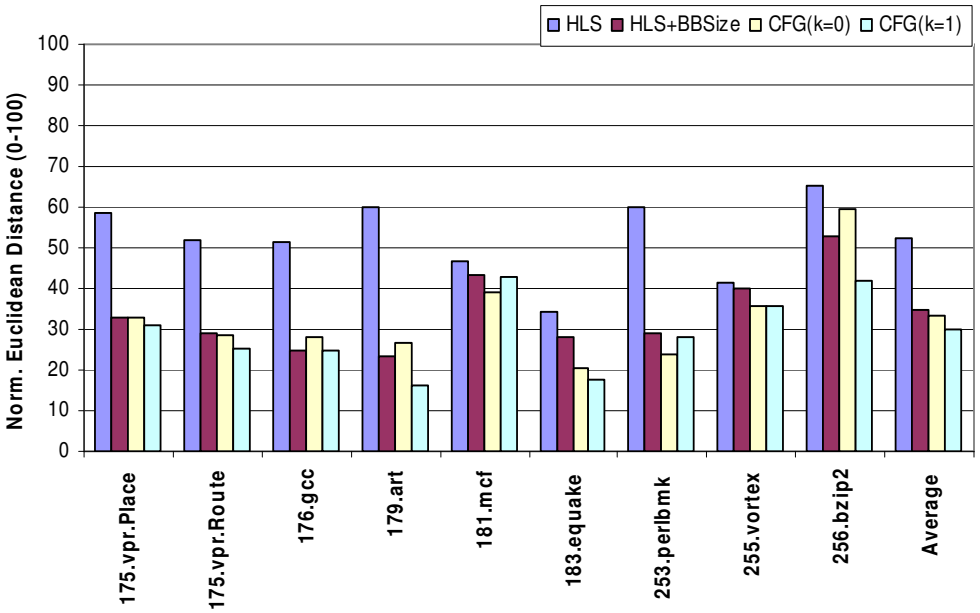
Order Control Flow Graph (CFG, k=1), only marginally improves the accuracy to of statistical simulation to identify the performance bottlenecks.



(a) Top 5 bottlenecks



(b) Top 20 bottlenecks



(c) Top 43 bottlenecks

Figure 3.7: Bottleneck characterization for 4 statistical simulation models.

In summary, from these results, we conclude that if absolute accuracy is the primary goal, then the computer architect should use the most detailed state-of-the art statistical simulation model, *Control Flow Graph (k=1)*. However, we observe that an increase in the absolute accuracy of statistical simulation does not result in a commensurate increase in its relative accuracy. Interestingly, a simple statistical model such as *HLS+BBSize* has the ability to yield very good relative accuracy, although the absolute accuracy is lower. Therefore, one key result from this chapter is that simple statistical simulations models have a good relative accuracy, which makes them an effective tool to make design decisions early in the design cycle when the time and resources for simulator development are very limited.

3.5 SUMMARY

Since detailed cycle-accurate simulation models require long simulation times, computer architects have proposed statistical simulation as a time-efficient alternative for performing early design space exploration studies. But the concern for many architects is that statistical simulation may not perform well for processor configurations that are drastically different than the ones that have been used in previous evaluations, *i.e.*, it is suited only for evaluating incremental changes in processor architectures. The objective of this chapter was to evaluate the efficacy of statistical simulation as a design space exploration tool, in wake of these issues and concerns to using statistical simulation.

In this chapter, we use the Plackett & Burman (P&B) design to measure the representativeness of the synthetic trace. The configurations used in P&B design provide a systematic way to evaluate the accuracy of statistical simulation by exposing various performance bottlenecks.

The key conclusions from this chapter are:

1) At the very least, synthetic traces stress the same 10 most significant processor performance bottlenecks as the original workload. Since the primary goal of early design space studies is to identify the most significant performance bottlenecks, we conclude that statistical simulation is indeed a very useful tool.
2) Statistical simulation has good relative accuracy and can effectively track design changes to identify feasible design points in a large design space of aggressive microarchitectures.
3) Our evaluation of four statistical simulation models shows that although a very detailed model is needed to achieve a good absolute accuracy in performance estimation, a simple model is sufficient to achieve good relative accuracy. This is

very attractive early in the design cycle when time and resources for developing the simulation infrastructure are limited.

From these results, we conclude that statistical simulation, with its ability to identify key performance bottlenecks and accurately track performance trends using a simple statistical workload model, is a valuable tool for making early microprocessor design decisions.

Chapter 4: Microarchitecture-Independent Workload Modeling

4.1 WORKLOAD CHARACTERIZATION

The metrics that are typically used to characterize an application can be broadly classified as microarchitecture-dependent and microarchitecture-independent. Programs can be characterized using microarchitecture-dependent characteristics such as cycles per instruction (CPI), cache miss-rate, and branch prediction accuracy, or microarchitecture-independent characteristics such as temporal or spatial data locality and instruction level parallelism. Microarchitecture-Independent characteristics are abstract metrics that measure program characteristics independent of the underlying hardware *i.e.* they only depend on the compiler and instruction-set-architecture (ISA).

Prior work in statistical simulation and workload synthesis has used a combination of microarchitecture-independent and microarchitecture-dependent characteristics. Typically, these techniques model synthetic memory access patterns and control-flow behavior to match a target metric e.g. cache miss-rate, branch prediction rate etc., and hence they also reflect machine properties rather than pure program characteristics. A major pitfall with using microarchitecture-dependent metrics to characterize an application is that it may hide the underlying program behavior *i.e.* although a microarchitecture-dependent characteristic may be the same for two programs the inherent behavior of the programs may be very different.

Consequently, the synthetic workloads generated from these models may yield large errors when cache and branch microarchitecture configurations are changed from the targeted configuration [Bell and John, 2005-1] [Bell and John, 2005-2]. Therefore, in order to enable the portability of the generated synthetic workload across a wide range of

microarchitectures, it is important to capture inherent program characteristics into the synthetic workload rather than generate a synthetic workload to match a target metric.

The importance of purely using microarchitecture-independent workload metrics to characterize a workload has also been emphasized by recent research whose objective was to measure similarity between programs [Joshi *et al.,* 2006-3] [Hoste and Eeckhout, 2006-2]. They show that conclusions based on performance characteristics such as execution time and cache miss-rate could categorize a program with unique characteristics as insignificant, only because it shows similar trends on the microarchitecture configurations used in the study. This indicates that an analysis based on microarchitecture-dependent characteristics could undermine the importance of a program that is really unique.

An objective of this dissertation is to develop a workload synthesis framework that can automatically construct synthetic workloads that are representative across a wide range of microarchitectures and platforms. In this chapter we characterize a set of embedded, general-purpose, and scientific programs and propose a set of microarchitecture-independent workload characteristics. We propose modeling approaches for incorporating synthetic instruction locality, data locality, and control flow predictability into synthetic workloads. The approach used in these models is to use an attribute to quantify and abstract code properties related to spatial locality, temporal locality, and branch predictability. These attributes are then used to generate a trace or a benchmark with similar properties. If the feature captured the program property very well the resulting performance metrics *e.g.* cache miss-rate and branch prediction rate will be similar to that of the original application program. We show in our evaluation, in Chapter 5, that modeling the microarchitecture-independent workload characteristics in a synthetic benchmark improves its representativeness across a wide range of cache, branch

predictor, and other microarchitecture configurations. This increases the usefulness of the synthetic benchmark during computer architecture research and development as it serves as a useful indicator of performance even when the underlying microarchitecture is altered.

4.2 BENCHMARKS

We characterize a set of embedded, general-purpose, and scientific benchmarks. We use the SPEC CPU 2000 Integer benchmarks as representative of general-purpose programs and SPEC CPU 2000 Floating-Point benchmarks as a representative of scientific applications. As a representative of the embedded application domain we use benchmarks from the MiBench and MediaBench suite. All benchmark programs were compiled on an `Alpha` machine using the native `Compaq cc v6.3-025` compiler with the `-O3` optimization setting and were run to completion.

4.3 APPLICATION BEHAVIOR SIGNATURE

The abstract microarchitecture-independent workload modeling approach is based on the premise that the behavior of programs can be characterized using a set of inherent workload characteristics. This requires that we develop a characterization of the behavior of programs to understand the set of inherent characteristics that correlate well with the performance exhibited by the program. The set of workload characteristics can be thought of as a signature that uniquely describes the workload's inherent behavior, independent of the microarchitecture.

We characterize the proprietary application by measuring its inherent, or microarchitecture-independent, workload characteristics that together can be considered as the application's behavior signature. The characteristics that we model in this study are a subset of all the microarchitecture-independent characteristics that can be

potentially modeled, but we believe that we model (most of) the important program characteristics that impact a program's performance; the evaluation in Chapter 5 in fact shows that this is the case for the general-purpose, scientific, and embedded applications that we target. The microarchitecture-independent characteristics that we measure cover a wide range of program characteristics, namely:

- Instruction Mix
- Control Flow Behavior
- Instruction Stream Locality
- Data Stream Locality
- Instruction-Level-Parallelism
- Control Flow Predictability

We discuss these characteristics in detail now.

4.3.1 Control Flow Behavior and Instruction Stream Locality

It has been well observed that the instructions in a program exhibit the *locality of reference* property. The locality of reference is often stated in the rule of thumb called the 90/10 rule, which states that a program typically spends 90% of its execution time in only 10% of its static code. In order to model this program property in a synthetic benchmark clone, it is essential to capture the control flow structure of the program, *i.e.*, we need to model how basic blocks are traversed in the program and how branch instructions alter the direction of control flow in the instruction stream. During the application profiling phase we capture the control flow information using the statistical flow graph (SFG) described in [Eeckhout *et al.,* 2004-2]. Each node in the SFG represents a unique basic block and the edges represent the transition probabilities between basic blocks. Figure 4.1 shows an example SFG that is generated by profiling the execution of a program. The probabilities marked on the edges of each basic block

indicate the transition probabilities, *e.g.*, if Basic Block 1 was executed, the probability that Basic Block 2 will be executed next is 70%. Thus, the SFG is a representation of the control flow graph of the program with the edges annotated with dynamic transition probabilities.

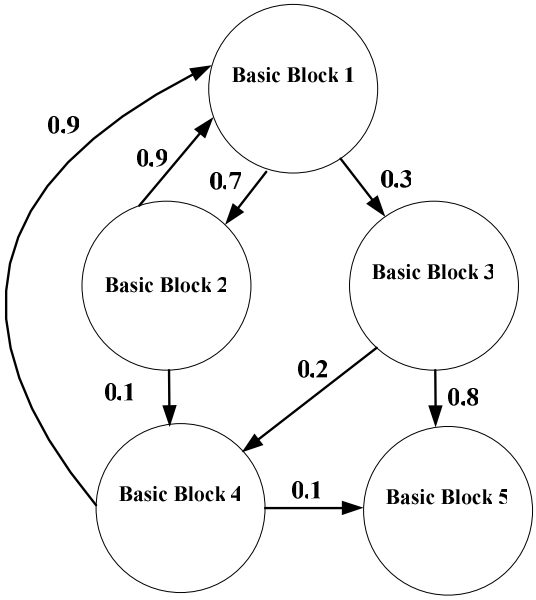


Figure 4.1: An example SFG used to capture the control flow behavior and instruction stream locality of a program.

We measure the workload characteristics described below per unique pair of predecessor and successor basic blocks in the control flow graph. For example, instead of measuring a single workload profile for Basic Block 4, we maintain separate workload characteristics profiles for Basic Block 4 dependent on its predecessor, Basic Block 2 or Basic Block 3. Gathering the workload characteristics at this granularity improves the modeling accuracy because the performance of a basic block is determined by the context in which it is executed [Eeckhout *et al.*, 2004].

Table 4.1 provides a summary of the information that can be gleaned from the SFGs for the complete executions of the SPEC CPU2000 integer, and floating-point programs, as well as a set of MiBench and MediaBench embedded programs. The table shows the number of unique basic blocks in each program, the number of basic blocks that account for 90% of the dynamic program execution, the average basic block size, and the number of successor basic blocks. Among all the SPEC CPU2000 programs, `gcc` has the largest footprint but still exhibits a very high instruction locality (only 9% of the basic blocks are required to account for 90% of the program execution). All the other benchmark programs have fairly small footprints, suggesting that they do not stress the instruction cache. The embedded benchmarks also have very modest footprints, with `ghostscript` having the largest footprint. Compared to the SPEC CPU benchmarks, the embedded benchmarks exhibit higher instruction locality; on average 90% of the time is spent in 13% of the basic blocks, compared to SPEC CPU benchmarks where 90% of the time is spent in 27% of the basic blocks. The average basic block size for the floating-point programs is 58.2, with `applu` (111.7), and `mgrid` (109.2) having very large basic blocks. On the other hand, the average basic block size for the integer programs is 7.9. The average basic block size for the embedded benchmarks, 10 instructions, is slightly larger than for the SPEC CPU integer programs – with `djpeg` (23.5) having a fairly large average basic block size.

The average number of successors for each basic block is a measure for the control flow complexity in the program – the higher the number of successor basic blocks, the more complex the control flow behavior. `crafty` and `gcc` are the two benchmarks in which basic blocks have a large number of successors, 7.4 and 10.5 basic blocks, respectively. This high number of basic blocks is due to returns from functions that are called from multiple locations in the code and multimodal switch statements. At

the other end of the spectrum, basic blocks in programs such as `applu`, `equake`, `gzip`, `mgrid`, `swim`, `perlbmk`, and `vpr` only have one or two successor basic blocks, suggesting that they execute in tight loops most of the time. Interestingly, some of the embedded benchmarks, `crc32`, `gsm`, `patricia`, `qsort`, and `rasta` have a large number of successor basic blocks, suggesting complex control flow. For the other embedded benchmarks the average number of successor basic blocks is 4, which is approximately the same as for the SPEC CPU benchmark programs.

The SFG thus captures a picture of the program structure and its control flow behavior. We will use the SFG to recreate this control flow structure of an application in its synthetic benchmark clone.

Table 4.1: Summary of information captured by the SFG.

(a) SPEC CPU2000 Integer and Floating Point Benchmarks.

Benchmark	**Number of Basic Blocks**	**Number of Basic Blocks that Account for 90% of Program Execution**	**Average Basic Block Size**	**Average Number of Successor Basic Blocks**
`applu`	348	104	112.3	1.8
`apsi`	262	127	28.6	3.3
`art`	69	6	7.7	2.7
`bzip2`	139	40	7.1	3.2
`crafty`	514	151	10.5	7.4
`eon`	303	119	9.3	4.7
`equake`	47	10	39.6	1.6
`gcc`	1088	98	6.9	10.5
`gzip`	163	30	8.7	2.3
`mcf`	178	23	4.2	4.4

mesa	211	74	16.4	3.2
mgrid	210	11	109.2	1.9
perlbmk	54	34	5.2	2.0
swim	63	17	41.5	1.1
twolf	155	55	8.2	5.1
vortex	626	44	4.9	6.6
vpr	84	21	7.2	2.2
wupwise	130	47	10.3	4.3

(b) Embedded benchmarks from MiBench and MediaBench benchmark suites.

Benchmark	**Number of Basic Blocks**	**Number of Basic Blocks that Account for 90% of Program Execution**	**Average Basic Block Size**	**Average Number of Successors**
basicmath	371	98	6.8	7.6
bitcount	240	8	8.3	3.0
crc32	311	146	6.2	11.2
dijkstra	366	15	6.9	2.3
fft	366	101	8.9	6.8
ghostscript	2549	62	6.7	4.9
gsm	312	142	6.2	10.3
jpeg	695	30	11.3	4.5
patricia	419	136	6.0	11.6
qsort	319	58	5.2	8
rsynth	536	22	10.2	2.9
stringsearch	160	46	6.7	5.1
susan	364	6	16.5	2.4
typeset	875	86	7.6	6.7
cjpeg	711	31	10.1	5.0
djpeg	702	45	23.5	5.1

epic	650	21	6.0	3.8
g721-decode	299	40	8.1	5.3
mpeg-decode	514	27	16.9	3.2
rasta	1089	215	10.6	9.7
rawaudio	119	3	19.1	2.2
texgen	886	71	10.7	3.9

4.3.2 Instruction Mix

The instruction mix of a program measures the relative frequency of operation types appearing in the dynamic instruction stream. We measure the percentage integer arithmetic, integer multiply, integer division, floating-point arithmetic, floating-point multiply, floating-point division operations, load, store, and branch instructions. The instruction mix is measured separately for every basic block.

4.3.4 Instruction-Level Parallelism

The dependency distance is defined as the number of instructions in the dynamic instruction stream between the production (write) and consumption (read) of a register and/or memory location. The goal of characterizing data dependency distances is to capture a program's inherent ILP. Figure 4.2 shows an illustration of how the dependency distance is measured. In Figure 4.2, there is a Read-After-Write (RAW) dependence on register R1 and the distance is equal to 4. While techniques such as value prediction reduce the impact of these dependencies on ILP, information on the dependency distance is very useful in understanding ILP inherent to a program. For every instruction, we measure the data dependency distance information on a per-operand basis as a distribution organized in eight buckets: percentage of dependencies that have a dependency distance of 1 instruction, and the percentage of dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions. This dependency

distance distribution is maintained separately for each unique pair of predecessor and successor basic blocks in the program.

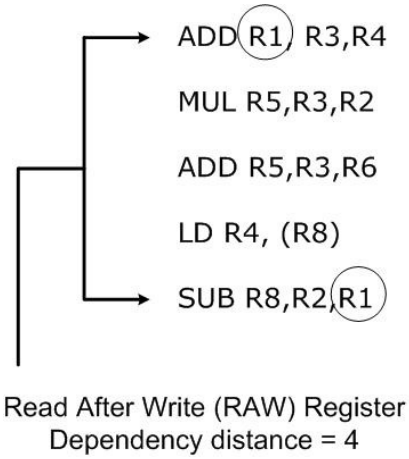


Figure 4.2: Illustration of measuring RAW dependency distance.

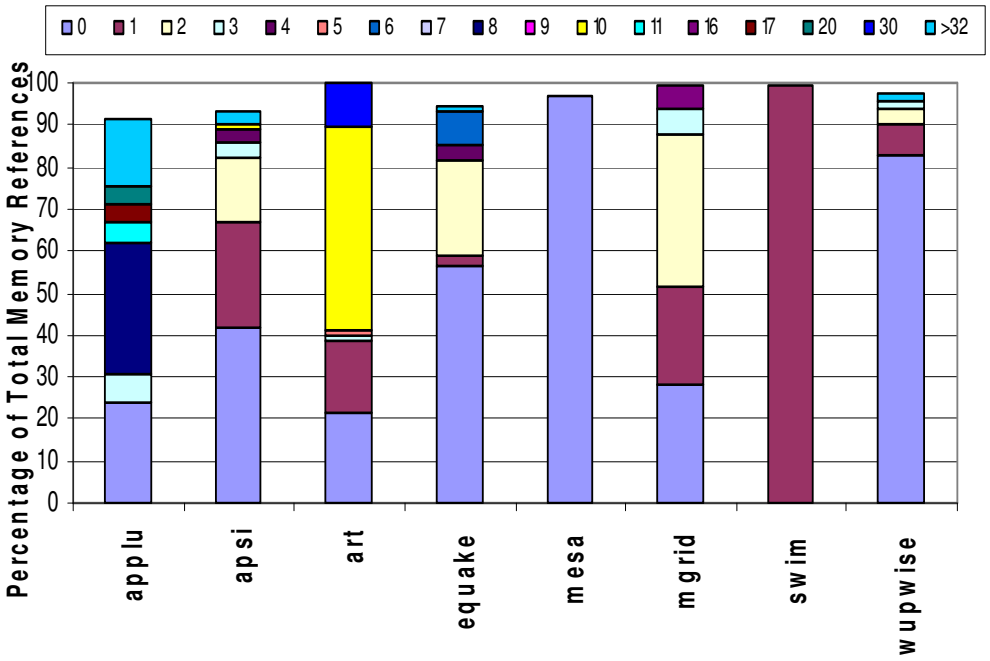
4.3.5 Data Stream Locality

The principle of data reference locality is well known and recognized for its importance in affecting an application's performance. Traditionally, data locality is considered to have two important components, temporal locality and spatial locality. Temporal locality refers to locality in time and is due to the fact that data items that are referenced now will tend to be referenced soon in the near future. Spatial locality refers to locality in space and is due to the program property that when a data item is referenced, nearby items will tend to be referenced soon. Previous work [Sorenson and Flanagan, 2002] shows that these abstractions of data locality and their measures are insufficient to replicate the memory access pattern of a program. Therefore, instead of quantifying temporal and spatial locality by a single number or a simple distribution, our approach for mimicking the temporal and spatial data locality of a program is to characterize the memory accesses of a program to identify the patterns with which a

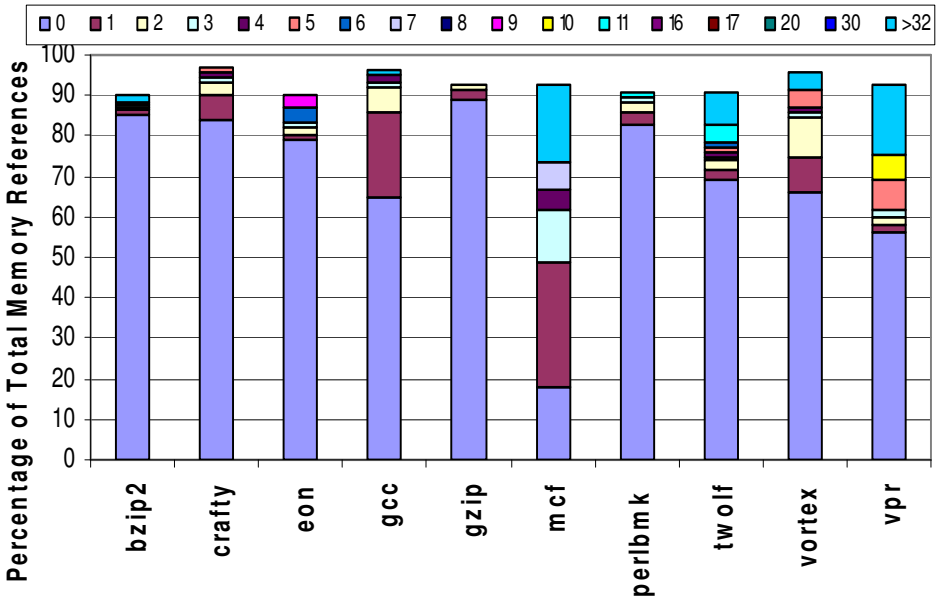
program accesses memory on a per-instruction basis and then replicate that in the synthetic benchmark.

The order in which a program accesses memory locations is a function of its dynamic traversal of the control flow graph and the memory access patterns of its individual load and store instructions. One may not be able to easily identify patterns or sequences when observing the global data access stream of the program though. This is because several memory access patterns co-exist in the program and are generally interleaved with each other. So, the problem that we are trying to address is how to efficiently extract patterns from the global sequence of memory addresses issued by the program. When a compiler generates a memory access instruction, load or store, it has a particular functionality – it accesses a global variable, stack, array, or a data structure. This functionality of the memory access instruction is consistent and stable, and determines how the instruction generates effective addresses. This suggests that rather than studying the global memory access stream of a program, it may be better to view the data access patterns at a finer granularity of individual memory access instructions.

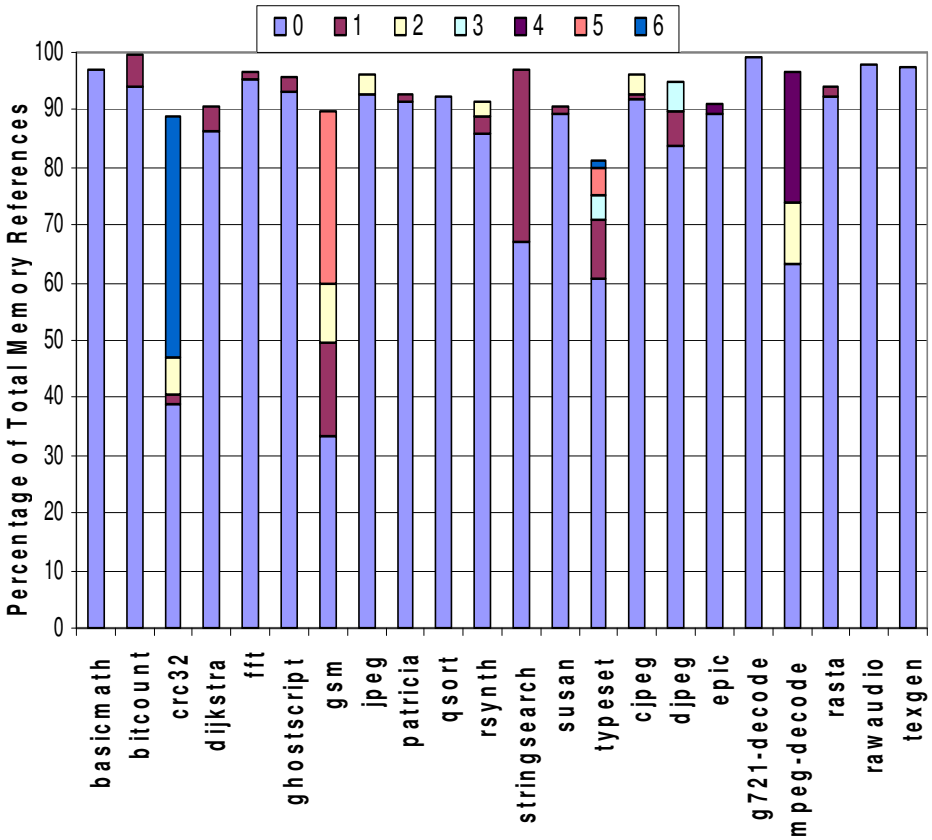
We profile general-purpose, scientific, and embedded benchmark programs (described later), and measure the stride values (differences between two consecutive effective addresses) per static load and store instruction in the program. We use this information to calculate the most frequently used stride value for each static load and store instruction, and the percentage of the memory references for which it was used. If a static memory access instruction uses the same stride more than 80% of the time, we classify the instruction as a *strongly strided instruction*. Then, we plot a cumulative distribution of the stride patterns for the most frequently used stride values for all of the strongly strided memory access instructions and the percentage of the dynamic memory access instructions that they represent.



(a) SPEC CPU 2000 Floating-Point Programs.



(b) SPEC CPU 2000 Integer Programs.



(c) Embedded Programs from MediaBench & MiBench Suites

Figure 4.3: Percentage breakdown of stride values per static memory access.

Figures 4.3 (a), (b), and (c) show this cumulative distribution for the SPEC CPU2000 floating-point, integer, and embedded benchmarks, respectively. The stride values are shown at the granularity of a 64 byte block (analogous to a cache line), *i.e.*, if the most frequently used stride value for an instruction is between 0 and 63 bytes, it is classified as a zero stride(consecutive addresses are within a single cache line distance), between 64 and 127 bytes as stride 1, etc.

The results from Figure 4.3 (a) show that for floating-point benchmarks almost all the references issued by a static load or store instruction can be classified as strides. This

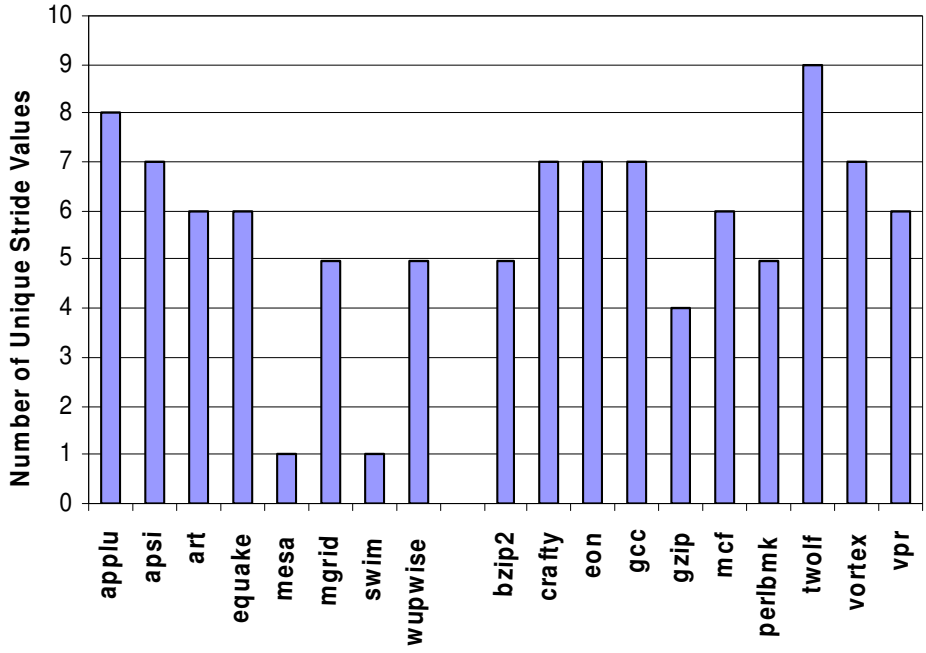
observation is consistent with the common understanding of floating-point program behavior – the data structures in these programs are primarily arrays and hence the memory instructions are expected to generate very regular access patterns. The access patterns of `swim` and `mesa` are very unique – all the static memory access instructions in `swim` walk through memory with a stride of one cache line size, and those in `mesa` with a zero stride. Also, `wupwise` has the same cache line access as its dominant stride. The number of prominent unique stride values in other floating-point program varies between 5 (`mgrid`) and 8 (`applu`). Benchmark `art`, a program that is known to put a lot of pressure on the data cache, also has a very regular access pattern with almost all the load and store instructions accessing memory with stride values of 0, 1, 10, or 30.

The static load and store memory access characterization behavior of the SPEC CPU2000 integer benchmarks, shown in Figure 4.3 (b), appear to be rather surprising at first sight. The results show that a large number of programs have just one single dominant access stride pattern – more than 80% of the accesses from static loads and stores for `bzip2`, `crafty`, `eon`, `gzip` and `perlbmk` have the same cache line access as their dominant stride. Also, a large percentage (more than 90%) of all memory references in programs such as `mcf`, `twolf`, `vpr` and `vortex` appear to exhibit regular stride behavior. These programs are known to have a lot of pointer-chasing code and it is expected that this results in irregular data access patterns. The observation made from Figure 3(b) suggests that in these pointer-intensive programs, linked list and tree structure elements are frequently allocated at a constant distance from each other in the heap. As a result, when linked lists are traversed, a regular pattern of memory accesses with constant stride emerges, and they manifest as stride patterns. Recent studies aimed at developing prefetching schemes have made similar observations, see for example [Collins *et al.,* 2001] [Stoutchinin *et al.*, 2001] [Wu *et al.*, 2002]; they developed stride-

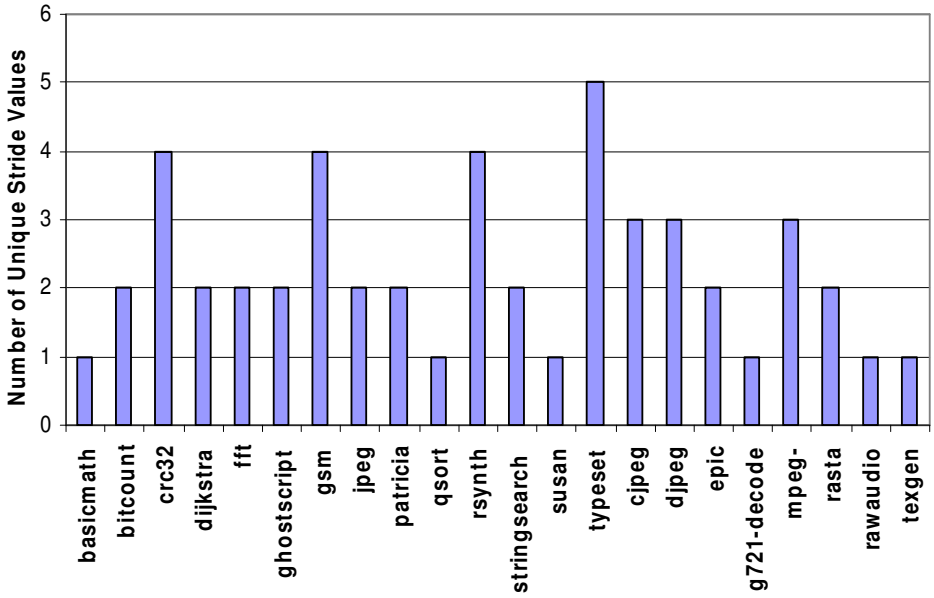
based prefetching schemes that improve the performance of pointer-intensive programs such as `mcf` by as much as 60%.

We observe in Figure 4.3(c) that for embedded benchmarks too more than 90% of the dynamic memory accesses originate from strongly strided static memory access instructions. Most of the embedded benchmarks have the same cache line access as their dominant stride. The exceptions are `crc32`, `dijkstra`, `rsynth` and `typeset` for which there are 4 to 5 different unique stride values.

From this characterization study of the memory access patterns we can infer that the memory access patterns of programs (both general-purpose integer, embedded and floating-point programs) are amenable to be modeled using a stride-based model on a per-instruction basis. We record the most frequently used stride value for every static memory access instruction in every node in the statistical flow graph. Also, during the profiling phase, the average length of the stride stream with which each static load or store instruction accesses data is recorded – this is measured by calculating the number of consecutive positive stride references issued by a static memory instruction before seeing a negative stride, or if there is a break in the stride pattern. Note, that the dominant stride value and the average stream length can be different for different static memory access instructions.



(a) SPEC CPU Floating-Point and Integer Benchmarks.



(b) Embedded benchmarks from MiBench & MediaBench suites.

Figure 4.4: Number of different dominant memory access stride values per program.

Figure 4.4 summarizes the number of different dominant strides with which static load and store instructions in integer, floating-point, and embedded programs access data. The vertical axis shows the number of different strongly strided values with which memory access instructions access data. We observe that a maximum of 9 stride values (twolf) are seen across the strongly strided memory access instructions. When constructing a synthetic benchmark clone, we model each static load/store instruction as accessing a bounded memory with its most frequent stride value and its average stream length obtained from this workload characterization. It will be clear when we describe the synthesis algorithm to incorporate these characteristics into synthetic benchmarks (Chapter 5) that having a small number of dominant stride values makes it feasible to incorporate the memory access model in the synthetic benchmark clone – only 9 registers are required to store the stride values.

4.3.6 Control Flow Predictability

In order to incorporate inherent branch predictability in the synthetic benchmark clone it is essential to understand the property of branches that makes them predictable. The predictability of branches stems from two sources: (i) most branches are highly biased towards one direction, *i.e.,* the branch is taken or not-taken for 80-90% of the time, and (ii) the outcome of branches may be correlated.

In order to capture the inherent branch behavior in a program, the most popular microarchitecture-independent metric is to measure the taken rate per static branch, *i.e.,* fraction of the times that a static branch was taken during the complete run of the program. Branches that have a very high or very low taken rate are biased towards one direction and are considered as highly predictable. However, merely using the taken rate of branches is insufficient to actually capture the inherent branch behavior. Consider two sequences of branches, one has a large number of taken branches followed by an equally

long number of not-taken branches, whereas the other sequence does not have such a regular pattern and switches randomly between taken and not-taken directions. Both sequences have the same taken rate (50%) but still have different branch misprediction rates. It is clear that the former sequence is better predictable than the latter. This suggests that branch predictability depends more on the sequence of taken and not-taken directions, and not just on the taken rate.

Therefore, in our control flow predictability model we also measure an attribute called *transition rate* [Haungs *et al.*, 2000] for capturing the branch behavior in programs. Transition rate of a static branch is defined as the number of times it switches between taken and not-taken directions as it is executed, divided by the total number of times that it is executed. By definition, branches with low transition rates are always biased towards either taken or not-taken. It has been well observed that such branches are easy to predict. Also, the branches with a very high transition rate always toggle between taken and not-taken directions and are also highly predictable. However, branches that transition between taken and not-taken sequences at a moderate rate are relatively more difficult to predict.

In order to incorporate synthetic branch predictability we annotate every node in the statistical flow graph with its transition rate. When generating the synthetic benchmark clone we ensure that the distribution of the transition rates for static branches in the synthetic clone is similar to that of the original program. We achieve this by configuring each basic block in the synthetic stream of instructions to alternate between taken and not-taken directions, such that the branch exhibits the desired transition rate. The algorithm for generating the synthetic benchmark program in the next section describes the details of this mechanism.

4.4 MODELING MICROARCHITECTURE-INDEPENDENT CHARACTERISTICS INTO SYNTHETIC WORKLOADS

In this section we propose basic algorithms for modeling the microarchitecture-independent characteristics, described in section 4.3, into synthetic workloads. In the next chapter we describe how these algorithms can be used to model characteristics into synthetic benchmark clones.

4.4.1 Data Locality

Based on the results of these characterization experiments we propose a first-order model to generate a synthetic trace of data address references in the program. In order to correctly model the locality of the data address stream it is important to ensure that the reference streams are correctly interleaved with each other. In order to capture this behavior we build a statistical flow graph of the program as described in section 4.3.1 and, for each static memory access instruction in every node record the following information during the statistical profiling phase: `<Access type (load/store), Most frequently used stride value, Stride length, Seed address (first data address issued by load/store instruction>`. The algorithm used to generate a stream of synthetic data address traces is outlined below.

---

(1) Generate a random number in the interval [0, 1] and use this value to select a node in the statistical flow graph based on the cumulative distribution function based on the occurrence frequency of each node.

(2) For each load/store instruction in the basic block represented by the selected node:

   (a) Generate a new reference address (last generated address by this instruction + stride) and output a synthetic entry in the form <Load/Store, New Address>. For the first reference, use the seed address as the last address generated for this instruction.

   (b) Increment counter for the current stream length.

   (c) Update last generated address counter with the new address. If the stream length has reached the maximum stream length, reset the last generated address counter to the seed address.

(3) The occurrence count of that node is then decremented.

(4) Increment count of the total number of data instruction addresses generated.

(5) A cumulative distribution function based on the probabilities of the outgoing edges of the nodes is then used to determine the next node. If the node does not have any outgoing edges, go to step 1.

(6) If the target number of instructions has not been generated, go to step 2. If the target number of addresses has been generated, the algorithm terminates.

The resulting synthetic trace can then be simulated through a trace driven cache simulator to obtain the miss-rates. Also, the generated synthetic instruction address trace can be annotated to the synthetic instruction stream [Eeckhout *et al.,* 2000] [Eeckhout *et al.,* 2004] [Bell *et al.,* 2004] instead of probabilistically modeling instruction cache misses. In summary, we have essentially created a congruence class for each static load or store instruction depending on its most frequently used stride and stride length. Each static load/store instruction will iterate through its congruence class.

4.4.2 Branch Predictability

In order to incorporate synthetic branch predictability we characterize the entire application and build a statistical flow graph structure of the entire program as described in section 4.3.1. In addition, every node in the statistical flow graph of the program is annotated with the following: `<taken rate, transition rate, static branch address, branch instruction opcode, instruction address of first instruction in the basic block>`. Once the statistical profile has been generated, the workload generator uses the following algorithm used to generate a trace that synthetically models branch behavior:

---

(1) Generate a random number in the interval [0, 1] and use this value to select a node in the statistical flow graph based on the cumulative distribution function of the occurrence frequency of each node.

(2) Use the *transition rate* of the branch instruction in the basic block represented by the node to determine whether the branch should be *taken* or *not-taken*. If the branch is determined as *taken*, the next node to be selected is determined using the cumulative

distribution function based on the probabilities of the outgoing edges to the *taken* target nodes. Otherwise, the *not-taken* target node is chosen as the next node.

(3) Using the profile information of the current and next node output a synthetic trace of the format: <PC of first instruction in basic block, PC of static branch instruction, Branch instruction opcode, Target Address of Branch>

(4) The occurrence count of that node is then decremented.

(5) Increment count of the total number of synthetic entries generated.

(6) If the target number of instructions has not been generated, go to step 2. If the target number of addresses has been generated, the algorithm terminates.

In order to perform a detailed trace driven cycle-accurate simulation instead of just the branch simulations, every branch instruction in synthetic trace generated in [Eeckhout *et al.,* 2000] [Eeckhout *et al.,* 2004] [Bell *et al.,* 2004] can be annotated with the branch instruction address, branch type, and the branch target address instead of using the probability that a branch direction will be correctly predicted.

4.5 SUMMARY

In this chapter we motivated the need for developing an abstract microarchitecture-independent workload model to characterize an application. The use of microarchitecture-independent characteristics makes it possible to describe the inherent behavior of a program, agnostic to the underlying architecture. We proposed a set of microarchitecture-independent workload characteristics and show that they can be effectively used to capture the key performance characteristics of general-purpose, scientific, and embedded applications. This chapter then proposes algorithms and describes how to model the key microarchitecture-independent characteristics into a synthetic workload. The abstract microarchitecture-independent characteristics and the algorithms to model them into a synthetic workload form the basis of generating synthetic benchmarks. In the following chapter we describe how the abstract microarchitecture independent model and the proposed algorithm can be used to capture

the essence of longer-running and proprietary applications into miniature synthetic benchmarks.

Chapter 5: Distilling the Essence of Workloads into Miniature Synthetic Benchmarks

In this chapter we propose an approach that automatically distills key inherent microarchitecture-independent workload characteristics, proposed in Chapter 4, into a miniature synthetic benchmark clone. The key advantage of the benchmark clone is that it hides the functional meaning of the code but exhibits similar performance characteristics as the target application. Moreover, the dynamic instruction count of the synthetic benchmark clone is substantially shorter than the proprietary application, greatly reducing overall simulation time – for SPEC CPU, the simulation time reduction is over five orders of magnitude compared to entire benchmark execution. By using a set of benchmarks representative of general-purpose, scientific, and embedded applications, we demonstrate that the power and performance characteristics of the synthetic benchmark clone correlate well with those of the original application across a wide range of microarchitecture configurations.

5.1. Disseminating Proprietary Applications as Benchmarks

Real-world applications tend to be intellectual property and application developers hesitate to share them with third party vendors and researchers. Moreover, enormous time and effort is required for engineering real-world applications into portable benchmarks. This problem is further aggravated by the fact that real-world applications are diverse and evolve at a rapid pace making it necessary to upgrade and maintain special benchmark versions very frequently. Another challenge in engineering benchmarks from real-world applications is to make them simulation friendly – a very large dynamic instruction count results in intractable simulation times even on today's fastest simulators running on today's fastest machines.

In this chapter we explore an automated synthetic benchmark generation as an approach to disseminate real-world applications as miniature benchmarks without compromising on the applications' proprietary nature. Moreover, automated synthetic benchmark generation significantly reduces the effort of developing benchmarks, making it possible to upgrade the benchmarks more often. In order to achieve this we propose a technique, called *benchmark cloning*, which distills key behavioral characteristics of a real-world application and models them into a synthetic benchmark. The advantage of the synthetic benchmark clone is that it provides code abstraction capability, *i.e.,* it hides the functional meaning of the code in the original application but exhibits similar performance characteristics as the real application. Source code abstraction prevents reverse engineering of proprietary code, which enables software developers to share synthetic benchmarks with third parties. Moreover, the dynamic instruction count of the synthetic benchmark clone is much smaller than the original application and significantly reduces the simulation time. These two key features of synthetic benchmark clones, source code abstraction and a small dynamic instruction count, enable the dissemination of a proprietary real-world application as a miniature synthetic benchmark that can be used by architects and designers as a proxy for the original application.

The key novelty in our benchmark cloning approach compared to prior work in synthetic benchmark generation is that we use the abstract microarchitecture-independent workload model to synthesize a benchmark clone. Since the design of the synthetic benchmark is guided only by the application characteristics and is independent of any hardware specific features, the benchmark can be used across a wide range of microarchitectures. We show in our evaluation that the synthetic benchmark clone shows good correlation with the original application across a wide range of cache, branch predictor, and other microarchitecture configurations.

The remainder of this chapter is organized as follows. In Section 5.2 we provide a high-level overview of the benchmark cloning approach. Section 5.3 then provides a detailed description of the algorithm used to generate the synthetic benchmark clone. In Section 5.4 we describe our simulation environment, machine configuration, and the benchmarks that we use to evaluate the efficacy of the synthetic benchmark clone for performance and power estimation. Section 5.5 presents a detailed evaluation of the synthetic benchmark cloning technique using various criteria. In Section 5.6 we provide a discussion on the strengths and limitations of the performance cloning technique. Finally, in Section 5.7 we summarize the key results from this chapter.

5.2. BENCHMARK CLONING APPROACH

Figure 5.1 illustrates the benchmark cloning approach that we propose in for generating synthetic benchmarks that abstract the functional meaning of a real-world application while mimicking its behavioral and performance characteristics. Benchmark cloning comprises of two steps: (1) Profiling the real-world proprietary workload to measure its inherent behavior characteristics, and (2) Modeling the measured workload attributes into a synthetic benchmark program.

In the first step we profile a set of workload characteristics described in Chapter 5. These set of workload characteristics can be thought of as a signature that uniquely describes the workload's inherent behavior, independent of the microarchitecture. This increases the usefulness of the synthetic benchmark clone during computer architecture research and development as it serves as a useful indicator of performance even when the underlying microarchitecture is altered. We measured these characteristics using a functional simulator. However, an alternative to simulation is to measure these characteristics using a binary instrumentation tool such as ATOM [Srivastava and Eustace, 1994] or PIN [Luk *et al.*, 2005].

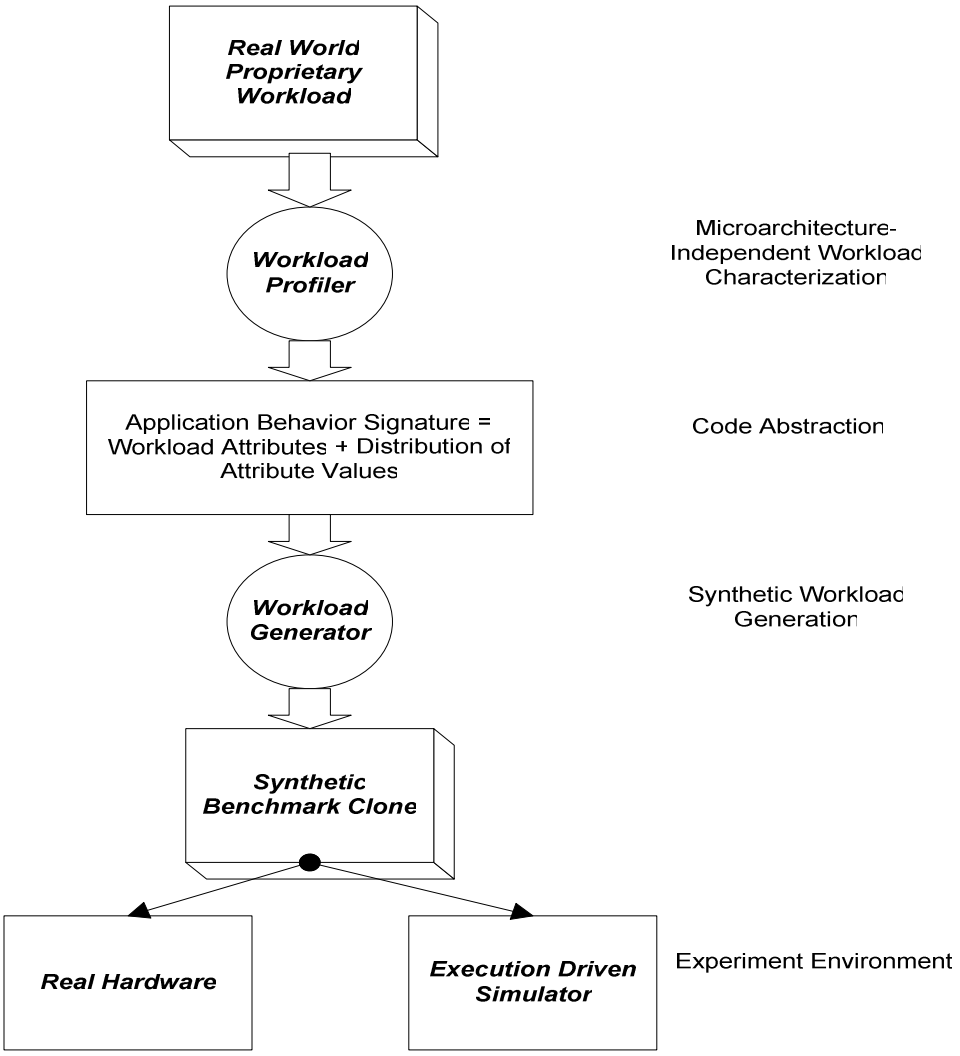


Figure 5.1: Framework for constructing synthetic benchmark clones from a real-world application.

The next step in the benchmark cloning technique is to generate a synthetic benchmark that embodies the application behavior signature of the proprietary workload. Ideally, if all the key workload attributes of the real application are successfully replicated into the performance clone, the synthetic benchmark should exhibit similar performance characteristics. However, the key challenge in developing an

algorithm for generating the synthetic benchmark is to ensure that all the characteristics from the application signature are retained when the benchmark clone is run on an execution-driven performance model or real hardware. This is achieved by generating the benchmark clone as a program in C-code with a sequence of *asm* calls that reproduce the behavior of native instructions. The use of the *volatile* construct ensures that the instruction sequences and their dependencies are not optimized by the compiler.

5.3. BENCHMARK CLONE SYNTHESIS

We now provide details on the algorithm that models these characteristics into a synthetic benchmark clone. The second step in our cloning methodology is to generate a synthetic benchmark by modeling all the microarchitecture-independent workload characteristics from the previous section into a synthetic clone. The basic structure of the algorithm used to generate the synthetic benchmark program is similar to the one proposed by [Bell *et al.*, 2005]. However, the memory and branching model is replaced with the microarchitecture-independent models described in the previous section.

The clone generation process comprises of five sub steps – SFG analysis, memory accessing pattern modeling, branch predictability modeling, register assignment, and code generation. Figure 5.1 illustrates each of these steps.

5.3.1 Statistical Flow Graph Analysis

In this step, the SFG profile obtained from characterizing the application is used for generating the basic template for the benchmark. The SFG is traversed using the branching probabilities for each basic block, and a linear chain of basic blocks is generated. This linear chain of basic blocks forms the spine of the synthetic benchmark program (refer to step (a) in Figure 5.1). The spine is the main loop in the synthetic clone that is repeatedly iterated during the program executing. The length of the spine should

be long enough to reflect the average basic block size and the representation of the most frequently traversed basic blocks in the program. The average basic block size and the number of basic blocks in the program, shown in Table 1, is used as a starting point to decide the number of basic blocks in spine for each program. We then tune the number of basic blocks to match the overall instruction mix characteristics by iterating through the synthesis a small number of times. The number of iterations over which the main loop is executed is set such that performance characteristics of the synthetic clone converge to a stable value. Our experiments, discussed in Section 5.4, show that a total of approximately 10 million dynamic instructions are required for convergence of the synthetic clone. This dynamic instruction count is used to determine the number of times the main loop of the synthetic clone is executed.

The algorithm used for instantiating the basic blocks in the synthetic clone is as follows:

---

(1) Generate a random number in the interval [0,1] and use this value to select a basic block in the statistical flow graph (SFG) based on the cumulative distribution function that is built up using the occurrence frequencies of each basic block.

(2) Output a sequence of instructions per basic block. Assign instruction types to the instructions using the instruction mix distribution for that basic block. Depending on the instruction type, assign the number of source operands for each instruction.

(3) For each source operand, assign a dependency distance using the cumulative dependency distance distribution. This step is repeated until a real dependency is found that ensures syntactical correctness, *i.e.*, the source operand of an instruction cannot be dependent on a store or branch instruction. If this dependency cannot be satisfied after 100 attempts, the dependency is simply squashed.

(4) A cumulative distribution function based on the probabilities of the outgoing edges of the nodes in the SFG is then used to determine the next node. If the node does not have any outgoing edges, go to step 1.

(5) If the target number of basic blocks (equal to the total number of basic blocks in the original program) has not been generated, go to step 2. If the target number of basic blocks has been generated, the algorithm terminates.

5.3.2 Modeling Memory Access Pattern

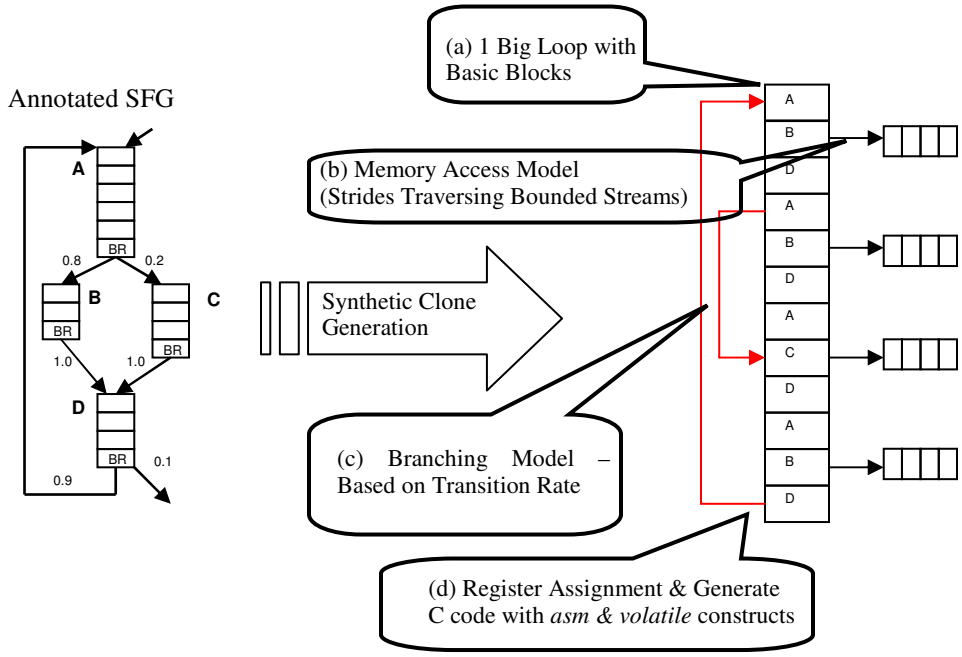
For each memory access instruction in the synthetic clone we assign its most frequently used stride along with its stream length. A load or store instruction is modeled as a memory operation that accesses a circular and bounded stream of references, *i.e.*, each memory access walks through an entire array using its dominant stride value and then restarts from the first element of the array (step (b) in Figure 5.2). An arithmetic instruction in each basic block is assigned to increment the stride value for the memory walk. The stride value itself is stored in a register. Since the maximum number of stride values in the program is 9, we do not need a large number of registers to store the various stride values in the synthetic benchmark.

5.3.4 Modeling Branch Predictability

For each static branch in the spine of the program we identify branches with very high or very low transition rates, and model them as always taken or not-taken (branches with transition rate of less than 10% or greater than 90%). Branches with moderate transition rates are configured to match the transition rate of the corresponding static branch. A register variable is assigned for each transition rate category (a maximum of 8 categories). For every iteration of the master loop, a modulo operation is used to determine whether the static branches belonging to a particular transition rate category will be taken or not-taken (step (c) in Figure 5.2). The static branches in the program use these register variables as a condition to determine the branch direction.

5.3.5 Register Assignment

In this step we use the dependency distances that were assigned to each instruction to assign registers. A maximum of 8 registers are needed to control the branch transition rate and a maximum of 9 registers are used for controlling the memory access patterns. The number of registers that are used to satisfy the dependency distances is typically kept to a small value (typically around 10) to prevent the compiler from generating stack operations that store and restore the values.



(a) Steps in benchmark clone synthesis

```
_asm_ _volatile_ ("$LINSTR126: addq $23, 64, $26" : "=r" (vout_26), "=r"
                  (vout_23): "r" (vout_26), "r" (vout_23));
_asm_ _volatile_ ("$LINSTR127: addq %0, 0, %0" : "=r" (vout_12): "r" (vout_12));
_asm_ _volatile_ ("$LINSTR128: ldl $27,0(%1)" : "=r" (vout_27), "=r" (vout_12):
                  "r" (vout_27), "r" (vout_12));
_asm_ _volatile_ ("$LINSTR129: addq %0, 0,%0": "=r" (vout_11): "r" (vout_11));
_asm_ _volatile_ ("$LINSTR130: addq %0, 4,%0" : "=r" (vout_14) : "r" (vout_14));
_asm_ _volatile_ ("$LINSTR131: ldl $25,0(%1)": "=r" (vout_25), "=r" (vout_14): "r"
```

```
                    (vout_25), "r" (vout_14)) ;
_asm_ _volatile_ ("$LINSTR132: addq %0, 0,%0": "=r" (vout_12) : "r" (vout_12));
_asm_ _volatile_ ("$LINSTR133: beq $12, $LINSTR149": "=r" (vout_12): "r"
                    (vout_12));
```

(b) Code snippet from one basic block of the synthetic clone

Figure 5.2: Illustration of the Synthetic Benchmark Synthesis Process.

5.3.5. Code Generation

During the code generation phase the instructions are emitted out with a header and footer. The header contains initialization code that allocates memory using the *malloc* library call for modeling the memory access patterns and assigns memory stride values to variables. Each instruction is then emitted out with assembly code using *asm* statements embedded in C code. The instructions are targeted towards a specific ISA, Alpha in our case. However, the code generator can be modified to emit instructions for an ISA of interest. The *volatile* directive is used to prevent the compiler from reordering the sequence of instructions and changing the dependency distances between instructions in the program.

Figure 5.1(b) shows a snippet of code for one basic block from the synthetic clone targeted towards Alpha ISA. Each instruction comprises of an assembler instruction template comprising of a label (*e.g.*, $LINSTR126), an instruction type (*e.g.,* addq), the input and output registers in assembly language (*e.g.,* $23) or operands corresponding to C- expressions (*e.g.,* %0, %1), operand constraint for register type (*e.g.*, 'r' for integer and 'f' for floating-point), and register variables in C-expressions (*e.g.,* vout_22). Please refer [Gcc-Inline, 2007] for details of the syntax assembler format and techniques for specifying the operand constraint string.

5.4. Experiment Setup

We use a modified version of the SimpleScalar [Burger *et al.,* 1997] functional simulator `sim-safe` to measure the workload characteristics of the programs. In order to evaluate and compare the performance characteristics of the real benchmark and its synthetic clone, we use SimpleScalar's `sim-outorder`. We use Wattch [Brooks *et al.*, 2000] to measure the power characteristics of the benchmarks, and consider the most aggressive clock gating mechanism in which an unused unit consumes 10% of its maximum power and a unit that is used only for a fraction *n* consumes only a fraction *n* of its maximum power.

In most of our experiments, we use one 100M-instruction simulation point selected using SimPoint [Sherwood *et al.,* 2002] for the SPEC CPU2000 benchmarks, see Table 5.1; we also consider complete simulation of the benchmarks to assess the synthetic benchmark cloning method for longer-running benchmarks. As a representative of the embedded application domain we use benchmarks from the MiBench and MediaBench suite, see Table 5.2. The MiBench and MediaBench programs were run to completion. All benchmark programs were compiled on an Alpha machine using the native `Compaq cc v6.3-025` compiler with the `-O3` optimization setting.

Table 5.1: SPEC CPU 2000 programs, input sets, and simulation points used in this study.

Program	Input	Type	SimPoint
applu	ref	FP	46
apsi	ref	FP	3408
art	110	FP	340
equake	ref	FP	812
mesa	ref	FP	1135
mgrid	ref	FP	3292
swim	ref	FP	2079
wupwise	ref	FP	3237
bzip2	graphic	INT	553
crafty	ref	INT	774

eon	rushmeier	INT	403
gcc	166.i	INT	389
gzip	graphic	INT	389
mcf	ref	INT	553
perlbmk	perfect-ref	INT	5
twolf	ref	INT	1066
vortex	lendian1	INT	271
vpr	route	INT	476

Table 5.2: MediaBench and MiBench programs and their embedded application domain.

Program	**Application Domain**
basicmath, qsort, bitcount, susan	Automotive
crc32, dijkstra, patricia	Networking
fft, gsm	Telecommunication
ghostscript, rsynth, stringsearch	Office
jpeg, typeset	Consumer
cjpeg, djpeg, epic, g721-decode, mpeg, rasta, rawaudio, texgen	Media

In order to evaluate the representativeness of the synthetic clones, we use a 4-way issue out-of-order superscalar processor as our baseline configuration, Table 5.3.

Table 5.3: Baseline processor configuration.

L1 I-cache & D-cache	16 KB/2-way/32 B
Fetch, Decode, and Issue Width	4-wide out-of-order
Branch Predictor	Combined (2-level & bimodal), 4KB
L1 I-cache & D-cache – Size/Assoc/Latency	32 KB / 4-way / 1 cycle
L2 Unified cache – Size/Assoc/Latency	4MB / 8-way / 10 cycles
RUU / LSQ size	128 / 64 entries
Instruction Fetch Queue	32 entries
Functional Units	2 Integer ALU, 2 Floating Point, 1 FP Multiply/Divide, and 1 Integer

	Multiply/Divide unit
Memory Bus Width, Access Time	8B, 150 cycles

5.5. EVALUATION OF SYNTHETIC BENCHMARK CLONE

We evaluate the accuracy and usefulness of the synthetic benchmark cloning approach by applying the technique to generate clones that are representative of general-purpose, scientific, and embedded domain benchmarks. In our evaluation we compare workload characteristics of the synthetic clone with those of the original program, absolute and relative accuracy in estimating performance and power, convergence characteristics of the synthetic benchmark clone, and the ability of the synthetic benchmark to assess design changes.

5.5.1 Workload Characteristics

In this section we evaluate the proposed memory access and branching models proposed in this chapter, by comparing the cache misses-per-instruction and branch direction prediction rates of the synthetic clone with those of the original program.

Cache behavior

Figure 5.3 & Figure 5.4 respectively shows the L1 and L2 data cache misses-per-thousand-instructions for the original benchmark and the synthetic benchmark clone of the SPEC CPU programs on the base configuration. The average absolute difference between the L1 misses-per-thousand-instructions between the actual benchmark and the synthetic clone is 2 misses-per-thousand-instructions, with `mcf` having a maximum error of 6 misses-per-thousand-instructions. Looking at the L2 cache misses, see Figure 5.4, we observe that `mcf`, `equake`, `swim` and `applu` are the only programs that cause a significant number of L2 cache misses – the rest of the programs have a footprint that is small enough to fit in the L2 cache. For the four programs that show a relatively high L2 miss rate, the average difference between the misses estimated by the synthetic

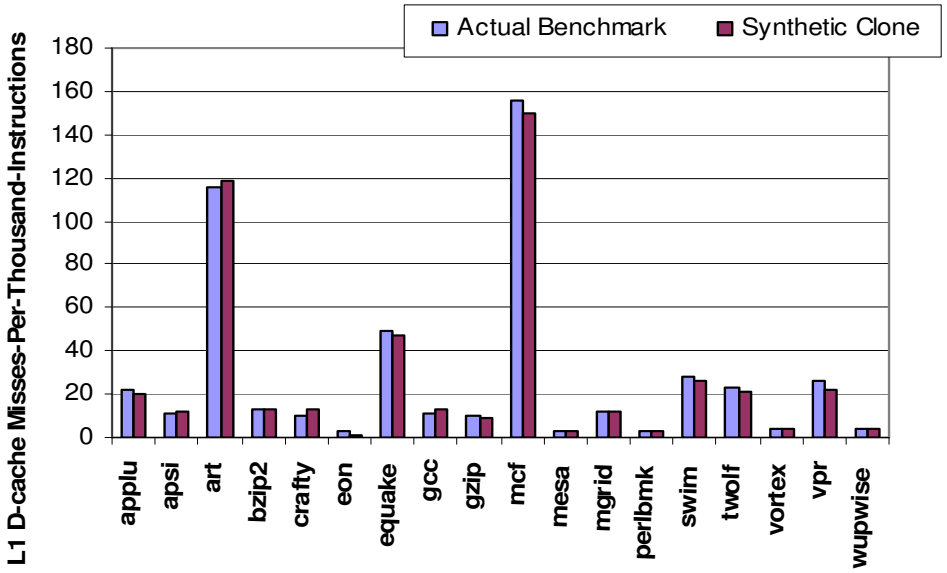


Figure 5.3: L1 data cache misses-per-thousand-instructions per benchmark and its synthetic clone for the SPEC CPU2000 benchmark programs.

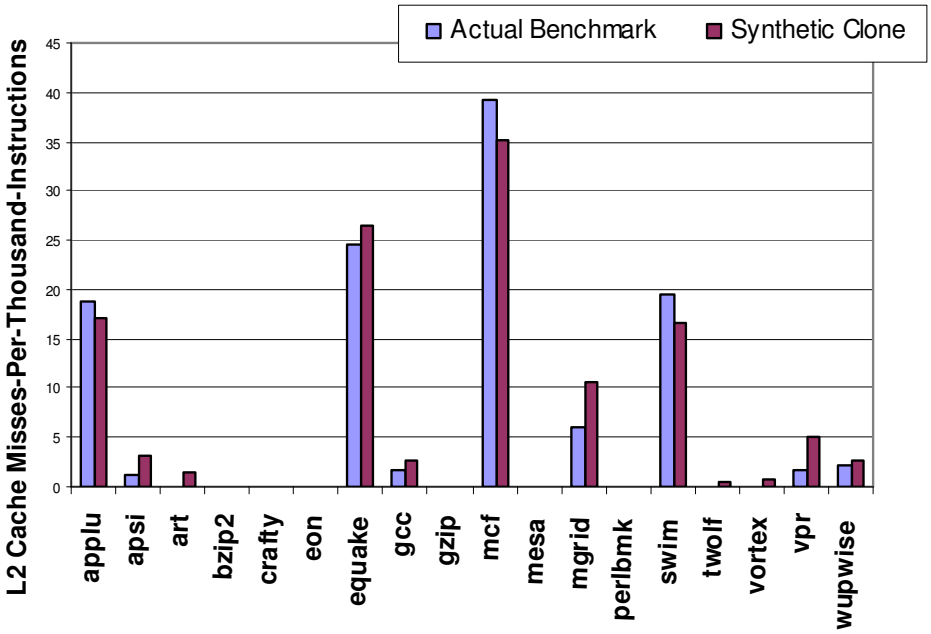


Figure 5.4: L2 unified cache misses-per-thousand-instructions per benchmark and its synthetic clone for the SPEC CPU2000 benchmark programs.

benchmark clone and the actual program is only about 3 misses-per-thousand-instructions.

For the embedded benchmarks, the L1 data cache misses-per-thousand-instructions are negligibly small, with a maximum of 8 misses-per-thousand-instructions for benchmark `typeset`. The base configuration that we use in our evaluation has a 32 KB 4-way L1 data cache, and the footprints of the embedded benchmarks are small and fit in this cache. Therefore, in order to make a meaningful comparison between the absolute errors in the difference between the misses-per-thousand-instructions estimated by the synthetic clone and the actual benchmark we use a 4KB, 2-way set-associative cache and also the ability to track design changes across a wide range of cache configurations representative of embedded systems.

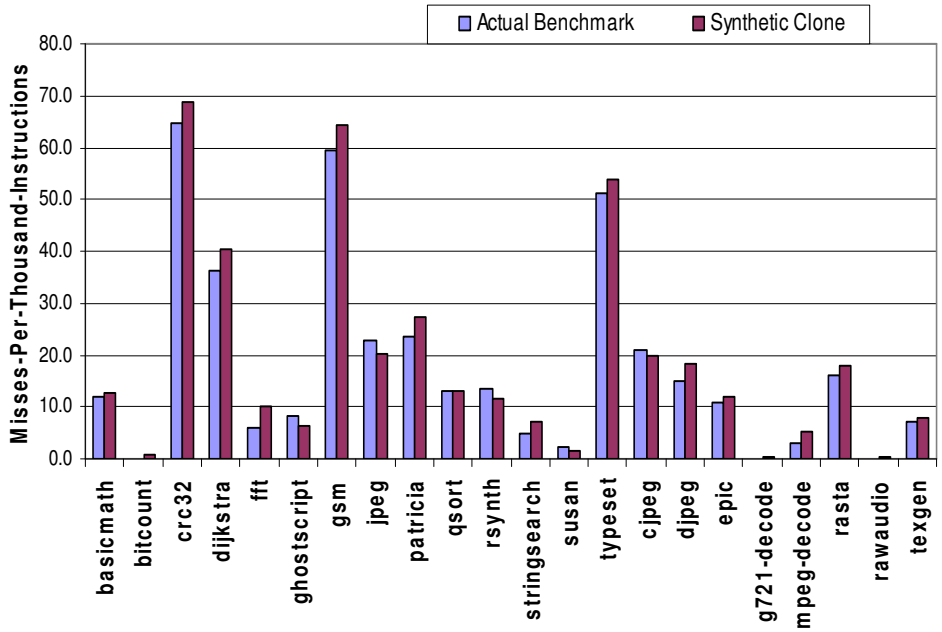


Figure 5.5: Cache misses-per-thousand-instructions per benchmark and its synthetic clone for the embedded benchmarks.

Figure 5.5 shows the L1 data misses-per-thousand-instructions for this cache configuration. The maximum absolute error is 4.7 misses-per-thousand-instructions, for benchmark `gsm`.

In order to evaluate the model for incorporating synthetic data locality we used 28 different L1 D-caches with sizes ranging from 256 Bytes to 16 KB with direct-mapped, 2-way set-associative, 4-way set-associative and fully associative configurations. The Least Recently Used replacement policy was used for all the cache configurations, and the cache line size was set to 32 bytes. We simulated the real benchmark program and the synthetic clone across these 28 different cache configurations and measured the number of misses-per-instruction. As described earlier, the primary objective of the synthetic benchmark clone is to be able to make design decisions and tradeoffs; where relative accuracy is of primary importance.

We quantify the relative accuracy for the synthetic benchmark clones using the Pearson's linear correlation coefficient between the misses-per-instruction metric for the 27 different cache configurations relative to the 256 Byte direct-mapped cache configuration - for the original benchmark and the synthetic benchmark clone. Specifically, the Pearson's correlation coefficient is given by: $R_P = S_{XY} / (S_X . S_Y)$, where X and Y respectively refer to the misses-per-instruction of the synthetic benchmark clone and the original benchmark relative to the 256 Byte direct-mapped cache configuration. The value of correlation, R, can range from -1 to 1. The Pearson's correlation coefficient reflects how well the synthetic benchmark clone tracks the changes in cache configurations – a high positive correlation indicates that the synthetic benchmark clone tracks the actual change in misses-per-instruction, *i.e.* perfect relative accuracy.

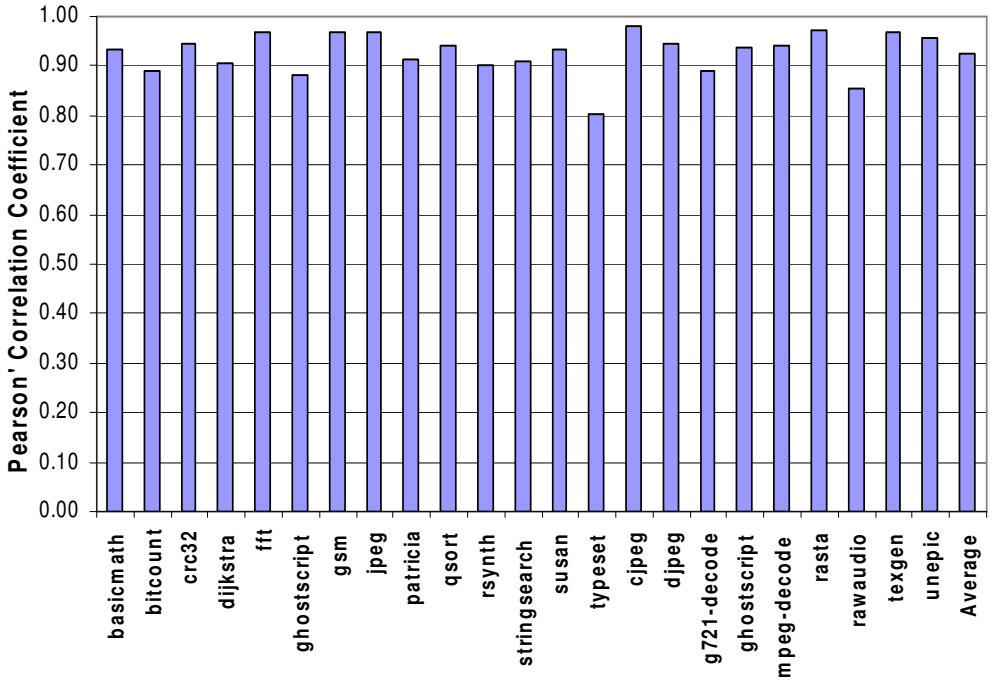


Figure 5.6: Pearson Correlation coefficient showing the efficacy of the synthetic benchmark clones in tracking the design changes across 28 different cache configurations.

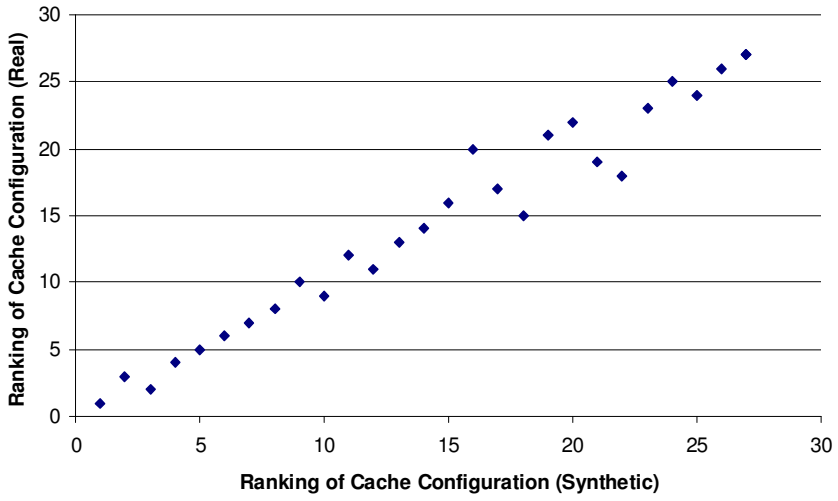


Figure 5.7: Scatter plot showing ranking of the cache configuration estimated by the synthetic benchmark clone and the real benchmark.

Figure 5.6 shows the Pearson's correlation coefficient for each benchmark program. The average correlation coefficient is 0.93, indicating very high correlation between the synthetic benchmark clone and the original benchmark application across all the applications. The benchmark typeset shows the smallest correlation (0.80) of all the benchmark suites. A plausible explanation for this observation is that the typeset benchmark needed 66 different unique streams to model its stride behavior, in compared to an average of 18 unique streams for the other benchmark program. This suggests that programs that require a larger number of unique stream values to capture the inherent data locality characteristics of a programs, introduce larger errors in the synthetic clone. This is perhaps due to the fact that having a large number of streams creates a larger number of possibilities of how the streams intermingle with each other, which is probably not accurately captured by our first-order synthetic benchmark generation method.

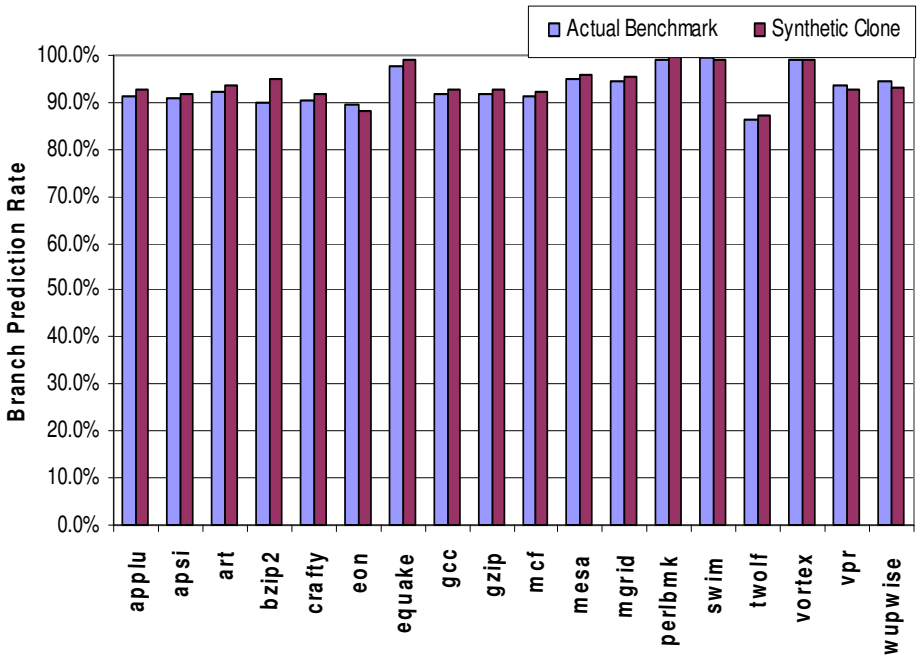
Figure 5.7 shows a scatter plot of the average rankings (cache with smallest misses-per-instruction is ranked the highest) of the 28 cache configurations predicted by the synthetic benchmark clones and the ones obtained using the real benchmark programs. Each point in the scatter plot represents a cache configuration. If the synthetic benchmarks accurately predicted all the rankings of the 28 cache configurations, all the points in the scatter plot will be along a line that passes through the origin and makes an angle of 45 degrees with the axes. From the chart it is evident that rankings predicted by the synthetic benchmark clone and those of the real benchmark are high correlated (all points are close to the 45 degree line passing through origin).

As such, based on the results in Figures 5.6 and 5.7, we can conclude that the synthetic benchmark clone is capable of tracking changes in cache sizes and associativity, and can be effectively used as a proxy for the real application in order to perform cache design studies.

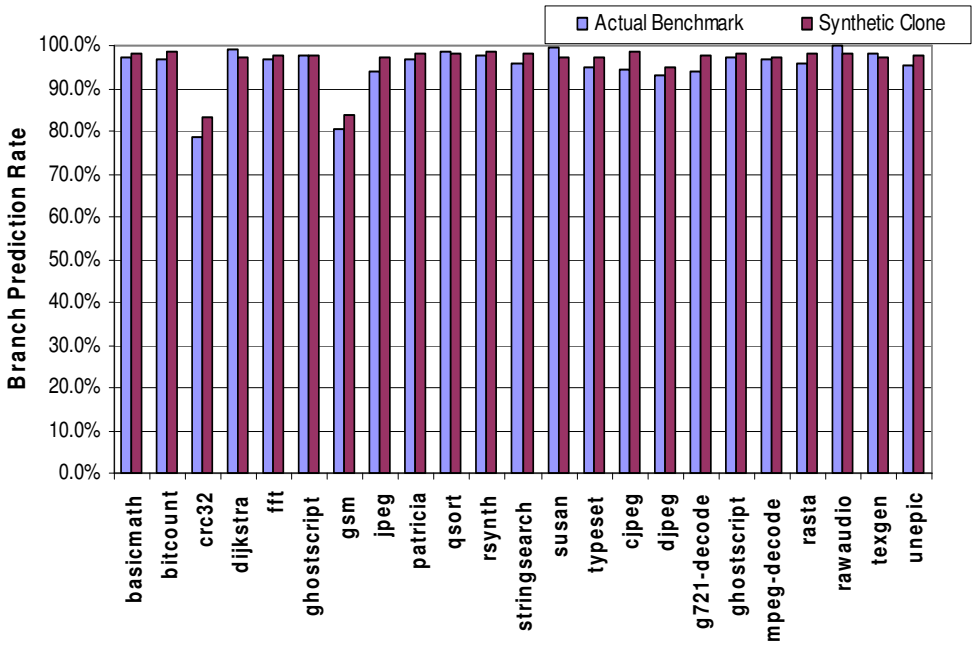
Branch behavior

Figure 5.8 shows the branch prediction rate of the synthetic clone and the original benchmark on the hybrid branch predictor considered in the baseline processor configuration. The average error shown by the synthetic clone in estimating the branch prediction rate is 1.2%, with a maximum error of 5.2% for `bzip2`. For the embedded benchmarks, the average error in the branch prediction rate is 2.1% with the maximum error of 5.3% for `crc32`, which has the lowest branch prediction rate in the entire suite.

In summary, based on the results presented in this section, we can conclude that the proposed memory access and branch models are fairly accurate, and capture the inherent workload characteristics into the synthetic benchmark clone.



(a) SPEC CPU2000 benchmarks.



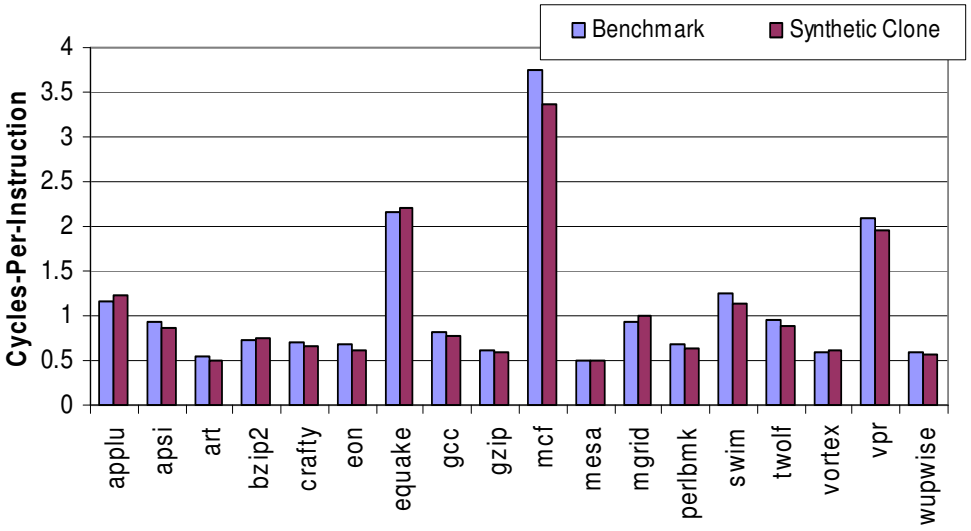
(b) Embedded benchmarks from MediaBench & MiBench suites.

Figure 5.8: Branch prediction rate per benchmark and its synthetic clone.

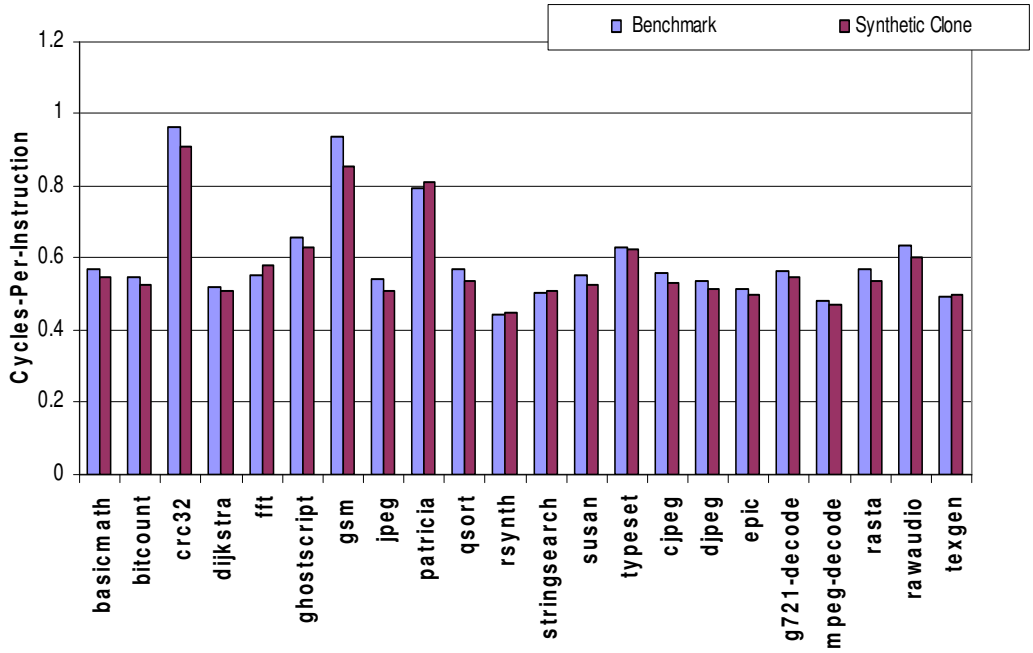
5.5.2 Accuracy in Performance & Power Estimation

We now evaluate the accuracy of synthetic cloning in estimating overall performance and energy consumption. To this end, we simulate the actual benchmark and its synthetic clone on the baseline processor configuration outlined in Table 5.3. Figures 5.9 and 5.10 respectively show the Cycles-Per-Instruction (CPI) and Energy-Per-Instruction (EPI) metrics. The average absolute error in CPI for the synthetic clone across all the SPEC CPU2000 benchmark configurations is 6.3%, with maximum errors of 9.9% for `mcf` and 9.5% for `swim`. The average absolute error in estimating the EPI is 7.3%, with maximum errors of 10.6% and 10.4% for `mcf` and `swim`, respectively. For the embedded benchmarks, the average error in estimating CPI and EPI using the

synthetic clone is 3.9% and 5.5%, respectively; the maximum error is for `gsm` (9.1% in CPI and 10.3% in EPI).

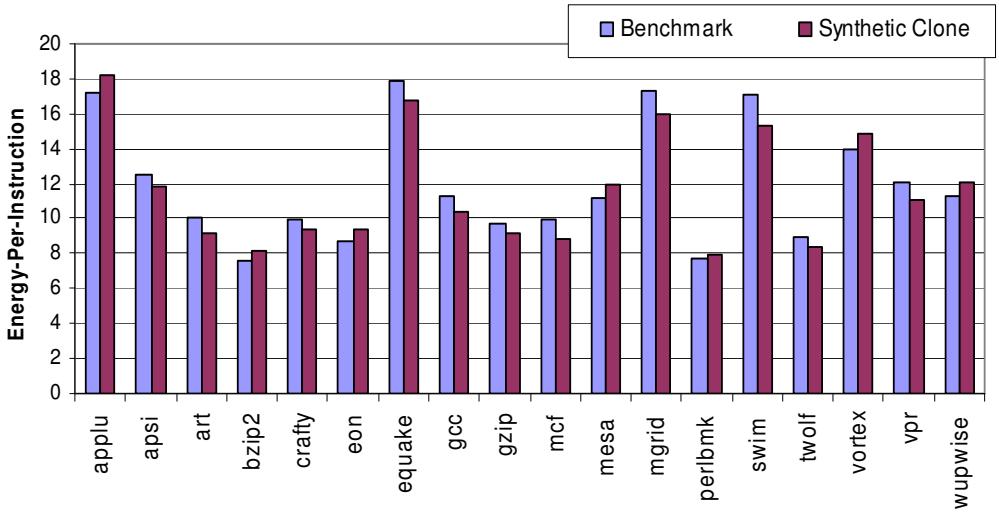


(a) SPEC CPU2000 benchmark programs.

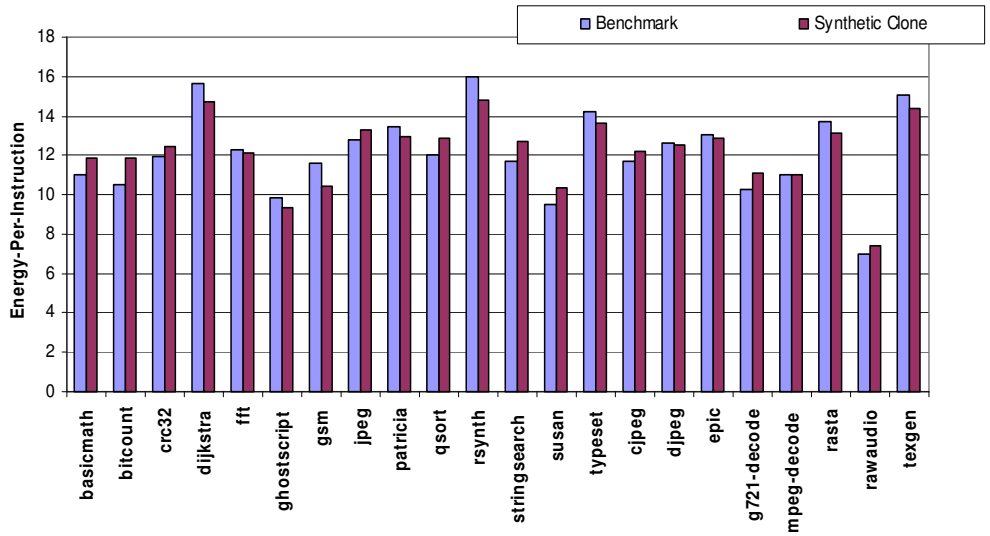


(b) Embedded benchmarks from MediaBench & MiBench benchmark suite.

Figure 5.9: Comparison of CPI of the synthetic clone versus the original benchmark.



(a) SPEC CPU2000 benchmark programs.



(b) Embedded benchmarks from MediaBench & MiBench benchmark suite.

Figure 5.10: Comparison of Energy-Per-Cycle of the synthetic clone versus the original benchmark.

In general we conclude that memory-intensive programs, such as `mcf`, `swim`, `art` and `twolf` have a higher than average absolute error in estimating CPI and EPI.

Also, for programs with very poor branch predictability, such as `gsm`, the errors are higher than average. On the other hand, the errors for control-intensive programs with moderate or high branch predictability, such as `bzip2, crafty, gcc, gzip` and `perlbmk` are relatively smaller. Overall, from these results we can conclude that the synthetic benchmark clone can accurately estimate performance and power characteristics of the original application.

5.5.3 Convergence Property of the Synthetic Benchmark Clone

The instructions in the synthetic benchmark clone are generated by probabilistically walking the SFG. In addition, the memory accesses are modeled as strides that traverse fixed-length arrays. Also, the branch instructions are configured to match a pre-set transition rate. So, if the entire spine of the program is executed in a big loop with a sufficient number of iterations, it will eventually reach steady state where the performance and power characteristics, such as CPI and EPI, converge.

Compared to the pipeline core structures, large caches will take a relatively longer time to reach steady state. Hence, in order to understand the upper bound on the number of instructions required for the program to converge, we select a memory-intensive benchmark that exhibits poor temporal locality. We have performed a similar analysis for all programs, but present the benchmark that took the largest number of instructions to converge. Benchmark `mcf` is one of the most memory-intensive programs with a very poor temporal locality, so we use this benchmark as an example. We simulated the synthetic benchmark clone for `mcf` on the base configuration described in Table 4, in a large outer loop and plotted the CPI against the dynamic instruction count, see Figure 11. The CPI initially increases and then stabilizes after around 9 million instructions; simulating more instructions only changes the CPI value by about 1%. The data cache misses in `mcf` are dominated by capacity misses, and the misses-per-instruction increases

during the course of the synthetic clone execution and eventually stabilizes at a steady state value. We experimented with other programs and all of them converged to a steady state value within 9 million instructions. So, for the benchmark programs that we studied we set 10 million instructions as an upper bound on the number of instructions required to converge. We can set this number as a guideline when selecting the outer loop count for the benchmark program – we typically set the value to 10 million instructions divided by the number of static instructions in the synthetic clone spine. If the L1 and L2 cache sizes are smaller than the one used in the configuration, the benchmark will converge faster, requiring less than 10 million instructions.

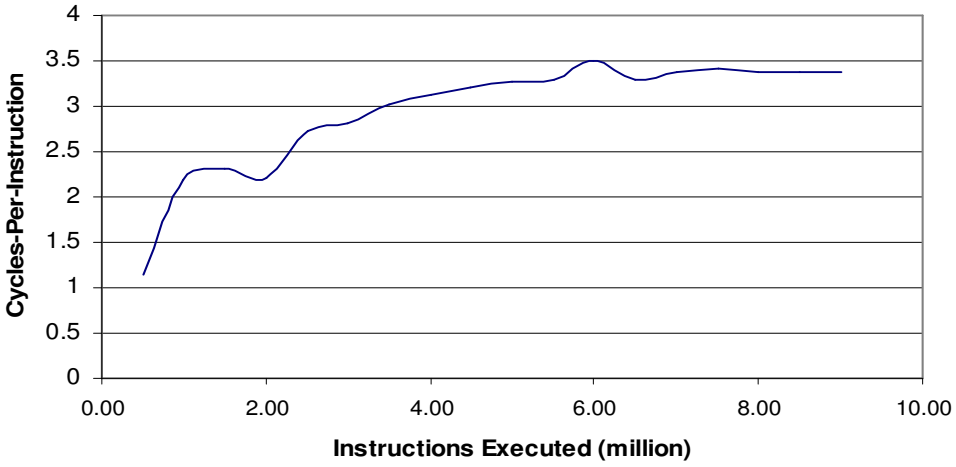


Figure 5.11: CPI versus instruction count for the synthetic clone of `mcf`.

The synthetic benchmark clones that we generate can therefore be considered as representative miniature versions of the original applications. In our setup, where we use one 100 million instruction trace, we obtain a simulation speedup of an order of magnitude. If larger streams of instructions are considered, as shown the Section 5.5.5, the saving in simulation time is over five orders of magnitude.

5.5.4 Relative Accuracy in Assessing Design Changes

In our prior evaluation we only measured the absolute accuracy of the synthetic benchmark clone in predicting performance and power for a single microarchitecture design point. However, in many empirical studies, predicting the performance trend is more important than predicting absolute performance. To evaluate the effectiveness of the synthetic benchmark clone in assessing design changes, we measure the relative accuracy by changing the cache sizes and associativity, reorder buffer size, processor width and branch predictor configuration. In each experiment, all parameters have the baseline value, except for the parameter that is being changed. When changing the RUU and LSQ size, we ensure that the LSQ size is never larger than the RUU size.

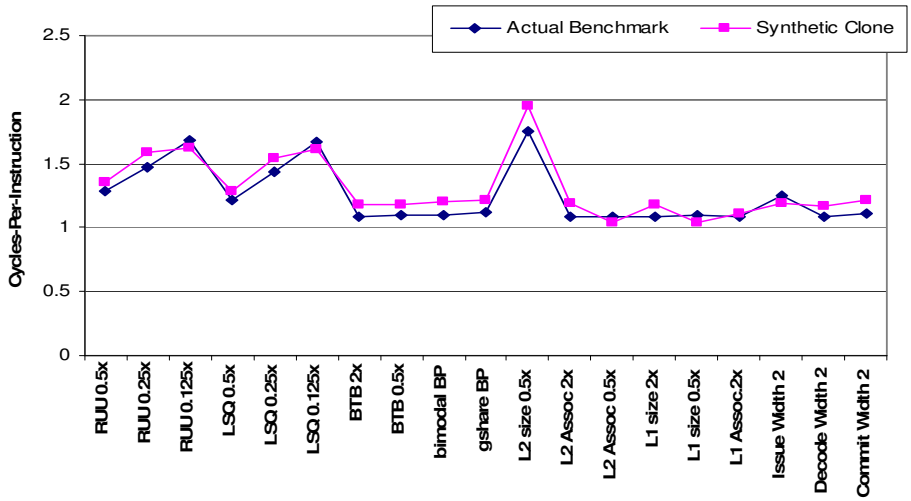


Figure 5.12: Response of synthetic benchmark clone to design changes in base configuration.

Figure 5.12 shows the design change on the horizontal axis, and the CPI of the actual benchmark and the synthetic clone on the vertical axis. The average error for the synthetic clone across all the design changes is 7.7%, with a maximum average error of 11.3% for the design change in which L2 cache size is reduced to half. As such, we

conclude that the synthetic benchmark clone is able to effectively track design changes to the processor core and the memory hierarchy.

5.5.5 Modeling long-running applications

In the prior sections we showed that the synthetic benchmark clone exhibits good accuracy when the performance characteristics of the original program are measured from one representative 100M-instruction simulation point. In order to evaluate the proposed synthetic benchmark generation methodology for modeling longer running benchmarks, we now generate a synthetic clone that is representative of the complete run of each program, and evaluate its representativeness and accuracy for the 8-way superscalar processor configuration from the SimPoint website using the published CPI numbers [SimPoint-Website]. Figure 5.13 shows the CPI estimated by the synthetic clone and that of the actual benchmark program. The trend in errors is the same as for the 100M-instruction simulation points, with mcf and swim having the highest maximum CPI prediction errors of 14.8% and 13.5%, respectively; the average CPI error is 8.2%. Table 5.4 shows the static instruction count of the synthetic clone and the simulation speedup through the synthetic cloning compared to the original benchmark. Recall that the static instruction count in the synthetic clone is the number of instructions in the program spine. These results show that the synthetic benchmark clones exhibit good accuracy even for long-running benchmarks, and result in a simulation time reduction by more than 5 orders of magnitude.

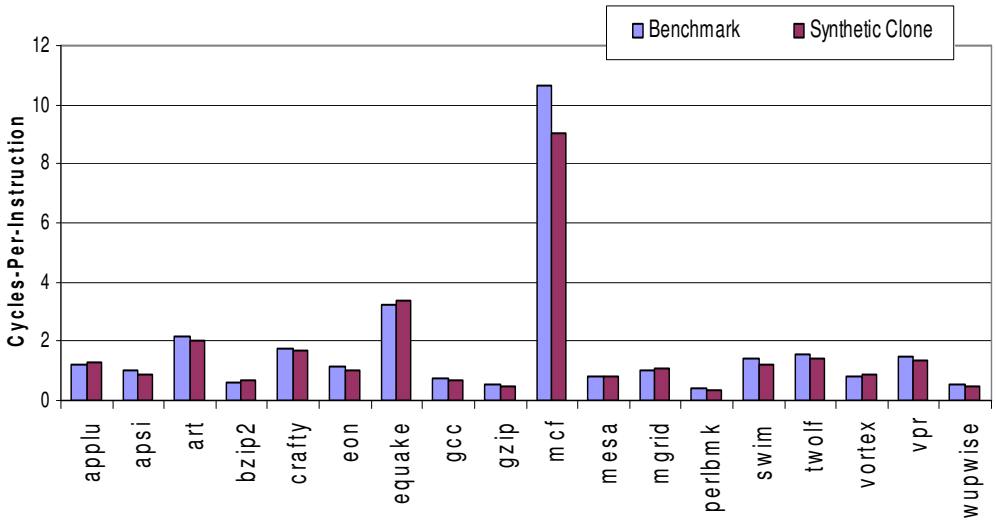


Figure 5.13: Comparing the CPI of the synthetic clone and the actual benchmark for entire SPEC CPU2000 benchmark executions.

Table 5.4: Speedup from Synthetic Benchmark Cloning.

Benchmark	**Instruction Footprint of Synthetic Clone**	**Dynamic Instruction Count Of Original Benchmark (Billion Instructions)**	**Speedup from Synthetic Benchmark Clone**
applu	9433	223	22,300
apsi	7646	347	34,700
art	554	45	4,500
bzip2	1023	128	12,800
crafty	4655	191	19,100
eon	2901	80	8,000
equake	1303	131	13,100
gcc	9177	46	4,600
gzip	1685	103	10, 300
mcf	713	61	6,100
mesa	2436	141	14,100
mgrid	2481	419	41,900

swim	3179	225	22,500
twolf	1320	346	34,600
vortex	3571	118	11,800
vpr	842	84	8,400
wupwise	1309	349	34,900

5.6. DISCUSSION

As mentioned before, the advantage of the benchmark cloning approach proposed in this chapter as compared to previously proposed workload synthesis techniques [Bell and John, 2005-1] [Bell and John, 2006] is that all the workload characteristics modeled into the synthetic benchmark clone are microarchitecture-independent. This makes the benchmarks portable across a wide range of microarchitecture configurations. However, a limitation of the proposed technique is that the synthetic benchmark clone is dependent on the compiler technology that was used to compile the original proprietary application. Therefore, the generated synthetic benchmark clone may have limited application to the compiler community for studying the effects of various compiler optimizations. Also, the synthetic benchmark clone only imbibes the characteristics that were modeled, *i.e.*, other characteristics such as value locality is not modeled and the benchmark clone cannot be used for studies exploiting these characteristics. However, if these characteristics are important and need to be modeled, one can always develop microarchitecture-independent metrics to capture their behavior and augment the benchmark cloning framework to mimic these characteristics into the synthetic benchmark clone.

A second note that we would like to make is that the synthetic benchmark clones that we generate contain instruction set architecture (ISA) specific assembly instructions embedded in C-code. Therefore, a separate benchmark clone would have to be synthesized for all target architectures of interest (*e.g.*, Alpha, PowerPC, IA-32, *etc.*).

Typically, every designer and researcher would be interested only in his particular architecture and therefore this may not be a severe problem in practice. However, if the synthetic benchmark clone is to be made truly portable across ISAs, it would be important to address this concern. One possibility could be to generate the synthetic benchmark clone using a virtual instruction set architecture that can then be consumed by compilers for different ISAs. Another possibility would be to perform binary translation of the synthetic benchmark clone binary to the ISA of interest.

A final note is that the abstract workload model presented in this chapter is fairly simple by construction, *i.e.*, the characteristics that serve as input to the synthetic benchmark generation, such as the branching model and the data locality model, are far from being complicated. We have shown that even the observed behavior of pointer-intensive programs can be effectively modeled using simple stride-based models. This was our intention: we wanted to build a model that is simple, yet accurate enough for predicting performance trends of workloads.

5.7. SUMMARY

In this chapter we explored a workload synthesis technique that can be used to clone a real-world proprietary application into a synthetic benchmark clone that can be made available to architects and designers. The synthetic benchmark clone has similar performance/power characteristics as the original application but generates a very different stream of dynamically executed instructions. By consequence, the synthetic clone does not compromise on the proprietary nature of the application. In order to develop a synthetic clone using pure microarchitecture-independent workload characteristics, we develop memory access and branching models to capture the inherent data locality and control flow predictability of the program into the synthetic benchmark clone. We developed synthetic benchmark clones for a set of benchmarks from the SPEC

CPU2000 integer and floating-point, and MiBench and MediaBench benchmark suites, and showed that the synthetic benchmark clones exhibit good accuracy in tracking design changes. Also, the synthetic benchmark clone runs more than five orders of magnitude faster than the original benchmark, and significantly reduces simulation time on cycle-accurate performance models.

The proposed technique will benefit architects and designers to gain access to real-world applications, in the form of synthetic benchmark clones, when making design decisions. Moreover, the synthetic benchmark clones will help the vendors to make informed purchase decisions, because they would have the ability to benchmark a processor using the synthetic benchmark clone as a proxy of their application of interest.

Chapter 6: Towards Scalable Synthetic Benchmarks

The focus of Chapter 4 and 5 was to improve the accuracy of the benchmark generation framework to improve the accuracy and representativeness of the benchmark clone. This is important for cloning the performance of an existing real-world application. However, in order to model emerging applications and futuristic workloads the flexibility to alter program characteristics is more important than the accuracy or representativeness of the synthetic workload.

This chapter shows that the benchmark generation strategy can be adapted to construct scalable synthetic benchmarks from a limited number of hardware-independent program characteristics. Essentially, we develop a parameterized workload model that enables the construction of benchmarks that allow researchers to explore a wider range of the application behavior space, even when no benchmarks (yet) exist. This chapter also demonstrates the applicability and the usefulness of BenchMaker for studying the impact of program characteristics on performance and how they interact with processor microarchitecture.

6.1 The Need For Developing A Parameterized Workload Model

The advent of standardized benchmark suites has streamlined the process of performance comparison between different computer systems, architects and researchers face several challenges when using benchmarks in industry product development and academic research. These problems primarily emerge from the fact that standardized benchmarks are rigid and it is not possible to alter their characteristics to study program behavior and model emerging workloads.

One of the approaches for addressing these limitations is to complement standardized benchmark suites with synthetic benchmarks. A synthetic program that can

be tuned to produce a variety of benchmark characteristics would be of great benefit to the computer architecture community. An approach to automatically generate scalable synthetic benchmarks can help in: (1) constructing synthetic benchmarks to represent application characteristics for which benchmarks do not (yet) exist, (2) isolating individual program characteristics into microbenchmarks, (3) altering hard-to-vary benchmark characteristics, and (4) modeling commercial workload that have large hardware requirements for full-scale setup. The objective of this chapter is to propose a framework, called BenchMaker, which adapts the benchmark generation strategy to construct scalable benchmarks whose code properties can easily be altered.

The synthesis approaches proposed in Chapter 4 and 5, and prior work in statistical simulation and benchmark synthesis has at least one shortcoming that limits its ability to generate scalable benchmarks by varying program characteristics. Firstly, in most of these approaches [Nussbaum and Smith, 2001] [Eeckhout *et al.,* 2004-2] [Bell and John, 2005-3], an application is characterized using a detailed workload model – a statistical flow graph captures the control flow behavior of a program and characteristics such as instruction mix, register dependency distribution, control flow predictability, and memory access pattern – that are measured at the granularity of a basic block. This involves specifying a large number of probabilities to describe a workload, which is highly impractical when using these frameworks for exploring workload behavior spaces by varying workload characteristics. Secondly, although some of the approaches for generating synthetic workloads [Oskin *et al.,* 2000] [Eeckhout *et al.,* 2001] show that applications can be modeled using a limited of number of program characteristics, they use a combination of microarchitecture-dependent and microarchitecture-independent program characteristics. Microarchitecture-dependent characteristics, such as branch misprediction rate and cache miss rate, do not capture the inherent program

characteristics and make it difficult to explore the entire application behavior space independently from the underlying hardware. Finally, a shortcoming of some of these techniques is that they generate synthetic workload traces, precluding their use on real hardware, execution-driven simulators, and RTL models.

The approach proposed in this chapter overcomes these shortcomings. Unlike prevailing approaches to generating synthetic benchmarks, the BenchMaker framework that we propose makes it possible to alter inherent workload characteristics of a program by varying a limited number of key microarchitecture-independent program characteristics in a synthetic benchmark – changing the workload behavior is done by simply 'turning knobs'. This ability to vary program characteristics makes it possible to efficiently explore the application behavior space. Specifically, this chapter makes the following contributions:

1) It shows that it is possible to adapt the workload model to capture a program behavior with just a few microarchitecture-independent workload characteristics, albeit at the cost of slightly reduced accuracy. This is much more efficient than the collection of distributions that need to be specified in the benchmark cloning approach.
2) It evaluates the usefulness of the BenchMaker framework by demonstrating its applicability to three different areas: (a) Studying the effect of inherent workload characteristics on performance, (b) Studying the interaction of microarchitecture-independent workload characteristics with the microarchitecture features of a processor, and (c) Accounting for workload drift during microprocessor design.

The remainder of this chapter is structured as follows. In section 6.2 we provide an overview of the proposed parameterized model for constructing scalable synthetic

benchmarks from program characteristics. In section 6.3 we describe our simulation environment, machine configuration, and the benchmarks used to evaluate the BenchMaker framework. In section 6.4 we evaluate the BenchMaker framework by demonstrating how it can be used to generate synthetic benchmarks that exhibit similar behavior to SPEC CPU2000 Integer benchmarks. In sections 6.5 we demonstrate the application of the BenchMaker framework to three challenging problems. Finally, in section 6.6 we summarize the key results from this chapter.

6.2 BENCHMAKER FRAMEWORK FOR PARAMETERIZED WORKLOAD SYNTHESIS

Figure 6.1 illustrates the approach used by the BenchMaker framework that we propose in this chapter for generating synthetic benchmarks from a set of microarchitecture-independent program characteristics. The program characteristics measure the inherent properties of the program that are independent from the underlying hardware. Collectively, these characteristics form an abstract workload model. This abstract workload model serves as an input to the synthetic benchmark generator. Our intention is to develop a workload model that is simple yet accurate enough for predicting performance trends across the workload space. Keeping the workload model simple makes it possible to not only accurately model the characteristics of an existing workload into a synthetic benchmark, but also provides the ability to conduct 'what-if' studies by varying program characteristics. In the following sections we describe the workload characteristics that serve as input to the synthetic workload generator and we also describe the algorithm used for modeling these characteristics into a synthetic workload.

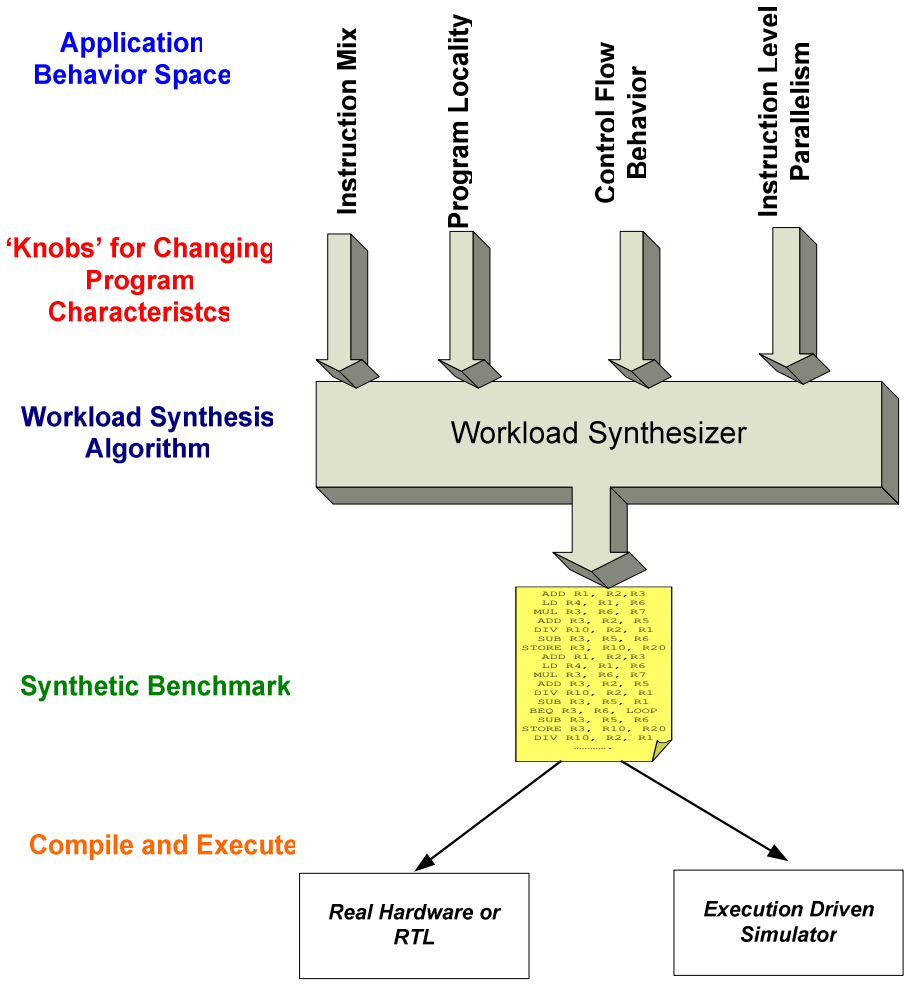


Figure 6.1: The BenchMaker framework for constructing scalable synthetic benchmarks.

6.2.1 Workload Characteristics

The characteristics that we propose to drive the benchmark synthesis process are a subset of all the microarchitecture-independent characteristics that can be modeled. However, we believe that our abstract workload model captures (most of) the important program characteristics that potentially impact a program's performance; the results from

evaluation of the synthetic benchmarks in this chapter in fact show that this is the case, at least for the benchmarks that we used.

Recall that the key goal of the parameterized framework is to show that it is possible to maintain good representativeness and good accuracy with a limited number of key workload characteristics. For limiting the number of program characteristics, we capture them at a coarse granularity using average statistics over the entire program. Although measuring program characteristics at a coarse granularity likely reduces the representativeness of the synthetic benchmarks compared to fine grained characteristics, this is key to enable the flexibility in BenchMaker for generating benchmarks with characteristics of interest. This will enable one to easily vary workload characteristics by 'turning knobs' and make it possible to answer 'what-if' questions. We propose to measure the following workload characteristics at the program level.

Instruction Mix. The instruction mix of a program measures the relative frequency of various operations performed in the program; namely the percentage of integer small latency, integer long latency, floating-point small latency, floating-point long latency, integer load, integer store, floating-point load, floating-point store, and branches in the dynamic instruction stream of a program.

Basic Block Size. A basic block is a section of code with one entry and one exit point. We measure the basic block size as the average number of instructions between two consecutive branches in the dynamic instruction stream of a program. We assume that the basic block sizes in the program have a normal distribution, and characterize them in terms of the average and standard deviation in the basic block size distribution of a program.

Instruction Level Parallelism. The dependency distance is defined as the number of instructions in the dynamic instruction stream between the production (write) and

consumption (read) of a register and/or memory location. The goal of characterizing the data dependency distances is to capture a program's inherent ILP. We measure the data dependency distance information on a per instruction basis and summarize it as a cumulative distribution organized in eight buckets: percentages of dependencies that have a dependency distance of 1 instruction, and the percentage of dependency dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions. Longer dependency distances permit more overlap of instructions in a superscalar out-of-order processor.

Data Footprint. We measure the data footprint of a program in terms of the total number of unique data addresses referenced by the program. The data footprint of a program gives an idea of whether the data set fits into the level-1 or level-2 caches.

Data Stream Strides. We model the data stream with respect to the distribution of the local data strides. A local stride is defined as the difference in the data memory addresses between successive memory addresses from a single static instruction. We describe the local strides in terms of 32-byte block sizes (analogous to a cache line size), i.e., stride 0 refers to a local data stride of 0 to 32 bytes (consecutive addresses are within one cache line distance). The local strides are summarized as a histogram showing the percentage of memory access instructions with stride values of 0, 1, 2, *etc.*

In order to capture the data access pattern of a program we measure a distribution of local strides in the program. Local stride value is the difference between two consecutive effective addresses generated by the same static load or store instruction. We measure the local strides in terms of 32-byte block sizes (analogous to a cache line size), i.e., if a local stride is between 0 or 31 bytes, it is classified as stride 0 (consecutive addresses are within one cache line distance), between 32 and 63 bytes as stride 1, etc. We summarize the local stride distance for the entire program as a histogram showing the

percentage of memory access instructions with stride value of 0, 1, 2, *etc.* Figure 2 shows the distribution of the data stride values of SPEC CPU2000 Integer Programs. From this figure we observe that for the `bzip2`, `crafty`, `gzip`, and `perlbmk` benchmarks, more than 80% of the local stride references are within a 32-byte block size, indicating very good spatial data locality. The `gcc`, `twolf`, and `vortex` benchmarks only have 60% of local stride values that are within a 32-byte block size, and exhibit moderate spatial data locality. The `vpr` benchmark shows two extremes, with approximately 50% of local strides accessing the same 32-byte block, and the other 50% with extremely large local stride values, indicating a mix of references with extremely poor and extremely high spatial locality. The `mcf` benchmark is an outlier and has very poor data locality, with most of the local stride values being extremely large.

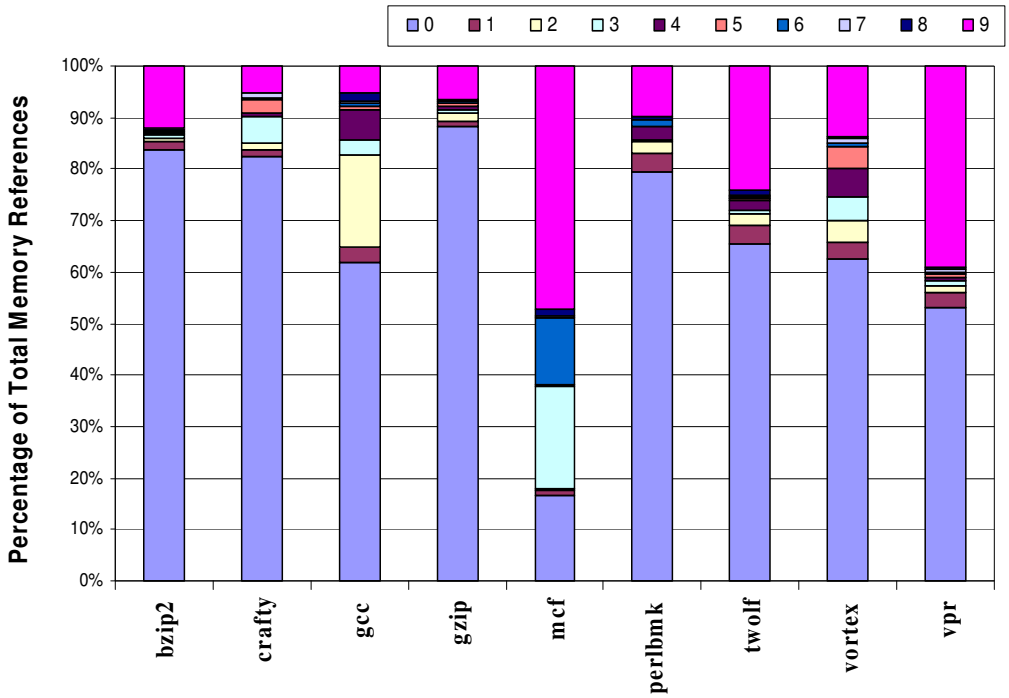


Figure 6.2: Percentage breakdown of local stride values.

The combination of data footprint and the stride value distribution captures the inherent data locality in the program. These two characteristics are typically very difficult to modify in standard benchmarks. In synthetic benchmarks it is easy to fix one of these parameters and study the effect of the other. For example, using BenchMaker, we can easily study the impact of changing stride values while keeping the data footprint the same. Or, if of interest, one can explore the combined effect of varying footprint and access patterns.

Instruction Footprint. We characterize the instruction footprint as the total number of unique instructions referenced by the program. The instruction footprint of a program gives an idea of whether the data set fits into the level-1 or level-2 caches. The instruction footprint of all the programs are very small (`gcc` has the highest instruction footprint) and do not stress the instruction cache.

Branch Transition Rate. In order to capture the inherent branch behavior in a program, the most popular microarchitecture-independent metric is to measure the percentage of taken branches in the program or the taken rate for a static branch, i.e., fraction of the times that a static branch was taken during the complete run of the program. Branches that have a very high or low taken rate are biased towards one direction and are considered to be highly predictable. However, merely using the taken rate of branches is insufficient to actually capture the inherent branch behavior. The predictability of the branch depends more on the sequence of taken and not-taken directions than just the taken rate.

Therefore, in our control flow predictability model we also measure an attribute called transition rate, due to [Haungs *et al.,* 2000], for capturing the branch behavior in programs. Transition rate of a static branch is defined as the number of times it switches between taken and not-taken directions as it is executed, divided by the total number of

times that it is executed. By definition, the branches with low transition rates are always biased towards either taken or not-taken. It has been well observed that such branches are easy to predict. Also, the branches with a very high transition rate always toggle between taken and not-taken directions and are also highly predictable. However, branches that transition between taken and not-taken sequences at a moderate rate are relatively more difficult to predict. In order to incorporate synthetic branch predictability we measure a distribution of transition rate of all static branches in the program. When generating the synthetic benchmark clone we ensure that the distribution of the transition rates for static branches in the synthetic stream of instructions is similar to that of the original program. We achieve this by configuring each basic block in the synthetic stream of instructions to alternate between taken and not-taken directions, such that the branch exhibits the desired transition rate.

Summary. To summarize the above discussion, the abstract model characterizing a workload consists of 40 numbers in total, as shown in Table 6.1. Collecting only 40 workload statistics results in a much more compact representation of a workload; compared to the benchmark cloning approach (Chapters 4 and 5) where most of these statistics are separately measured for every basic block resulting in typically several thousands of numbers to characterize a workload. Consequently, the BenchMaker framework has 40 'knobs' that can be controlled to efficiently explore the application behavior space.

Table 6.1: Microarchitecture-independent characteristics that form an abstract workload model.

Category	Num.	Characteristic
instruction mix	10	percentage of integer short latency percentage of integer long latency percentage of floating-point short latency percentage of floating-point long latency percentage of integer load percentage of integer store percentage of floating-point load percentage of floating-point store percentage of branches
instruction level parallelism	8	register-dependency-distance – 8 distributions for register dependencies. Register dependency distance equal to 1 instruction, and the percentage of dependency dependencies that have a distance of up to 2, 4, 6, 8, 16, 32, and greater than 32 instructions.
data locality	1 10	data footprint distribution of local stride values
instruction locality	1	instruction footprint
branch predictability	10	distribution of branch transition rate

6.2.2 Synthetic Benchmark Construction

The benchmark synthesis algorithm is similar the one described in Chapter 4 and 5 except that the workload characteristics measured at a coarser granularity are used. Recall that in the benchmarking cloning approach the synthetic clone is generated using workload characteristics measured at a finer granularity – basic block level. The 40 workload characteristics or knobs serve as an input the synthesis algorithm compared to thousands of statistics for the benchmark cloning synthesis algorithm. This makes it easy to synthesize a new benchmark by altering a particular workload characteristic.

6.3 EXPERIMENT SETUP

In all of our experiments we use the `sim-alpha` simulator [Desikan *et al.,* 2001] from the `SimpleScalar` Tool Set [Burger and Austin, 1997]. The `sim-alpha` simulator is an execution driven performance model that has been validated against the superscalar out-of-order Alpha 21264 processor. In order to measure the

abstract workload characteristics of a program we used a modified version of the `sim-safe` simulator. In our experiments we use benchmarks from the SPEC CPU Integer benchmark suite that are representative of general purpose application programs. In most of our experiments we use one 100M-instruction simulation point selected using `SimPoint` [Sherwood *et al.,* 2002]. However, when comparing programs from two generations of SPEC CPU Integer benchmark suites we use multiple simulation points. All the SPEC CPU2000 Integer benchmark programs were compiled on an `Alpha` machine using the native `Compaq cc v6.3-025` compiler with `-O3` compiler optimization. The SPEC CPU95 Integer benchmark program, `gcc`, was compiled using a native circa 1995 compiler, `gcc 2.6.3`. Table 6.2 summarizes the benchmarks and the simulation points that were used in this study. We also used traces of three commercial workloads – SPECjbb2005 (representative of JAVA server workloads), DBT2 (representative of an OLTP workload), and DBMS (a database management system workload). The traces for the commercial workloads were generated using the SIMICS full-system simulator and simulated using a modified version of a trace driven sim-outorder simulator.

Table 6.2: SPEC CPU programs, input sets, and simulation points used in study.

Benchmark	**Input**	**SimPoint(s)**
SPEC CPU2000 Integer		
`bzip2`	graphic	553
`crafty`	ref	774
`eon`	rushmeier	403
`gcc`	166.i	389
`gzip`	graphic	389
`mcf`	ref	553
`perlbmk`	perfect-ref	5
`twolf`	ref	1066
`vortex`	lendian1	271
`vpr`	route	476
`gcc`	expr	8, 24, 47, 51, 56, 73,

		87, 99
SPEC CPU95 Integer		
gcc	expr	0, 3,5,6,7,8,9,10,12

6.4 EVALUATION OF BENCHMAKER FRAMEWORK

In this section we evaluate BenchMaker's accuracy by using it to generate synthetic benchmark versions of general-purpose (SPEC CPU INT2000) and commercial (SPECjbb2005, DBT2, and DBMS) workloads. We measure the program characteristics of the SPEC CPU2000 and commercial workloads and feed this abstract workload model to the BenchMaker framework to generate a synthetic benchmark; we then compare the performance/power characteristics of the synthetic benchmark against the original workload.

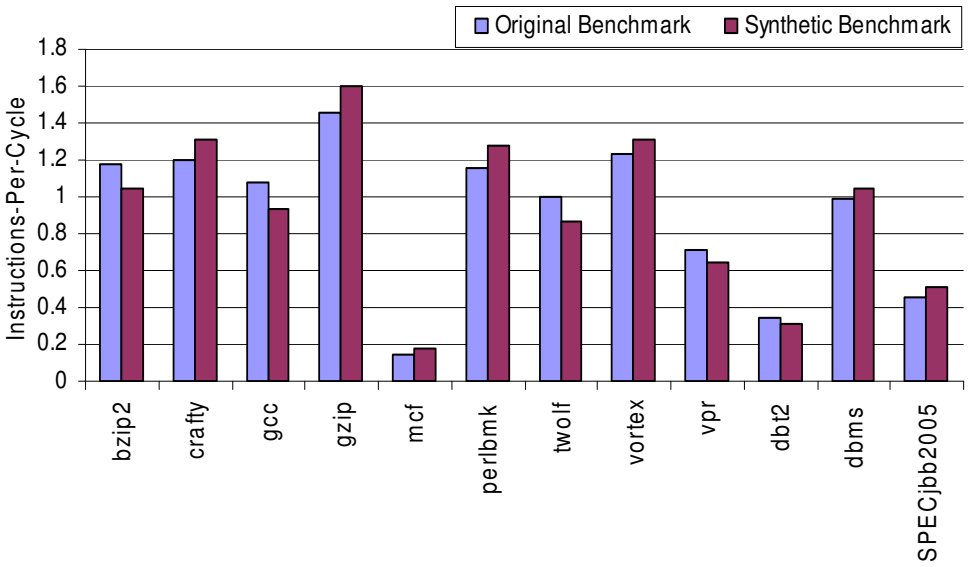
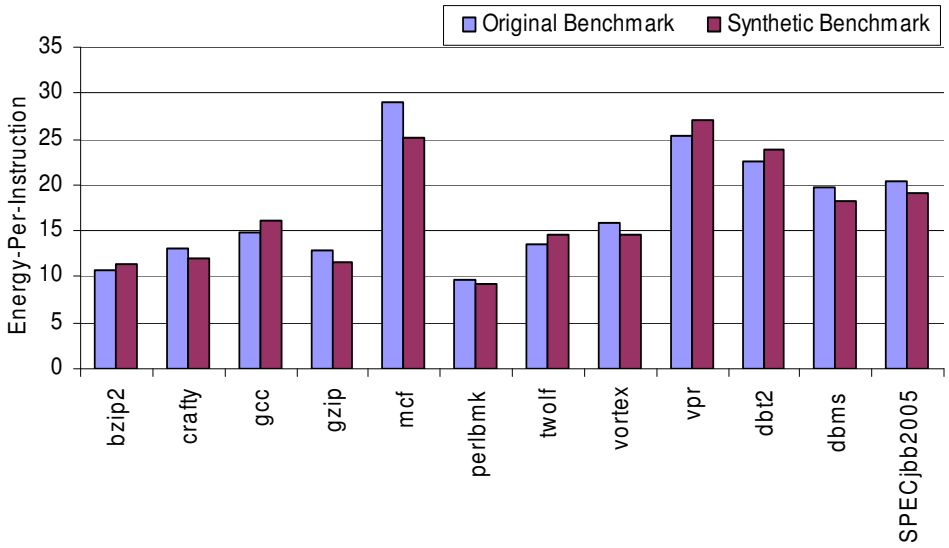
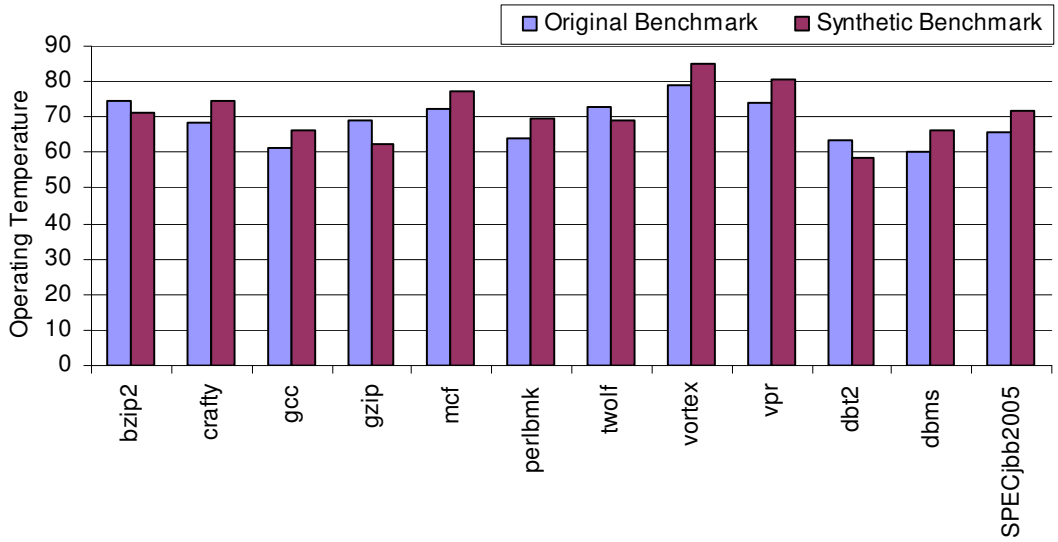


Figure 6.3: Comparison of Instructions-Per-Cycle (IPC) of the actual benchmark and its synthetic version.



(a) Energy-Per-Instruction



(b) Operating Temperature

Figure 6.4: Comparison of Energy-Per-Instruction (EPI) and Operating Temperature of the actual benchmark and its synthetic version.

Figure 6.3 evaluates the accuracy of BenchMaker for estimating the pipeline instruction throughput measured in instructions-per-cycles (IPC). We observe that the synthetic benchmark performance numbers track the real benchmark performance

numbers very well. The average IPC prediction error is 10.9% and the maximum error is observed for mcf (19.9%). Figure 6.4 shows similar results for the Energy-Per-Instruction (EPI) metric and the average operating temperature (details of the microarchitecture level temperature modeling tools are described in Chapter 7). The average error in estimating EPI from the synthetic version is 7.5%, with the maximum error of 13.1% for mcf. The average error in estimating the average operating temperature is 8.1%.

Parameterization of workload metrics make it possible to succinctly describe an application's behavior using a limited number (40) of fundamental coarse-grain program characteristics instead of having several thousands of fine-grain program metrics. BenchMaker trades accuracy (10.9% average error in IPC compared to less than 6% error in the benchmarking cloning approach in Chapters 4 and 5) for the flexibility to enable one to easily alter program characteristics and workload behavior.

Figure 6.5 shows similar results for the L1 D-cache performance: the number of L1 D-cache misses per one thousand instructions is shown on the vertical axis for the various benchmarks. Again, the synthetic benchmark numbers track the real benchmark numbers very well. The maximum error in predicting the number of L1 cache misses-per-1K instructions is observed for mcf for which the difference between the real and the synthetic benchmark is 9 misses-per-1K-instructions (or less than 4% in relative terms). We obtain similar results for the L2 cache performance. All of the benchmarks except for mcf and vpr have a negligibly small miss-rate at the L2 cache level; mcf shows 120 L2 misses-per-1K-instructions, and vpr shows 8 L2 misses-per-1K instructions. The synthetic benchmark accurately tracks this trend, and shows 114 and 5 L2 misses-per-1K instructions respectively for mcf and vpr benchmarks. Also, the L1 instruction cache miss-rate is negligible for all programs, with gcc having the highest miss-rate of 1.3%.

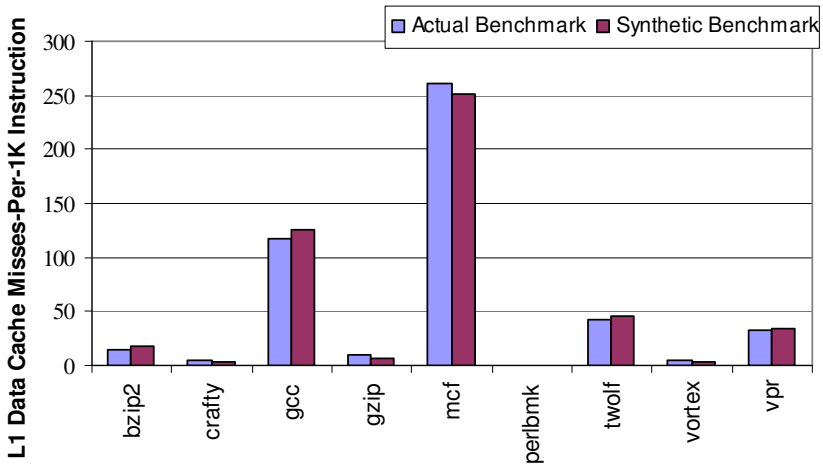


Figure 6.5: Comparison of the number of L1 D-cache misses-per-1K-instructions for the actual benchmark and its synthetic version.

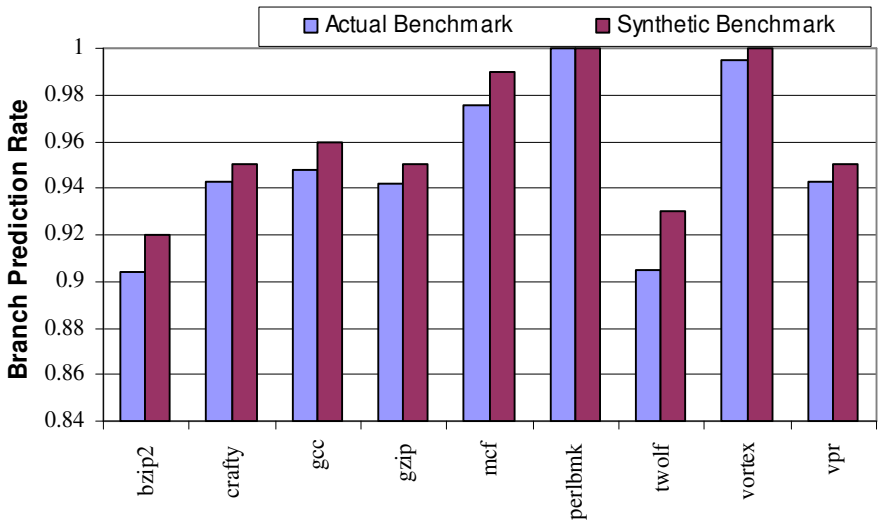


Figure 6.6: Comparison of the branch prediction rate for the actual benchmark and its synthetic version.

Figure 6.6 evaluates the accuracy of BenchMaker for replicating the branch behavior of a real benchmark into a synthetic benchmark. Here again, we observe that the synthetic versions of the benchmark track the real benchmark numbers very well. One

particularity to note here is that the branch prediction rates are always higher for the synthetic benchmarks than for the real benchmarks. This suggests that some of the difficult to predict branch sequences in the program are not captured in the synthetic benchmark. The branches in the synthetic benchmark tend to be relatively easier to predict than is the case for the original benchmark.

6.5 APPLICATIONS OF BENCHMAKER FRAMEWORK

6.5.1 Program Behavior Studies

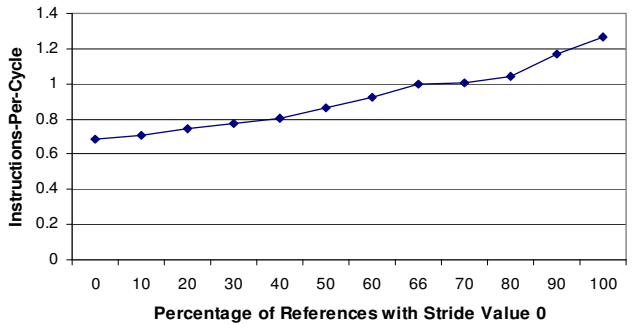
In order to demonstrate the usefulness of the BenchMaker framework we show how it can be applied for studying workload behavior and its interaction with microarchitecture. It is extremely difficult to conduct comparable 'what-if' studies using a set of standardized benchmarks because their characteristics form an essential part of the benchmark application and cannot be easily altered. On the contrary, using BenchMaker, it is possible to easily generate a benchmark program from a limited list of characteristics.

We generate a synthetic benchmark using the average of all the characteristics across the SPEC CPU Integer benchmark programs. The synthetic benchmark, AvgSynBench, modeling the average characteristics shows a pipeline throughput of 1.1 IPC on the Alpha 21264 processor. In our study we use the characteristics of this benchmark as our baseline characteristics and alter them to study the effect of each program characteristic on performance, their interaction with each other, and their interaction with the microarchitecture.

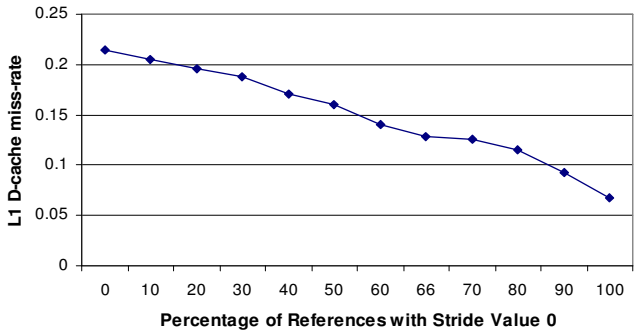
6.5.1.1 Impact of Individual Program Characteristics on Performance

In this section we use BenchMaker to study the impact of data locality and control flow predictability by varying memory access patterns and branch transition rates,

respectively. Figure 6.7 shows how the change in percentage of references with zero strides (subsequent executions of the same static memory operations access memory within a 32-byte block size) affects IPC and L1 D-cache miss-rate. We observe that as the percentage of references with zero stride varies from 0 (no accesses to the same cache line) to 100 (all executions of the same static memory operation access the same cache line), the IPC of the program linearly increases. Interestingly, the drop in L1 data cache miss-rate is also linear with the increase in percentage of references with stride value 0. This suggests that if all other characteristics remain constant, the L1 data cache miss-rate and IPC have an almost perfect negative linear correlation (-0.99).



(a) Impact of the percentage of references with zero stride value on IPC



(b) Impact of the percentage of references with zero stride value on L1 D-cache miss-rate.

Figure 6.7: Studying the impact of data spatial locality by varying the local stride pattern.

Next we study how the branch transition rate affects performance. Recall, that the branch transition rate of a program is measured as a distribution. We experimented with a number of random combinations of distribution of transition rates. We observed that with these random combinations, the branch prediction rate varies between 0.99 and 0.82, and correspondingly the variation in IPC was a factor of 1.61 (61% dip in performance if branch prediction rate falls to 0.82).

Based on these studies we can conclude that the BenchMaker framework is a useful tool for isolating and studying the behavior of individual program characteristics and their impact on performance.

6.5.1.2 Interaction of Program Characteristics

In our abstract workload model we characterize the data locality of a program by measuring its data footprint (an indicator of temporal locality) and the distribution of local stride pattern (an indicator of spatial locality).

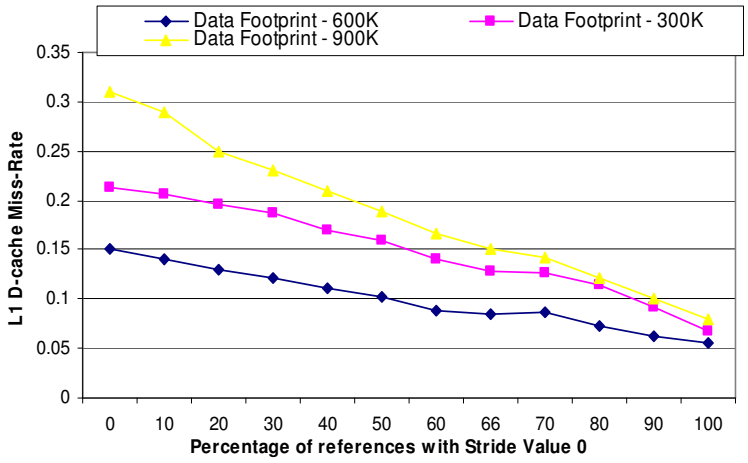


Figure 6.8: Interaction of local stride distribution and data footprint program characteristics.

In this section we analyze how the local stride distribution pattern and the data footprint of a program interact with each other. Figure 6.8 shows the effect of changes in percentage of references with zero strides for three different data footprints. From this graph we observe that for larger footprints, we see a steeper fall in L1 D-cache miss-rate as the percentage of references with stride value 0 increases. For the case where 100% of the references access the same cache line, the footprint does not seem to have an impact on the L1 D-cache miss-rate.

6.5.1.3 Interaction of Program Characteristics with Microarchitecture

A benchmark synthesis framework is not only useful for isolating and studying the impact of program characteristics on performance, but is also an invaluable tool to understand how program characteristics interact with microarchitectural structures. For example, BenchMaker can be used to find a combination of program characteristics that interact poorly with a given microarchitecture. More in particular, automatically generating a benchmark that 'stresses' the microarchitecture can give insight into critical program-microarchitecture interactions. The 'stress' benchmarks can help in exposing performance anomalies and understanding the limitations of a given microarchitecture.

As an example, in order to find a benchmark that stresses the branch predictor, we generated a number of synthetic benchmarks that contain randomly generated distributions of transition rates. Interestingly, the transition rate distribution that resulted in the lowest prediction rate was the case where 100% of the branches have a transition rate between 90% and 100%. In this configuration, every branch in the synthetic benchmark continuously toggles between taken and not-taken directions. This sequence of branches heavily stresses the Alpha 21264 branch predictor (which is a tournament branch predictor that chooses between local and global history to predict the direction of a given branch): it achieves a branch prediction rate of only 82%. Similarly,

this approach can be extended to stress-test different microarchitectural structures for performance, power, energy and temperature studies

6.5.2 Workload Drift Studies

Research work [Yi *et al.,* 2006-1] has shown that it is important to account for the potential impact of workload drift when designing a microprocessor. This section demonstrates how BenchMaker can be used to study workload drift.

6.5.2.1 Analyzing the impact of benchmark drift

As a first case study, we use the `gcc` benchmark with the `expr` input set from the SPEC CPU95 and SPEC CPU00 benchmark suites. The `gcc-expr95` benchmark shows an IPC throughput of 1.54 on the Alpha 21264; `gcc-expr00` shows an IPC throughput of 1.11. This clearly shows that a new release of the same application program (with the same input) can result in significant performance degradation (36% degradation in the case of `gcc`). To understand this behavior, we now compare the abstract workload model for `gcc-expr95` and `gcc-expr00`. Most of the program characteristics are more or less the same across the two `gcc` versions. Even the local stride values (indicative of spatial locality) exhibit a similar distribution. However, the data footprint (indicative of temporal locality) appears to have increased by a factor of 3. Based on this observation, we constructed a synthetic benchmark with the same characteristics as `gcc-expr95` but with three times its data footprint. This benchmark shows an IPC throughput of 1.19 (an error of only 7.2% compared to IPC of `gcc-expr00`).

This result demonstrates that BenchMaker can be a useful tool to generate futuristic workloads in the anticipation of changes in program characteristics, and can help in projecting the impact of workload drift on performance.

6.5.2.2 Analyzing the impact of increase in code size

Previous characterization studies [Phansalkar *et al.,* 2005] have pointed out that although the dynamic instruction count has increased by a factor 100 over the four generations of SPEC CPU benchmark suites, the static instruction count of the programs has not significantly grown. However, in general, the static instruction count of any commercial software application tends to increase with every generation as the application evolves with the advent of new features and functionality. The absence of any benchmarks that stress the instruction cache makes it difficult to analyze the performance impact of an application that could result from code footprints that are substantially larger than available benchmarks. To illustrate the application of BenchMaker to study the impact of potential increase in code size on program performance, we use the *AvgSynBench* benchmark and vary its code footprint. Figure 6.8 shows different flavors of the AvgSynBench benchmark with varying instruction footprints to stress the instruction cache. The graph shows that increases in code size can have a significant impact on performance and must be taken into account if application code size is expected to increase.

As such, we can conclude that in absence of any SPEC CPU Integer benchmarks that stress the instruction cache; this is a plausible approach to project the impact of I-cache misses on the performance of an application

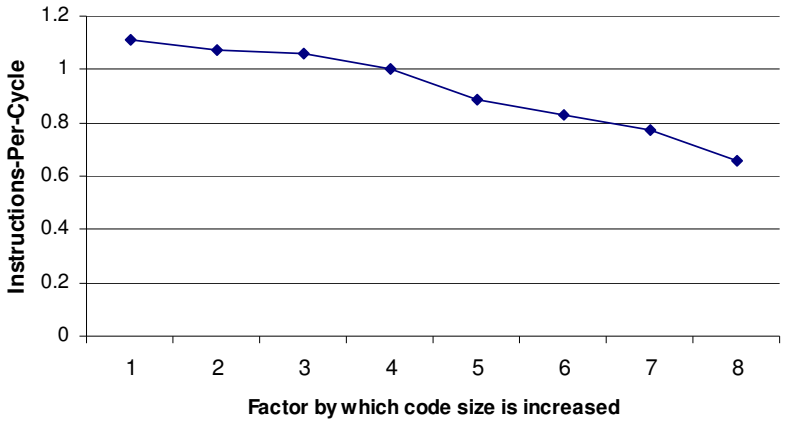


Figure 6.8: Effect of increasing instruction footprint on program performance.

6.6 SUMMARY

The objective of this chapter was to develop a framework that adapts the benchmark cloning strategy to construct scalable synthetic benchmarks. One of the key results from this chapter is that it is possible to fully characterize a workload by only using a limited number of microarchitecture-independent program characteristics, and still maintain good accuracy. Moreover, since these program characteristics are measured at a program level they can be measured more efficiently and are amenable to parameterization. We implement this approach in a framework called BenchMaker and demonstrate various applications that help in studying program characteristics that are typically difficult to vary in standardized benchmarks.

Chapter 7: Power and Temperature Oriented Synthetic Workloads

Estimating the maximum power and thermal characteristics of a microarchitecture is essential for designing the power delivery system, packaging, cooling, and power/thermal management schemes for a microprocessor. Typical benchmark suites used in performance evaluation do not stress the microarchitecture to the limit, and the current practice in industry is to develop artificial benchmarks that are specifically written to generate maximum processor (component) activity. However, manually developing and tuning such synthetic benchmarks is extremely tedious, requires an intimate understanding of the microarchitecture, and is therefore very time-consuming.

In this chapter we apply the parameterized workload model developed in Chapter 6 to propose a framework, StressBench, which can be used to automatically construct stress benchmarks for measuring the maximum power and temperature characteristics of a given microarchitecture design. The framework uses machine learning algorithms to optimize workload characteristics to stress the microarchitecture. This chapter demonstrates that StressBench is very effective in automatically generating stress benchmarks in a limited amount of time.

7.1 THE NEED FOR STRESS BENCHMARKS

In recent years, power, energy, and temperature have emerged as first class constraints in designing microprocessors. At one end of the spectrum, namely in the domain of hand-held and portable devices, battery life and system cost drive the design team to develop power and energy efficient systems. Also, with the rise of mobile computing and pervasive connectivity, devices are becoming smaller and more mobile, making it essential for platforms to consume less energy, reduce the power density, and produce less heat. At the other end of the spectrum, in high performance workstation and

server machines, the complexity of designs, shrinking die sizes, and higher clock speeds, have rapidly increased the packaging and cooling cost of microprocessors. Computer architecture research has demonstrated that it is important for a microprocessor design team to consider power consumption and dissipation limits to adopt a microarchitecture that balances performance, power, and operating temperature constraints [Brooks and Martonosi, 2001] [Skadron *et al.,* 2003-2] [Gunther *et al.,* 2001]. As a result, along with performance, it has become important to measure and analyze the impact of design on power, energy, and temperature at all stages in a microprocessor design flow – from microarchitecture definition, register-transfer-level (RTL) description, to circuit-level implementation.

In order to design a temperature- and power-aware microprocessor it is not only important to characterize the design's power consumption, dissipation, and operating temperature when executing a typical workload, but also to evaluate its maximum power and operating temperature characteristics. Although a microprocessor is generally designed to exhibit optimal power/energy-efficient performance on a typical workload, it is also important to analyze the impact of application code sequences that could stress the microarchitecture's power and thermal characteristics to its limit – although these code sequences are infrequent and may only occur in a short burst [Vishwanathan *et al.,* 2000] [Rajgopal, 2006] [Gowan *et al.,* 1998]. Therefore, having knowledge of the worst case maximum power dissipation and operating temperature is essential for evaluating dynamic power and temperature management strategies, even during the early stages of microarchitecture definition. Also, large instantaneous power dissipation can cause overheating (local hot-spots) that can reduce the lifetime of a chip, degrade circuit performance, or even result in chip failure [Skadron *et al.,* 2003-2]. Having knowledge about the maximum power requirements can therefore also serve as a

guideline for understanding the limits and boundaries of a circuit. Estimating the maximum power dissipation and operating temperature of a microarchitecture is also vital for designing the thermal package (heat sink, cooling, *etc.*) for the chip and the power supply for the system [Vishwanathan *et al.,* 2000]. As such, characterizing the maximum thermal characteristics and power limits is necessary for microarchitects, circuit designers, and electrical engineers responsible for thermal packaging and power delivery system design.

Industry-standard benchmarks that are typically used in performance evaluation of computer systems are representative of workloads that will be executed on the target system and can be used for estimating the typical power consumption and operating temperature of a microprocessor design. However, these benchmarks do not stress the microarchitecture design to its limit and are not particularly useful when characterizing the maximum power and thermal requirements of a design. Standardized benchmarking committees such as the Standard Performance Evaluation Consortium (SPEC) and EDN Embedded Microprocessor Benchmark Consortium (EEMBC) have recognized the need for power and energy oriented benchmarks, and are in the process of developing such benchmark suites [Spec, 2007] [Kanter, 2006]. However, these benchmarks too will only represent the average power consumption and not the worst case maximum power dissipation requirement. Due the lack of any standardized stress benchmarks, current practice in industry is to develop hand-coded synthetic 'max-power' benchmarks that are specifically written to generate maximum processor activity for a particular microarchitecture [Vishwanathan *et al.,* 2000] [Rajgopal, 1996] [Gowan *et al.,* 1998] [Bhattacharya and Williamson, 2007].

Developing synthetic benchmarks for characterizing maximum power consumption is non-trivial because the instruction sequence has to simultaneously

generate maximum processor activity. This requires a very detailed knowledge of the microarchitecture design [Gowan *et al.,* 1998] and, given the complexity of modern day out-of-order superscalar microprocessors, writing and tuning different flavors of such benchmarks for different microarchitectures can take up to several weeks– impacting the time-to-market [Bhattacharya and Williamson, 2007]. Furthermore, manually developing a similar benchmark for stressing thermal characteristics would be even more difficult, time consuming, and error prone. This is primarily because the operating temperature is not only dependent on power, but also on lateral coupling among microarchitecture blocks, role of the heat sink, *etc.* [Skadron *et al.,* 2003-2], and hence tedious to vary by manually writing a synthetic sequence of instructions.

In this chapter we address the problem of developing stress benchmarks by proposing a framework, *StressBench*, which automates the process of generating stress benchmarks for measuring the maximum power and thermal characteristics of a microarchitecture. StressBench synthesizes a benchmark from a specified set of parameterized workload characteristics (proposed in Chapter 6), and uses machine learning algorithms to explore attribute values for the workload characteristics that stress the microarchitecture. The use of StressBench to generate stress benchmarks has four key advantages over hand-coded synthetic stress benchmarks: (1) StressBench significantly reduces the time required to develop a stress benchmark and therefore enables developing a wider spectrum of benchmarks to stress various aspects of the microarchitecture at earlier stages in the design cycle, (2) StressBench uses automatic design space exploration algorithms that prevent getting stuck in a local minimum (which is very likely in a hand-coded test), explores a wider workload space, and increases the confidence that the generated stress test indeed characterizes the maximum power or operating temperature of a design, (3) StressBench makes it possible to develop stress

benchmarks for cases where manually writing a test case is not feasible because the interaction of program characteristics with the parameter to be stressed (*e.g.*, hot spots) is not very well understood, and (4) StressBench generates stress tests from a list of inherent program characteristics and therefore provides insight into the combination of workload characteristics that result in worst case power dissipation and thermal characteristics.

7.2 STRESS BENCHMARK GENERATION APPROACH

The flow chart in Figure 7.1 illustrates the approach used by StressBench to generate benchmarks for stressing a particular microarchitecture design. StressBench iterates over four steps: (1) Workload Synthesis, (2) Simulation, (3) Evaluating quality of benchmark, and (4) Workload Space Exploration.

In the first step, Workload Synthesis, a benchmark is synthesized from a parameterized set of fundamental program characteristics. These workload characteristics can be considered as a signature that uniquely describes a stress benchmark. The parameterized nature of these workload characteristics makes it possible to alter workload characteristics to vary the stress that a benchmark places on the microarchitecture. In the second step, the stress benchmark is simulated on the microprocessor model and the value of the parameter to be stressed, power, energy, or temperature, is measured. As mentioned earlier, a microprocessor design team may want to estimate the maximum power and temperature of a microarchitecture right from the early design stage exploration to the final circuit level implementation. The model used for simulation can thus be a high-level performance model, an RTL-level Verilog model, or a circuit-level implementation. In the third step, a decision to continue or stop is made based on the stress level placed by the benchmark and the simulation budget. In the fourth step, Workload Space Exploration, design space exploration or machine learning

algorithms are used to alter the workload characteristics and improve the quality of the stress benchmark. This iterative process continues till the design space exploration algorithm converges or a maximum exploration time is reached determined by time constraints.

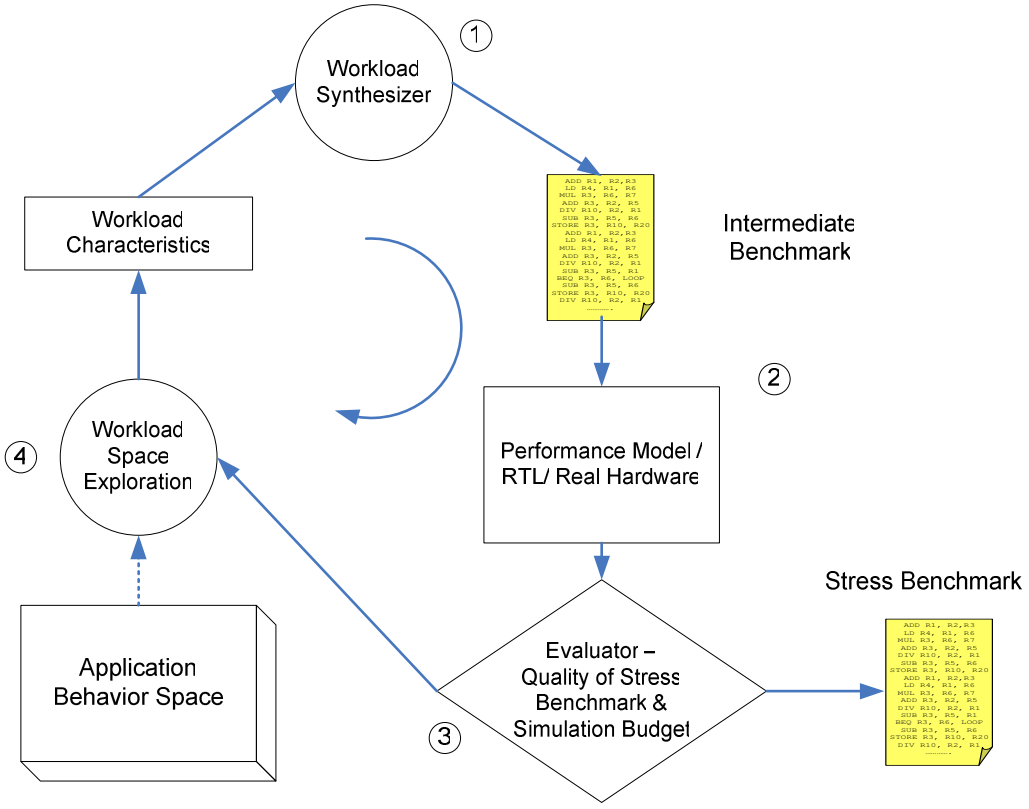


Figure 7.1: Automatic stress benchmark synthesis flow.

In the following section we describe the machine learning algorithms that we used for automatically searching the application behavior space for finding the characteristics to stress the microarchitecture.

7.3 AUTOMATIC EXPLORATION OF WORKLOAD ATTRIBUTES

The design space comprising of the workload characteristics described in the previous section is extremely large and it is impossible to evaluate every design point. Therefore, we use automated design space exploration algorithms to efficiently search and prune the workload space to converge on a set of workload attributes to maximize an objective function (*e.g.*, thermal stress placed on the microarchitecture). These design space exploration algorithms are described below – we will evaluate their efficacy in the evaluation section:

Random Descent (RD) randomly selects one workload characteristic that is randomly incremented or decremented. The change is only accepted if the objective function improves. The algorithm iterates till the objective function no longer improves.

Steepest Descent (SD) is similar to the random descent algorithm, but instead of randomly selecting a dimension it selects the dimension along which the objective function improves the most. All neighboring design points thus need to be evaluated in each iteration of the algorithm, which likely makes steepest descent slower than random descent.

One Parameter at a Time (OP) algorithm successively optimizes each dimension in a fixed order. Once all dimensions have been optimized, the entire process is repeated till the objective function no longer improves.

Tabu Search (TS) algorithm is similar to SD except that it always goes in the steepest descent direction irrespective of whether the objective function improves or not – this is to avoid getting stuck in a local minimum. A small history of recently visited design points (called the *tabu* list) is maintained and these design points are not accepted to prevent circulating around a local minimum.

Genetic Search (GS) initially randomly selects a set of design points, called a generation, which are subsequently evaluated according to the objective function, also called the fitness function. To form a new population, an *offspring*, which is a subset of these design points, is probabilistically selected by weighting their *fitness function*, *i.e.,* a fitter function is more likely to be selected. Selection alone cannot introduce new design points in the search space, therefore mutation and crossover is performed to build the offspring generation. *Crossover* is performed, with probability pcross, by randomly exchanging parts of two selected design points from the current generation. The *mutation* operator prevents premature convergence to local optima by randomly altering parts of a design point, with a small probability pmut. The generational process is continued until a specified termination condition has been reached. In our experiments we specify the termination condition as the point when there is little or no improvement in the objective function across successive generations. We use the genetic search algorithm with pcross and pmut set to 0.95 and 0.02, respectively.

7.4 EXPERIMENTAL SETUP

7.4.1 Simulation Infrastructure

For our StressBench experiments we use the `sim-outorder` simulator from the `SimpleScalar Toolset v3.0`. In order to estimate the power characteristics of the benchmarks we use an architectural power modeling tool, namely `Wattch v1.02` [Brooks and Martonosi, 2000] which was shown to provide good relative accuracy. In `Wattch` we consider an aggressive clock gating mechanism (`cc3`). For measuring the thermal characteristics we use the `HotSpot v3.1` infrastructure [Skadron *et al.,* 2003-2]. We use the `hotfloorplanner` tool [Skadron *et al.,* 2003-2] to develop a layout for the `sim-outorder` pipeline and use the

HotSpot tool to estimate the steady-state operating temperature based on the average power. The synthesized stress tests are compiled using gcc on an Alpha machine and are simulated for 10 million dynamic instructions. This small dynamic instruction count serves the needs in this evaluation; however, in case longer-running applications need to be considered, *e.g.*, when studying the effect of temperature on (leakage) power consumption, the stressmarks can also be executed in a loop for a longer time. It should also be noted that StressBench is agnostic to the underlying simulation model, and can be easily ported to a more accurate industry-standard simulators and/or power/temperature models.

7.4.2 Benchmarks

In order to evaluate the parameterized workload synthesis framework, we consider all SPEC CPU2000 benchmarks and select one representative 100M-instruction simulation point selected using SimPoint [Sherwood *et al.,* 2002]. We also use traces from three commercial workloads – SPECjbb2005 (representative of Java server workloads), DBT2 (representative of an OLTP workload), and DBMS (a database management system workload). The commercial workload traces represent 30 million instructions once steady-state has been reached (all warehouses have been loaded), and were generated using the SIMICS full-system simulator.

7.4.3 Stress Benchmark Design Space

The workload characteristics form a multi-dimensional space (instruction mix, ILP, branch predictability, instruction footprint, data footprint, and data strides). We bound the stressmark design space by discretizing and restricting the values along each dimension, see Table 7.1. This discretization does not affect the generality of the proposed methodology – its purpose is to keep the evaluation in this chapter tractable.

The total design space comprises of 250K points. We will evaluate the efficacy of the genetic search algorithm used in StressMaker against an exhaustive search in this 250K design space.

Table 7.1: Stress benchmark design space.

Dimension	**Num. Points**	**Values/Ranges**
instruction mix and basic block size	10	Combinations where integer, floating-point, load, store, and branch instructions are set to low (10%), moderate (40%), and high (80%)
instruction-level-parallelism	10	Varying from all instructions with virtually no dependencies (dependency distance > 64 instructions) to all instructions are dependent on the prior instruction (dependency distance of 1)
data footprint	5	50K, 100K, 500K, 2M, and 5M unique data addresses
local stride distribution	10	Varying from 100% references with stride 0, up to 10% with stride 0 and 90% with stride 10.
instruction footprint	5	600, 1800, 6000, and 20000 unique instructions
branch predictability	10	Varying from 100% branches with transition rate below 10% to equal distribution of transition rate across all 10 transition rate categories (0-10%, 10-20%, *etc.*)

7.4.4 Microarchitecture Configurations

Table 7.2 summarizes the three different microarchitecture configurations, ranging from a modest 2-way configuration representative of an embedded microprocessor, to a very aggressive 8-way issue high performance microprocessor. We used `Config 2` as the base configuration for our experiments.

Table 7.2: Microarchitecture configurations evaluated.

	Config 1	Config 2	Config 3
L1 I-cache & D-cache Size/Assoc/Latency	16 KB/2-way/32 B	32 KB / 4-way / 1 cycle	64 KB / 4-way / 1 cycle
Fetch, Decode, and Issue Width	2-wide out-of-order	4-wide out-of-order	8-wide out-of-order
Branch Predictor	2-level	Combined (2-level & bimodal), 4KB	Combined (2-level & bimodal), 4KB
L2 Unified cache – Size/Assoc/Latency	256KB / 4-way / 10 cycles	4MB / 8-way / 10 cycles	4MB / 8-way / 10 cycles
RUU / LSQ size	16 / 8 entries	128 / 64 entries	256 / 128 entries
Instruction Fetch Queue	8 entries	32 entries	64 entries
Functional Units	2 Integer ALU, 1 FP Unit	4 Integer ALU, 2 Floating Point, 1 FP Multiply/Divide, and 2 Integer Multiply/Divide unit	8 Integer ALU, 2 Integer Multiply/Divide, 2 Floating Point, 2 FP Multiply/Divide units
Memory Bus Width, Access Time	8B, 40 cycles	8B, 150 cycles	8B, 150 cycles

7.5 EVALUATION OF STRESSBENCH FRAMEWORK

We now evaluate the application and usefulness of StressBench by applying the methodology to generate various flavors of power and thermal stress benchmarks. Specifically, we apply StressBench to automatically construct benchmarks for characterizing the maximum power of a microprocessor, creating thermal hotspots, and thermal stress patterns. We also compare the characteristics of the stress benchmarks across microarchitectures and evaluate the efficacy of various stress benchmark design space exploration algorithms.

7.5.1 Maximum Sustainable Power

The maximum sustainable power is the maximum average power that can be sustained indefinitely over many clock cycles. Estimating the maximum sustainable power is important for the design of the power delivery system and also the packaging

requirements for the microprocessor. We applied the StressBench methodology to construct a stress benchmark for characterizing the maximum sustainable power of the 4-way issue microarchitecture (`Config 2`) outlined in Table 7.2. Figure 7.2 shows a plot of the value of the best *fitness function* (maximum power consumption) in each generation during the iterative process of stress benchmark synthesis using the genetic algorithm. We terminate the search after 15 generations, requiring a total of 225 simulations. The number of generations required before the fitness function can be accepted is dependent on the search space and the microarchitecture. However, our experiments on three very different microarchitectures, outlined in Table 7.2, suggest that there is little improvement beyond 15 generations and therefore for our experiments we terminate the search after 15 generations.

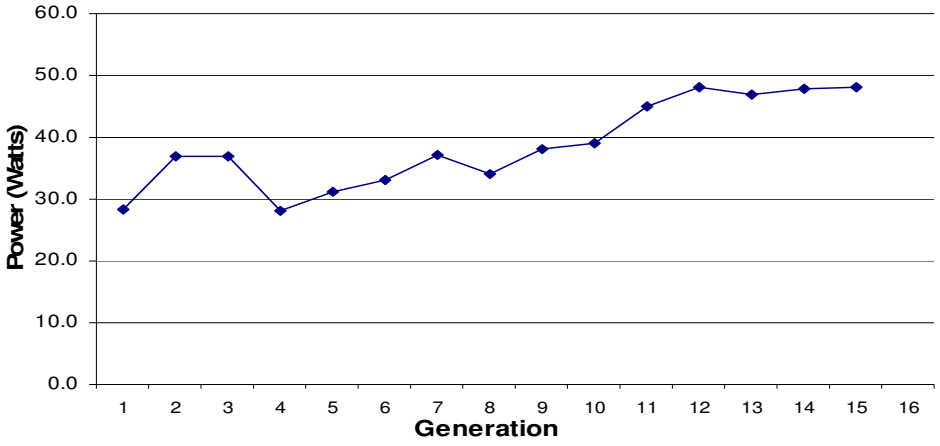


Figure 7.2: Convergence characteristics of StressBench.

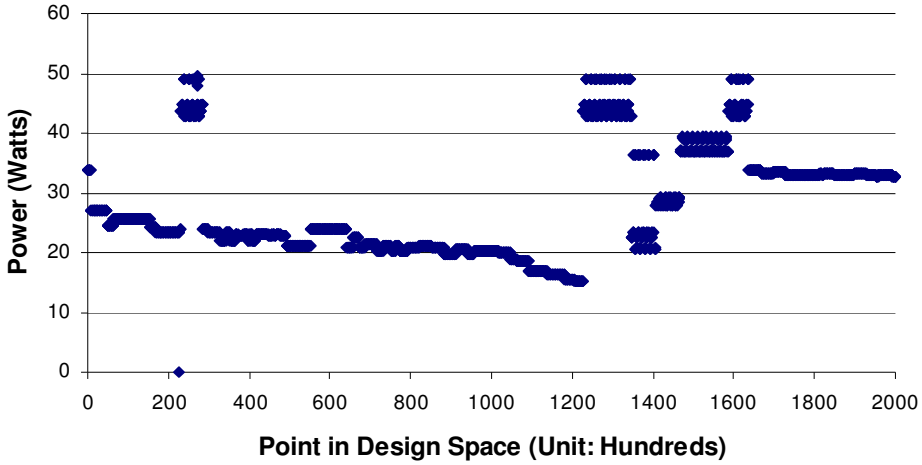


Figure 7.3: Scatter plot showing distribution of power consumption across 250K points in the design space.

This 'maximum sustainable power' search process results in a stressmark that has a maximum average sustainable power-per-cycle of 48.8 W. Figure 7.3 shows the results of an exhaustive search across all the 250K design points. A comparison of these results shows that the power of the stressmark is within 1% of the maximum power of the design point. This suggests that the StressMaker approach is highly effective in finding a stressmark, and also results in a three orders of magnitude speedup compared to an exhaustive search. Automatically generating the stressmark on a 2GHz Pentium Xeon processor using a cross compiler for Alpha and sim-outorder performance model, typically takes 2.5 hours. Therefore, we believe StressMaker is an invaluable approach for an expert, because it can quickly narrow down a design space, and provide a stressmark that can be hand tuned to exercise worst-case behavior.

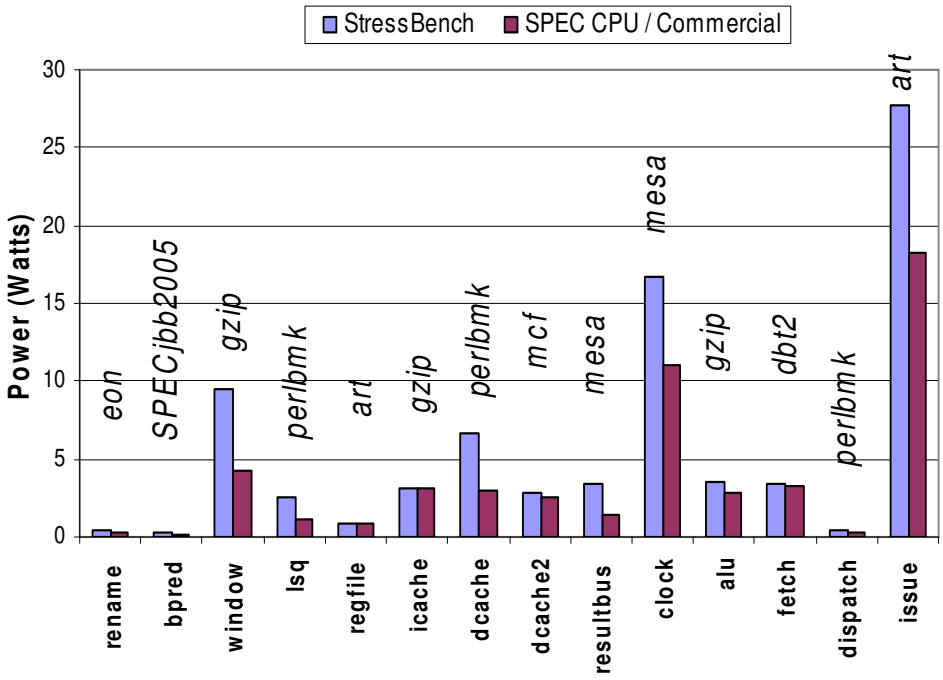


Figure 7.4**:** Comparison of power dissipation of different microarchitecture units using stress benchmark with the maximum power consumption across SPEC CPU2000.

Figure 7.4 shows the maximum power dissipation of different microarchitecture units using the stress benchmark, along with the maximum power dissipation of that unit across *all* SPEC CPU2000 integer and floating-point benchmarks, and commercial workloads. The stressmark exercises all the microarchitecture units more than any of the SPEC CPU benchmarks. It is interesting that other than branch predictor and the fetch unit, SPEC CPU2000 benchmarks are more effective in stressing the microarchitecture units than commercial workloads. Especially, the stressmark causes significantly higher power dissipation in the instruction window, L1 data cache, clock tree, and the issue logic.

The workload characteristics of this stress benchmark are – (1) Instruction mix of 40% short latency floating point operations, 40% short latency integer operations, 10%

branch instructions, and 10% memory operations (2) Register dependency distance of greater than 64 instructions (*i.e.*, very high level of ILP), (3) 80% of branches having a transition rate of less than 10% and the remaining 20% branches have a transition rate between 10-20% (recall that branches with very low transition rates are highly predictable), (4) Data strides having 95% of the references to the same cache line and 5% with references to the next cache line, (5) Instruction footprint of 1800 instructions, and (6) Data Footprint of 100K bytes.

These workload characteristics suggest that the stress benchmark creates a scenario where the control flow of the program is highly predictable and hence there are no pipeline flushes, the floating-point units are kept busy due to a large percentage of floating point operations, the issue logic does not stall due to large dependency distances, and the locality of the program is such that the data and instruction cache hit rates are extremely high. The characteristics of this stress benchmark are similar to the hand-crafted tests [Gowan *et al.,* 1998] [Bhattacharya and Williamson, 200] that are tuned to maximize processor activity by fully and continuously utilizing the instruction issue logic, all of the execution units, and the major buses. However, the advantage over current practice in building hand-coded max-power benchmarks is that StressBench provides an automatic process and does not require an intimate understanding of the microarchitecture, resulting in substantial savings in time and effort. Also, the search through a large design space increases confidence in the results.

7.5.2 Maximum Single-Cycle Power

Maximum single-cycle power is defined as the maximum total power consumed during one clock cycle, and is important to estimate the maximum instantaneous current that can be drawn from the power supply. This characterization is

also important to understand current variability, referred to as the dI/dt problem, which the power supply and voltage regulation systems should be able to handle.

We apply the StressBench framework to automatically construct a stress benchmark that maximizes single-cycle power. The search process results in a benchmark that has a maximum single-cycle power dissipation of 72W. The workload characteristics of this benchmark are (1) Instruction mix of 40% long latency operations, 20% branches, and 40% memory operations, (2) Register dependency distance of greater than 64 instructions (*i.e.*, very high level of ILP), (3) Equal distribution of branch transition rate across all the 10 categories, (4) 10% of the data references have a local stride of 0, 10% a stride of 1, and 80% have a stride of 3 cache lines, (5) Instruction footprint of 1800 instructions, and (6) Data footprint of 5M unique address. These characteristics suggest that the benchmark does not yield the best performance due to a mix of easy and difficult to predict branches (evenly distributed transition rates), possible issue stalls (large percentage of long latency operations), and data cache misses (large footprint and strides). Therefore, it is not surprising that the average power consumption of this benchmark is only 32W. However, the overlapping of various events creates a condition where all units are simultaneously busy.

Interestingly, the stress benchmark that maximizes the average sustainable power (section 7.5.1) only has a maximum single-cycle power of 59.5W, and cannot be used to estimate maximum single-cycle power. Also, the maximum single-cycle power requirement of a SPEC CPU benchmark, `mgrid`, is only 57W. This demonstrates that the sequence of instructions resulting in maximum single-cycle power is very timing sensitive (even benchmarks that run for billions of cycles may not probabilistically hit upon this condition) and is therefore extremely difficult to manually construct.

Prior work [Joseph *et al.,* 2003] has expressed the need for constructing a 'dI/dt stressmark', and argues that manually developing such a benchmark is extremely difficult due to knowledge required about the power, packaging, and timing characteristics of the targeted processor. In order to study the applicability of StressBench to automatically develop such a 'dI/dt stressmark', we used the framework to generate two sequences of 200 instructions – one for maximizing single-cycle power and the other for minimizing single-cycle power. We then concatenated these two sequences of instructions and evaluated its power characteristics. Our experiments show that the power consumption in the benchmark shows a cyclic behavior at a period of 400 instructions - with 72W and 16W as the maximum and minimum single-cycle power consumption. Also, it is possible to change the frequency of the power oscillations by varying the number of instructions of the individual (maximum and single-cycle power) stress tests. These experiments show that it is indeed possible to automatically generate a 'dI/dt stressmark', which is typically very difficult to hand-craft and tune.

7.5.3 Comparing Stress Benchmarks Across Microarchitectures

We generate stress benchmarks for three different microarchitectures described in Table 3 and analyze whether the stress benchmarks are similar or different across microarchitectures. The stress benchmarks generated for `Config1`, `Config2`, and `Config3` are called `StressBench1`, `StressBench2`, and `StressBench3`, respectively. We then execute the 3 stress benchmarks on all the three configurations. Figure 7.5 shows the average power consumption of each of the benchmarks across all the 3 configurations. We observe that the stress benchmark synthesized for each microarchitecture configuration always results in maximum power consumption compared to the other two stress benchmarks, *i.e.*, a stress benchmark generated for one microarchitecture does not result in maximum power for another

microarchitecture. In fact, a stress benchmark developed for one microarchitecture can result in extremely low power consumption on another microarchitecture, *e.g.*, `StressBench1` on `Config3`.

The three stress benchmarks are similar in that they have highly predictable branches, small instruction and data footprints, and very large register dependencies. However, their instruction mixes of computational operations are very different – `StressBench1` comprises of 80% short latency integer operations, `StressBench2` comprises of 40% short latency floating point operations, 40% short latency integer operations, and `StressBench3` has 40% short latency and 40% long latency floating-point operations. This is intuitive because, in order to minimize any structural hazards, the instruction mix of the stress benchmark will depend on the number of functional units.

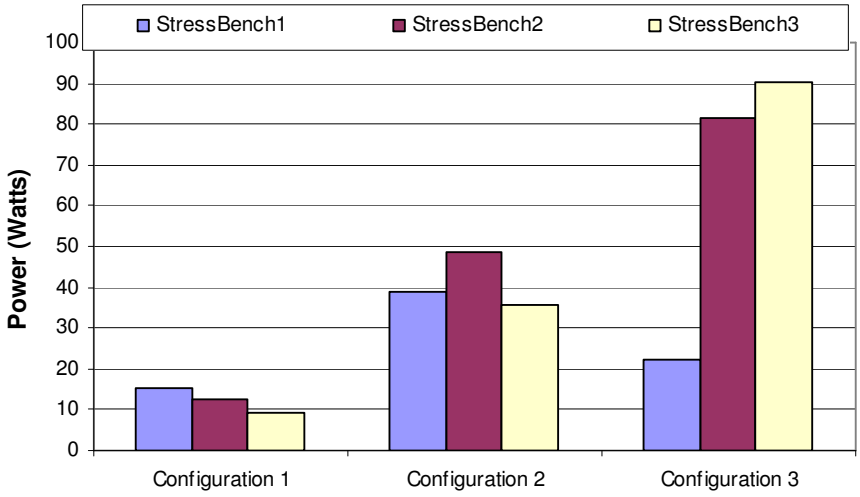


Figure 7.5: Comparison of stress benchmarks across three very different microarchitectures.

We conclude that the characteristics of benchmarks that cause maximum power dissipation vary across microarchitecture designs. Therefore, separate custom

stress benchmarks have to be constructed for different microarchitectures. This further motivates the importance of having an automated framework to generate stress benchmarks.

7.5.4 Creating Thermal Hotspots

Applications can cause localized heating of specific units of a microarchitecture design, called hotspots, which can cause permanent chip damage. Therefore, to study the impact of hotspots in different microarchitecture units it is important to design benchmarks that can be used to vary the location of a hotspot [Skadron *et al.,* 2003-2]. We apply StressBench to generate benchmarks that can create hotspots across different microarchitecture units on the floorplan.

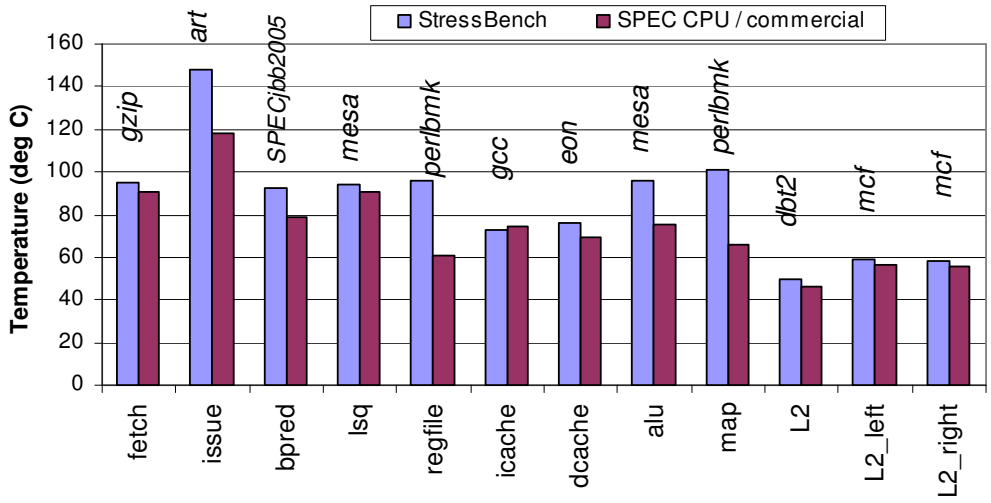


Figure 7.6: Comparison of hotspots generated by stress benchmarks and SPEC CPU2000

Figure 7.6 compares hotspots generated by StressBench with the hotspots generated by SPEC CPU2000 benchmarks. As compared to the SPEC CPU2000 benchmarks, the stress benchmarks are especially very effective in creating hotspots in the issue, register file, execution, and register remap units. The stress benchmarks can be effectively used for studying the effect of hotspots in different microarchitecture units.

7.5.5 Thermal Stress Patterns

In order to support dynamic thermal management schemes it has become important to place on-chip sensors to monitor temperature at different locations on the chip. Conceptually, there can be applications that only stress a particular unit that is far from a sensor, causing hotspots that may not be visible to the distant sensor causing permanent damage to the chip [Lee *et al.,* 2005] [Gunther *et al.,* 2001]. Typically, only a few sensors can be placed on a chip. Therefore, the placement of sensors needs to be optimized based on the maximum thermal gradient that can exist between different units on the chip. Hand-crafted tests have been typically used to develop such gradients [Lee *et al.,* 2005]. StressBench seems to be a natural way to optimize a complex objective function such as the temperature gradient between two microarchitecture units. We selected a set of microarchitecture units and generated stress benchmarks to maximize the temperature difference between units that are not adjacent to each other. Table 4 shows the pair of units, maximum temperature gradient created by the automatically generated stress benchmark, and the key stress benchmark characteristics.

Table 7.3: Developing thermal stress patterns using StressBench

Pair of Units	Temperature Differential (ºC)	Key characteristics of the stressmarks that are automatically synthesized by StressMaker
L2 & Instruction Fetch	44.6	(1) Small data footprint and short local strides that result in high L1 d-cache hit-rates with almost no L2 activity, and (2) 80% short latency operations with large dependences and highly predictable branches – keeping fetch busy without any pipeline stalls.
L2 & Register Remap	48.4	(1) 40% memory operations, large data footprint, and long local strides that result in a large percentage of L1 cache misses and stress L2, and (2) 40% short latency memory operations with very large dependency distances that put minimal stress on the register remap
Instruction Cache & Issue	60.1	(1) 40% short latency integer operations, 40% short latency floating-point operations with very large dependency distances – preventing any structural hazards due to dependencies and hence stressing the issue unit.
L2 & Execution	44.4	(1) No memory operations, so no stress on L2, and (2) 40% short latency integer operations and 40% short latency floating-point operations that stress the execution unit.
Branch	41.3	(1) 80% branches with transition rate equally distributed between all

Predictor & L2		buckets (0-10% … 90-100%) – a mix of difficult and easy to predict branches that stress the branch predictor, and (2) No memory operations resulting almost on L2 activity.
Issue & LSQ	61.0	(1) 80% memory operations with small data footprint and short local strides that result in high L1 d-cache activity and hence stress the load store.

7.5.6 Quality and Time Complexity of Search Algorithms

So far we have used the genetic search algorithm to explore the workload behavior space. In this section we compare the quality and time complexity of various search algorithms described earlier in the chapter. We implemented each of these algorithms in the StressBench framework and used them to generate a stress benchmark that causes maximum power dissipation for the 3 configurations described in Table 7.2. Figures 7.7 and 7.8 respectively show the time complexity (number of simulations) and quality (maximum sustainable power of stress benchmark found) of the search algorithms.

The genetic algorithm requires the maximum number of simulations, but always yields the best solution. On the other hand the random descent algorithm requires the least number of simulations, but always yields a suboptimal solution. The reason is that the performance of the random descent algorithm is very sensitive to the starting point and it can easily get stuck in local maxima. Automatically generating benchmark using StressBench on a 2GHz Pentium Xeon processor using a cross compiler for Alpha and `sim-outorder` performance model, typically takes 2.5 hours.

The other algorithms (tabu search, steepest descent, and one parameter at time) are able to construct good stress benchmarks for some configurations, but do not generate the best solution on the other configurations. Overall, we conclude that the genetic algorithm is most effective in finding a stress benchmark, albeit at the cost of a larger number of simulations.

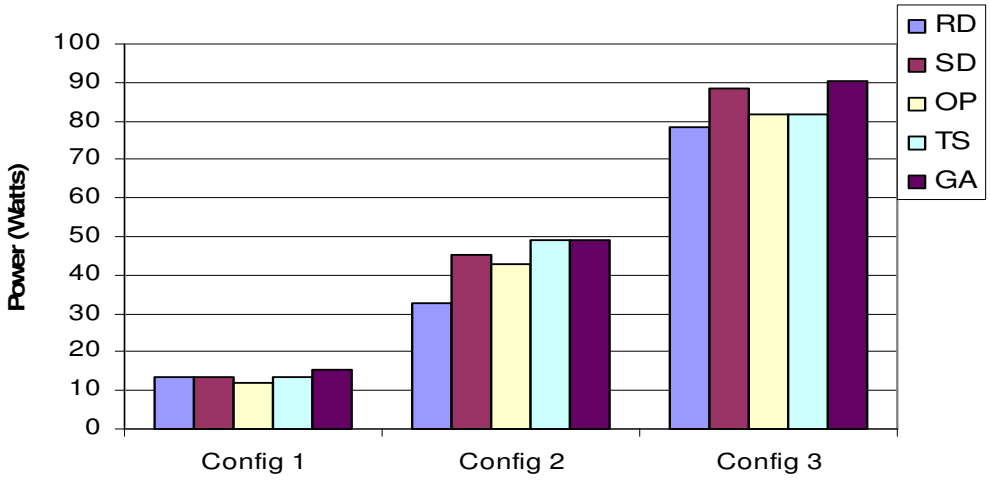


Figure 7.7: Number of simulations required for different search algorithms.

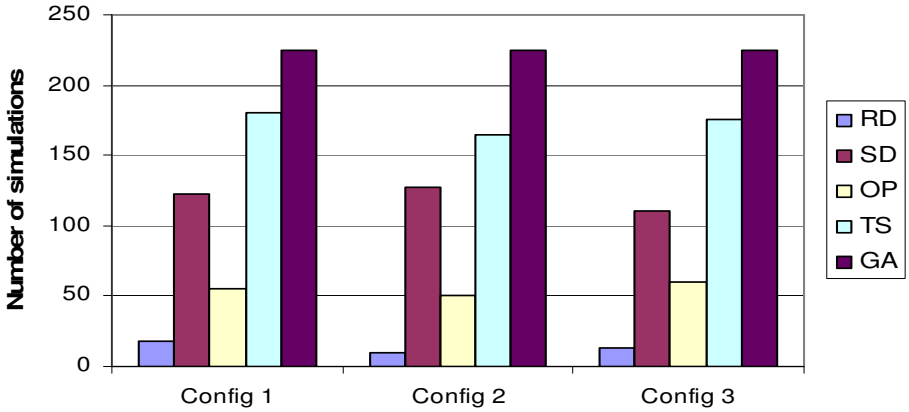


Figure 7.8: Comparison of quality of stress benchmark for maximum sustainable power constructed using different search algorithms.

7.6 SUMMARY

Characterizing the maximum power dissipation and thermal characteristics of a microarchitecture is an important problem in industry. Typically, hand-coded synthetic streams of instructions have been used to generate maximum activity in a processor to estimate the maximum power dissipation. However, due to the increase in complexity of

microprocessors, and the need to construct synthetic test cases to vary complex parameters such as thermal characteristics, it is extremely tedious to manually develop and tune stress benchmarks for different microarchitectures.

In this chapter we presented StressBench, a framework that synthesizes a stress benchmark from fundamental program characteristics using machine learning algorithms to tune the program characteristics to stress the microarchitecture under study. We showed that StressBench is very effective in constructing stress benchmarks for measuring maximum average power dissipation, maximum single-cycle power dissipation, and temperature hot spots. The automated approach to stress benchmark synthesis can eliminate the time- consuming complex task of hand-coding a stress benchmark, and also increase the confidence in the quality of the stress benchmark.

Chapter 8: Conclusions and Directions for Future Research

8.1 CONCLUSIONS

Over the last two decades the advent of standardized application benchmark suites such as SPEC, TPC, EEMBC *etc.* have streamlined the process of computer performance evaluation and benchmarking. The use of benchmarks performance evaluation has now become the *de jure* standard in academic research and industry product development. However, the increase in complexity of computer systems and the size of application benchmarks has resulted in prohibitive simulation times. Moreover, due to proliferation in the diversity of software application domains, it is becoming increasingly difficult develop, maintain, and upgrade standardized benchmark suites.

Consequently, there has been a revival of interest in the computer architecture community to develop statistical workload models to generate synthetic traces and benchmarks. The primary motivation for prior research was to reduce the simulation to a tractable amount of time by automatically generating representative synthetic traces and benchmarks. The objective of this dissertation work was to advance the state-of-the-art in benchmark synthesis by increasing the confidence in the use of synthetic workloads, increase the representativeness of synthetic workload across different microarchitectures, develop the ability to model emerging applications and futuristic workloads, and develop temperature and power oriented workloads. Overall, this dissertation improves the application of synthetic benchmarks beyond reduction in simulation time.

The following are the major findings and contributions of this dissertation work to the area of performance evaluation and benchmarking:

- **Efficacy of Statistical Workload Modeling for Design Space Exploration**

We apply the P&B statistical design of experiments technique to evaluate the ability of statistical workload modeling for early design space studies. P&B provides a systematic way to evaluate the accuracy and representativeness of statistical workload modeling by exposing different processor bottlenecks. We draw three key inferences from this study:

1) At the very least, synthetic traces stress the same 10 most significant processor performance bottlenecks as the original workload. Since the primary goal of early design space studies is to identify the most significant performance bottlenecks, we conclude that statistical simulation is indeed a very useful tool.
2) Statistical simulation has good relative accuracy and can effectively track design changes to identify feasible design points in a large design space of aggressive microarchitectures.
3) Our evaluation of four statistical simulation models shows that although a very detailed model is needed to achieve a good absolute accuracy in performance estimation, a simple model is sufficient to achieve good relative accuracy. This is very attractive early in the design cycle when time and resources for developing the simulation infrastructure are limited.

- **Microarchitecture-Independent Workload Modeling**

We show that it is possible to completely characterize the performance of an application using microarchitecture-independent workload attributes. We develop a set of workload characteristics that can be used to capture the data locality and control flow predictability of a program using microarchitecture-independent characteristics. These set of characteristics can be considered as a signature that uniquely characterize the

performance of an application. We characterize a set of embedded, general-purpose, and scientific benchmarks using these characteristics.

- **Distilling the Essence of Proprietary Applications into Miniature Synthetic Benchmarks**

We explored a workload synthesis technique that can be used to clone a real-world proprietary application into a synthetic benchmark clone that can be made available to architects and designers. The synthetic benchmark clone has similar performance/power characteristics as the original application but generates a very different stream of dynamically executed instructions. By consequence, the synthetic clone does not compromise on the proprietary nature of the application. In order to develop a synthetic clone using pure microarchitecture-independent workload characteristics, we develop memory access and branching models to capture the inherent data locality and control flow predictability of the program into the synthetic benchmark clone. We developed synthetic benchmark clones for a set of benchmarks from the SPEC CPU2000 integer and floating-point, MiBench and MediaBench benchmark suites, and showed that the synthetic benchmark clones exhibit good accuracy in tracking design changes. Also, the synthetic benchmark clone runs orders of magnitude faster than the original benchmark and significantly reduces simulation time on cycle-accurate performance models.

The technique proposed in this paper will benefit architects and designers to gain access to real-world applications, in the form of synthetic benchmark clones, when making design decisions. Moreover, the synthetic benchmark clones will help the vendors to make informed purchase decisions, because they would have the ability to benchmark a processor using a proxy of their application of interest.

- **Adapting the Benchmark Generation Approach to Synthesize Scalable Benchmarks**

A key result from this dissertation is that it is possible to fully characterize a workload by only using a limited number of microarchitecture-independent program characteristics, and still maintain good accuracy. Moreover, since these program characteristics are measured at a program level they can be measured more efficiently and are amenable to parameterization. This makes it possible to adapt the benchmark generation strategy to develop a parameterized workload model that can synthesize scalable benchmarks. We demonstrate various applications of this technique that help in studying program characteristics that are typically difficult to vary in standardized benchmarks. Also, the ability to parameterize workloads makes it possible to extract key workload characteristics from commercial applications, which are typically very difficult to setup in a simulation environment, and model them in a synthetic benchmark. This makes it possible to use commercial workloads in simulation based research and early design space performance and power studies.

- **Power and Temperature Oriented Characterization Benchmarks**

We a novel technique, StressBench, that can be used to synthesize a stress benchmark from fundamental program characteristics. The StressBench approach uses machine learning algorithms to tune the program characteristics to stress the microarchitecture under study. We showed that StressBench is very effective in constructing stress benchmarks for measuring maximum average power dissipation, maximum single-cycle power dissipation, and temperature hot spots. The automated approach to stress benchmark synthesis can eliminate the time- consuming complex task of hand-coding a stress benchmark, and also increase the confidence in the quality of the stress benchmark.

8.2 DIRECTIONS FOR FUTURE RESEARCH

- **Multithreaded and Multi-core Workload Synthesis**

The improvement of single-thread performance with power and complexity of microarchitectures as first class constraints, is reaching a point of diminishing returns. The future trend in computer architecture is towards exploiting the coarse-grain parallelism and concurrency in workloads. To take full advantage of this there is a consensus towards developing multi-core and multithreaded microprocessors. The design, evaluation, and optimization of multi-core and multithreaded architectures poses a daunting challenge to architectures and researchers. At this time representative benchmarks oriented towards performance evaluation of multi-core architectures are lacking at this time. Developing and standardizing such benchmarks is a non-trivial task, and are much more complex to develop in comparison to single-threaded benchmarks. Synthetic benchmarks would be useful tool to model the behavior of multithreaded workloads. An interesting and challenging direction for future work would be to develop approaches to characterize and identify the key performance attributes of multithreaded workloads and develop approaches to model them into representative synthetic benchmarks.

- **Portability of Synthetic Benchmarks Across Architectures**

The benchmark synthesis approach in this dissertation uses instruction set architecture (ISA) specific instructions embedded in C-code. Therefore, the only way to port the synthetic benchmark across different architectures would be to generate a separate benchmark for each ISA. Typically, every microprocessor designer would be interested only in one particular architecture and therefore this may not be a severe

problem in practice. However, if the synthetic benchmark clone is to be made truly portable across ISAs, it would be important to address this concern. One possibility to address this challenge would be to generate the synthetic benchmark clone in a virtual instruction set architecture or an intermediate compiler format that can be consumed by compilers for different ISAs. Another possibility would be binary translating the synthetic benchmark clone binary to the ISA of interest. The key challenge in these approaches is to be able to retain the inherent program characteristics of the synthetic benchmark across different architectures. Investigating these approaches would be an interesting direction for future research work.

Bibliography

[Barford and Crovella, 1998] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1998, pp. 151-160.

[Bell *et al.,* 2004] R. Bell Jr., L. Eeckhout, and L. John. Deconstructing and Improving Statistical Simulation in HLS. *Workshop on Deconstructing, Duplicating, and Debunking,* 2004.

[Bell and John, 2005-1] R. Bell Jr. and L. John. Efficient Power Analysis using Synthetic Testcases. *Proceedings of International Symposium on Workload Characterization,* 2005, pp. 110-118

[Bell and John, 2005-2] R. Bell Jr. and L. John. Improved Automatic Test Case Synthesis for Performance Model Validation. *Proceedings of International Conference on Supercomputing*, 2005, pp. 111-120.

[Bell and John, 2005-3] R. Bell Jr. and L. John. The Case for Automatic Synthesis of Miniature Benchmarks. *Proceedings of Workshop on Modeling of Benchmarks and Systems*, 2005.

[Bell *et al.,* 2006] R. Bell Jr., R. Bhatia, L. John, J. Stuecheli, R. Thai, J. Griswell, P. Tu, L. Capps, and A. Blanchard. Automatic Testcase Synthesis and Performance Model Validation for High-Performance PowerPC Processors. *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2006, pp. 154-165.

[Bodnarchuk and Bunt, 1991] R. Bodnarchuk and R. Bunt. A synthetic workload model for a distributed system file server. *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* 1991, pp. 50-59.

[Bose, 1998] P. Bose. Performance Test Case Generation for Microprocessor. *Proceedings of the IEEE VLSI Test Symposium Tutorial,* 1998.

[Bose and Abraham, 2000] P. Bose and J. Abraham. Performance and Functional Verification of Microprocessors. *Proceedings of IEEE VLSI Design Conference*, 2000, pp. 58-93.

[Brooks and Martonosi, 2001] D. Brooks and M. Martonosi. Dynamic Thermal Management for High-Performance Microprocessors. *Proceedings of the International Symposium on High Performance Computer Architecture,* 2001, pp. 171-184.

[Brooks *et al.,* 2003] D. Brooks, V. Tiwari, and M. Martonosi. A framework for

architectural level power analysis and optimizations. *Proceedings of the Annual International Symposium on Computer Architecture,* 2000, pp. 83-94.

[Burger and Austin, 1997] D. Burger and T. Austin. The SimpleScalar Toolset, version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342*,* 1997.

[Carl and Smith, 1998] R. Carl and J. Smith. Modeling Superscalar Processors via Statistical Simulation, *Workshop on Performance Analysis and its Impact on Design (PAID-98),* 1998.

[Chandra *et al,* 2005] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. *Proceedings of the International Symposium on High Performance Computer Architecture*. 2005, pp. 340-351.

[Chen and Patterson, 1994] P. Chen and D. Patterson. A New Approach to I/O Performance Evaluation – Self-Scaling I/O Benchmarks, Predicting I/O Performance. *ACM Transactions on Computer Systems,* 1994, vol. 12(4), pp. 308-339.

[Chilimbi *et al.,* 2001] T. Chilimbi. Efficient representations and abstractions for quantifying and exploiting data reference locality. *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation,* 2001, pp. 191-202.

[Chou and Roy, 1996] T. Chou and K. Roy. Accurate power estimation of CMOS sequential circuits. *IEEE Transactions on VLSI Systems,* 1996, vol. 4(3), pp. 369-380.

[Collins *et al.,* 2001] J. Collins, J. Wang, H. Christopher, D. Tullesen, C. Huges, Y. Lee, D. Lavery, and J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *Proceedings of the Annual International Symposium on Computer Architecture,* 2001, pp. 14-25.

[Conte and Hwu, 1990] Conte T. and Hwu W-M. Benchmark Characterization for Experimental System Evaluation. *Proceedings of the 1990 Hawaii International Conference on System Sciences (HICSS),* Architecture Track, 1990, vol. I, pp. 6-16.

[Conte *et al.,* 1996] T. Conte, M. Hirsch, and K. Menezes. Reducing state loss for effective trace sampling of superscalar processors. *Proceedings of the International Conference on Computer Design*, 1996, pp. 468-477.

[Curnow and Wichman, 1976] H. Curnow and B. Wichman. A Synthetic Benchmark. *Computer Journal*, 1976, vol. 19(1), pp. 43-49.

[Denning, 1968] P. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 1968, vol 2(5), pp. 323-333.

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

[Eeckhout *et al.,* 2000] L. Eeckhout, K. De Bosschere, and H. Neefs. Performance Analysis through Synthetic Trace Generation. *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2000, pp. 1-6.

[Eeckhout and Bosschere*,* 2001] L. Eeckhout and K. De Bosschere. Hybrid Analytical-Statistical Modeling for Efficiently Exploring Architecture and Workload Design Spaces. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* 2001, pp. 25-34.

[Eeckhout *et al.,* 2002] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload Design: Selecting Representative Program-Input Pairs. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques,* 2002, pp. 83-92.

[Eeckhout *et al.,* 2003-1] L. Eeckhout, S. Naussbaum, J.E. Smith, and K. De Bosschere. Statistical Simulation: Adding Efficiency to the Computer Designer's Toolbox. *IEEE Micro*, 2003, vol. 23(5), pp. 26-38.

[Eeckhout *et al.,* 2003-2] L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Quantifying the Impact of Input Data Sets on Program Behavior and its Applications. *Journal of Instruction-Level Parallelism,* 2003, vol. 5, pp. 1-33.

[Eeckhout *et al.,* 2004-1] L. Eeckhout and K. De Bosschere. How accurate should early design stage power/performance tools be? A case study with statistical simulation. *Journal of Systems and Software. Elseiver,* 2004, vol 73(1), pp 45-62.

[Eeckhout *et al.,* 2004-2] L. Eeckhout, R. Bell Jr., B. Stougie, K. De Bosschere, and L. John. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. *Proceedings of International Symposium on Computer Architecture*, 2004, pp. 350-361.

[Eeckhout *et al.,* 2005] L. Eeckhout, J. Sampson, B. Calder. Exploiting Microarchitecture Independent Characteristics and Phase Behavior for Reduced Benchmark Suite Simulation. *Proceedings of the IEEE International Symposium on Workload Characterization*, 2005, pp. 2-12.

[Eyerman *et al.,* 2006] S. Eyerman, L. Eeckhout, and K. De Bosschere. Efficient Design Space Exploration of High-Performance Embedded Out-of-Order Microprocessors. *Proceedings of Design, Automation, and Test in Europe*, 2006, pp. 351-356.

[Felter and Keller] W. Felter and T. Keller. Power Measurement on the Apple Power Mac G5. *IBM Technical Report RC23276 (W0407-046).* 2004.

[Ferrari, 1984] D. Ferrari. On the foundations of artifical workload design. *Proceedings of AMC SIGMETRICS* Conference on Measurement and Modeling of Computer Systems, 1984, pp. 8-14.

[Ganger, 1995] G. Ganger. Generating Representative Synthetic Workloads: An Unsolved Problem. *Proceedings of Computer Management Group Conference*, 1995, pp. 1263-1269.

[Gowan *et al.,* 1998] M. Gowan, L. Biro, D. Jackson, Power Considerations in the Design of the Alpha 21264 Microprocessor. *Proceedings of the Design Automation* Conference, 1998, pp. 726-731.

[Genbrugge *et al.*, 2006] D. Genbrugge, L. Eeckhout, and K. De. Bosschere. Accurate Memory Data Flow Modeling in Statistical Simulation. *Proceedings of the International Conference on Supercomputing*, 2006, pp. 87-96.

[Genbrugge and Eeckhout, 2007] D. Genbrugge and L. Eeckhout. Statistical Simulation of Chip Multiprocessors Running Multi-Program Workloads. *Proceedings of the 25th IEEE International Symposium on Computer Design,* 2007, pp. 464-471.

[Gunther *et al.,* 2001] S. Gunther, F. Bins, D. Carmean, and J. Hall. Managing the Impact of Increasing Microprocessor Power Consumption. *Intel Technology Journal,* Q1 2001.

[Hammerstrom and Davidson, 1997] D. Hammerstrom and E. Davidson. Information content of CPU memory referencing behavior. *Proceedings of International Symposium on Computer Architecture*, 1997, pp. 184-192.

[Haungs *et al.,* 2000] M. Haungs, P. Sallee, and M. Farrens. Branch Transition Rate: A New Metric for Improved Branch Classification Analysis. *Proceedings of International Symposium on High Performance Computer Architecture*, 2000, pp. 241-250.

[Henning *et al.,* 2000] J. Henning. SPEC CPU2000: Measuring CPU Performance in the New Millennium. *IEEE Computer*, vol. 33(7), 2000, pp. 28-35.

[Hoste *et al.,* 2006-1] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L.K. John, and K.D. Bosschere. Performance Prediction Based on Inherent Program Similarity.

Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2006, pp. 114-122.

[Hoste and Eeckhout, 2006] K. Hoste and L. Eeckhout. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. *Proceedings of the IEEE International Symposium on Workload Characterization,* 2006, pp. 83-92.

[Hsiao *et al.,* 2000] M. Hsiao, E. Rudnick, and J. Patel. Peak power estimation of VLSI circuits: New peak power measures. *IEEE Transactions on VLSI* Systems, 2000, vol. 8(4), pp. 439-445.

[Hsieh and Pedram*,* 1998] 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*, 1998, vol. 17(11), pp. 1080-1089.

[Iyengar *et al.,* 1996] V. Iyengar, L. Trevillyan, and P. Bose. Representative traces for processor models with infinite cache. *Proceedings of the International Symposium on High Performance Computer Architecture*, 1996, pp. 62-73.

[Iyengar and Trevillyan, 1996] V. Iyengar and L. H. Trevillyan. Evaluation and generation of reduced traces for benchmarks. *Technical Report RC 20610, IBM Research Division, T. J. Watson Research Center*, Oct. 1996.

[John *et al.,* 1995] L. John, V. Reddy, P. Hulina, and L. Coraor. Program Balance and its impact on High Performance RISC Architecture. *Proceedings of the International Symposium on High Performance Computer Architecture,* 1995, pp.370-379.

[John *et al.,* 1998] L. John, P. Vasudevan, and J. Sabarinathan. Workload Characterization: Motivation, Goals, and Methodology. *Proceedings of the Workshop on Workload Characterization*, 1998, pp. 3-14.

[Joseph *et al.,* 2003] R. Joseph, D. Brooks, and M. Martonosi. Control Techniques to eliminate voltage Emergencies in High Performance Processors. *Proceedings of the International Symposium on High Performance Computer* Architecture, 2003, pp. 79-90.

[Joshi *et al.,* 2006-1] A. Joshi, J. Yi, R. Bell Jr., L. Eeckhout, L. John, and D. Lilja. Evaluating the Efficacy of Statistical Simulation for Design Space Exploration. *Proceedings of the International Symposium on Performance Analysis of Systems and Software,* 2006, pp. 70-79.

[Joshi *et al.,* 2006-2] A. Joshi, L. Eeckhout, R. Bell Jr., and L. John. Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks. *Proceedings of the IEEE International Symposium on Workload Characterization*, 2006, pp. 105-115.

[Joshi *et al.,* 2006-3] A. Joshi, A. Phansalkar, L. Eeckhout, and L. John. Measuring Program Similarity Using Inherent Program Characteristics. *IEEE Transaction on Computers,* 2006, vol. 55(6), pp. 769-782.

[Joshi *et al.,* 2007-1] A. Joshi, Y. Luo, and L. John. Applying Statistical Sampling for Fast and Efficient Simulation of Commercial Workloads. *To appear in IEEE Transaction on Computers,* 2007.

[Joshi *et al.,* 2007-2] A. Joshi, L. Eeckhout, R. Bell Jr., L. John. Distilling the Essence of Proprietary Workloads into Miniature Benchmarks. *To appear in Transactions on Architecture and Code Optimization,* 2007.

[Joshi *et al.,* 2007-3] A. Joshi, L. Eeckhout, and L. John. Exploring the Application Behavior Space Using Parameterized Synthetic Benchmarks. *Accepted as Poster in ACM International Conference on Parallel Architectures and Compilation Techniques,* 2007.

[Joshi *et al.,* 2007-4] A. Joshi, L. Eeckhout, L. John, and C. Isen. Parameterized Workload Modeling for Stressing Microarchitectures. *To appear in International Symposium on High Performance Computer Architecture*, 2008.

[Kanter 2006] D. Kanter. EEMBC Energizes Benchmarks. *Microprocessor Report.* July 2006.

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

[KleinOsowski and Lilja, 2002] AJ KleinOsowski and D. Lilja. MinneSPEC: A New SPEC Benchmark Workload Simulation-Based Computer Architecture Research. *Computer Architecture Letters,* vol.1, 2002.

[Kurmas *et al.,* 2003] Z. Kurmas, K. Keeton, and K. Mackenzie. Synthesizing Representative I/O Workloads Using Iterative Distillation. *Proceedings of International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2003, pp. 6-15.

[Hoste *et al.,* 2006-1] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L.K. John, and K.D. Bosschere. Performance Prediction Based on Inherent Program Similarity. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 114-122.

[Hoste and Eeckhout, 2006] K. Hoste and L. Eeckhout. Comparing Benchmarks Using Key Microarchitecture-Independent Characteristics. *Proceedings of the IEEE International Symposium on Workload Characterization,* 2006, pp. 83-92.

[Lafage and Seznec, 2000] T. Lafage and A. Seznec. Choosing Representative Slices of Program Execution for Microarchitecture Simulations: A Preliminary Application to the Data Stream. Workload Characterization of emerging computer applications, Kluwer Academic Publishers, 2001, pp. 145-163.

[Lee *et al.,* 2005] K. Lee, K. Skadron, and W. Huang. Analytical Model for Sensor Placement on Microprocessors. *Proceedings of International Conference on Computer Design,* 2005, pp. 24-27.

[Lim *et al.,* 2002] C. Lim, W. Daasch, and G. Cai. A thermal-aware superscalar microprocessor. *Proceedings International Symposium on Quality Electronic Design,* 2002, pp. 517-522.

[Luk *et al.,* 2005] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp.190-200.

[Luo *et al.,* 2005] Y. Luo, A. Joshi, A. Phansalkar, L. John, and J. Ghosh. Analyzing and Improving Clustering Based Sampling for Microprocessor Simulation. *Proceedings of International Symposium on Computer Architecture and High Performance Computing,* 2005, pp. 193-200.

[Najm *et al.,* 1995] F. Najm, S. Goel, and I. Hajj. Power estimation in sequential circuits. *Proceedings of Design Automation Conference*, 1995, pp. 253-259.

[Noonburg and Shen, 1997] D. Noonburg and J. Shen. A Framework for Statistical Modeling of Superscalar Processors. *Proceedings of the International Symposium on High Performance Computer Architecture,* 1997, pp. 298-309.

[Noonburg and Shen, 1994] D. Noonburg and J. Shen. Theoretical Modeling of Superscalar Processor Performance. *Proceedings of the International Symposium on Microarchitecture,* 1994, pp. 52-62.

[Nussbaum and Smith, 2001] S. Nussbaum and J. Smith. Modeling Superscalar Processors via Statistical Simulation. *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2001, pp 15-24.

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

[Qui *et al.,* 1998] Q. Qui, Q.Wu, and M. Pedram. Maximum Power Estimation Using the Limiting Distributions of Extreme Order Statistics. *Proceedings of the Design*

Automation Conference, 1998, pp. 684-689.

[Phansalkar *et al.,* 2005] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John. Measuring Program Similarity – Experiments with SPEC Benchmark Suites. *Proceedings of the International Symposium on Performance Analysis of Systems and Software,* 2005, pp. 10-20.

[Phansalkar *et al.,* 2007-1] A. Phansalkar, A. Joshi, and L. John. Subsetting the SPEC CPU2006 benchmark suite. *ACM Computer Architecture News,* 2007, vol. 35(1), 2007.

[Phansalkar *et al.,* 2007-2] A. Phansalkar, A. Joshi, and L. John. Analyzing the Application Balance and Redundancy in SPEC CPU2006. *Proceedings of the International Symposium on Computer Architecture,* 2007, pp. 412-423, 2007.

[Plackett and Burman, 1946] R. Plackett and J. Burman. The Design of Optimum Multifactorial Experiments. *Biometrika*, 1946, vol. 33(4), pp. 305-325.

[Ringenberg *et al.,* 2005] J. Ringenberg, C. Pelosi, D. Oekmke, and T. Mudge. Intrinsic Checkpointing: A Methodology for Decreasing Simulation Time Through Binary Modification. *Proceedings of International Symposium on Performance Analysis of Systems and Software,* 2005, pp. 78-88.

[Rajgopal, 1996] Challenges in Low-Power Microprocessor Design. *Proceedings of the International Conference on VLSI Design: VLSI in Mobile Communication*, 1996, vol. 3(6), pp. 329-330.

[Sair *et al.,* 2002] S. Sair, T. Sherwood, and B. Calder. Quantifying Load Stream Behavior. *Proceedings of the International Symposium on High Performance Computer Architecture*, 2002, 197-208.

[Sakamoto *et al.,* 2002] 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 International Symposium on High Performance Computer Architecture,* 2002, pp. 81-91.

[Saveedra and Smith, 1996] R. Saveedra and A. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *Proceedings of the ACM Transactions on Computer Systems*, 1996, vol. 14 (4) pp. 344-384.

[Shao *et al.,* 2005] M. Shao, A. Ailamaki, B. Falsafi. DBmbench: Fast and Accurate Database Workload Representation on modern microarchitecture. *Proceedings of the IBM Center for Advanced Studies Conference,* 2005.

[Sherwood *et al.,* 2002] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *Proceedings of the*

International Conference on Architectural Support for Programming Languages and Operating System, 2002, pp. 45-57.

[Skadron *et al.,* 2003-1] K. Skadron, M. Martonosi, D. August, M. Hill, D. Lilja, and V. Pai. Challenges in Computer Architecture Evaluation. *IEEE Computer*, 2003, vol. 36(8), pp. 30-36.

[Skadron *et al.,* 2003-2] K. Skadron, M. Stan, W. Huang, S. Velusamy, Sankaranarayanan and D. Tarjan. Temperature-aware microarchitecture. *Proceedings of International Symposium on Computer Architecture,* 2003, pp. 2-13.

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

[Spirn, 1972] J. Spirn and P. Denning. Experiments with Program Locality. The Fall Joint Conference, 1972, pp. 611-621.

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

[Srivastava and Eustace, 1994] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, March 1994.

[Stoutchinin *et al.,* 2001] A. Stoutchinin, J. Amaral, G. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative Prefetching of Induction Pointers. *Proceedings of Compiler Construction 2001*, *European Joint Conferences on Theory and Practice of Software*, 2001, pp. 289-303.

[Thiebaut, 1989] D. Thiebaut. On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio. *IEEE Transaction on Computers*, 1989, vol. 38(7), pp. 1012-1026.

[Tsui *et al.,* 1995] C. Tsui, J. Monteiro, M. Pedram, A Despain, and B. Lin. Power Estimation Methods for Sequential Logical Ciruits. *IEEE Transactions on VLSI* Systems, 1995, vol 3(3), pp. 404-416.

[Vishwanathan *et al.,* 2000] R. Vishwanath, V. Wakharkar, A. Watwe, V. Lebonheur. Thermal Performance Challenges from Silicon to Systems. *Intel Technology Journal Q3.* 2000.

[Weicker, 1984] R. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. Communications of the ACM, 1984, pp. 1013-1030.

[Weicker, 1990] R. Weicker. Overview of Common Benchmarks. *IEEE Computer,* 1990, vol. 23(12), pp. 65-75.

[Weicker, 1997] R. Weicker. On the use of SPEC benchmarks in computer architecture research. *Computer Architecture News,* 1997, vol. 25(1), pp. 19-22.

[Wong and Morris, 1998] W. Wong and R. Morris. Benchmark Synthesis Using the LRU Cache Hit Function. *IEEE Transactions on Computers*, 1998, vol. 37(6), pp. 637-645.

[Wu, 2002] Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002, pp. 210-221.

[Wunderlich *et al.,* 2003] R. Wunderlich, T. Wenish, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the International Symposium on Computer Architecture*, 2003, pp. 84-95.

[Yi *et al.*, 2003] J. Yi, D. Lilja, and D. Hawkins. Improving Computer Architecture Simulation Methodology by Adding Statistical Rigor. *IEEE Transactions on Computers*, 2003, vol.54(11) pp. 1360-1373.

[Yi *et al.*, 2005] J. Yi, S. Kodakara, R. Sendag, D. Lilja, and D. Hawkins. Characterizing and Comparing Prevailing Prevailing Simulation Techniques. Proceedings *of International Symposium on High Performance Computing*, 2005, pp. 266-277.

[Yi *et al.,* 2006-1] J. Yi, H. Vandierendonck, L. Eeckhout, and D. J. Lilja. The Exigency of Benchmark and Compiler Drift: Designing Tomorrow's Processors with Yesterdays Tools. *International Conference on Supercomputing*, 2006, pp. 75-86.

[Yi *et al.,*2006-2] J. Yi, R. Sendag, L. Eeckhout, A. Joshi, D. Lilja, and L. John. Evaluating Benchmark Subsetting Approaches. *Proceedings of the IEEE International Symposium on Workload Characterization,* 2006, pp. 93-104.

[Yi *et al.,* 2006-3] J. Yi, R. Sendag, L. Eeckhout, and L. John. Analyzing the Processor Bottlenecks in SPEC CPU2006. *Standard Performance Evaluation Corporation Annual Workshop*, 2006.

[SimPoint, 2007] SimPoint Website. http://www-cse.ucsd.edu/~calder/simpoint/

[Gcc-Inline, 2007] GCC-INLINE. www.cs.virginia.edu/~clc5q/gcc-inline-asm.pdf

[Bhattacharya & Williamson, 2007] Personal communication with Aparajita Bhattacharya (Senior Design Engineer) and David Williamson (Consulting Engineer), ARM Inc.

[Spec, 2007] http://www.spec.org/specpower/

Vita

Ajay Manohar Joshi was born in Pune, India, on November 25, 1976, as the son of Dr. Manohar Vasant Joshi and Mrs. Madhuri Manohar Joshi. He received his Higher Secondary-school Certificate (HSC) from St. Vincent's Jr. College, Pune, India, Bachelor of Engineering (BE) degree in Instrumentation Engineering from University of Pune, India, and Master of Science (MS) in Electrical Engineering from The Ohio State University, Columbus, Ohio. Before entering the doctoral program at The University of Texas at Austin in September 2003, he worked as a Software Engineer at Tata Infotech, India, and Hewlett-Packard, Richardson, Texas. During the summer of 2004 he was an intern at ARM Inc., Austin, and in summer 2005 an intern at International Business Machines Corp. He currently works in the Performance and Benchmarking group at ARM Inc., in Austin, Texas. He is a student member of the IEEE and ACM.

Permanent address: 7300 Rolling Stone Cove
Austin, TX 78739

This dissertation was typed by the author.