Copyright
by
Karthik Ganesan
2011

The Dissertation Committee for Karthik Ganesan
certifies that this is the approved version of the following dissertation:

# Automatic Generation of Synthetic Workloads for Multicore Systems

Committee:

______________________________
Lizy K. John, Supervisor

______________________________
Vijay K. Garg

______________________________
Adnan Aziz

______________________________
Sarfraz Khurshid

______________________________
Byeong K. Lee

# Automatic Generation of Synthetic Workloads for Multicore Systems

by

**Karthik Ganesan, B.E., M.S.E.**

**DISSERTATION**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**DOCTOR OF PHILOSOPHY**

THE UNIVERSITY OF TEXAS AT AUSTIN

December 2011

Dedicated to my parents,

Mr. Ganesan Swaminathan and Mrs. Radhamani Ganesan

# Acknowledgments

I would like to thank my advisor, Dr. Lizy John, for being a great mentor throughout the PhD program. She has given me invaluable guidance, financial support and has always been very motivating. She has been a great inspiration and was always available to answer questions and provide feedback. I would also like to thank (in alphabetical order) Dr. Vijay K Garg, Dr. Adnan Aziz, Dr. Sarfraz Khurshid and Dr.Byeong Lee for serving on my dissertation committee and providing invaluable comments and feedback.

I would like to thank Dr. Dimitris Kaseridis for helping me setup GEMS and Simics framework aiding in immensely accelerating my research. He has also been a great friend and has provided valuable feedback for my research. I would like to thank Dr. Ajay Joshi for providing me his simulation tools and framework, which served as the starting point for my research. I am thankful to Dr. Lloyd Bircher for helping me with hardware measurements at AMD. I also enjoyed working with Jungho Jo and Dr. Zhibin Yu on the different projects related to synthetic benchmarks. I am also thankful to the current and past members of the Laboratory of Computer Architecture, Dr. Jian Chen, Dr. Ciji Isen, Jungho Jo, Arun Nair, Faisal Iqbal, Youngtaek Kim for providing comments and feedback during the various practice talks of mine.

Amy Levin, Melanie Gulick, Deborah Prather, and Melissa Campos

were very helpful whenever I had any administrative issues and questions.

Prof. Venkateswaran Nagarajan of Waran Research Foundation, Chennai, India has played a key role in helping me understand the importance of research and the value of a PhD. I cannot thank him enough for his advise during my PhD program that kept me motivated to pursue my PhD to completion.

I am very grateful to my father Mr. Ganesan Swaminathan for his unswerving encouragement throughout my PhD program. He has always been a great source of inspiration and his encouragement has been one of the most significant motivating factors in my pursuit towards this doctorate. I am also very grateful to my mother Mrs. Radhamani Ganesan for providing me the moral support whenever I needed and also for enabling me to strive to become a well rounded personality. I am also thankful for the encouragement and support given by my brother and his family.

**Automatic Generation of Synthetic Workloads for Multicore Systems**

Karthik Ganesan, Ph.D.
The University of Texas at Austin, 2011

Supervisor: Lizy K. John

When designing a computer system, benchmark programs are used with cycle accurate performance/power simulators and HDL level simulators to evaluate novel architectural enhancements, perform design space exploration, understand the worst-case power characteristics of various designs and find performance bottlenecks. This research effort is directed towards automatically generating synthetic benchmarks to tackle three design challenges: 1) For most of the simulation related purposes, full runs of modern real world parallel applications like the PARSEC, SPLASH suites cannot be used as they take machine weeks of time on cycle accurate and HDL level simulators incurring a prohibitively large time cost 2) The second design challenge is that, some of these real world applications are intellectual property and cannot be shared with processor vendors for design studies 3) The most significant problem in

the design stage is the complexity involved in fixing the maximum power consumption of a multicore design, called the Thermal Design Power (TDP). In an effort towards fixing this maximum power consumption of a system at the most optimal point, designers are used to hand-crafting possible code snippets called power viruses. But, this process of trying to manually write such maximum power consuming code snippets is very tedious.

All of these aforementioned challenges has lead to the resurrection of synthetic benchmarks in the recent past, serving as a promising solution to all the challenges. During the design stage of a multicore system, availability of a framework to automatically generate system-level synthetic benchmarks for multicore systems will greatly simplify the design process and result in more confident design decisions. The key idea behind such an adaptable benchmark synthesis framework is to identify the key characteristics of real world parallel applications that affect the performance and power consumption of a real program and create synthetic executable programs by varying the values for these characteristics. Firstly, with such a framework, one can generate miniaturized synthetic clones for large target (current and futuristic) parallel applications enabling an architect to use them with slow low-level simulation models (e.g., RTL models in VHDL/Verilog) and helps in tailoring designs to the targeted applications. These synthetic benchmark clones can be distributed to architects and designers even if the original applications are intellectual property, when they are not publicly available. Lastly, such a framework can be used to automatically create maximum power consuming code snippets to be able

to help in fixing the TDP, heat sinks, cooling system and other power related features of the system.

The workload cloning framework built using the proposed synthetic benchmark generation methodology is evaluated to show its superiority over the existing cloning methodologies for single-core systems by generating miniaturized clones for CPU2006 and ImplantBench workloads with only an average error of 2.9% in performance for up to five orders of magnitude of simulation speedup. The correlation coefficient predicting the sensitivity to design changes is 0.95 and 0.98 for performance and power consumption. The proposed framework is evaluated by cloning parallel applications implemented based on p-threads and OpenMP in the PARSEC benchmark suite. The average error in predicting performance is 4.87% and that of power consumption is 2.73%. The correlation coefficient predicting the sensitivity to design changes is 0.92 for performance. The efficacy of the proposed synthetic benchmark generation framework for power virus generation is evaluation on SPARC, Alpha and x86 ISAs using full system simulators and also using real hardware. The results show that the power viruses generated for single-core systems consume 14-41% more power compared to MPrime on SPARC ISA. Similarly, the power viruses generated for multicore systems consume 45-98%, 40-89% and 41-56% more power than PARSEC workloads, running multiple copies of MPrime and multithreaded SPECjbb respectively.

# Table of Contents

**Acknowledgments** **v**

**Abstract** **vii**

**List of Figures** **xiii**

**Chapter 1. Introduction** **1**

1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

1.1.1 Prohibitive Simulation Time . . . . . . . . . . . . . . . . 3

1.1.2 Proprietary Applications . . . . . . . . . . . . . . . . . . 5

1.1.3 Worst-case Power Characteristics . . . . . . . . . . . . . . 5

1.2 Objectives . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

1.2.1 Power Virus Generation . . . . . . . . . . . . . . . . . . 12

1.2.2 Workload Cloning . . . . . . . . . . . . . . . . . . . . . 15

1.3 Thesis Statement . . . . . . . . . . . . . . . . . . . . . . . 16

1.4 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . 16

1.5 Organization . . . . . . . . . . . . . . . . . . . . . . . . . . 18

**Chapter 2. Related Research and Background** **20**

2.1 Statistical Simulation, Benchmark Synthesis and Workload Cloning . . . . . . . . . . . . . . . . . . . . . . 20

2.2 Other Simulation Time Reduction Techniques . . . . . . . . . . 22

2.3 Power Virus Generation . . . . . . . . . . . . . . . . . . . . . 24

2.4 Hiding Intellectual Property in Applications . . . . . . . . . . 25

2.5 ImplantBench Workloads . . . . . . . . . . . . . . . . . . . . 26

**Chapter 3. Synthetic Benchmark Generation Framework** **28**
3.1 Abstract Workload Model . . . . . . . . . . . . . . . . . . . . . 28
3.1.1 Stride Based Memory Access Behavior . . . . . . . . . . 31
3.1.2 Model for the Memory Level Parallelism . . . . . . . . . 32
3.1.3 Transition Rate Based Branch Behavior . . . . . . . . . 35
3.1.4 Dimensions of the Abstract Workload Model . . . . . . 35
3.2 Code Generation . . . . . . . . . . . . . . . . . . . . . . . . . . 44

**Chapter 4. Workload Cloning** **50**
4.1 Improved Workload cloning for Single-cores . . . . . . . . . . . 50
4.1.1 Benchmark Characterization . . . . . . . . . . . . . . . 50
4.1.2 Results and Analysis . . . . . . . . . . . . . . . . . . . . 61
4.1.2.1 Accuracy in the representativeness of the synthetic clones . . . . . . . . . . . . . . . . . . . . 61
4.1.2.2 Accuracy in the sensitivity to design changes . . 70
4.1.2.3 Cloning selected full runs of CPU2006 . . . . . 75
4.2 Workload cloning for Multicores . . . . . . . . . . . . . . . . . 77
4.2.1 Benchmark Characterization . . . . . . . . . . . . . . . 78
4.2.2 Results and Analysis . . . . . . . . . . . . . . . . . . . . 83
4.2.2.1 Accuracy in assessing performance . . . . . . . 83
4.2.2.2 Accuracy in assessing power consumption . . . 86
4.2.2.3 Accuracy in assessing sensitivity to design changes 87
4.2.2.4 Speedup achieved in using the synthetics . . . . 90
4.2.3 Proxies for Proprietary Applications . . . . . . . . . . . 92

**Chapter 5. Power Virus Generation** **94**
5.1 Abstract Workload Model . . . . . . . . . . . . . . . . . . . . . 96
5.2 Genetic Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . 100
5.3 Simulation Infrastructure . . . . . . . . . . . . . . . . . . . . . 103
5.4 State-of-the-art Power viruses . . . . . . . . . . . . . . . . . . 104
5.5 SYstem-level Max POwer (SYMPO) - Power Viruses for Single-core systems . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.5.1 Results on SPARC ISA . . . . . . . . . . . . . . . . . . 107

5.5.2 Results on Alpha ISA . . . . . . . . . . . . . . . . . . . . 111
5.5.3 Suitability of Genetic Algorithm for SYMPO . . . . . . 114
5.5.4 Validation of SYMPO using measurement on instrumented real hardware . . . . . . . . . . . . . . . . . . . . . . . . 115
5.6 MAximum Multicore POwer (MAMPO) - Power Viruses for Multicores . . . . . . . . . . . . . . . . . . . . . . . . . . . 119
5.6.1 Experimental Setup . . . . . . . . . . . . . . . . . . . . . 120
5.6.2 Results and Analysis . . . . . . . . . . . . . . . . . . . . . 122

**Chapter 6. Conclusions and Future Research** **128**
6.1 Workload Cloning . . . . . . . . . . . . . . . . . . . . . . . . 130
6.2 Power Viruses for Single-core Systems . . . . . . . . . . . . . . 132
6.3 Power Viruses for Multicore Systems . . . . . . . . . . . . . . 133

**Bibliography** **135**

**Vita** **148**

# List of Figures


1.1 Adaptable synthetic benchmark generation framework . . . . . 10

1.2 Breakdown of power consumption of the PARSEC benchmark *fluidanimate* on typical octcore and sixteen core systems . . . 11

3.1 List of metrics to characterize the execution behavior of workloads that significantly affect the performance and power consumption . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.2 Comparison of the MLP behavior of synthetics generated by previous approaches to that of a real single-threaded workload 33

3.3 Multithreaded synthetic workload generation . . . . . . . . . . 43

4.1 Overall workload cloning methodology . . . . . . . . . . . . . 51

4.2 Captured SFG information and branch transition rate for CPU2006 and ImplantBench workloads on a single-core system . . . . . 53

4.3 Dependency distance distribution for SPEC CPU2006 on a single-core system . . . . . . . . . . . . . . . . . . . . . . . . . . 54

4.4 Dependency distance distribution for ImplantBench workloads on a single-core system . . . . . . . . . . . . . . . . . . . . . 55

4.5 Memory access stride distribution for SPEC CPU2006 on single-core systems . . . . . . . . . . . . . . . . . . . . . . . . . . 57

4.6 Memory access stride distribution for ImplantBench workloads on single-core systems . . . . . . . . . . . . . . . . . . . . . 58

4.7 Captured MLP information as box plots showing the distribution of the burstiness of long-latency loads for CPU2006 workloads on a single-core system . . . . . . . . . . . . . . . . . . . 59

4.8 Captured MLP information as box plots showing the distribution of the burstiness of long-latency loads for ImplantBench workloads on a single-core system . . . . . . . . . . . . . . . . 59

4.9 Machine configurations used for cloning experiments on singlecore systems: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads . . . . . . . . . . . . . . . . . . . 60

4.10 Comparison of the basic block size between the synthetic and the original workloads for CPU2006 on single-core systems . . 61

4.11 Comparison of the Instruction mix of the original (bar on left) and the synthetic workloads (bar on right) for CPU2006 . . . 62
4.12 Comparison of the Instruction mix of the original (bar on left) and the synthetic workloads (bar on right) for ImplantBench . 62
4.13 Machine configurations used: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads . . . . . . . . . . 64
4.14 Comparison of IPC between the synthetic and the original workloads on single-core system configurations for Alpha ISA . . . 65
4.15 Comparison of power-per-cycle between the synthetic and the original workloads for CPU2006 on single-core system configuration for Alpha ISA . . . . . . . . . . . . . . . . . . . . . . 67
4.16 Comparison of power-per-cycle between the synthetic and the original workloads for ImplantBench on single-core system configuration for Alpha ISA . . . . . . . . . . . . . . . . . . . . . 68
4.17 Comparison of DL1 missrate, UL2 missrate and branch misprediction rate for CPU2006 and ImplantBench on single-core system configurations for Alpha ISA . . . . . . . . . . . . . . 69
4.18 Comparison of the variation of IPC and power-per-cycle for 433.milc between the synthetic and the original on single-core system configurations for Alpha ISA . . . . . . . . . . . . . . 72
4.19 Comparison of the variation of IPC and power-per-cycle for 445.gobmk between the synthetic and the original on single-core system configurations for Alpha ISA . . . . . . . . . . . . . . 73
4.20 Correlation coefficient between synthetic and the original for design changes on single-core system configurations for Alpha ISA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
4.21 Comparison of IPC between the synthetic and the original full runs for CPU2006 on single-core system configuration for Alpha ISA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
4.22 Speedup information for complete runs of some CPU2006 workloads on single-core system configuration for Alpha ISA . . . . 76
4.23 Instruction mix distribution for a 8-threaded version of various PARSEC workloads . . . . . . . . . . . . . . . . . . . . . . . 79
4.24 Spatial distribution of the accessed memory addresses into sharing patterns for various a 8-threaded version of PARSEC workloads . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
4.25 Temporal distribution of the various memory accesses in a 8-threaded version of PARSEC workloads into different sharing patterns for reads . . . . . . . . . . . . . . . . . . . . . . . . 82

4.26 Temporal distribution of the various memory accesses in a 8-threaded version of PARSEC workloads into different sharing patterns for writes . . . . . . . . . . . . . . . . . . . . . . . . . 82
4.27 Comparison of IPC between original and synthetic for various threads of benchmark Blackscholes in the PARSEC suite on a 8-core system configuration . . . . . . . . . . . . . . . . . . . . 84
4.28 Average Error in IPC between synthetic and original for the PARSEC benchmarks on a 8-core system configuration . . . . 84
4.29 Comparison of L1 missrate between the synthetic clones and that of the original PARSEC workloads on a 8-core system configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
4.30 Comparison of branch prediction rate between the synthetic clones and that of the original PARSEC workloads on a 8-core system configuration . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.31 Power-per-cycle for various PARSEC workloads along with a breakdown of the power consumption in various components on a 8-core system . . . . . . . . . . . . . . . . . . . . . . . . . . . 88
4.32 Multicore machine configurations used to evaluate the accuracy in assessing the impact of design changes by the synthetic in comparison to original PARSEC workloads . . . . . . . . . . . 88
4.33 Correlation coefficients for the sensitivity to design changes between the synthetic and the original using various multicore machine configurations for the workloads in the PARSEC suite 89
4.34 Comparison of sensitivity to design changes using various multicore machine configurations for the workload Streamcluster in PARSEC suite . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
4.35 Comparison of sensitivity to design changes using various multicore machine configurations for the workload Raytrace in PARSEC suite . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 91
4.36 Speedup achieved by using the synthetic proxies over the full run of the PARSEC workloads on a 8-core system configuration 91

5.1 Multithreaded power virus generation framework . . . . . . . 95
5.2 Abstract workload space searched through by the machine learning algorithm including the range of values used for each of the different knobs . . . . . . . . . . . . . . . . . . . . . . . . . 97
5.3 Single-threaded power viruses widely used in the industry . . . 106
5.4 Evaluation of SYMPO on SPARC ISA for single-core systems 108
5.5 Single-core machine configurations used to evaluate SYMPO . 109

5.6 Evaluation of SYMPO on Alpha ISA using SimpleScalar for single-core systems . . . . . . . . . . . . . . . . . . . . . . . . . 112
5.7 Breakdown of power consumption of SYMPO and MPrime for single-core systems on SPARC and Alpha ISAs . . . . . . . . . . 116
5.8 Machine configuration of AMD Phenom II . . . . . . . . . . . 117
5.9 Power measurement on quad-core AMD Phenom II . . . . . . 118
5.10 Multicore system configurations for which power viruses are generated to evaluate the efficacy of MAMPO on SPARC ISA 120
5.11 Interconnection networks used in the multicore system configurations for evaluating the efficacy MAMPO . . . . . . . . . . . 121
5.12 MAMPO virus generation and evaluation on multicore systems on SPARC ISA . . . . . . . . . . . . . . . . . . . . . . . . . 125
5.13 Breakdown of power consumption of MAMPO virus for various multicore system configurations and comparison to MPrime on SPARC ISA . . . . . . . . . . . . . . . . . . . . . . . . . . . 126

6.1 Summary of the power consumption of the single-threaded power virus generated by SYMPO in comparison to *MPrime* on Alpha, SPARC and x86 ISAs . . . . . . . . . . . . . . . . . . . . . . 132

# Chapter 1

# Introduction

Performance evaluation and benchmarking of computer systems has been a challenging task for designers and is only expected to increase in future due to the ever increasing complexity of modern computer systems. Understanding program behavior through simulations is the foundation for computer architecture research and program optimization. Thus, it is very common to have models written for the designed systems at various levels of abstractions in the design stage of a processor to enable simulations. Functional models, which stand at the highest level of abstraction are typically written using higher level languages like C, C++ and could be of varying levels of accuracy based on the models being cycle-accurate, trace or execution driven, bare-metal or include a full system. At the lowest level of abstraction, are the most detailed models written at the Register Transfer Level (RTL) using languages like VHDL or Verilog. These aforementioned models play a key role in evaluating novel architectural enhancements, perform design space exploration, understand the worst-case power characteristics and identify performance bottlenecks of various designs by enabling an architect to simulate the runs of the most representative set of target workloads.

Identifying the correct set of workloads to use with these models is in itself a more challenging task than even developing the models for the processors. Though microbenchmarks and kernels, which are hand coded code snippets that represent the most commonly used algorithms in real world applications are small and easy to use with the performance models, they may not be comprehensive enough to cover various execution behaviors to be representative of the real target applications. For this purpose, there have been benchmark suites, developed and maintained by academia and organizations like Standard Performance Evaluation Corporation (SPEC) containing the most commonly used applications in various domains. Some of the most popular benchmark suites are SPEC CPU2006 [1] [2], Splash-2 [3] [4], PARSEC [5], EEMBC and ImplantBench [6], which represent the most commonly used desktop, embedded and futuristic applications. The SPEC CPU2006 suite, released in Aug 2006 is a current industry-standard, CPU-intensive benchmark suite, created from a collection of popular modern single-threaded workloads. The EEMBC benchmarks contain workloads from the embedded domain targeting telecom/networking, digital media, Java, automotive/industrial, consumer, and office equipment products. The ImplantBench suite proposed by Jin et al. [6] is a collection of futuristic applications that will be used in futuristic bio-implantable devices. Splash-2 is a collection of multithreaded workloads developed at Stanford targeting shared memory systems. The Princeton Application Repository for Shared-Memory Computers (PARSEC) is a benchmark suite developed at Princeton University composed of emerging multithreaded

workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors.

There are many challenges involved in using these real applications with the various performance models in the design stage to analyze the performance and power characteristics of the designs under study. The most significant challenges addressed by this dissertation are that these workloads have a prohibitively large run time on the performance models, some of them are not available to architects as they are proprietary and most of them cannot be of much use in analyzing the worst-case power characteristics of designs. This dissertation addresses each of these challenges by distilling the most important characteristics of real world applications and using them to construct an adaptable synthetic benchmark generation framework, which will be a valuable tool in the design stage of processors.

## 1.1 Motivation

### 1.1.1 Prohibitive Simulation Time

For most of the simulation related purposes, full runs of modern real world applications like the SPEC CPU2006, PARSEC suites cannot be used as they take machine weeks of time on cycle accurate and HDL level simulators incurring a prohibitively large time cost. The prohibitive simulation time of the real-world applications [5] [2] can be attributed to the fact that they contain thousands of billions of instructions [1]. Design models of modern multicore systems at RTL level are many orders slower than cycle accurate

simulators. For example, the design of IBM POWER6 has 3.3 million lines of VHDL to model 790 million transistors and it is almost impossible to use full runs of modern workloads for design studies. The advent of the multicore processors and heterogeneity in the cores has made the simulations for design space exploration even more challenging. This has driven architects to use samples/traces of important parts of the target applications instead of complete runs. It is to be noted that even after 5 years after the release of the SPEC CPU2006 suite, we do not see many simulation based papers using these more representative modern workloads and rather architects tend to use the older version CPU2000 due to the availability of miniaturized samples/traces.

To reduce simulation time, sampling techniques like simulation points [7] and SMARTS [8] are well known and widely used. But, the problem with such sampling techniques is that most of them are restricted to phase behavior analysis and check-pointing of single-threaded applications and none of them can be directly used for sampling multithreaded applications or simultaneous execution of independent programs. Though there has been some efforts towards extending such sampling techniques for multicore architectures as in the work by Biesbrouck et al [9], but it is all still in infancy. Another problem with such sampling techniques is that huge trace files for the particular dynamic execution interval have to be stored or they require the simulator to have the capability to fast-forward until it reaches the particular interval of execution that is of interest to the user. The problem with other techniques like benchmark subsetting [10] is that the results are still whole programs and are too

big to be directly used with design models.

### 1.1.2 Proprietary Applications

The previously mentioned simulation time problem is augmented with the unavailability of some of the real target applications due to being proprietary. For example, in case where a vendor is designing a system for a defense application or for military purposes, it is not possible to have these target applications in hand for performance analysis. In such cases, the architect will end up using the publicly available similar applications or the most generic benchmark suites. But, these proprietary target applications may have some unique characteristics that is not accounted for, and could result in the architects ending up with a non-optimal design.

### 1.1.3 Worst-case Power Characteristics

Excessive power consumption and heat dissipation have been a critical problem faced by computer designers in the past decade. Due to power delivery, thermal and cooling issues along with a world-wide initiative towards green computing, power consumption is a first class design parameter in high end server systems and it has always been a significant constraint in low end embedded system design. More specifically, the maximum power consumption for which computer systems are designed, called the Thermal Design Power (TDP) is one of the most important of the different design parameters and is something that is very carefully determined by the computer architects. This

worst-case power consumption has a direct impact on attainable microprocessor performance and implementation cost. Current generation multi-core performance is almost universally limited by power delivery, cooling and reliability rather than critical path delay. The cooling systems of these modern processors/memories are designed in such a way, that these systems are deemed to safely operate only within this power cap and are equipped with the capability to automatically throttle down the operating frequency when the system is driven to reach this maximum power. This maximum power consumption for which a system is designed cannot just be fixed as the sum of the power consumption of the various components in the system, but rather it has to be the maximum attainable power consumption that a user workload could practically achieve in the system under design. This is due to the fact that this maximum attainable power consumption is quite low compared to the sum of the power consumption of various micro-architectural components as it is almost impossible to keep all these components of a system simultaneously active by any workload. The process of determining the maximum power for a design is very complicated due to it's dependence on multiple factors like the workload that could be executed, the configuration of the system, the power saving features implemented in hardware and the way some of these features are exercised by the operating system.

If the maximum power of a design is fixed too high, a designer will end up wasting a lot of resources by over-provisioning the heat sinks, cooling system, power delivery system and various other system level power management

utilities. A related example will be the design of external power supplies to server systems. Due to incognizance of the precise maximum attainable power of a system, a power supply could be designed to handle a high load and when the typical usage scenario is far below that load, the efficiency of the power supply is known to drop many folds [11]. It is to be noted that over provisioning of these power related utilities could result in substantial increase in maintenance costs. The recent trend towards consolidation in server systems (e.g. blade servers), has resulted in an explosion in power density leading to high costs related to electricity and cooling system. The 'power dissipation per square foot' of recent server systems is estimated to be 160 Watts per square foot. Data center energy costs are starting to exceed hardware costs and it was estimated that in 2010, the power a server burns over its lifetime will cost more than the server itself. It is estimated that for every watt of power used by the computing infrastructure in a data center, another 0.33 to 0.5 watt of power is required by the cooling system [12] [13] due to the ongoing rack-level compaction [14]. This problem has driven data center based companies to set up sites near power stations and design some of them to be wind-cooled naturally to save on cooling costs. On the other hand, if this maximum power consumption is underestimated, the architect will be unnecesarily limiting the performance of the system due to frequency throttling or in case of unavailability of such features, it results in affecting the overall system reliability and availability due to overheating. When the ambient temperature increases beyond the safe operating limits, it could result in early failure of

the micro-architectural components resulting in sporadic system freezes and crashes.

Identifying this attainable worst-case power in current generation microprocessors is a challenging task that will only become more daunting in the future. As additional system components are integrated into single packages, it becomes increasingly difficult to predict aggregate worst-case power. Existing designs integrate multiple cores and memory controllers on a single die. The trend for future designs is to include a wider array of components including graphics processors and IO bus controllers [15] [16]. It is to be noted that the worst-case power of a system is not simply the sum of the maximum power of each component. Due to underutilization of resources and contention for shared resources, such as caches or memory ports, the aggregate worst-case is significantly less than the sum.

In an effort towards fixing the maximum power consumption of systems at the most optimal point, architects are used to hand-crafting possible code snippets called power viruses [17] [18]. But, this process of trying to manually write such maximum power consuming code snippets is very tedious [19]. This tedium is due to the fact that there are so many components that interact when a workload executes on a processor/system making it intractable to model all these complex interactions and requires a profound knowledge about these interactions to be able to write a code snippet that will exactly exercise a given execution behavior. Adding to this complexity are the various power saving features implemented in the hardware like clock gating, demand based

switching, enhanced speed step technology and the various power states of the CPUs exercised by the operating system. Lastly, one cannot be sure that the manually written power virus is the practically possible maximum case to be able to safely design the processor for this particular maximum power. As a result of this, designers tend to end up in the aforementioned wasteful over-provisioning.

## 1.2 Objectives

During the design stage of a multicore system, availability of a framework to automatically generate system-level synthetic benchmarks for multicore systems will greatly simplify the design process and result in more confident design decisions. The key idea behind such an adaptable benchmark synthesis framework is to identify the key characteristics of real world applications such as instruction mix, memory access behavior, branch predictability, thread level parallelism etc that affect the performance and power consumption of a real program and create synthetic executable programs by varying the values for these characteristics as shown in Figure 1.1. Firstly, with such a framework, one can generate miniaturized synthetic clones for large target (current and futuristic) applications enabling an architect to use them with slow low level simulation models (e.g., RTL models in VHDL/Verilog) and helps in tailoring designs to the targeted applications. These synthetic benchmark clones can be distributed to architects and designers even if the original applications are proprietary that are not publicly available. These clones can-

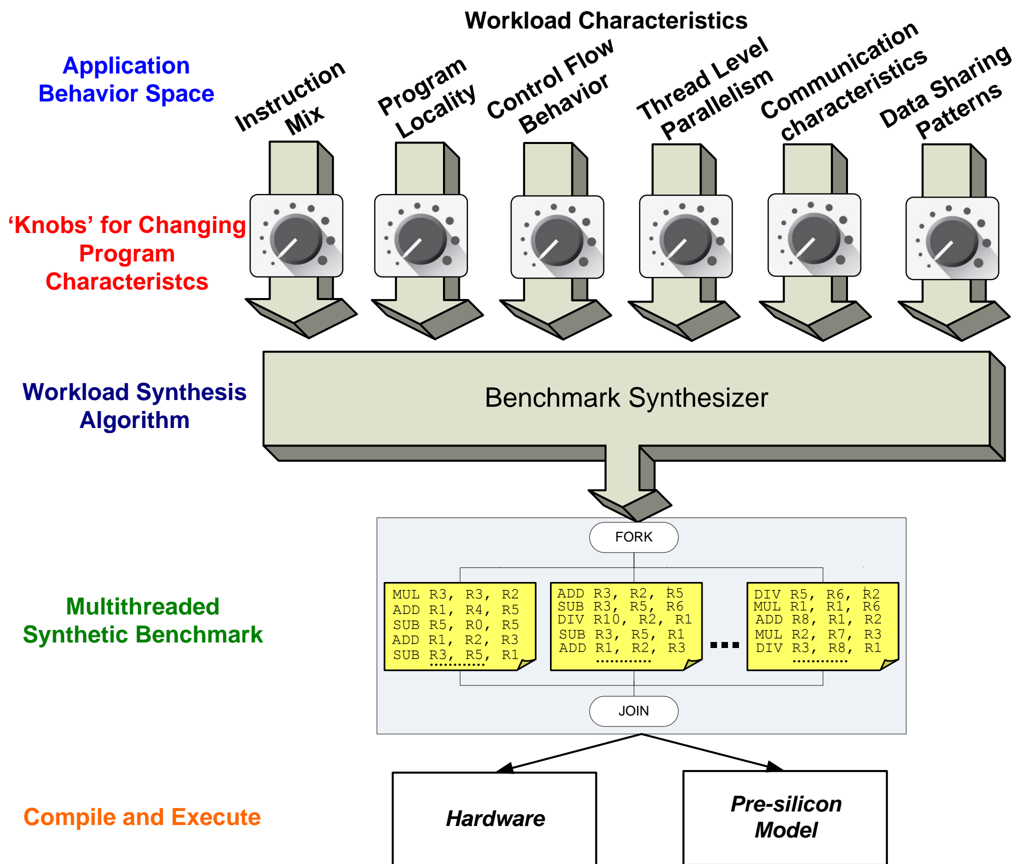



Figure 1.1: Adaptable synthetic benchmark generation framework

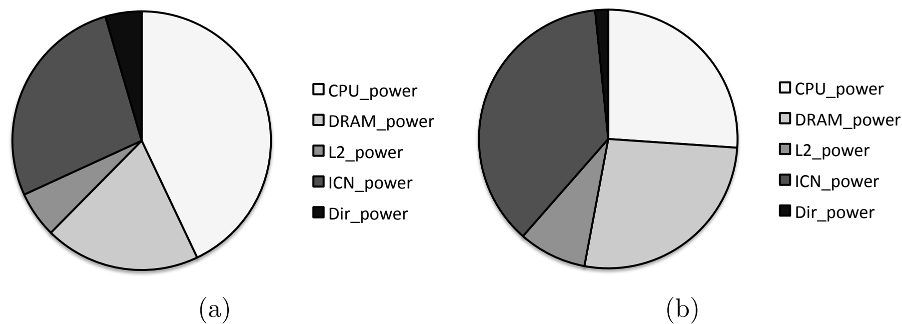



Figure 1.2: Breakdown of power consumption of the PARSEC benchmark *fluidanimate* on typical octcore and sixteen core systems

not be reverse engineered in any way to obtain any useful information about their original counterparts. Secondly, such a framework can be used to automatically create maximum power consuming code snippets to be able to help in fixing the Thermal Design Point, heat sinks, cooling system and other power related features of the system. The synthetic benchmarks that are provided are space efficient in terms of storage and do not require any special capability in a simulator as required by other simulation time reduction techniques [7] [8].

Though the applications of an automatic system-level synthetic benchmark generation framework are numerous, there has not been any efforts towards synthesizing workloads at system-level or for multicore systems. All the previous efforts towards synthesizing workloads [20] [21] [22] are all restricted to only the behavior of a single core CPU. It is to be noted that there are many components like the interconnection network, shared caches, memory subsystem and cache coherence directory other than the CPU that

significantly contribute to the overall performance and power consumption of a multicore parallel system. To emphasize the importance of the components other than the CPU, the breakdown of power consumption of the PARSEC [5] benchmark *fluidanimate* is shown on two typical modern multicore systems with eight and sixteen cores in Figures 1.2(a) and 1.2(b) respectively. The eight core system has eight 4-wide out-of-order cores with 4MB L2 and 8GB DRAM and the sixteen core system has sixteen 2-wide out-of-order cores with 8MB L2 and 16GB DRAM. One can see that the total power consumption of all the cores sum up to only 41% and 21% of the whole system power for the oct-core and sixteen-core systems showing the importance of the other components in the system. In today's multicore systems, it is important to characterize the behavior of the workloads in the shared caches, inter-connection network, coherence logic and the DRAM to be able to generate reasonable proxies for modern workloads. To achieve this, the synthetic benchmarks should be multithreaded and access shared addresses to be able to exercise various shared data access patterns.

### 1.2.1 Power Virus Generation

The objective is to automatically generate stressmarks by using a machine learning based search through a workload space constructed with microarchitecture independent characteristics broadly classified into instruction mix, thread level parallelism, instruction level parallelism, control flow behavior, shared and private memory access patterns and memory level parallelism.

Joshi et al. [22] also presented an automatic approach to the generation of power viruses, but Joshi's work was limited to the power consumption of the CPUs and has a few more limitations: 1) they had not modeled the burstiness of memory accesses or the Memory Level Parallelism (MLP) of the workloads 3) the framework was tested only by comparing with SPEC workloads and not with industry standard hand crafted power viruses 4) the results were only based on a simulator and was not validated on real hardware 5) it was done only for the Alpha ISA.

This dissertation aims at overcoming the aforementioned limitations and generating system level power stressmarks including components outside the CPU. Though the CPU consumes the maximum power among the various subsystems of a system, recent trends have shown that the power consumption of other subsystems like the DRAM is also significantly high [23] [24] and is predicted to increase in the future. Thus, it is important to characterize the power consumption of the entire system rather than just the CPU while constructing max-power viruses. Our metrics include the burstiness of accesses to DRAM by characterizing the memory level parallelism of the synthetic. The metrics used in this dissertation also include characteristics of workloads to stress the shared caches, coherence logic and the interconnection network. The granularity of the instruction mix in the generated synthetic is very important to generate good power viruses and our synthetic benchmark generation is more robust in terms of the number of instruction types generated than compared to Joshi et al's work [22]. In this work, we validate the power virus

generation framework using real hardware than just simulators. Three ISAs namely Alpha, SPARC and x86 are used for validation against industry grade power viruses than regular workloads.

The results in this work has shown that running multiple copies of these single-core power viruses like MPrime [25] on multiple cores is not even close to the power consumption of a power virus generated specifically for a given multicore parallel system. This is due to fact that such a single-core power virus like MPrime is very compute-bound lacking in data movement resulting in a reduced activity in the shared caches and the interconnection network. Due to upcoming memory hungry technologies like virtualization, the continuously more memory-seeking nature of today's search and similar Internet based applications along with a shift in paradigm from multicore to many-core, we see that only the power levels of processors being controlled and capped, while we do not see any signs of slow down in the increase in power consumption of memory and interconnects making it more important to be aware of their worst-case power characteristics. This is the first attempt towards answering many questions about how to efficiently search for a power virus for multicores viz., i) which are the most important dimensions of the abstract workload space that should be modeled for a multicore system, ii) what is the required amount of granularity in each dimension and especially the detail at which the core level out-of-order execution related workload characteristics should be modeled iii) if it is worthwhile to make the threads heterogeneous and deal with state space explosion problem or should the threads be homogeneous iv)

what are the data sharing patterns (producer-consumer, migratory etc) that should be exercised to stress the interconnection network, shared caches and DRAM effectively, and many other similar questions, each of which are further elaborated later in this paper.

### 1.2.2 Workload Cloning

In terms of workloads cloning, the objective is to characterize the long running original workloads for the identified metrics of interest. Then, these metrics are fed to the synthetic benchmark generation framework to generate a clone for each of these workloads. The fidelity of each of these clones are verified by comparing the most important microarchitecture dependent metrics like Instruction Per Cycle (IPC), total power consumption, missrates at caches, branch predictability etc. These clones should also be evaluated for their relative accuracy or the sensitivity to design changes, proving their utility in design space exploration for systems. The speedup achieved in using the provided miniaturized clones over using the original applications are reported to show the reduction in runtime.

The cloning framework is validated by cloning applications in three benchmark suites namely SPEC CPU2006, ImplantBench and PARSEC, representing the single threaded compute intensive application, embedded application and multithreaded parallel application domains respectively.

## 1.3 Thesis Statement

With knobs for thread-level parallelism, memory level parallelism, communication characteristics, synchronization characteristics and data sharing patterns included, a parameterized workload synthesis framework is a valuable tool in the design stage of multicore computer systems to generate representative miniaturized clones for long running modern applications and to automatically generate max-power stressmarks to help in fixing the Thermal Design Power for a given microarchitecture design.

## 1.4 Contributions

In this research, a system-level synthetic benchmark generation framework targeting both single-core and multicore systems is proposed. Amongst the different applications of such a framework, its efficacy for miniaturized workload clone generation and power virus generation are evaluated, each of which is elaborated below:

The workloads cloning framework will be very useful for architects, validation engineers, benchmarking engineers and performance architects in the design stage to miniaturize the long running workloads. Also, it should be noted that such a workload cloning framework will be more useful to software vendors who would like to disseminate their software to processor manufacturers even if it is proprietary. The synthetic clones that are generated cannot be reverse engineered in anyway as they only have the performance characteristics of the applications and do not retain any of the higher level information like

identifier names, function names or even instruction sequences. This cloning framework can significantly miniaturize applications and will also promote architecture research in both industry and academia by making simulations more feasible.

The power virus generation framework will be very useful for architects who manually write code snippets for power virus generation. This automation can reduce a lot of tedium and also provide enough confidence in the worst case behavior exercised by the synthetic power virus, avoiding the need to over-provision power related utilities. This need to over-provisioning the power related utilities will save a lot of power in data centers and reduce the cooling costs significantly.

The major contributions of this dissertation are,

- Proposal of the system-level synthetic benchmark generation framework, which includes an abstract workload model and a code generator to synthesize workloads for modern systems including multicores.

- The proposed framework is evaluated to show its superiority over the existing cloning methodologies for single-core systems by generating miniaturized clones for CPU2006 and ImplantBench workloads with only an average error of 2.9% in performance for up to five orders of magnitude of simulation speedup. The correlation coefficient predicting the sensitivity to design changes is 0.95 and 0.98 for performance and power consumption.

- The proposed framework is evaluated by cloning parallel applications implemented based on p-threads and OpenMP in the PARSEC benchmark suite. The average error in predicting performance is 4.87% and that of power consumption is 2.73%. The correlation coefficient in tracking the performance for design changes by the synthetic is 0.92.

- The proposed framework is further leveraged with the help of machine learning to build SYstem-Level Max POwer (SYMPO) and MAximum Multicore POwer (MAMPO) to automatically generate power viruses for single-core and multicore systems respectively.

- Validation of these power virus generation frameworks using SPARC, Alpha and x86 ISAs using full system simulators and also using real hardware. The results show that the usage of SYMPO results in the generation of power viruses that consume 14-41% more power compared to MPrime on SPARC ISA for single-core systems. Similarly, the MAMPO power viruses consume 45-98%, 40-89% and 41-56% more power than PARSEC workloads, running multiple copies of MPrime and multithreaded SPECjbb respectively.

## 1.5 Organization

- Chapter 2 elaborates on the synthetic benchmark generation framework starting with the most significant metrics relevant to the performance and power consumption of systems that are used in the abstract work-

load model for this dissertation. Later in the chapter, the synthetic code generation is explained, which is the process of translating the characteristics provided in the abstract workload model into synthetic code.

- Chapter 3 first provides an overview of the workload cloning framework which includes a profiler to profile the characteristics of the original application, the benchmark generator and the processor simulators used to evaluate the representativeness of the synthetics. The chapter also provides the accuracies of the clones generated for SPEC CPU2006, ImplantBench and PARSEC workloads.

- Chapter 4 discusses the power virus generation framework including the genetic algorithm toolset, simulators used to estimate the power consumption, the experimental setup to evaluate the power virus generation framework along with the results.

- Chapter 5 provides a brief overview of previous research in this area and along with related work.

- Chapter 6 summarizes the dissertation with conclusions and provides future directions

# Chapter 2

# Related Research and Background

## 2.1 Statistical Simulation, Benchmark Synthesis and Workload Cloning

Oskin et al. [26] and Nussbaum et al. [27] introduced the idea of statistical simulation to guide the process of design space exploration. Eeckhout et al [28] proposed the use of Statistical Flow Graphs (SFG) in characterizing the control flow behavior of a program in terms of the execution frequency of basic blocks annotated with their mutual transition probabilities. A SFG consists of nodes that are the basic blocks in the program and the edges represent the mutual transition probabilities between the basic blocks. Wong et al. introduced the idea of synthesizing benchmarks [29] [30] [31] based on the workload profiles. Bell and John [21] and Joshi et al. [20] synthesized benchmark clones for the workloads in the SPEC CPU2000 suite by using a technique in which one loop is populated with embedded assembly instructions based on the instruction mix, control flow behavior, the memory behavior and the branch behavior of the original workload. This generated synthetic loop was iterated until the performance characteristics became stable. In the work by Bell and John [21], most of the metrics that were used to characterize the behavior in the caches were based on microarchitecture dependent metrics like

miss rates. They used branch misprediction rate to characterize the control flow predictability, which is also a microarchitecture dependent metric. Since the synthetic clones that are generated are proposed to be used for design space exploration, a more robust framework was employed by Joshi et al. [20] by using metrics that were independent of the underlying microarchitecture like branch transition rate, a stride access pattern in terms of static load stores etc.

Even for single-core systems, the previous synthetic benchmark generation efforts [20] [21] suffer from a major limitation. Their methodologies characterize the memory access, control flow and the instruction level parallelism information of the workload, but do not characterize or use the miss pattern information of the last level cache, viz., Memory Level Parallelism (MLP) information. As a result, the synthetics generated using these previous approaches always have misses in the last level cache happening at a constant frequency without much burstiness. For example, when cloning workloads that have high MLP (bursty misses), the generated synthetic results in having an entirely different execution behavior compared to the original workload even in single core systems as shown in Figure 3.2. The proposed system-level multithreaded synthetic benchmark generation methodology overcomes this important shortcoming by modeling the MLP in the synthetic using load-load dependencies.

## 2.2 Other Simulation Time Reduction Techniques

Simulation time problem has been addressed by the computer architecture community and there has been a lot of previous work aimed at solving this problem. To reduce simulation time, sampling techniques like simulation points [7] and SMARTS [8] are well known and widely used. Considerable work has been done in investigating the dynamic behavior of the current day programs to address the prohibitive simulation time problem. It has been seen that the dynamic behavior varies over time in a way that is not random, rather structured [32] [33] as sequences of a number of short reoccurring behaviors. The SimPoint [7] [34] tool tries to intelligently choose and cluster these representative samples together, so that they represent the entire execution of the program. These small set of samples are called simulation points that, when simulated and weighted appropriately provide an accurate picture of the complete execution of the program with large reduction in the simulation time.

To analyze the similarity between two execution intervals in a microarchitecture independent manner, the Simpoint tool uses a signature for an execution interval called as a Basic Block Vector [35]. A basic block vector characterizes an execution interval based on the parts of the underlying static code, which is absolutely microarchitecture independent. The SimPoint tool [34][7][36] employs the K-means clustering algorithm to group intervals of execution such that the intervals in one cluster are similar to each other and the intervals in different clusters are different from one another. The Manhattan distance between the Basic Block Vectors serve as the metric to know

the extent of similarity between two intervals. The SimPoint tool takes the maximum number of clusters as the input and generates a representative simulation point for each cluster. The representative simulation point is chosen as the one which has the minimum distance from the centroid of the cluster. Each of the simulation points is assigned a weight based on the number of intervals grouped into its corresponding cluster. These weights are normalized such that they sum up to unity.

But, the problem with such sampling techniques is that huge trace files for the particular dynamic execution interval have to be stored or requires the simulator to have the capability to fast-forward until it reaches the particular interval of execution that is of interest to the user. But rather, the synthetic benchmarks that we provide are space efficient in terms of storage and do not require any special capability in a simulator. Also, most of these sampling techniques are restricted to single threaded applications and there has been very little work regarding runtime reduction for multithreaded applications. It is to be noted that most of these sampling techniques, when applied to different threads of multithreaded applications separately, still result in so many combinations of execution scenarios due to different possible starting points for each threads in the multithreaded program. The problem with other techniques like benchmark subsetting [10] is that the results are still whole programs and are too big to be directly used with design models.

## 2.3 Power Virus Generation

Joshi et al. [22] introduced the idea of automatic stressmark generation using an abstract workload generator. Joshi et al. also show that the characteristics of stressmarks significantly vary across microarchitecture designs, emphasizing the fact that separate custom stressmarks should be developed for different microarchitectures. In the same paper, they also show that machine learning can be used to generate stressmarks with maximum single cycle power. They also generated dI/dt stressmarks that will have an alternating behavior of maximum power in one cycle and minimum power in the next cycle, causing ripples in the power delivery network. Similarly, hotspots were created in various parts of the chip using the same methodology.

In the VLSI community, there has been a lot of research to estimate the power consumption of a given CMOS circuit [37] [38]. To maximize the switching activity in these circuits, test vector patterns are generated using heuristics and statistical methods. Our approach and goals in this paper are similar to these previous research, except the fact that we generate embedded assembly instructions that can be compiled into a legal program instead of the generation of test vectors. The advantage of using legal programs to search for a stressmark is that it guarantees that the maximum power consumption is achieved within the normal operating constraints. Industry has developed hand-crafted power viruses [39] [40] [41] [42] [43] to estimate the maximum power dissipation and thermal characteristics of their microprocessors. Hand-crafted benchmarks are also used in generating temperature differentials across

microarchitecture units [44]. Stability testing tools written for overclockers like CPUBurnin [17] and CPUBurn [18] are also popular power viruses. The program MPrime [25], which searches for mersenne prime number is popularly called the torture test and is a well known power virus used in the industry.

## 2.4 Hiding Intellectual Property in Applications

There has been a lot of efforts towards hiding the intellectual property in software applications [45] [46] [47] when distributing the binary. Most of these techniques try to confuse some one that is trying to reverse engineer the application by using many code obfuscation techniques. Some of the most popular techniques are,

- Layout Obfuscation: The higher level information like the identifier names, comments etc are altered to make them less meaningful. The C shroud system [48] is an example of a code obfuscator that does layout obfuscation.
- Data Storage Obfuscation: This technique aims at garbling the way in which data is stored in the memory to confuse some one that is reverse engineering. The data structures used by the program are altered for obfuscation. For example a two dimensional array can be converted to a one dimensional array, convert local variables to global variables etc.
- Control Aggregation Obfuscation: This technique tries to change the way in which the statements of a program are grouped together. A good

example is inlining some procedures.

- Control Ordering Obfuscation: Control ordering obfuscations change the order in which the statements of a program get executed. A good example is to iterate a loop backward instead of forward.

- Control Computation Obfuscation: These techniques try to alter the control flow of the program by performance code changes. Some examples are inserting dead code, adding unnecessary loop termination instructions that will not possibly happen etc.

All of the aforementioned techniques are used to avoid reverse engineering of code without any concern about changing the performance or power characteristics of the workloads. But, our aim in this approach is to disseminate intellectual property applications with the same power/performance characteristics for better processor design. Though the aforementioned techniques throw perspectives on what are the characteristics that should be looked at when it comes to hiding intellectual applications, they are aimed at something completely different than what is targeted in this dissertation.

## 2.5 ImplantBench Workloads

Further in this Section, we provide some background on the ImplantBench suite. The ImplantBench suite proposed by Jin et al. [6] is a collection of futuristic applications that will be used in bio-implantable devices. Bio-implantable devices are planted into human body to collect, process and

communicate realtime data to aid human beings in recovering from various types of defects. A few examples are retina implants, functional electrical stimulation implants and deep brain stimulation implants. ImplantBench is a collection of applications falling into the categories: security, reliability, bioinformatics, genomics, physiology and heart activity. Security algorithms are used in these devices for a safe and secure transfer of data from these implanted devices to the outside world. Reliability algorithms take care of the integrity of the data transferred to and from the implanted devices due to using wireless techniques. Bioinformatics applications are the ones that extract and analyze genomic information. At times a part of a genomic application may be added into the implanted device for some real time uses. Physiology includes the job of collecting and analyzing physiological signals like Electrocardiography (ECG) and Electroencephalography (EEG). Heart activity applications diagnose heart problems by analyzing the heart activity. Jin et al. [6] provide a detailed characterization of these applications, but most of their characterization is based on microarchitecture dependent metrics, whereas our characterization is mostly independent of the microarchitecture.

# Chapter 3

# Synthetic Benchmark Generation Framework

Our synthetic benchmark generation framework consists of two main components, namely, an abstract workload model and a code generator. The abstract workload model is formulated based on the most important characteristics of modern workloads in terms of performance and power consumption. A code generator is developed to synthesize workloads for a given set of characteristics in terms of the defined abstract workload model.

## 3.1 Abstract Workload Model

For both the purposes of cloning and power virus generation, the effectiveness of the synthetic benchmark generation framework lies in the efficacy of the abstract workload model that is formulated. The dimensions of this abstract workload space should be as much microarchitecture independent as possible to enable this framework to be able to generate synthetic benchmarks for different types of microarchitectures for the purposes of design space exploration. These dimensions should also be robust enough to be able to vary the execution behavior of the generated workload in every part of a multicore system. In earlier approaches for synthetic benchmark generation at core-level

for uniprocessors, researchers came up with metrics to characterize the execution behavior of programs on single core processors [20] [49] [50] [21]. In this research, we come up with similar metrics for the generation of system-level synthetics and for multicore systems. We first begin by explaining the intuition behind the design of this abstract workload space in terms of our memory access model, branching model and shared data access patterns.

Investigation in previous research [51][52][53][54][55] about the communication characteristics of the parallel applications has showed that there are four significant data sharing patterns that happen, namely,

1. **Producer-consumer sharing pattern**: One or more producer threads write to a shared data item and one or more consumers read it. This kind of sharing pattern can be observed in the SPLASH-2 benchmark *ocean*.

2. **Read-only sharing pattern**: This pattern occurs when the shared data is constantly being read and is not updated. SPLASH-2 benchmark *raytrace* is a good example exhibiting this kind of a behavior.

3. **Migratory sharing pattern**: This pattern occurs when a processor reads and writes to a shared data item within a short period of time and this behavior is repeated by many processors. A good example of this behavior will be a global counter that is incremented by many processors.

4. **Irregular sharing**: There is not any regular pattern into which the this access behavior can be classified into. A good example will be a global

| No. | Metric | Category |
|---|---|---|
| 1 | Dynamic execution freq. of basic blocks | Control flow predictability |
| 2 | Average basic block size | |
| 3 | Branch taken rate for each branch | |
| 4 | Branch transition rate | |
| 5 | Instruction pattern in a basic block | Instruction mix |
| 6 | INT ALU proportion | |
| 7 | INT MUL proportion | |
| 8 | INT DIV proportion | |
| 9 | FP ADD proportion | |
| 10 | FP MUL proportion | |
| 11 | FP DIV proportion | |
| 12 | FP MOV proportion | |
| 13 | FP SQRT proportion | |
| 14 | LOAD proportion | |
| 15 | STORE proportion | |
| 16 | Dependency distance distribution per instruction type | Instruction level parallelism |
| 17 | Private stride value per static load/store | Data locality |
| 18 | Data Footprint of the workload | |
| 19 | Mean and standard deviation of the MLP | Memory Level Parallelism (MLP) |
| 20 | MLP frequency | |
| 21 | Number of threads | Thread level parallelism |
| 22 | Thread class and processor assignment | Shared data access pattern and communication characteristics |
| 23 | Percentage loads to private data | |
| 24 | Percentage loads to read-only data | |
| 25 | Percentage migratory loads | |
| 26 | Percentage consumer loads | |
| 27 | Percentage irregular loads | |
| 28 | Percentage stores to private data | |
| 29 | Percentage producer stores | |
| 30 | Percentage irregular stores | |
| 31 | Shared stride value per static load/store | |
| 32 | Data pool distribution based on sharing patterns | |
| 33 | Number of lock/unlock pairs | Synchronization Characteristics |
| 34 | Number of mutex objects | |
| 35 | Number of Instructions between lock and unlock | |

Figure 3.1: List of metrics to characterize the execution behavior of workloads that significantly affect the performance and power consumption

task queue, which can be enqueued or dequeued by any processor which does not follow a particular order.

Though the above said patterns are the most commonly occurring sharing patterns, subtle variations of each one or more than one sharing pattern may be occurring in a multicore system.

### 3.1.1 Stride Based Memory Access Behavior

Capturing the data access pattern of the workload is critical to replay the performance of the workload using a synthetic benchmark. The data access pattern of a benchmark affects the amount of locality that could be captured at various levels of the memory hierarchy. Though locality is a global metric characterizing the memory behavior of the whole program, our memory access model is mainly based on a 'stride' based access pattern [20] in terms of static loads and stores in the code. When profiling a modern workload, one can observe that each of the static loads/stores access the memory like in an arithmetic progression, where the difference between the addresses of two successive accesses is called the stride. It is known that the memory access pattern of most of the SPEC CPU2000 and the SPEC CPU2006 workloads can be safely approximated to be following a few dominant stride values [56] [49].

In our abstract workload model, the stride values of the memory accesses to the private and shared data are handled separately. Each of the static loads and stores in the synthetic benchmark walk one of the allocated

shared/private memory arrays in a constant strided pattern until the required data foot print of the application is touched and after which, they again start from the beginning of the array. The other integer ALU instructions in the generated synthetic are used to perform the address calculation for these loads/stores. Along with the stride access patterns, the proportion of loads and stores in each thread also affect the data sharing pattern of the synthetic workload. For example, to achieve the producer-consumer sharing pattern between two threads, one will have to configure the instruction mix in such a way that the loads to shared data in the consumer and the stores to shared data in producer are in the right proportion and also configure the remaining knobs like the percent memory accesses to shared data, strides to shared data, thread assignment to processors and data footprint to enable these threads to communicate the right amount of data between each other in a given pattern. Though our model is robust enough to model parallel applications and their behavior, it can also be configured to model loosely related threads of commercial applications by increasing the private data accesses high enough.

### 3.1.2 Model for the Memory Level Parallelism

Even for single-core systems, the previous synthetic benchmark generation efforts [20] [21] suffer from a major limitation. Their methodologies characterize the memory access, control flow and the instruction level parallelism information of the workload, but do not characterize or use the miss pattern information of the last level cache, viz., Memory Level Parallelism

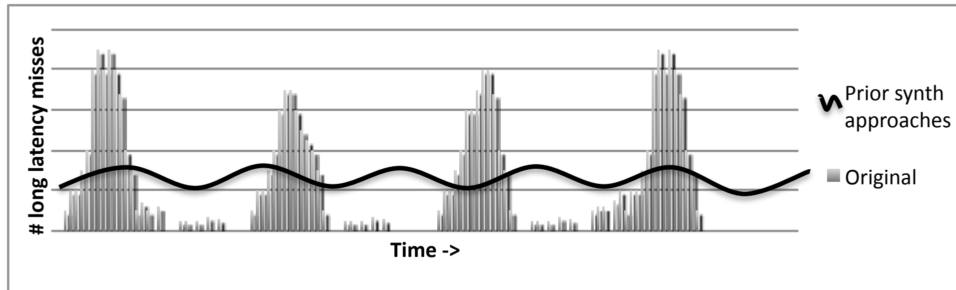



Figure 3.2: Comparison of the MLP behavior of synthetics generated by previous approaches to that of a real single-threaded workload

(MLP) information. The memory model used by the previous approaches [20] [21] consists of a set of static loads/stores that access a series of memory locations in a stride based access pattern. Even though the loads within this single loop are populated in such a way that they match the miss rates of the original application, they may not necessarily match the performance of the original application precisely. We classify loads into two categories. The loads that miss in the last level of the on-chip cache and result in an off-chip memory access are called 'long-latency' loads and the other set of loads that hit in the caches. Since these previous synthetic benchmark generation approaches do not model the burstiness of these long-latency loads, the long-latency loads are distributed randomly throughout the synthetic loop. These long-latency loads keep missing in a constant frequency as this loop is being iterated without much overlap in their execution. But the original workloads with the same miss rates may not necessarily have such a behavior. As already shown in Figure 3.2, the typical memory access behavior of the synthetics generated by the previous techniques can be entirely different compared to the case of many

of the original workloads. The original workloads can have a set of bursty long-latency loads in one time interval of execution and none of them at all for another interval of execution. In the original, though the pipeline may be clogged in this first interval due to the long-latency miss, the instructions may flow freely through the pipeline in the second. Rather, in the synthetic generated by previous approaches, there is a constant clog in the pipeline throughout the execution resulting in an entirely different execution behavior. In Section 3, we characterize the burstiness of misses in the target workloads and show real cases with the behavior (high MLP) as shown in Figure 3.2.

Since a long-latency load incurs hundreds of cycles due to the off-chip memory access, the performance of a workload varies significantly based on the amount of overlap present in the execution of these long-latency load instructions. The average number of such long-latency loads outstanding when there is at least one long-latency load outstanding is called the Memory Level Parallelism (MLP) present in a workload. Both of the cited previous approaches only characterize and model the Instruction-Level-parallelism in the workloads and fail to characterize and to model the Memory Level Parallelism (MLP) in the workloads. Eyerman and Eeckhout [57] show the impact of MLP on the overall performance of a workload. They show that there can be performance improvements ranging from 10% to 95% for various SPEC CPU2000 workloads if we harness the amount of MLP in the applications efficiently. This brings out the importance of characterizing the MLP in workloads. We characterize and model this MLP information in our synthetic generation framework.

For some workloads, we also require more than one loop to mimic the MLP behavior of the original workloads, upon which we elaborate in Section 3.

### 3.1.3 Transition Rate Based Branch Behavior

The branch predictability of the benchmark can be captured independent of the microarchitecture by using the branch transition rate [58]. The branch transition rate captures the information about how quickly a branch transitions between taken and not-taken paths. A branch with a lower transition rate is easier to predict as it sides towards taken or not-taken for a given period of time and rather a branch with a higher transition rate is harder to predict. First, the branches that have very low transition rates, can be generated as always taken or always not taken as they are easily predictable. The rest of the branches in the synthetic need to match the specified distribution of transition rate, which is further explained in the next Subsection.

### 3.1.4 Dimensions of the Abstract Workload Model

Our workload space consists of a set of 17 dimensions falling under the categories of control flow predictability, instruction mix, instruction level parallelism, data locality, memory level parallelism, shared access patterns, synchronization as shown in Figure 3.1. Further in this Subsection, each of these dimensions or what we call as the 'knobs' of our workload generator in this framework are explained along with their importance based on their power consumption compared to the overall power of the processor:

1. **Number of threads**: The number of threads knob controls the amount of thread level parallelism of the synthetic workload. This varies from only one thread up to 32 threads executing in parallel.

2. **Thread class and processor assignment**: This knob controls how the threads are mapped to different processors in the system. There are many thread classes to which each thread gets assigned. The threads in the same class share the same characteristics. This dimension is very useful when searching for a power virus, which will be detailed in Chapter 4.

3. **Number of basic blocks**: The number of basic blocks in the program combined with the basic block size determines the instruction footprint of the application. The number of basic blocks present in the program has a significant impact on the usage of the instruction cache affecting the performance and power consumption based on the Instruction cache missrates.

4. **Shared memory access stride values**: As mentioned earlier, two bins of stride values are specified for the shared memory accesses and every such memory access can be configured to have any one of the two bins with equal probability. This knob can also be configured separately for each of the different threads, to be able to allow each one of them to uniquely stress differ levels in the memory hierarchy.

5. **Private memory access stride values**: Similar to the stride values to the shared memory, two bins of stride values are specified for the private memory accesses and every such memory access can be configured to have the stride from any one of the two bins with equal probability. This knob can also be configured separately for each thread class to be able to stress different levels of the memory hierarchy separately.

6. **Data footprint**: This knob controls the data footprint of the synthetic. The data footprint of the application controls the number of cache lines that will be touched by the different static loads and stores. Also, it has a direct impact on the power consumption of the data caches. The correspondence of this knob to the real implementation in terms of number of iterations of one of the nested loops in the synthetic will be explained in detail in the next Subsection. This knob can be configured separately for different thread classes to be able to allow various cache resource sharing patterns in terms of varying data footprints.

7. **Memory Level Parallelism (MLP)**: This knob controls the amount of Memory Level Parallelism (MLP) in the workload, which is defined as the number of memory operations that can happen in parallel and is typically used to refer to the number of outstanding cache misses at the last level of the cache. The number of memory operations that can occur in parallel is controlled by introducing dependency between memory operations. The memory level parallelism of a workload also

affects the power consumption due to its impact on the DRAM power and also the pipeline throughput. This knob can also be configured separately for every thread class to enable the threads to have various access patterns to the DRAM.

8. **MLP frequency**: Though the MLP knob controls the burstiness of the memory accesses, one needs one more knob to control how frequently these bursty behaviors happen.

9. **Basic block size and execution frequency**: Basic block size refers to the average and standard deviation of number of instructions in a basic block in the generated embedded assembly based synthetic code. Execution frequency of basic block is used when detailed instruction pattern information has to be reproduced in the synthetic while cloning. The power consumption of a typical branch predictor which depends on the basic block size is usually around 4%-5% of the overall processor power.

10. **Branch predictability**: The branch predictability of a workload is an important characteristic that also affects the overall throughput of the pipeline. When a branch is mispredicted, the pipeline has to be flushed and this results in a reduced activity in the pipeline.

11. **Instruction mix**: The Instruction mix is decided based on the proportions of each of the instruction types INT ALU, INT MUL, INT DIV,

FP ADD, FP MUL, FP DIV, FP MOV and FP SQRT. Each of the instruction type in the abstract workload model has a weight associated with it ranging from 0 to 4. The proportion of this instruction type in the generated synthetic is not only governed by this weight, but also based on the weights associated with the remaining instruction types as they are correlated with each other. As different instruction types have different latencies and power consumption, the instruction mix has a major effect on the overall power consumption of a workload. Since the code generator generates embedded assembly, we have direct control over the instruction mix of the generated workload. Based on a static analysis of the power consumption, the typical power consumption of integer and floating point ALUs for an out-of-order superscalar processor is around 4%-6% and 6%-12% respectively. Some restrictions are placed on the instruction mix by writing rules in the code generator like a minimum number of INT ALU instructions should be present if there are any memory operations in the code to be able to perform the address calculation for these memory operations.

12. **Register dependency distance**: This knob refers to the average number of instructions between the producer and consumer instruction for a register data. The proportion of instructions that have an immediate operand is also used along with this distribution. This distribution is binned at a granularity of 1, 2, ... 20, 20-100 and greater than 100. This knob is required to be configured separately for different thread classes,

as different threads having different memory latencies may need to have different amounts of ILP. If the register dependency distance is high, the Instruction Level Parallelism (ILP) in the synthetic is high resulting in a high activity factor in the pipeline of the core. But, if the register dependency distance is low, the out-of-order circuitry like the ROB and other buffers may have higher occupancy resulting in a higher activity factor in these parts of the core. The activity factor also affects the clock power of a processor. The instruction window and the clock power are significant contributors to the power consumption of a processor ranging around 8% and 20% respectively.

13. **Random seed**: This knob controls the random seed that is used as an input to the statistical code generator, which will generate different code for the same values for all the other knobs. It mostly affects the alignment of the code or the order in which the instructions are arranged.

14. **Percentage loads to private data**: This dimension refers to the proportion of load accesses are to the private data and the rest of the memory accesses are directed to shared data. This knob can be configured separately for each thread class to allow the sharing percentage to be heterogeneous across thread classes. This heterogeneity may help the threads to be configured to differently stress the private and shared caches.

15. **Percentage loads to read-only data**: This dimension refers to the percentage of loads that access read-only data. Since this part of the

data does not have any writes, they do not cause any invalidation traffic in the interconnection network. The main traffic that will be generated by this kind of data will be capacity misses and data refills from other caches.

16. **Percentage migratory loads**: This dimension refers to the percentage of loads that are coupled with stores to produce a migratory sharing pattern. We not separately use a knob for migratory store percentage as it is co-dependent on this knob. This migratory sharing pattern can create huge amounts of traffic when coherence protocols like MESI is used where there is not a specific state for a thread to own the data.

17. **Percentage consumer loads**: This dimension refers to the percentage of loads that access the producer consumer data. The stores are configured to write to this producer consumer data and some loads are configured to read from them to reproducer the producer-consumer sharing pattern.

18. **Percentage irregular loads**: This dimension refers to the percentage of loads that fall into the irregular sharing pattern category and they just access the irregular data pool based on the shared strides specified.

19. **Percentage stores to private data**: This knob controls what proportion of stores access the private data and the rest of the memory accesses are directed to shared data. This knob can be configured separately for each thread class to allow the sharing percentage to be heterogeneous

across thread classes. This heterogeneity may help the threads to be configured to differently stress the private and shared caches.

20. **Percentage producer stores**: This dimension refers to the percentage of stores that write to the producer consumer data to replay the producer-consumer sharing pattern.

21. **Percentage irregular stores**: This dimension refers to the percentage of stores that fall into the irregular sharing pattern category and they write to the irregular data pool.

22. **Data pool distribution based on sharing patterns**: This dimension controls how the spatial data is divided in terms of the different sharing pattern access pools. It determines the number of arrays that are assigned to private, read-only, migratory, producer-consumer and the irregular access pools for the synthetic.

23. **Number of lock/unlock pairs**: This dimension refers to the number of lock/unlock pairs present in the code for every million instructions. This dimension is very important to make synthetics that are representative of multithreaded programs that have threads synchronizing with each other using locks.

24. **Number of mutex objects**: This dimension controls the number of objects that will be used by the locks in the different threads. It controls the conflict density between the threads when trying to acquire a lock.

```
pthread_barrier_t barr;
void** shared_array_ptrs;

main_function()
{
 shared_array_ptrs =
         allocate_arrays(sizes[]);

 //Barrier synchronization
 init_barrier(barr,
          number_of_threads);

 for every thread i in N; do
   create_pthread(thr_function_i);
 done

 join_pthreads (num_threads);
}
```

```
void thr_function_i()
{
   private_array_ptrs =
            allocate_arrays(sizes[]);

   processor_bind(My_threadID,
                      prscr_number);

   pthread_barrier_wait(&barr);

   while(out_cntr <= loop_count)
   {
      out_cntr++;

          /*....EMBEDDED...
            ....ASSEMBLY...
            ..INNER LOOPS..*/

      Reset_array_pointers();
    }
 }
```

Figure 3.3: Multithreaded synthetic workload generation

The more the number of mutex objects, lesser is the conflict density and in turn the workload executes much more efficiently resulting in a higher per-thread IPC.

25. **Number of Instructions between lock and unlock**: This dimension controls the number of instructions in the critical section of the workload. The bigger the critical section, the longer will be the wait to acquire locks by threads as it takes longer to finish executing all the instructions in the critical section and release a lock.

## 3.2 Code Generation

This section elaborates on how the final code generation happens based on the knob settings given in terms of the abstract workload parameters. Figure 3.3 shows an overview of code generation. The generated code consists of the main function and a function for each thread that is spawned from the main function using the pthread_create() system call. The required amount of shared data is declared and allocated in the main function as a set of integer/floating point arrays and the pointers to these arrays are available to each of the threads. The private data that is supposed to be used by every thread is declared and allocated within the function for each thread. Each of the threads also bind themselves with the processor number specified when the code was generated based on the thread class and processor assignment knob. A barrier synchronization is used to synchronize all the threads after they finish their respective system calls for allocating their private data arrays and binding themselves to the assigned processor.

The body of each thread consists of two inner loops filled with embedded assembly and one outer loop encompassing these inner loops. As previously mentioned, our memory model is a stride based access model, where the loads and stores in the generated synthetic access the elements of the private/shared arrays, each static load/store with a constant stride. The address calculation for the next access of each load/store is done by using other ALU instructions in the generated code for each of the array pointers by using the assigned stride value. When the specified data footprint is covered, the point-

ers that are used are reset to the beginning of the array. This pointer reset is done outside the inner loops and inside the encompassing outer loop enabling us to control the data footprint with the number of iterations of the inner loop and the number of dynamic instructions with the number of iterations of the outer loop. The embedded assembly contents of the two inner loops are the same if the MLP frequency knob is set to high. If the MLP frequency knob is set to low, the memory operations in the second loop are removed so that the bursty memory access behavior happening in the first loop occurs at a lower frequency.

Out of the total number of registers in the ISA, a set of registers are allocated to hold the base addresses of these allocated memory arrays and another set of registers are used to implement the predictability of the branches. The structure of our inner most loop is similar to that of the one proposed by Bell, et al. [21], but with an improved memory access, branching and ILP models. The required branch predictability or the control flow behavior in the synthetic is achieved by grouping branches into pools with each pool assigned to a constantly incremented register and a modulo operation on the register is used to decide if that branch is taken or not taken. The only information that is required to generate the main function is the biggest shared data footprint amongst the different threads to be able to allocate the shared arrays. The following steps are followed to generate the code for every thread based on the corresponding knob settings for each:

1. Generate the code to allocate the required amount of space for private

data accesses based on the percent private accesses, proportion of memory operations in instruction mix and the data footprint. The number of 1-D shared arrays are further subdivided into pools for each of the sharing patterns based on the spatial shared data access information.

2. Generate the processor_bind() system call using the assigned processor number and then a barrier synchronization system call is generated.

3. Generate the code for outer-loop based on the dynamic number of instructions desired taking into account the average basic block size and the number of basic blocks.

4. Fix the code spine for the first inner loop based on a fixed number of basic blocks and the average basic block size knob.

5. For each of the basic block in the first inner loop, configure the instruction type of each instruction by stochastically choosing from the instruction mix information. If the coupled load-stores is true, the instructions are swapped based on a bubble sort fashion in such a way that a store is made to follow a load and they are made to access the same address.

6. The basic blocks are bound together by using conditional branches at the end of each of the basic block. The number of branch groups and the modulo operation are fixed based on the required average branch predictability. The modulo operation for each of the branch groups are generated at the beginning of the inner loop based on the loop count

and a register is set/unset to decide if those branches for this particular group are going to be taken or not taken for this loop iteration. Branches are generated to fall through or take the target to another basic block based on their assigned register value.

7. Using the average dependency distance knob, each of the operands of every instruction is assigned with a previous producer instruction. Some rules are used to check the compatibility between producer and the consumer in terms of the data that is produced by the producer instruction and that consumed by the consumer. If two instructions are found to be not compatible, the dependency distance is incremented or decremented until a matching producer is found for every instruction. The memory level parallelism information is also used to assign load-load dependencies in this process.

8. Based on the percent private accesses knob, each of the memory operations are classified into the ones that access shared data and the ones that access private data. Based on the stride value of the corresponding memory operation (shared or private and based on the assigned bin), their corresponding address calculation instructions are given the stride values as immediate operands. If a load, store is classified to access shared data, the sharing pattern pool that they should access is determined by rolling a dice and using the shared data access pattern information.

9. Register assignment happens by first assigning the destination registers

in a round robin fashion. The source register for each operand of an instruction is assigned as the destination register of the producer instruction based on the corresponding dependency assignment.

10. The loop counters for the inner loops are set based on the specified data footprint and the compare instructions for loop termination are generated by choosing an integer ALU instruction in the code.

11. The second inner loop is also generated, which is a copy of the first loop without the memory operations if the MLP frequency is low or if it is set to be high, the second loop is generated just as a copy of the first loop.

12. Outside each of these inner loops, the memory base registers are reset to the first element of the memory arrays to enable temporal locality for the next loop or the next iteration of the outer wrapper loop.

13. Based on the number of locks, instructions between lock/unlock pairs provided as input to the code generator, the pthread_lock and pthread_unlock function calls are inserted in between the embedded assembly instructions. The mutex object to be used is determined by rolling a dice and choosing from the number of mutex objects specified.

During the synthesis of the workload, one can achieve the desired MLP in the synthetic by having control over the following: 1) the placement of highly strided loads (closer to each other or farther from each other) 2) the number of load-load dependencies. The highly strided loads are the long-latency loads

which miss in the last level of the cache. By grouping these long-latency loads, MLP of the synthetic can be controlled. When a load is dependent on another load, these two load instructions cannot be outstanding misses at the same time and thus, this also controls the amount of MLP in the synthetic. Though the first of the above said techniques are relatively easier to implement, it is not trivial to make a load instruction dependent on another load in our memory model due to walking an uninitialized memory array. If such a dependency is assigned directly, the 'consumer' load could access an invalid memory location. Initializing the memory array in the header of the synthetic alters the locality behavior of the synthetic. These special dependencies are handled as indirect dependencies in our framework by introducing an existing ALU instruction in the dependence chain to ensure the access to a valid memory location.

# Chapter 4

# Workload Cloning

Figure 4.1 shows the cloning methodology that is used. As the first step, the target application is profiled to collect the range of characteristics as specified in the abstract workload model. Then, this information is fed to the code generator to generate the synthetic. This final synthetic is compared with the original and the accuracies are reported for various machine configurations.

## 4.1 Improved Workload cloning for Single-cores

To clone the workloads for single core systems, a subset of the metrics (first 20) that are only relevant to single core systems in the abstract workload model are used.

### 4.1.1 Benchmark Characterization

To capture the various profile information of the workloads, we use modified versions of the different simulators in the SimpleScalar [59] simulation framework. Figure 3.1 delineates the different metrics, amongst which a subset of the metrics relevant to single-core systems are recorded for each of the workloads. Further in this Section, we explain each of these metrics and

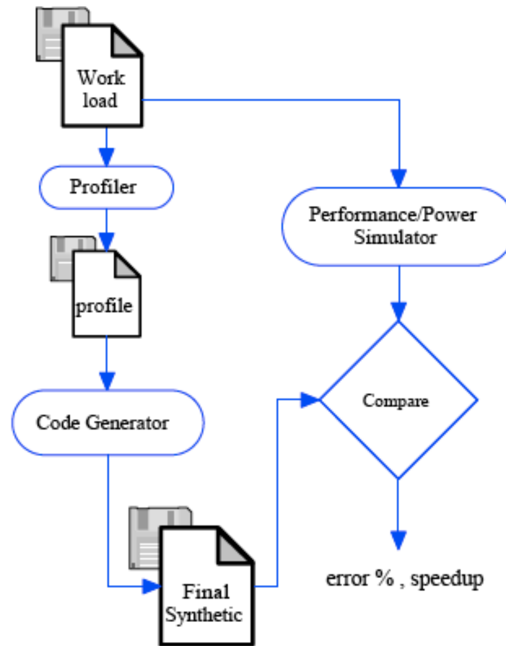



Figure 4.1: Overall workload cloning methodology

in tandem, provide the corresponding information captured for each of the metrics for the target workloads.

To capture the control flow behavior of a workload, the locality in the underlying static code being executed needs to be captured. A Statistical Flow Graph (SFG) [28] of the workload is constructed for capturing the control flow behavior of the workload. A SFG consists of nodes that are the basic blocks in the program and the edges represent the mutual transition probabilities between the basic blocks. We also record the average and the standard deviation of the size of the basic block along with the instruction pattern in the basic block in terms of the instruction type. The instruction mix of the original workload is a significant microarchitecture-independent metric that captures

the frequencies of various instruction types, namely: integer ALU operation, integer multiply, integer divide, floating point ALU operation, floating point multiply, floating point divide, load, store and branch.

For our experiments, we use the alpha binaries generated on an alpha machine running the Tru64 UNIX operating system using gcc 4.2 with an optimization level of -O2. A few of the SPEC CPU2006 workloads could not be compiled on the alpha architecture and we show the results for a set of 22 SPEC CPU2006 workloads. The tables in Figures 4.2(a) and 4.2(b) show the SFG information captured for the most representative 100 million instruction simulation point of the workloads in the CPU2006 benchmark suite and that of the ImplantBench suite respectively. We can see that the number of basic blocks that account for 90% of execution are only 6% of the total number of basic blocks in that interval of execution showing the amount of redundancy. The tables also show the average basic block size calculated based on both the number of instructions in each of the basic blocks and their dynamic execution frequency. As seen in 4.2(a), the floating point benchmarks of CPU2006 tend to have bigger basic block sizes compared to their integer counterparts in the same suite. The average number of successor basic blocks is another measure of the control flow complexity of the program. Programs that have complicated switch statements result in more successors and predicting the control flow becomes complicated. Also, when a function is called at multiple sites and each time it returns to different locations, it results in more successors.

The branch predictability of the benchmark can be captured indepen-

| Benchmark | No. of B.Blks | # B.Blks for 90% of Prog Exec. | Branch Transition Rate | Average B.Blk Size | Average Number of Succ. B.Blks |
|---|---|---|---|---|---|
| CINT 2006 | | | | | |
| 400.perlbench | 1620 | 169 | 0.07 | 5.99 | 1.93 |
| 401.bzip2 | 93 | 15 | 0.06 | 7.91 | 1.94 |
| 445.gobmk | 617 | 129 | 0.15 | 6.97 | 2.42 |
| 456.hmmer | 142 | 12 | 0.10 | 8.12 | 1.49 |
| 458.sjeng | 281 | 70 | 0.17 | 5.97 | 2.54 |
| 462.libquantum | 29 | 3 | 0.03 | 4.59 | 1.41 |
| 464.h264ref | 1074 | 55 | 0.09 | 15.13 | 1.69 |
| 471.omnetpp | 443 | 112 | 0.08 | 5.25 | 2.06 |
| 473.astar | 96 | 16 | 0.19 | 8.77 | 1.43 |
| 483.xalancbmk | 62 | 11 | 0.00 | 3.54 | 2.38 |
| 429.mcf | 179 | 96 | 0.01 | 10.58 | 1.21 |
| 403.gcc | 698 | 363 | 0.08 | 7.14 | 1.76 |
| CFP 2006 | | | | | |
| 410.bwaves | 32 | 10 | 0.26 | 32.61 | 1.50 |
| 433.milc | 42 | 18 | 0.44 | 15.32 | 1.48 |
| 434.zeusmp | 48 | 13 | 0.02 | 43.29 | 1.67 |
| 435.gromacs | 6 | 3 | 0.07 | 518.06 | 1.33 |
| 436.cactusADM | 22 | 7 | 0.18 | 247.16 | 1.27 |
| 437.leslie3d | 533 | 13 | 0.02 | 20.43 | 1.74 |
| 444.namd | 70 | 8 | 0.03 | 18.95 | 1.61 |
| 450.soplex | 358 | 14 | 0.03 | 7.79 | 1.45 |
| 459.GemsFDTD | 119 | 1 | 0.01 | 48.81 | 1.40 |
| 482.sphinx3 | 544 | 20 | 0.07 | 12.76 | 1.70 |

(a) SPEC CPU2006 workloads

| Benchmark | No. of B.Blks | # B.Blks for 90% of Prog. Exec. | Branch Transiti on Rate | Average B.Blk Size | Average # of Succ. B.Blks |
|---|---|---|---|---|---|
| AI_Adaline | 342 | 6 | 0.05 | 39.22 | 1.63 |
| AI_BPN | 410 | 13 | 0.06 | 54.27 | 1.80 |
| AI_GA | 503 | 160 | 0.23 | 7.7 | 2.26 |
| Bioinformatics_ELO | 194 | 15 | 0.05 | 9.42 | 1.72 |
| Bioinformatics_LMGC | 498 | 111 | 0.25 | 7.55 | 1.91 |
| Genomics_HMM | 463 | 119 | 0.28 | 8.03 | 1.80 |
| Genomics_NJ | 196 | 18 | 0.17 | 9.41 | 2.03 |
| HeartActivity_pNNx | 479 | 97 | 0.17 | 6.41 | 1.68 |
| Physiology_AFVP | 679 | 105 | 0.26 | 7.07 | 1.77 |
| Physiology_ECGSYN | 647 | 142 | 0.18 | 8.99 | 1.65 |
| Reliability_alder32 | 143 | 101 | 0.43 | 6.81 | 1.88 |
| Reliability_crc | 172 | 89 | 0.36 | 6.1 | 2.20 |
| Reliability_luhn | 155 | 90 | 0.40 | 6.46 | 2.10 |
| Reliability_reed_solm | 255 | 4 | 0.02 | 4.71 | 1.93 |
| Security_haval | 338 | 82 | 0.25 | 8.1 | 1.94 |
| Security_KHAZAD | 198 | 17 | 0.14 | 121.65 | 1.97 |
| Security_sha2 | 313 | 7 | 0.06 | 37.32 | 1.76 |

(b) ImplantBench workloads

Figure 4.2: Captured SFG information and branch transition rate for CPU2006 and ImplantBench workloads on a single-core system

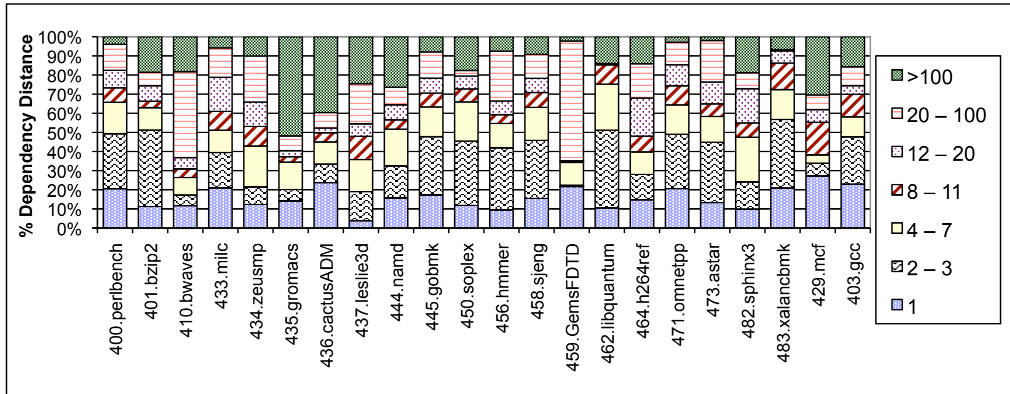



Figure 4.3: Dependency distance distribution for SPEC CPU2006 on a single-core system

dent of the microarchitecture by using the branch transition rate [58]. The branch transition rate captures the information about how quickly a branch transitions between taken and not-taken paths. A branch with a lower transition rate is easier to predict as it sides towards taken or not-taken for a given period of time and rather a branch with a higher transition rate is harder to predict. The branch transition rates for the CPU2006 workloads as given in Figure 4.2(a) average around 0.11 with few benchmarks like 433.milc, 410.bwaves having a transition rate above 0.25. Similarly, the branch transition rates of the ImplantBench suite are shown in Figure 4.2(b). It can be noted that the a few reliability applications have branches with high transition rates.

To capture the Instruction Level Parallelism (ILP) in the workload, we capture the dependency distance or the register reuse distance of the workload as a distribution. This information is captured for each type of instruction.

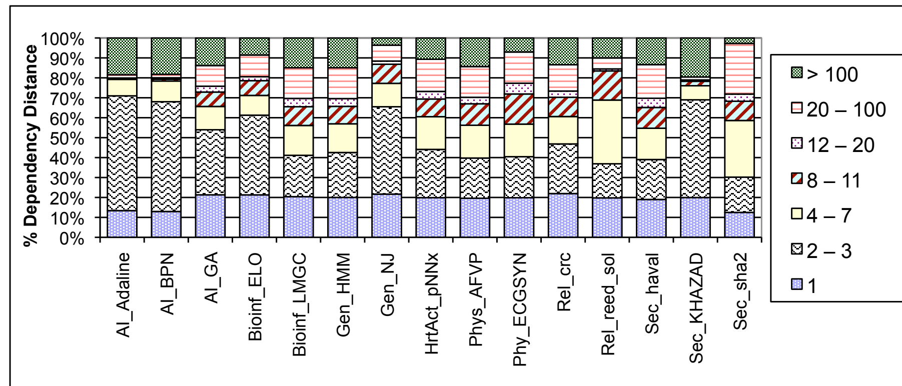



Figure 4.4: Dependency distance distribution for ImplantBench workloads on a single-core system

This corresponds to the number of instructions between the production and the consumption of a data value at the register level. The proportion of instructions that have an immediate operand is also recorded along with this distribution. This distribution is binned at a granularity of 1, 2, ... 20, 20-100 and greater than 100. The Figures 4.3 and 4.4 show a histogram of the dependency distances for the workloads in CPU2006 and the ImplantBench respectively. It can be observed that a few benchmarks like 436.cactusADM and 435.gromacs, which have very large basic block sizes (518 and 247 respectively), tend to have larger dependency distances. 435.gromacs and 436.cactusADM have respectively 50% and 40% of their dependencies that can be potentially resolved before 100 instructions. Still, it is to be noted that 436.cactusADM has more than 20% of dependencies just before one instruction. Such benchmarks with high instruction level parallelism will be sensitive to the out-of-order resources available in a processor and modeling their dependency distance distribution

plays a key role in mimicking the behavior of the original workload. In general, it can be observed that the ImplantBench and SPEC CINT2006 workloads that have smaller basic block sizes have shorter dependency distances compared to the SPEC CFP2006 workloads that have larger basic block sizes. For the ImplantBench and the CINT2006 workloads, more than 50% of their dependency distances are within 2-3 instructions.

Capturing the data access pattern of the workload is critical to replay the performance of the workload using a synthetic benchmark. The data access pattern of a benchmark affects the amount of locality that could be captured at various levels of the memory hierarchy. Though locality is a global metric characterizing the memory behavior of the whole program, previous studies [20] have resorted to characterizing the access behavior at per static load/store basis in terms of strides (differences between two consecutive effective addresses) to effectively model it again in the synthetic. Joshi et al. [49] identify that for the SPEC CPU2000 workloads, most of the load and store instructions have a dominant stride based memory access. We also observe a similar behavior in the SPEC CPU2006 workloads. Figure 4.5 shows the breakdown of the stride access patterns at a granularity of a 64Byte block by different load/store instructions binned into categories 0, 1-3, 4-7, 8-15, 16-31 and $> 31$. This way of characterizing and portraying the stride access patterns in terms of 64 byte block sizes is similar to the previous work as in Joshi et al. [49] for SPEC CPU2000. It is clearly evident that most of the benchmarks exhibit a stride based behavior. Benchmarks 456.hmmer, 473.astar, 436.cac-

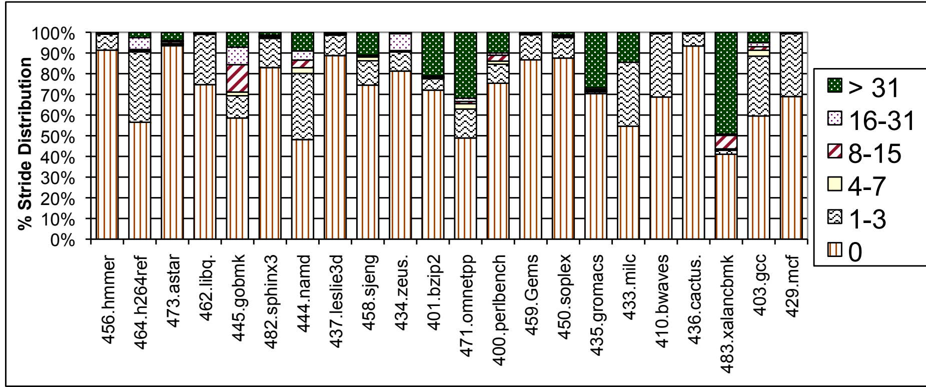



Figure 4.5: Memory access stride distribution for SPEC CPU2006 on single-core systems

tusADM have a dominant stride of zero for more than 90% of the memory accesses meaning that 90% of their accesses are within the same 64 byte block and will probably result in a lot of hits in the cache due to spatial locality. Totally, 12 benchmarks out of the 22 benchmarks studied have dominant stride for more than 75% of the memory accesses. For the ImplantBench suite, Figure 4.6 shows the break down of the stride at a 64 byte block granularity and it can be observed that the ImplantBench suite also has the same dominant stride behavior as CPU2006 and CPU2000 suites.

*Capturing Memory Level Parallelism Information:* Memory Level Parallelism information of the workloads is captured and the Figures 4.7 and 4.8 show the distribution of number of outstanding long-latency loads as box plots for the different workloads in the CPU2006 and ImplantBench suites respectively. By using the stride information, the synthetics mimic the hit/miss rate behavior of the original workloads. The impact of these hit/miss rates on the

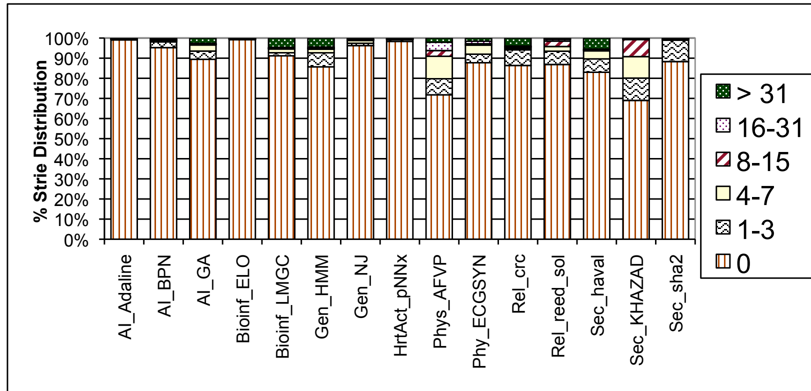



Figure 4.6: Memory access stride distribution for ImplantBench workloads on single-core systems

execution time is taken care of by capturing the memory level parallelism information. It can be noted that 483.xalancbmk, 410.bwaves, 436.cactusADM, 433.milc, 437.leslie3d, 459.GemsFDTD, 462.libquantum have relatively higher amounts of MLP compared to other benchmarks. It is to be noted that most of these benchmarks are floating point benchmarks except 462.libquantum and 483.xalancbmk. We also record the number of consecutive dynamic instructions when there are no outstanding long-latency loads to model the frequency of the bursty misses.

Also, to match the MLP of the original workloads, recording and modeling the load-to-load dependencies that exist in the original application plays a significant role. When a long-latency load is dependent on another long-latency load, there cannot be any overlap in execution between these loads. The previous synthetic benchmark generation approaches modeled the dependency distances only at the detail of the consumer instruction type and did

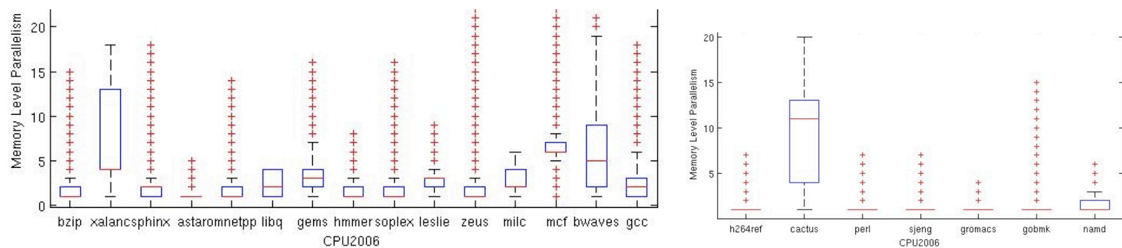



Figure 4.7: Captured MLP information as box plots showing the distribution of the burstiness of long-latency loads for CPU2006 workloads on a single-core system

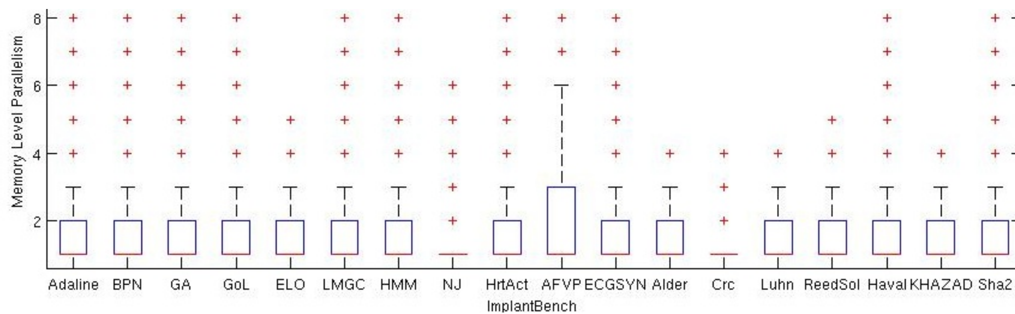



Figure 4.8: Captured MLP information as box plots showing the distribution of the burstiness of long-latency loads for ImplantBench workloads on a single-core system

| Configuration | Machine-A | Machine-B |
|---|---|---|
| Machine widths | Four wide out-of-order | one wide out-of-order |
| RUU size | 128 entries | 16 entries |
| LSQ size | 64 entries | 8 entries |
| Functional Units | 4 Integer ALUs, 1 Integer Mul/Div, 4 FP ALUs and 1 FP Mul/Div | 2 Integer ALUs, 1 Integer Mul/Div, 1 FP ALU and 1 FP Mul/Div |
| Memory | 8B bus and 80 cycles access time | 8B bus and 40 cycles access time |
| Branch Predictor | Bimod with table size 2048 | 2-level Gap predictor |
| IFQ size | 32 entries | 8 entries |
| L1 Data Cache | 32 KB, 4 way, 32B line size, 2 cycle access time | 16 KB, 2 way, 32B line size, 1 cycle access time |
| Unified L2 Cache | 4MB, 4 way, 64B line size, 12 cycle access time | 64 KB, 4 way, 64B line size, 6 cycle access time |
| L1 Instruction Cache | 16 KB, block size 32 Bytes, 1 cycle access time | 16 KB, block size 32 Bytes, 1 cycle access time |

Figure 4.9: Machine configurations used for cloning experiments on singlecore systems: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads

not record the type of the producer. Our experimental results (next section) show that, at least this information has to be captured for load instructions to be able to match the memory behavior of some of the modern workloads. By feeding this profiled data to the code generator as described in the previous chapter, clones were generated for SPEC CPU2006 and ImplantBench workloads. In the rest of this chapter, we compare both the microarchitecture dependent and microarchitecture-independent characteristics of the synthetic benchmark to that of the original workload.

To show the superiority of the proposed framework over previous cloning methodologies, SPEC CPU2006 and the ImplantBench workloads are cloned and the clones are evaluated for the machine configurations as shown in Figure

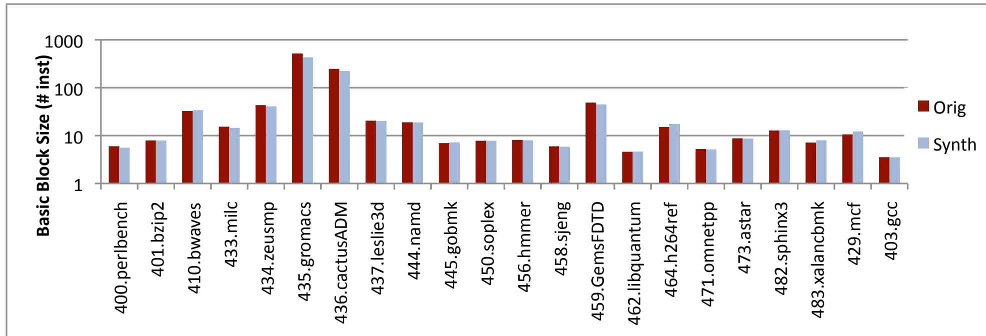



Figure 4.10: Comparison of the basic block size between the synthetic and the original workloads for CPU2006 on single-core systems

4.9.

### 4.1.2 Results and Analysis

#### 4.1.2.1 Accuracy in the representativeness of the synthetic clones

The accuracies of the synthetic benchmarks in capturing the performance of the original application is evaluated by comparing both microarchitecture dependent and independent metrics. First, we compare the microarchitecture-independent metrics like the basic block size, instruction mix, and dependency distance distribution of the original to that of the synthetic. Figure 4.10 shows the arithmetic mean of the basic block sizes of the original and the synthetic in the logarithmic scale. These arithmetic means were calculated based on the number of instructions in the basic blocks and the dynamic frequency of execution of each of the basic blocks. It can be observed that the basic block sizes of the synthetic match the basic block sizes of the original with an average error of 3.9%.

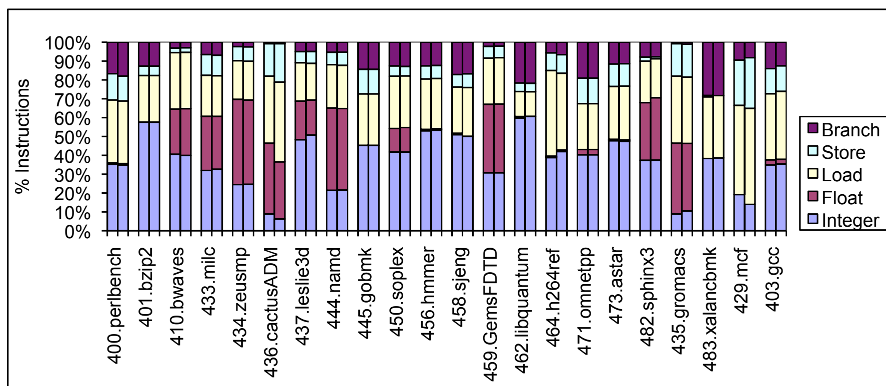



Figure 4.11: Comparison of the Instruction mix of the original (bar on left) and the synthetic workloads (bar on right) for CPU2006

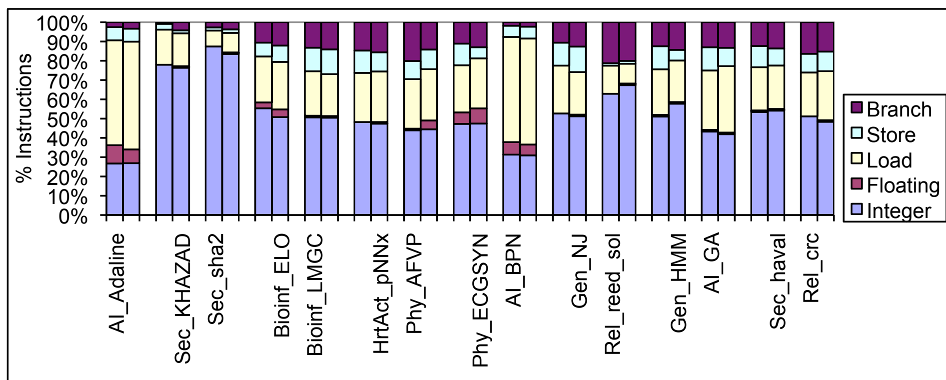



Figure 4.12: Comparison of the Instruction mix of the original (bar on left) and the synthetic workloads (bar on right) for ImplantBench

The Figures 4.11 and 4.12 show the instruction mix of the synthetic benchmark and that of the original benchmarks for the CPU2006 and Implant-Bench workloads. It can be found that the instruction mix of the synthetic matches that of the original very closely and the average errors are within 5%. Even the minimal error in the instruction mix occurs when the effective address calculation for loads/stores or the modulo operation for the branch needs to be done and there are not enough integer ALU instructions in the original benchmark. The dependency distances of the original and the synthetic are compared based on each of the instruction type and the error is evaluated. Usually the dependency assigning algorithm does not have to move up/down more than 2-3 instructions before it successfully assigns the dependency for our target workloads. While the average error in dependency distances for various types is within 7%, the main source of the error is the first operand of the integer ALU operations. This is due to the changes in the dependency distances that happen when an integer ALU instruction is made as a load/store effective address calculation instruction. In that case, the original dependency distance of the integer instruction is overridden by the distance from the producer of the base address.

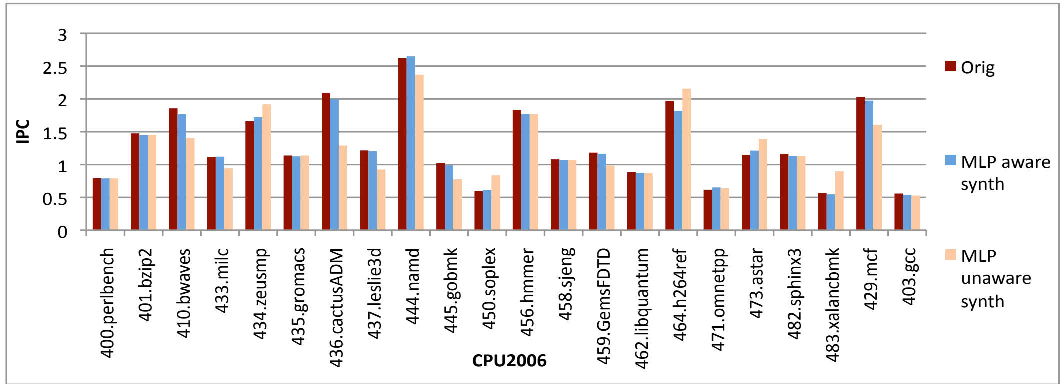
The execution time and power consumption of a benchmark are the first class performance metrics used in computer architecture to assess the performance of a benchmark on a processor/system. Since, we aim at miniaturization of the workloads in terms of the execution time, Instruction-Per-Cycle (IPC) and power-per-cycle are the metrics that we have used to compare the

| Configuration | Machine-A | Machine-B |
|---|---|---|
| Machine widths | Four wide out-of-order | one wide out-of-order |
| RUU size | 128 entries | 16 entries |
| LSQ size | 64 entries | 8 entries |
| Functional Units | 4 Integer ALUs, 1 Integer Mul/Div, 4 FP ALUs and 1 FP Mul/Div | 2 Integer ALUs, 1 Integer Mul/Div, 1 FP ALU and 1 FP Mul/Div |
| Memory | 8B bus and 80 cycles access time | 8B bus and 40 cycles access time |
| Branch Predictor | Bimod with table size 2048 | 2-level Gap predictor |
| IFQ size | 32 entries | 8 entries |
| L1 Data Cache | 32 KB, 4 way, 32B line size, 2 cycle access time | 16 KB, 2 way, 32B line size, 1 cycle access time |
| Unified L2 Cache | 4MB, 4 way, 64B line size, 12 cycle access time | 64 KB, 4 way, 64B line size, 6 cycle access time |
| L1 Instruction Cache | 16 KB, block size 32 Bytes, 1 cycle access time | 16 KB, block size 32 Bytes, 1 cycle access time |

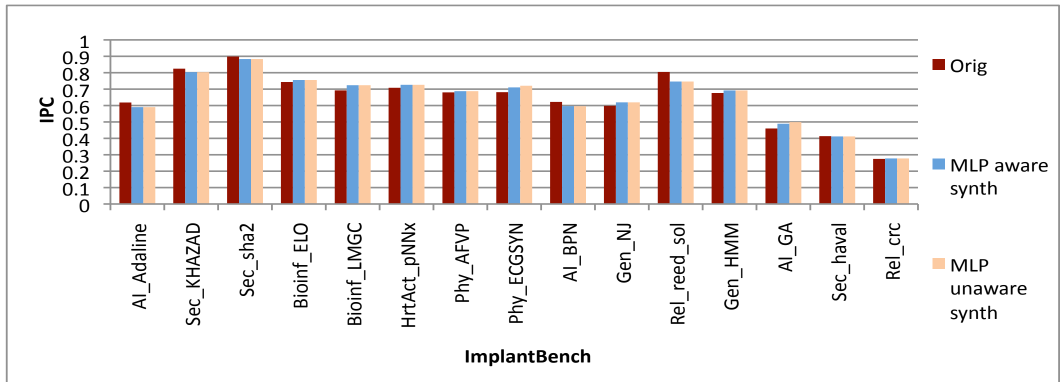
Figure 4.13: Machine configurations used: Machine-A for SPEC CPU2006 and Machine-B for ImplantBench workloads

performance of the original and the synthetic workloads to show the efficacy of the generated synthetics. We determine both the accuracies of using the synthetics as proxies to evaluate the performance of a given microarchitecture and the sensitivity to various micro-architectural design changes. To evaluate the performance of the CPU2006 workloads, we simulate both the original and the synthetic on the simoutorder simulator of SimpleScalar [59] for a typical modern machine configuration (Machine-A) given in the figure 5.10. For the experiments on ImplantBench, we use a typical configuration of an embedded processor (Machine-B) as given in Figure 5.10. These machine configurations are the same as used in some previously published work [49].

As shown in Figure 4.14(a), the synthetics for CPU2006 have an average error of 2.8% and a maximum error of 7.7% for the benchmark 464.h264ref




(a) SPEC CPU2006




(b) ImplantBench

Figure 4.14: Comparison of IPC between the synthetic and the original workloads on single-core system configurations for Alpha ISA

when using the MLP information in the synthetics. While using the previous synthetic generation methodologies (without MLP information) as in previous work [21] [20], the average error in IPC is 15.3% clearly showing the importance of an MLP aware workload generation. It should be noted that the benchmarks 410.bwaves, 456.cactusADM and 483.xalancbmk that have high MLP as shown in Figure 4.7 have decreased error rates while using our MLP aware synthetic clone generation. The importance of modeling the type of the producer instruction for a consumer load instruction while modeling the dependency distances can be explained with 450.soplex as an example. The benchmark 450.soplex solves a linear program using a simplex algorithm and sparse linear algebra and it has a lot of load instructions that are dependent on other load instructions. When this load-load dependency information is not incorporated into the synthetic, this benchmark results in 40% error compared to 2.7% when using this information. For IPC and power results, 15 benchmarks benefit from the 3 automated MLP techniques, 4 benchmarks benefit from all the 4 MLP techniques (with two loops) and 6 benchmarks do not benefit from the MLP techniques. There is an error of 4.43% in IPC when only automatic MLP techniques are used as opposed to 15.3% for MLPunaware. The usage of two loops with manual intervention reduces the IPC error further to 2.8%. The accuracies in the IPCs of the ImplantBench suites are given in Figure 4.14(b). The average IPC error for the workloads in the ImplantBench suite is 2.9% and a maximum error of 7.2%.

To evaluate the power consumption of the synthetic and the original

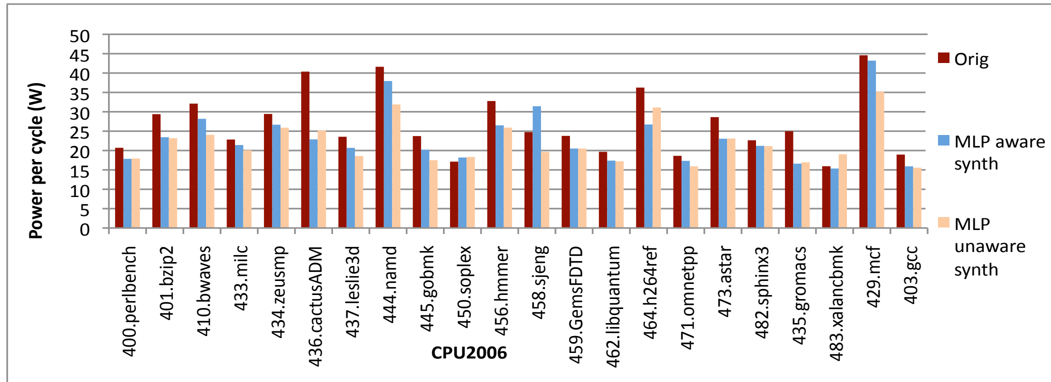



Figure 4.15: Comparison of power-per-cycle between the synthetic and the original workloads for CPU2006 on single-core system configuration for Alpha ISA

workloads, we use the Wattch [60] simulator extension of the SimpleScalar tool set. We use the most aggressive clock gating setting in Wattch and compare the average power consumption per cycle of the synthetic and that of the original workload. Figure 4.15 shows this comparison for CPU2006 and it is to be noted that the average error in power per cycle is 14% and the maximum error is 33% for the benchmark 435.gromacs. The average size of basic blocks in this benchmark is 512 instructions in the original and when we try to miniaturize the benchmark based on the execution frequencies of the basic blocks, we lose some long basic blocks that have a significant impact on the power characteristics. If a user is more concerned about the errors in these benchmarks being so high, the only solution is to compromise on the speedup to achieve higher accuracies by including more basic blocks into the synthetic. The other significant source of error in power-per-cycle for the remaining benchmarks is due to the fact that the long running original applications have higher power consump-

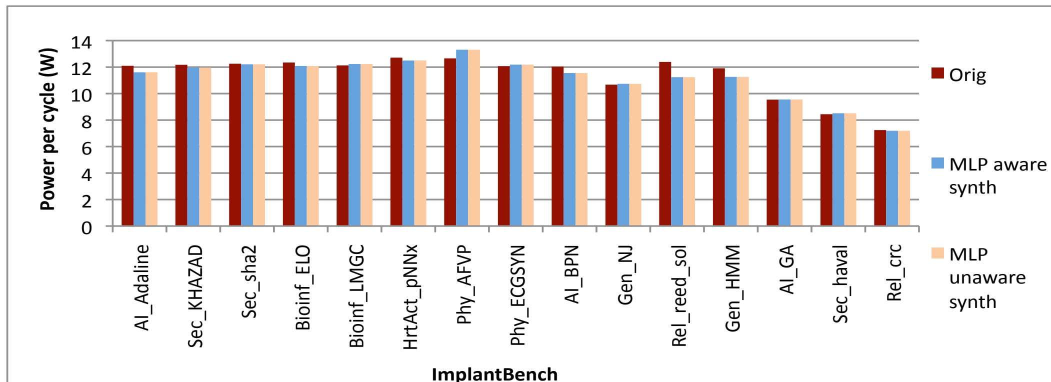
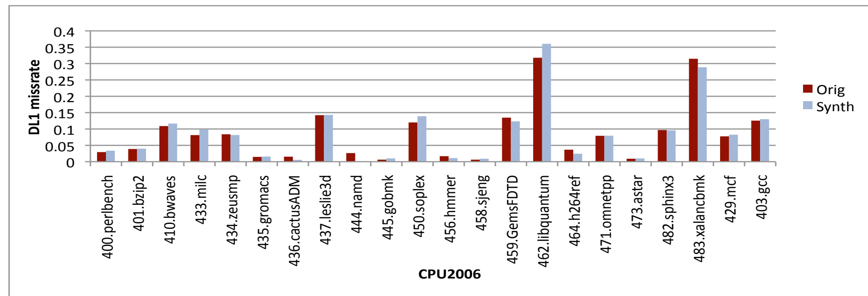


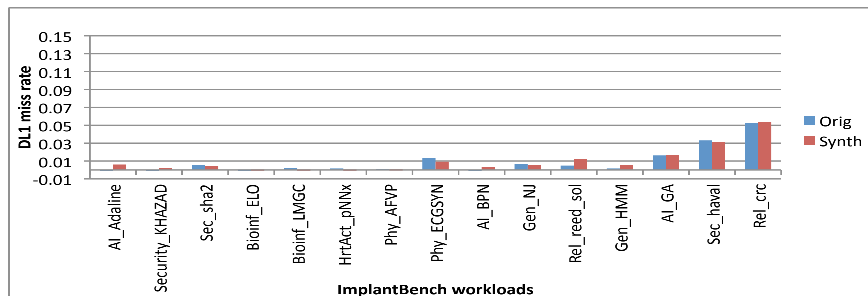

Figure 4.16: Comparison of power-per-cycle between the synthetic and the original workloads for ImplantBench on single-core system configuration for Alpha ISA

tion in the instruction cache than these relatively very small synthetic clones. It can be observed that power consumption is mostly underestimated by the synthetic, bringing up the possibility of correcting it. For the ImplantBench suite, the average error in power consumption is 2.5% and a maximum error of 9.2% which can be seen in Figure 4.16. This error is less than CPU2006, since these workloads have relatively lower dynamic number of instructions.

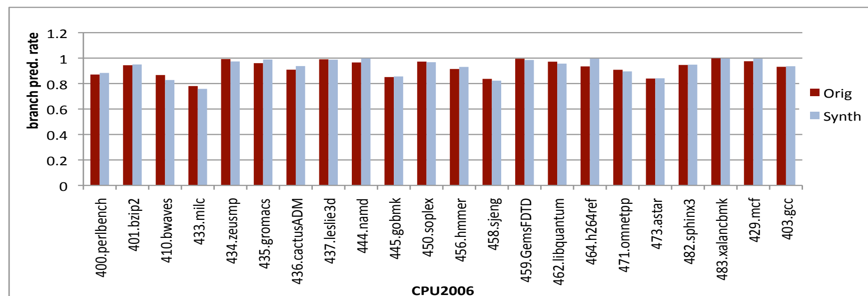
Figures 4.17(a) and 4.17(c) show the error in the miss rates in the Data Level 1 (DL1) cache and the branch misprediction rates of the synthetic compared to the original workload for SPEC CPU2006. The average error in the DL1 hit rate for CPU2006 is 1.06% and that in the branch predictability is 1.7%. The DL1 miss rate comparison for ImplantBench is shown in Figure 4.17(b). Most of the ImplantBench workloads being simple have very low L2 miss rates and high branch predictability. Thus, we only show the accuracies in L2 cache miss rates and branch predictability of the SPEC CPU2006



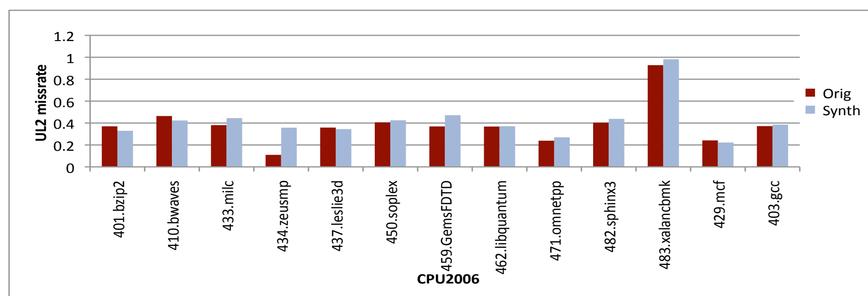

(a) DL1 missrate comparison for CPU2006




(b) DL1 missrate comparison for ImplantBench




(c) Branch misprediction rate comparison for CPU2006




(d) UL2 miss rate comparison for CPU2006

Figure 4.17: Comparison of DL1 missrate, UL2 missrate and branch misprediction rate for CPU2006 and ImplantBench on single-core system configurations for Alpha ISA

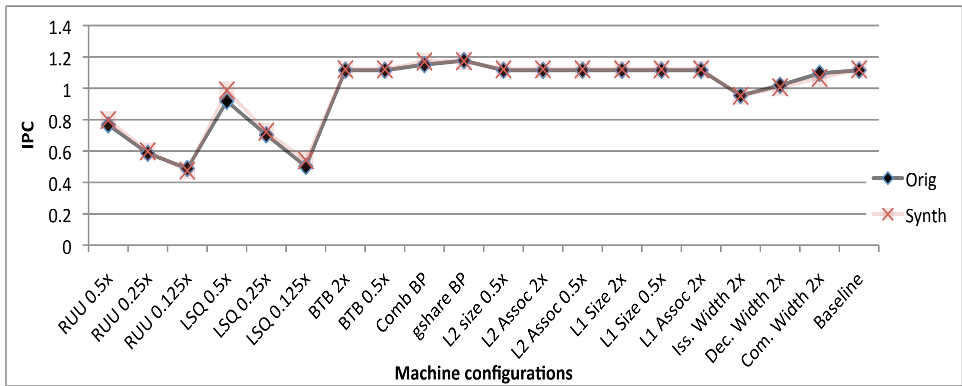
workloads. Figure 4.17(d) shows the error the miss rate in the Unified Level 2 (UL2) cache compared to the originals for those benchmarks that have at least more than 3% of the DL1 accesses reaching UL2. When the number of UL2 accesses are too small, the impact of the accuracy in UL2 miss rates on IPC is also small. The benchmark 434.zeusmp has a high error in the UL2 miss rate compared to the original benchmark. It is a computational fluid dynamics application that is used for the simulation of astrophysical phenomena. This benchmark has a very large data footprint compared to any of the other benchmarks that we have used in this study. It has an almost 1GB of data foot print for the top 100 million instruction simulation point. This benchmark has a miss rate of 8.5% in the DL1 and has a miss rate of only 10% in the UL2. A very detailed modeling of the working set size at a much smaller granularity in terms of the dynamic execution interval is required for this benchmark to capture its overall memory access behavior more precisely than what is dealt with, in this paper. We do not show the error rates in the Instruction Level 1 cache because we found that the number of misses is very small for a typical modern processor configuration.

#### 4.1.2.2 Accuracy in the sensitivity to design changes

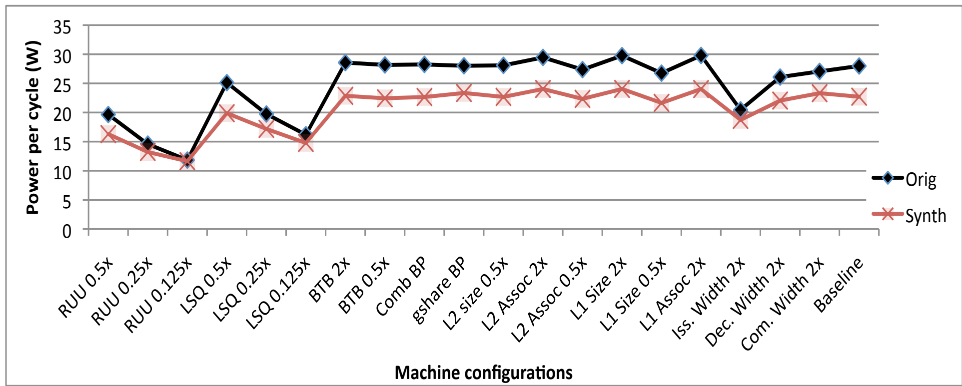
In an architecture study, the accuracy in assessing the performance impact of design changes [61] [62] is more important than assessing the performance for a particular microarchitecture. We evaluate the synthetics to see the sensitivity to various design changes. We study accuracies for changes in

the size of the Register Update Unit (RUU), Load Store Queue (LSQ), Branch Target Buffer (BTB), the type of the branch predictor used, size of the Unified L2 cache, Unified L2 associativity, Data L1 cache size, Data L1 associativity, issue width, decode width and the commit width of the machine. The IPC and power variation for the CPU2006 floating point benchmark 433.milc to design changes are given in the Figures 4.18(a) and 4.18(b) respectively. 433.milc is one of the benchmarks that is very sensitive to the different design changes under study. The IPC and power variation for the CPU2006 benchmark 445.gobmk to design changes are given in Figures 4.19(a) and 4.19(b) repectively. 445.gobmk is one of the benchmarks that has the least of the correlation coefficients in terms of IPC.

The correlation coefficients in IPC between the synthetic and the original for the set of 19 design points as used in Figure 4.18(a) are shown in the Figure 4.20(a) for CPU2006 workloads. The correlation coefficient is directly proportional to correctness of the synthetic in following the trends of the original for the different design points. The average of the correlation coefficient for IPC is 0.95 for all the workloads in CPU2006. Similarly, Figure 4.20(b) shows the correlation coefficients of the synthetic with the original in assessing the power per cycle metric for CPU2006 workloads. The average correlation coefficient for power is 0.98 for all the workloads in CPU2006. Figures 4.20(c) and 4.20(d) show the correlation coefficients for IPC and power consumption for the ImplantBench workloads. The average correlation coefficient for IPC is 0.94 and that for power-per-cycle is 0.97 for the ImplantBench workloads.

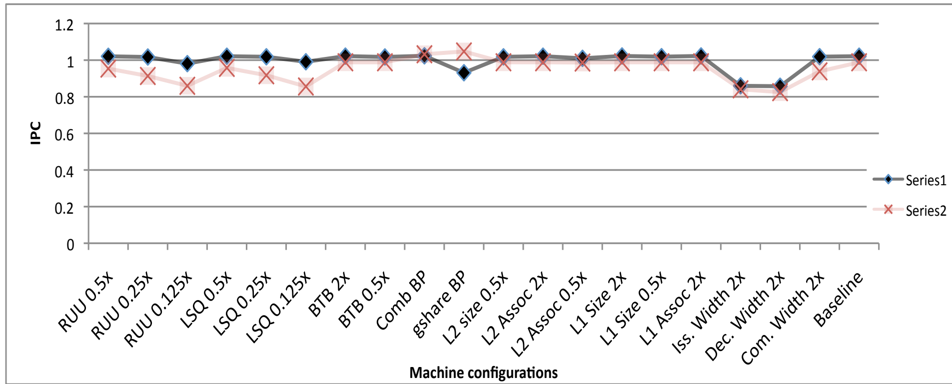



(a) IPC

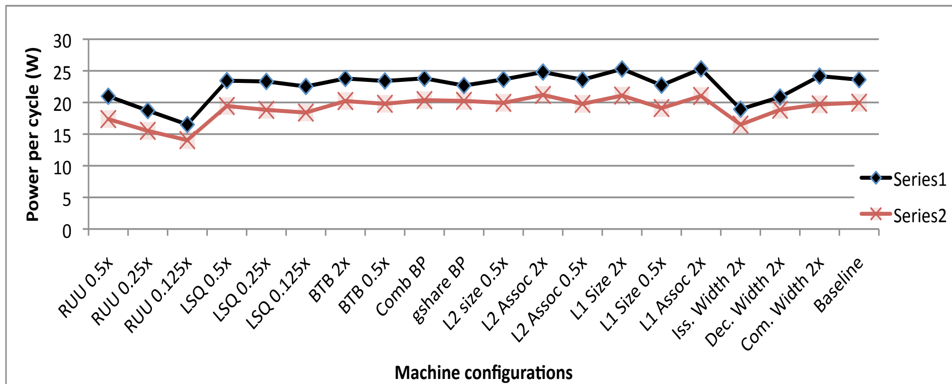



(b) Power-per-cycle

Figure 4.18: Comparison of the variation of IPC and power-per-cycle for 433.milc between the synthetic and the original on single-core system configurations for Alpha ISA

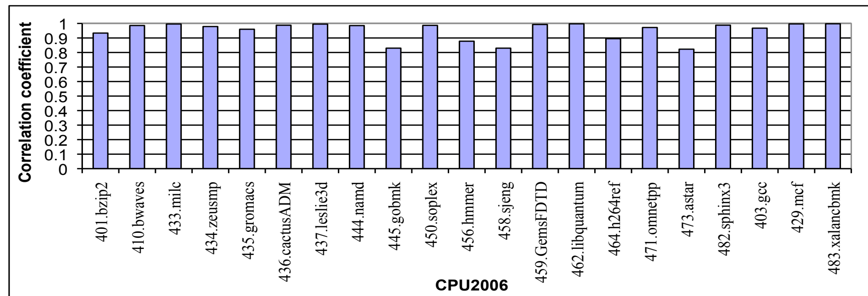



(a) IPC

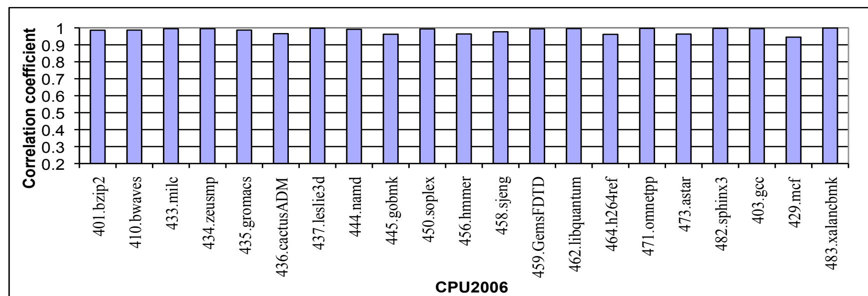



(b) Power-per-cycle

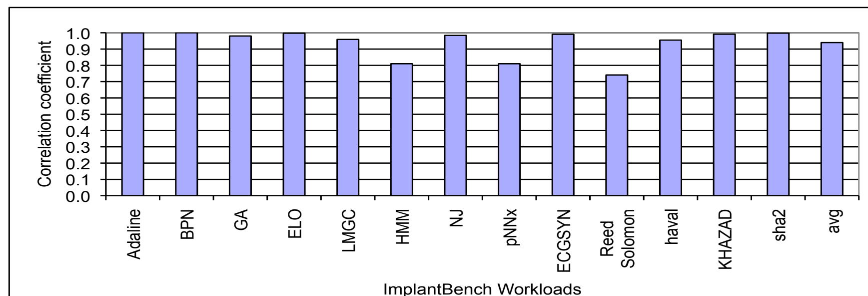
Figure 4.19: Comparison of the variation of IPC and power-per-cycle for 445.gobmk between the synthetic and the original on single-core system configurations for Alpha ISA



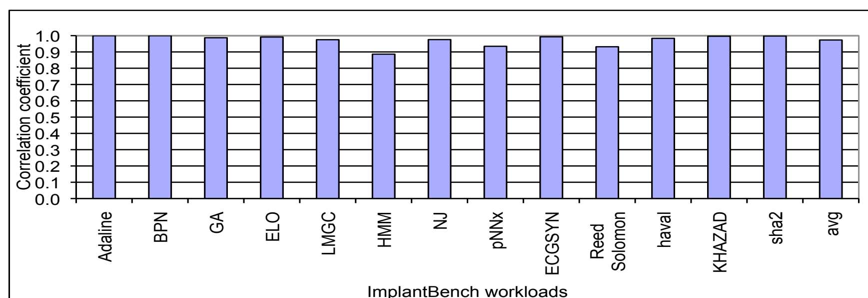

(a) IPC for CPU2006




(b) Power-per-cycle for CPU2006




(c) IPC for ImplantBench




(d) Power-per-cycle for ImplantBench

Figure 4.20: Correlation coefficient between synthetic and the original for design changes on single-core system configurations for Alpha ISA

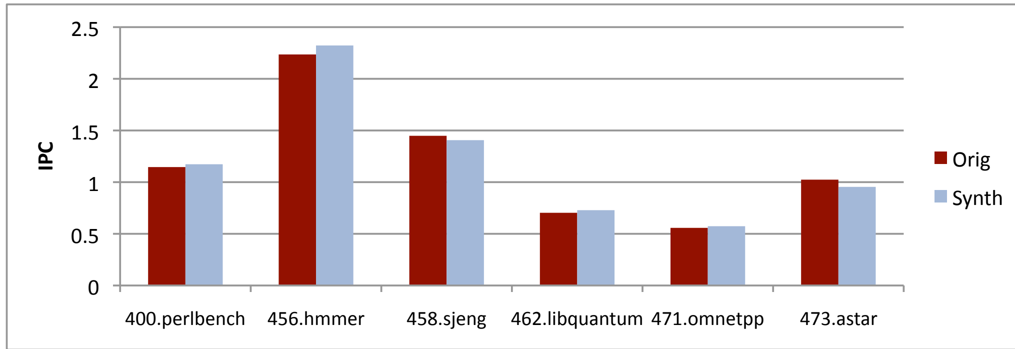



Figure 4.21: Comparison of IPC between the synthetic and the original full runs for CPU2006 on single-core system configuration for Alpha ISA

#### 4.1.2.3 Cloning selected full runs of CPU2006

Previously, we have shown the efficacy of our synthetic benchmark generation methodology by cloning the top simulation point of the different workloads in the SPEC CPU2006 suite. This was due to the prohibitive simulation time that is required to profile the CPU2006 workloads completely for various machine configurations used in this study. To bring out the effectiveness of the methodology for cloning complete runs, we have profiled the complete runs of six workloads that have relatively less simulation time than others and generated synthetics for these workloads. We have compared the performance of these synthetics with the original complete run in terms of the IPC. Figure 4.21 shows the IPC comparison results. The average error is 3.74% in IPC. The table in Figure 4.22 shows the dynamic number of instructions in the full run and that of the synthetic and the speedup that is achieved. An average speedup of 5 million (in terms of instructions) is achieved for the six selected workloads.

| Benchmark | # of Instns in billions (original) | # of Instns in millions (synthetic) | Speedup |
|---|---|---|---|
| 400.perlbench | 184.5 | 0.19 | 936238 |
| 456.hmmer | 2593.1 | 0.29 | 8724843 |
| 458.sjeng | 3187.7 | 0.30 | 10357323 |
| 462.libquantum | 1989.0 | 0.56 | 3495214 |
| 471.omnetpp | 730.0 | 0.12 | 5692522 |
| 473.astar | 966.5 | 0.25 | 3830291 |

Figure 4.22: Speedup information for complete runs of some CPU2006 workloads on single-core system configuration for Alpha ISA

Our methodology is found to be superior in both accuracy and miniaturization compared to simulation points. For the 100 Million instruction simpoints used in the study [1], the average error when using all the simulation points (generated with max number of simpoints=30) is around 5%. If, say a typical benchmark had 15 simulation points, the number of dynamic instructions simulated will be 1500 million instructions. It is very common to use only one simulation point and the error should be much higher when only one simulation point is used. Rather our methodology gives only an error of 3.7% for synthetics of length less than a million instructions. This could be attributed to the reason that these synthetic instruction sequences are constructed based on characteristics of the whole program, rather simpoint methodology is forced to leave some characteristics to be able to choose one contiguous dynamic instruction chunk.

## 4.2 Workload cloning for Multicores

Amongst the various applications that target Multicore systems, multithreaded applications are becoming increasingly common. Multithreaded applications have a varied set of characteristics in terms of the sharing patterns etc that have an impact in the performance of the shared caches, the interconnection network, coherence logic and DRAM. We use all the characteristics as specified in the abstract workload model specified in 3.1 in Chapter 3. The cloning methodology is the same as specified for the single threaded applications, consisting of a profiler that is used to get the characteristics of the long running original applications, which are fed to the synthetic benchmark generation framework to generate clones. These clones are compared with the original applications based on both micro-architecture dependent and independent characteristics to evaluate their representativeness to their original counterparts.

To show the efficacy of the proposed multithreaded synthetic benchmark generation to clone the multithreaded applications, clones are generated for the benchmarks of the PARSEC suite. The PARSEC benchmark suite is a collection of applications targeting a shared memory multicore systems. Most of these applications are representative of the workloads that will be running on multicore desktop and server systems. The PARSEC suite also includes many emerging workloads that are expected to be more commonly used in the future than today. The benchmarks in the suite are not restricted to any single application domain, rather they are quite varied in terms of that usage.

For example, PARSEC includes applications from the finance domain namely Blackscholes and Swaptions, that target option pricing using partial differential equations and a portfolio of swaptions respectively. The suite includes data mining applications like Streamcluster, Freqmine and Ferret targeting data clustering algorithms, itemset mining and content similarity search server respectively. The suite includes a workload Canneal that is used extensively in the chip design industry for optimizing routing cost of a chip design using simulated annealing. Compression algorithms are included like the benchmark Dedup. Image processing, video encoding and real time ray tracing algorithms are included, which are Vips, X264 and Raytrace respectively in the suite. A few applications from the physics domain like fluid dynamics for animation (Fluidanimate), body tracking of a person (Bodytrack) and simulation of face movements as in Facesim are also included. Further in this chapter, the first step in cloning, which is the benchmark characterization is explained followed by clone generation and analysis.

### 4.2.1 Benchmark Characterization

This subsection elaborates on how each of the different metrics of the abstract workload model are captured and also provide the characterized data for the PARSEC workloads. The full system simulator called Windriver Simics is used along with the processor, memory and interconnection network simulation model called GEMS from Wisconsin Madison University for profiling the workloads. An instruction trace and a memory access address trace

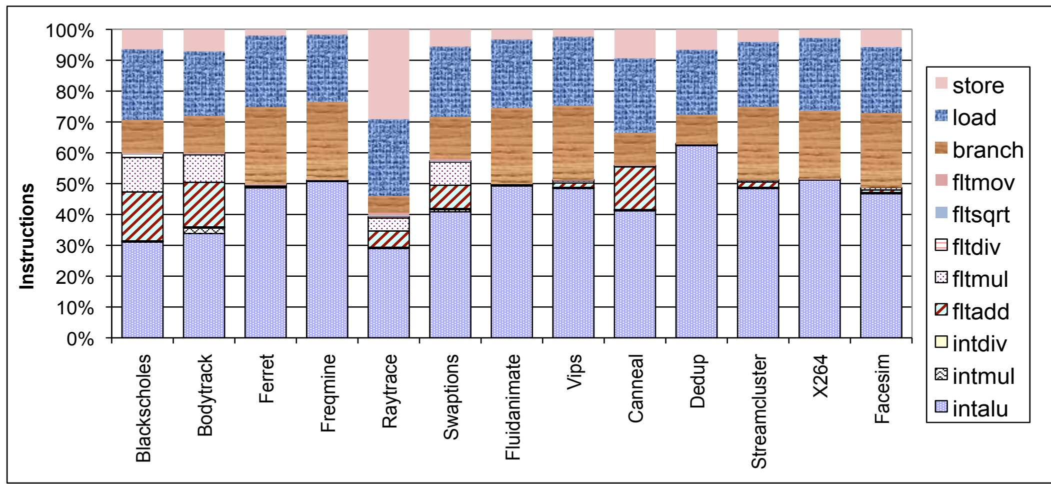



Figure 4.23: Instruction mix distribution for a 8-threaded version of various PARSEC workloads

are captured to record most of the significant characteristics. The instruction mix, register dependency distance distribution and the various synchronization characteristics are recorded based on the instruction trace. The figure 4.23 shows the distribution of the instruction into various categories of instruction types for the Parsec workloads. It can be noted that most of the PARSEC applications are quite compute intensive in terms of the integer operations. Only a few workloads have a considerable amount of floating point operations namely Blackscholes, Bodytrack and Canneal. Most of the Parsec workloads have 20% to 25% load operations and mostly less than 10% stores operations. It can be noted that Raytrace is the only application that has a considerable amount of stores of 29%, which is even greater than the percentage load operations in Raytrace. Most of these workloads have a high percentage of branch instructions. The basic block sizes vary between 4 instructions to at most 18

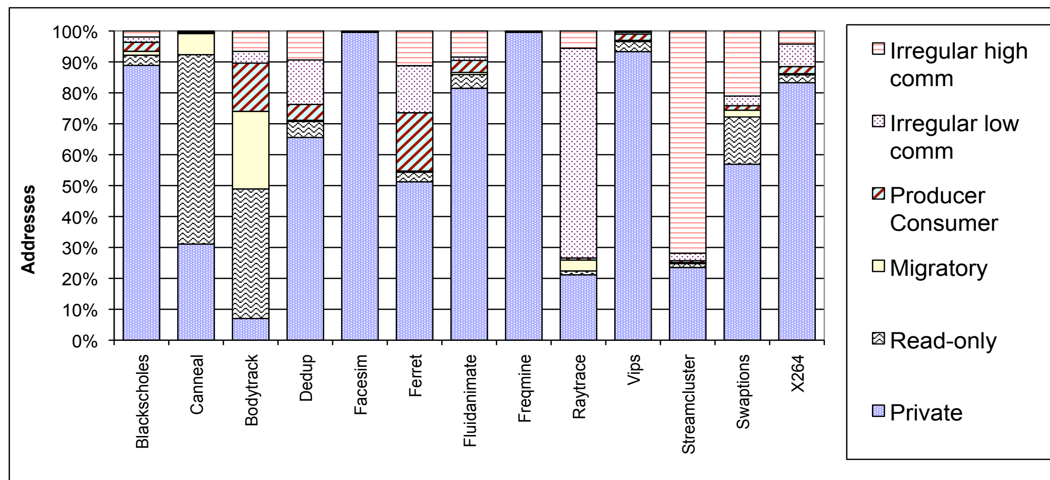



Figure 4.24: Spatial distribution of the accessed memory addresses into sharing patterns for various a 8-threaded version of PARSEC workloads

instructions, showing that these workloads will be quite sensitive to the branch predictability of a machine configuration.

For the synchronization characteristics, the calls to the system call functions pthread_mutex_lock and pthread_mutex_unlock are recorded using the instruction trace. The number of instructions between the lock and unlock calls is recorded and is averaged to be the size of the critical section. Inside the lock and unlock function calls, the mutex object address to which the exclusive locks and unlocks happen are also recorded. This information gives an idea about the conflict density in across the synchronization events happening in various threads of the workload. All the synchronization metrics are recorded relative to the dynamic number of instructions to be able to replay it in the synthetic to clone these workloads.

The memory access trace is post processed to record the memory access strides. The same memory access trace is then post processed to classify the different addresses that are accessed into various categories of producer-consumer, migratory, read-only and irregular sharing patterns. Each address is examined to see if the accesses follow any of the three major sharing patterns and are added to that particular class. If the addresses do not show any conceivable pattern, they are classified to be following an irregular access pattern. Based on the spatial distribution of this data, i.e, the number of addresses that belong to each of the sharing pattern classes, the data footprint of the synthetic will also be distributed. The category irregular is once again broken into two classes, one which has a high communication overhead and the one that has low communication overhead based on the fact whether the data is accessed by more than one processors within a given number of accesses. The figure 4.24 shows this spatial distribution of the accesses data in terms of various sharing patterns. Based on this data, we can see that the private data footprint of applications like Blacksholes, Facesim, Fluidanimate, Freqmine, Vips and X264 are considerably high compared to the shared data footprint. The workloads Canneal, Bodytrack and Streamcluster are the only workloads where the shared data footprint is higher than the private data footprint. In the cases of Canneal and Bodytrack the read-only shared data content is considerably high compared to other workloads. Bodytrack also has a considerable amount of data that are classified into migratory pattern.

Then, based on the number of accesses to each of these addresses, the

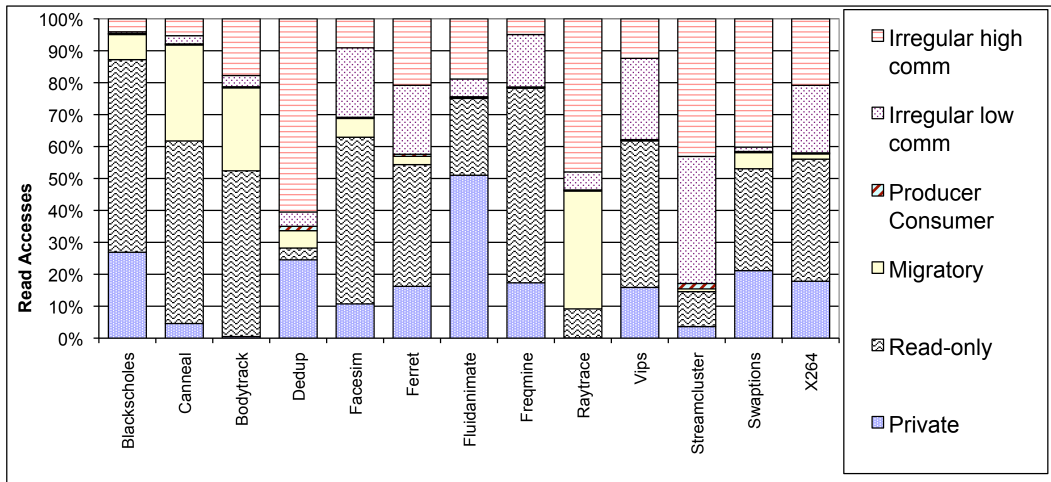



Figure 4.25: Temporal distribution of the various memory accesses in a 8-threaded version of PARSEC workloads into different sharing patterns for reads

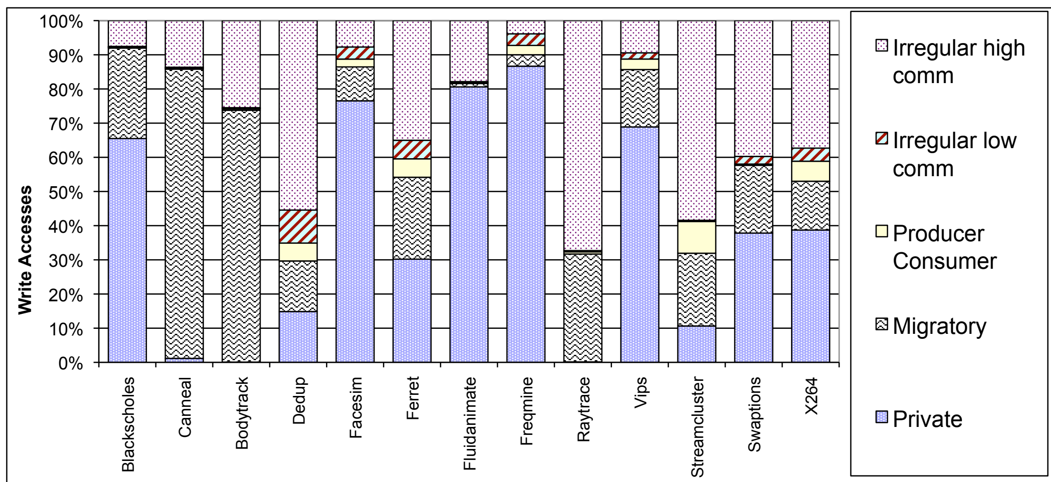



Figure 4.26: Temporal distribution of the various memory accesses in a 8-threaded version of PARSEC workloads into different sharing patterns for writes

proportion of load accesses and store accesses to each of these sharing patterns is determined. This information is much more important than the spatial distribution of the data into different sharing patterns. Many workloads may not have a huge shared data footprint, but can have more shared data accesses than private data accesses. The Figures 4.25 and 4.26 show this temporal distribution of the accesses into various sharing patterns. Good examples of a workloads with a low shared data footprint, but with a high amount of accesses to shared data are Facesim, Freqmine and Vips. The memory level parallelism, control flow predictability metrics are recorded using the information provided by the processor and memory models in the GEMS infrastructure.

### 4.2.2 Results and Analysis

The characterized data for the PARSEC applications are fed to the synthetic benchmark generation framework to generate clones for the different PARSEC applications. To be able to assess the efficacy of the cloning methodology for multithreaded applications, the next step is to assess the representativeness of the generated clones to that of their original counterparts. This is accomplished by comparing the microarchitecture dependent and independent characteristics of the clones to the original applications.

#### 4.2.2.1 Accuracy in assessing performance

An typical 8-core modern system configuration is used to get the microarchitecture dependent metrics for comparison. The machine configuration

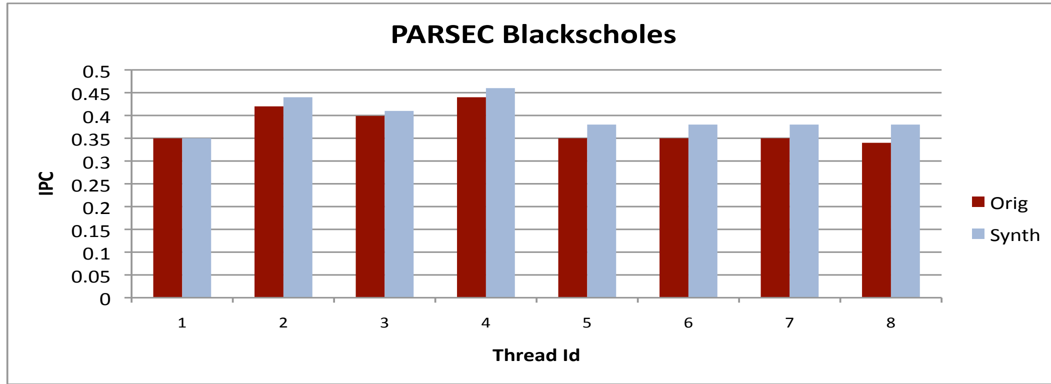



Figure 4.27: Comparison of IPC between original and synthetic for various threads of benchmark Blackscholes in the PARSEC suite on a 8-core system configuration

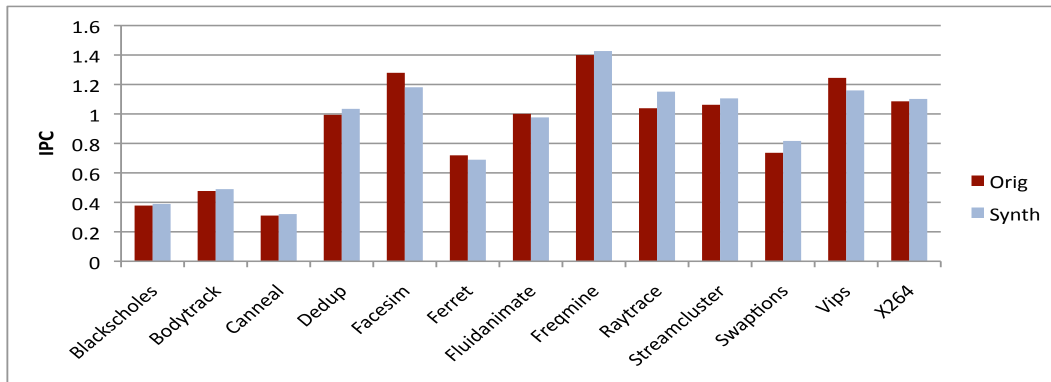



Figure 4.28: Average Error in IPC between synthetic and original for the PARSEC benchmarks on a 8-core system configuration

that is used has a 8GB DRAM, 32KB, 4-way, 1 cycle latency L1 cache, a 4MB, 8-way 8-banked L2 cache, 32 MSHRS, 64 entry reorder buffer size with a machine width of 4 instructions per cycle. The branch predictor used is a YAGS branch predictor with a 11 bit addressable Pattern History Table (PHT). The configuration has a 512 entry Branch Target Buffer (BTB), 3 integer ALUs with one integer divide, 2 floating point ALUs with one FP multiply and one FP divide units. The topology that is used to connect the various memory components is a hierarchical switch. The original and the synthetic workloads are run on this configuration and the execution time in terms of number of cycles is recorded. Based on the number of instructions executed, the Instruction-Per-Cycle is computed for each of the threads. The Figure 4.27 shows the comparison of IPC between the synthetic and the original for various threads of a randomly chosen PARSEC benchmark Blackscholes. The average error when considering all the threads is 2.9% for Blackscholes. The Figure 4.28 shows the comparison of IPC between original and synthetic averaged over all the threads. The error in the IPC when averaged over all the 13 benchmarks is 4.87% with a maximum error of 10.8% for the workload Raytrace. It should be noted that Raytrace is unique in terms of the number of writes that it does to memory as previously discussed about the instruction mix of Raytrace.

Other microarchitecture metrics like the miss rates in L1 and the branch prediction rates are also compared between the original and synthetic workloads for various PARSEC applications. The Figure 4.29 shows the comparison of the L1 cache missrate between the original and the synthetic applications.

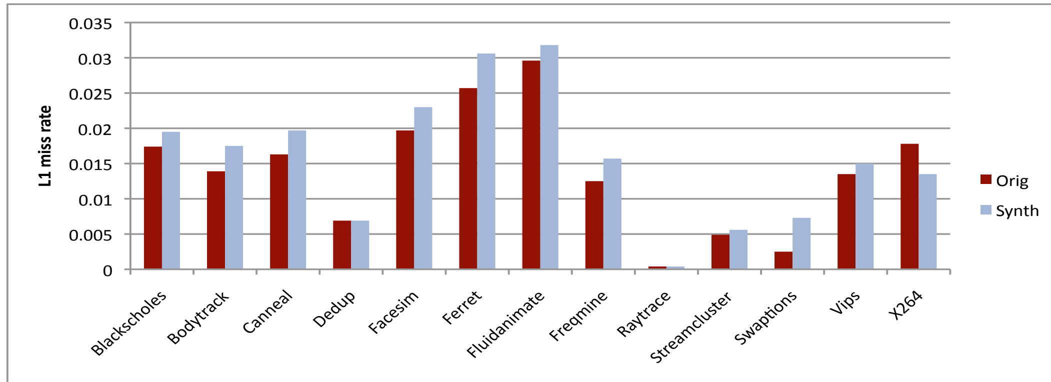



Figure 4.29: Comparison of L1 missrate between the synthetic clones and that of the original PARSEC workloads on a 8-core system configuration

Since the L1 missrates for many of the applications are quite small, the average in the L1 hit rate is computed to assess the representativeness. The average error in the L1 hit rate across all the PARSEC workloads is 0.67% with a maximum of 1.83% for the application Facesim. The Figure 4.30 shows the comparison of the branch prediction rate between the original and the synthetic applications. The average error in the branch prediction rate is 0.52%.

#### 4.2.2.2 Accuracy in assessing power consumption

To see how effectively the synthetic benchmarks can be used as proxies for the original PARSEC workloads for power modeling, the power consumption of various workloads is compared to that of their synthetic clones. The figure 4.31 shows the comparison of the total system power consumption between the original and synthetic workloads. To show how effectively the synthetic models the execution behavior in each of the system components, the

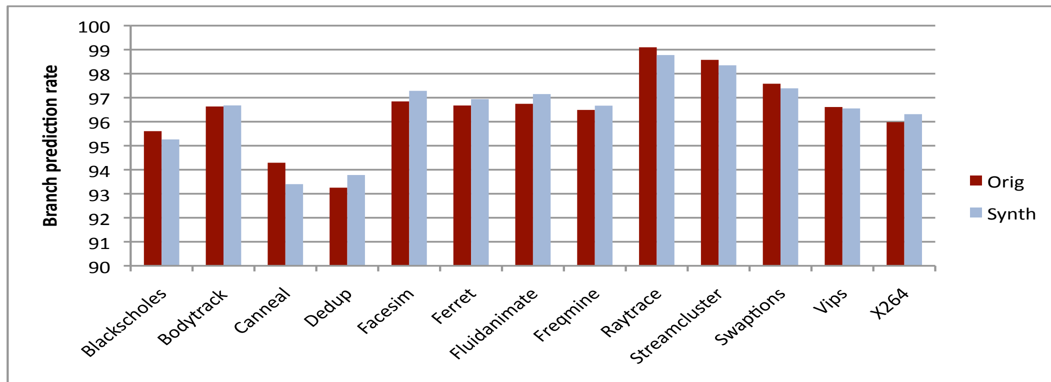



Figure 4.30: Comparison of branch prediction rate between the synthetic clones and that of the original PARSEC workloads on a 8-core system configuration

same figure is also annotated with the breakdown of the power consumption in the different system components for both the synthetics and the originals. The average error in the total power consumption between the synthetic and the original workloads is 2.73% with a maximum error of 5.5% for the application Raytrace. It should be noted that Raytrace is the application that also has a maximum error in performance and in most of the cases the power consumption of workloads are quite proportional to their performance.

#### 4.2.2.3 Accuracy in assessing sensitivity to design changes

In computer architecture, estimating the performance of a workload on one machine configuration is less important comparing to the ability to estimate the sensitivity of a workload's performance to various design changes. Thus, it is important to evaluate the representativeness of the synthetic workloads to their original counterparts in terms of their sensitivity to design

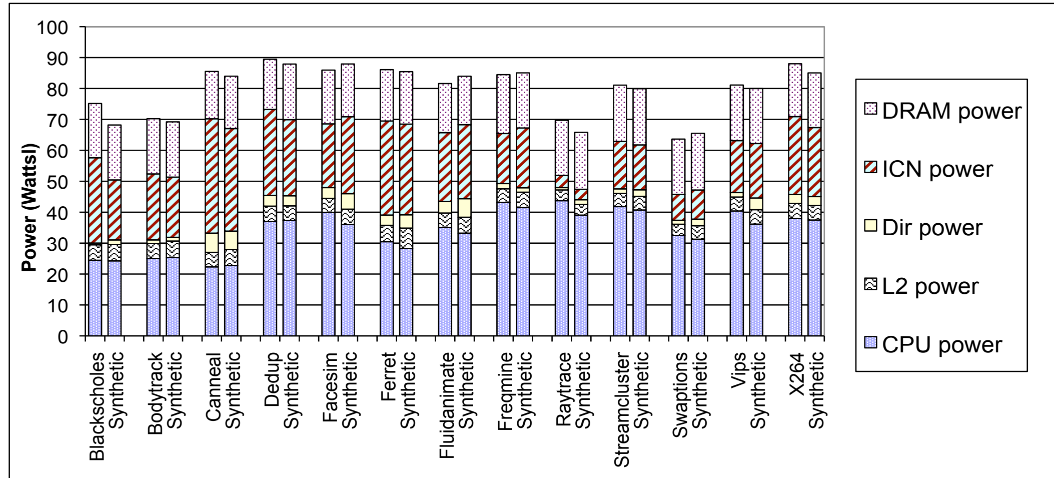



Figure 4.31: Power-per-cycle for various PARSEC workloads along with a breakdown of the power consumption in various components on a 8-core system

| Parameter | System – A | System – B | System – C |
|---|---|---|---|
| No. of cores | 8 | 8 | 8 |
| DRAM | 16 GB | 8 GB | 4 GB |
| L1 cache | 64 KB, 4 way, 2 cycles | 32 KB, 4 way, 1 cycle | 16 KB, 2 way, 1 cycle |
| L2 cache | 8 MB, 16 way, 16 banks | 4 MB, 8 way, 8 banks | 2 MB, 8 way, 8 banks |
| L1, L2 MSHRs | 48 | 32 | 24 |
| ROB | 128 | 64 | 32 |
| Mach-width | 8 | 4 | 2 |
| Branch pred. | YAGS, 12 bit PHT | YAGS, 11 bit PHT | YAGS, 10 bit PHT |
| BTB size | 1024 | 512 | 256 |
| Int ALUs | 4 ALU, 2 Int div | 3 ALU, 1 Int div | 2 ALU, 1 Int div |
| Topology | Crossbar | Hierarchical switch | Hierarchical switch |
| FP ALUs | 2 ALU, 2 Mul, 2 div | 2 ALU, 1 Mul, 1 div | 1 ALU, 1 Mul, 1 div |

Figure 4.32: Multicore machine configurations used to evaluate the accuracy in assessing the impact of design changes by the synthetic in comparison to original PARSEC workloads

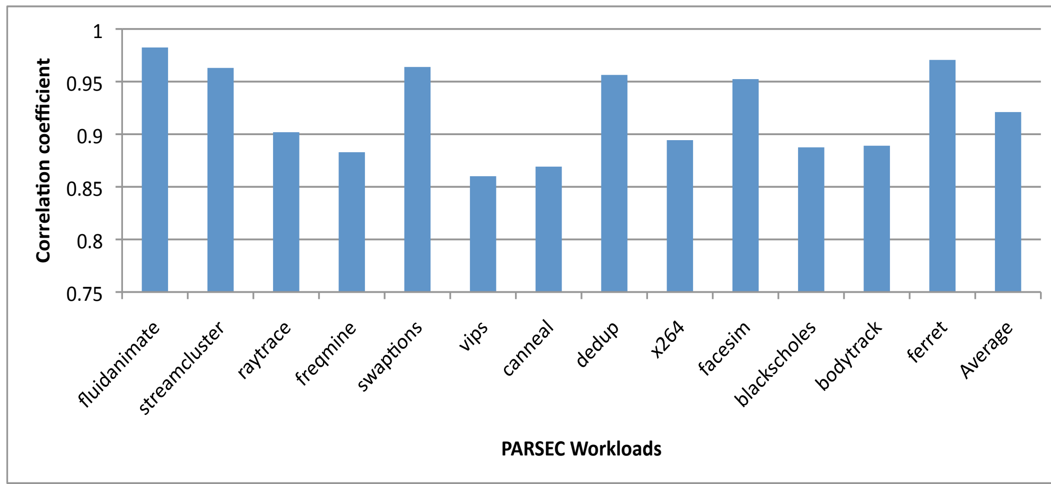



Figure 4.33: Correlation coefficients for the sensitivity to design changes between the synthetic and the original using various multicore machine configurations for the workloads in the PARSEC suite

changes. To accomplish this, we use three system configurations as shown in Figure 4.32 to analyze performance variations for design changes. The three system configurations have varying microarchitecture settings in terms of cache sizes, machine width, branch predictor, topology of the interconnection network etc. To make more design points, the system configuration B was mutated as following to form nine more configurations: 0.5X L1 cache size, 2X DRAM size, 2X L2 cache size, 2X machine width and ROB size, 2X PHT size for branch predictor, more ALUS, ICN crossbar and 0.5X L2 cache size. The performance of the workloads on each of these configurations were recorded for both original and synthetic workloads using the metric IPC. The correlation between the trends followed by original and the synthetic is determined by finding the correlation coefficient.

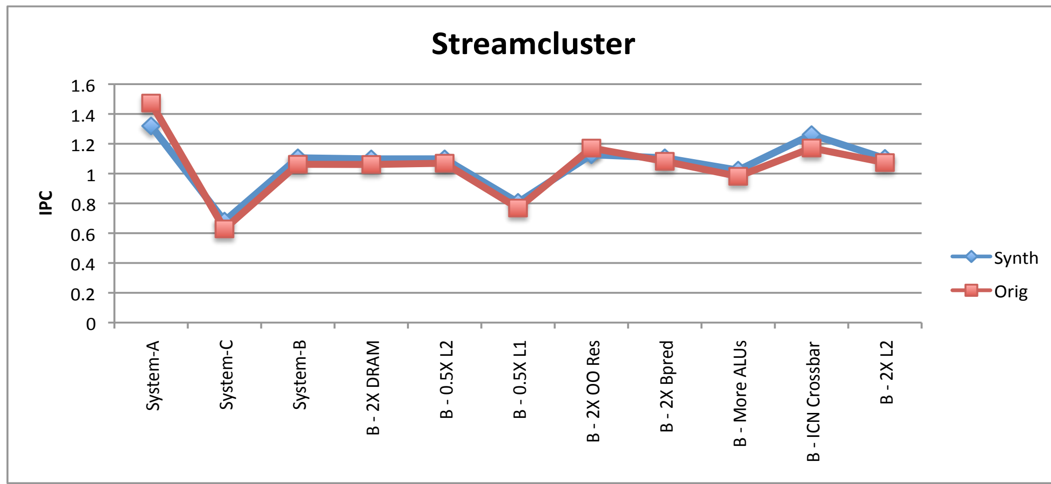



Figure 4.34: Comparison of sensitivity to design changes using various multicore machine configurations for the workload Streamcluster in PARSEC suite

The Figure 4.33 shows the correlation coefficient between the trends followed by the synthetic and the original workloads for the various design changes. The average of the correlation coefficient for all the workloads in the PARSEC suite is 0.92. The Figures 4.35 and 4.34 show the comparison of sensitivity to design changes using various multicore machine configurations by mutating system configuration B for the randomly chosen workloads Raytrace and Streamcluster in PARSEC suite respectively. This brings out the utility value of the synthetics to be used as proxies for the PARSEC workloads for the most invasive design space exploration studies.

#### 4.2.2.4 Speedup achieved in using the synthetics

The most important advantage of using the synthetic proxies over the long running original PARSEC applications is the speedup achieved in simu-

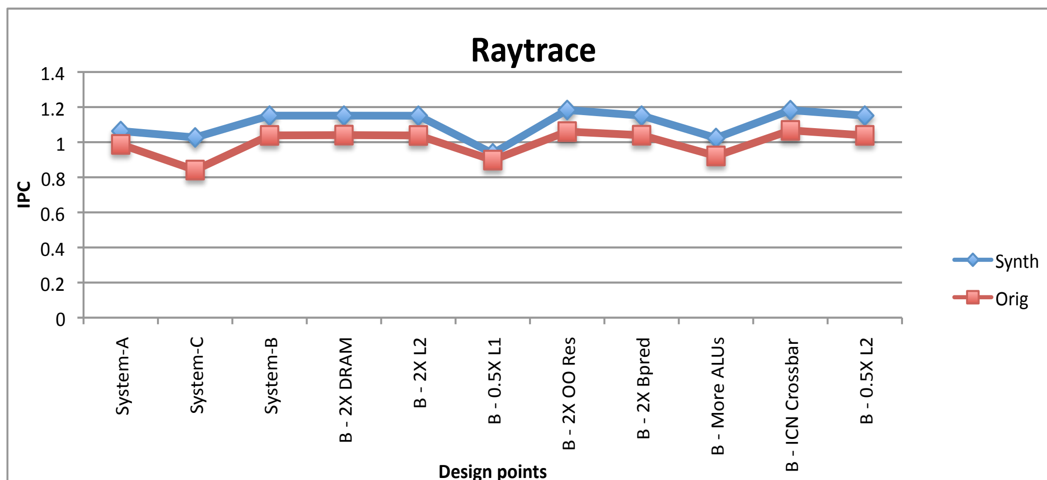



Figure 4.35: Comparison of sensitivity to design changes using various multi-core machine configurations for the workload Raytrace in PARSEC suite

| Benchmarks | No. of Instructions (Millions) | | Speedup |
|---|---|---|---|
| | Synthetic | Original | |
| Blackscholes | 3.80 | 13028 | 3429 |
| Bodytrack | 5.96 | 56918 | 9552 |
| Canneal | 8.29 | 10484 | 1264 |
| Dedup | 4.41 | 8428 | 1912 |
| Facesim | 7.45 | 1151026 | 154450 |
| Ferret | 4.42 | 58398 | 13227 |
| Fluidanimate | 7.81 | 72669 | 9300 |
| Freqmine | 6.93 | 5588 | 807 |
| Raytrace | 4.02 | 223560 | 55585 |
| Streamcluster | 4.51 | 52527 | 11641 |
| Swaptions | 7.01 | 11020 | 1572 |
| Vips | 7.50 | 37135 | 4952 |
| X264 | 3.75 | 13001 | 3465 |

Figure 4.36: Speedup achieved by using the synthetic proxies over the full run of the PARSEC workloads on a 8-core system configuration

lations when using these miniaturized clones. The table in Figure 4.36 shows the speedup achieved in terms of reduction in the number of instructions when using the synthetic proxies over the original PARSEC applications. The synthetic benchmarks generally have three to eight million instructions, when the original workloads have a few thousand million instructions. The speedup achieved is at least four orders of magnitude to a maximum of about 5 orders of magnitude for the application Facesim.

### 4.2.3 Proxies for Proprietary Applications

As described before, one of the applications of our cloning methodology is to disseminate proprietary applications to processor architects for better performance analysis of target workloads. The most important feature about our synthetic benchmark generation is the claim that they cannot be reverse engineered to find any information about the original applications. There has been a lot of related research in terms of code obfuscation techniques where one tries to obfuscate the code to hide the intellectual property in applications when distributing binaries to make it harder to reverse engineer. Though these techniques have been researched a lot, most of them suffer from increasing the execution time of the program or code size in some way. In our synthetic benchmark generation case, it is to be noted that only the performance characteristics of the workloads are distilled into the synthetic benchmark and all the other higher level information about the original workload are lost, making any kind of reverse engineering quite meaningless. We majorly obfuscate the

organization of the data in the original applications, which is popularly called 'Data storage obfuscation' by converting all the data structures into a single one dimensional array. This removes the information like the presence of a class (in C++ or Java) and the presence of a structure in C. Also the name of the identifiers, function calls etc are also not passed on to the synthetic. This final piece of code is independent of most of the higher level constructs of loops, function calls and other possible organization in the code. The most relevant information related to reverse engineering that is passed on the synthetic are instruction sequence information and control flow behavior. But, it is to be noted that the instruction sequence information is in terms of instruction types than real instruction at all. For example, in our synthetic benchmark generation, all single cycle latency integer operations are grouped into one category, and are generated as just integer adds. Regarding the control flow information, reverse engineering any algorithm just based on the basic block size or branch predictability is quite impossible. Thus, the proposed synthetic benchmark cloning is quite robust to disseminate proprietary applications without the need to worry about being reverse engineered for information about the original applications.

# Chapter 5

# Power Virus Generation

A power virus for a modern multicore system has to stress different parts of the system in such a way that the overall power consumption is maximized. As mentioned already, keeping all the components of a system simultaneously active is not possible. For example, to be able to stress the DRAM of a system, the processor may have to stall for many cycles for those long-latency loads to complete. Any program that is completely memory-bound cannot consume much power in the cores and a program that is completely compute-bound cannot consume much power in the memory, caches or the interconnection network. Thus, the power virus has to strike the right balance between stressing different power consuming components in the system to be able to maximize the overall power. There are many latency hiding mechanisms implemented throughout a modern computer system starting from the out-of-order execution circuitry in the cores, various buffers, the miss status handling registers in the caches, pipelining in the interconnection network to various optimizations implemented in the DRAM controller and all these numerous features should be exploited to the right extent by this power virus to achieve maximum overall power. Mainly to avoid the need to model all these complex interactions, we propose the use of a black-box approach that employs

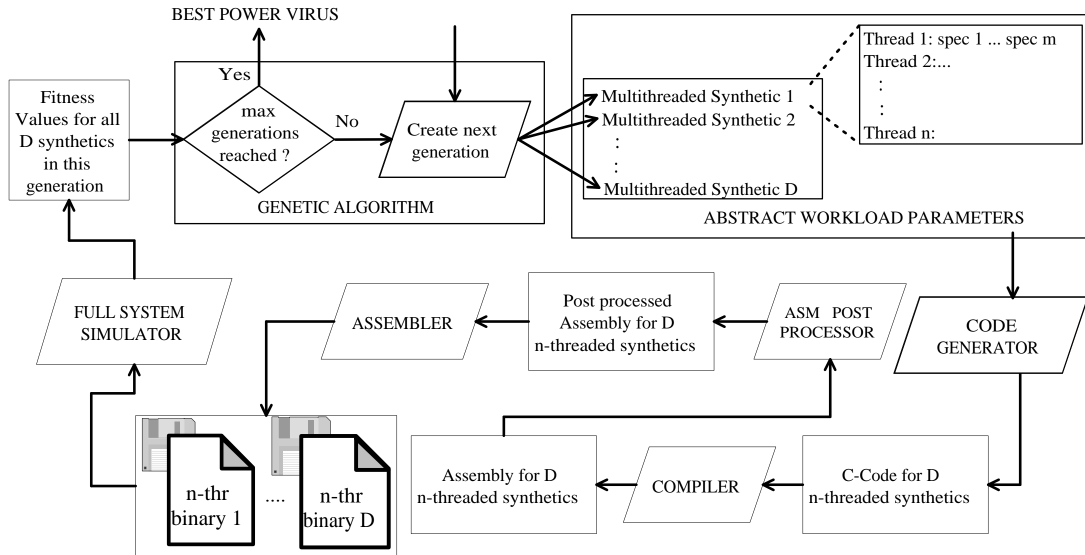



Figure 5.1: Multithreaded power virus generation framework

a machine learning based search technique along with the previously proposed multithreaded workload generator to automatically search for a power virus given a system configuration. Our power virus generation framework to generate power viruses for multicores is called Multicore MAx POwer (MAMPO) and the same for single-core systems which uses only a subset of the abstract workload parameters is called SYstem-level Max POwer (SYMPO).

The main components of the MAMPO framework are, i) the machine learning technique employed in the framework, Genetic Algorithm (GA), ii) the abstract workload model along with the code generator, compiler and the assembly post processor iii) the full system simulator with detailed power models used to evaluate the power consumption of the multithreaded synthetics. Figure 5.1 shows the flowchart of the power virus generation framework. The

Genetic Algorithm (GA) [63] generates the parameter values for the potential candidates for the synthetic power virus case as it iteratively searches through the abstract workload space. These generated abstract workload characteristics are fed to the code generator that generates a multi-threaded synthetic workload based on these specified characteristics. This multi-threaded C code is then translated to direct assembly code with the help of a compiler. At times, the compiler introduces some spurious stack operations amidst the set of instructions that are incorporated as embedded assembly and this assembly code has to be post processed to remove such unnecessary instructions and then it is further compiled into a binary. This multi-threaded binary is then executed on a full system simulator with cycle accurate power models for various system components to evaluate the power consumption of the generated synthetic on the system configuration under study. These power consumption numbers are fed back to the Genetic Algorithm to intelligently choose the next set of potential candidates for the power virus and this process iteratively continues until the search converges to find the best power virus for a given system configuration. Each of the components of this framework will be explained in detail further in this Section.

## 5.1 Abstract Workload Model

The effectiveness of this kind of power virus generation framework lies mainly in the efficacy of the abstract workload space that is being searched through by machine learning. Firstly, the dimensions of this abstract workload

| # | Knob name | Knob range | # Thr classes | Category |
|---|---|---|---|---|
| 1 | Number of threads | 1, 4, 8, 16, 32 | 4 | Parallelism |
| 2 | Thread class & proc. assignment | 1, 2, … 12 | - | Shared data access pattern and communication characteristics |
| 3 | Percent accesses to shared data | 10, 30, 50, 60, 70, 90 | 4 | |
| 4 | Shared memory access strides in two buckets | 0, 4, 8, 12, 16, 32, 64 | 4 | |
| 5 | Coupled load-stores | True/false | 1 | |
| 6 | Private memory access strides in two buckets | 0, 4, 8, 12, 16, 32, 64 | 4 | Private data access pattern |
| 7 | Working set size (# loop iterations before array ptr. reset) | 1, 10, 20, 40, 100, 200 | 4 | Memory footprint |
| 8 | Memory Level Parallelism (MLP) | 1, 2, 3, 4, 6 | 4 | Memory level parallelism |
| 9 | MLP frequency | High, low | 1 | |
| 10 | Average basic block size | 10, 20, 30, 50, 100 | 1 | Control flow predictability |
| 11 | Average branch predictability | 0.8, 0.86, 0.92, 0.96, 0.98, 0.99, 1.0 | 4 | |
| 12 | INT ALU proportion | 0 - 4 | 4 | Instr. mix |
| 13 | INT MUL proportion | 0 - 4 | 4 | |
| 14 | INT DIV proportion | 0 - 4 | 4 | |
| 15 | FP ADD proportion | 0 - 4 | 4 | |
| 16 | FP MUL proportion | 0 - 4 | 4 | |
| 17 | FP DIV proportion | 0 - 4 | 4 | |
| 18 | FP MOV proportion | 0 - 4 | 4 | |
| 19 | FP SQRT proportion | 0 - 4 | 4 | |
| 20 | LOAD proportion | 0 - 4 | 4 | Instruction mix, data access pattern and communication characteristics |
| 21 | STORE proportion | 0 - 4 | 4 | |
| 22 | Register dependency distance (number of instructions) | 1, 2, 4, 8, 16, 32, 64 | 4 | Instr. level parallelism |
| 23 | Random seed | 1, 2, 3 | 1 | Code alignment |

Figure 5.2: Abstract workload space searched through by the machine learning algorithm including the range of values used for each of the different knobs

space should be as much microarchitecture independent as possible to enable this framework to be able to generate the best power virus for different types of microarchitectures. It is the job of the machine learning algorithm to take care of tailoring the parameters of the abstract workload model to maximize the power consumption for a given microarchitecture based on power estimates provided by the simulator for this microarchitecture under study. But, it is also important that these dimensions of the abstract workload space be robust enough to be able to vary the execution behavior of the generated workload in every part of the multicore system. It is to be noted that the dimensions should also not be too many as it could also result in a situation where the search would never converge due to a state space explosion. The characteristics of real-world programs that affect performance and in turn the power consumption are carefully studied and is used as a guide to design these dimensions as it is important that the generated power virus should still be a realistic workload depicting the practically attainable maximum power.

In the abstract workload model, we have the choice of searching for a multithreaded power virus with homogeneous thread characteristics or provide the GA with the flexibility to configure the threads to be heterogeneous. It is to be noted that, when the threads are made heterogeneous, almost we multiply the number of dimensions in the abstract workload space for every thread by the number of threads. This could possibly result in a state space explosion and the GA may never converge. But, on the other hand, most of the real world parallel applications have heterogeneous thread characteristics [5] at least in

their data access pattern. For example, one of the most commonly used data access pattern is the producer-consumer relationship between simultaneously executing threads, where one or more producer threads write data, which is read by one or more consumer threads. To be able to exercise such a behavior in the synthetic, there should be some amount of heterogeneity in the threads to be able to act as a producer and a consumer thread. At a minimum, there should be some heterogeneity in the instruction mix in terms of the number of loads or stores. But, due to this heterogeneity in the instruction mix, the other core-level dimensions may also need be adjusted heterogeneously to be able to consume maximum power. For example, the producer threads may need to have a different register dependency distance or branch predictability than the consumer thread to be able maximize the power consumption of the core, in turn to keep the system at its maximum attainable power. Figure 5.2 shows the different dimensions of our abstract workload model and their granularity. Further in this Section, we explain each of these dimensions or what we call as the 'knobs' of our workload generator. We first begin by explaining the intuition behind the design of this abstract workload space.

In our abstract workload model, we have a controlled amount of heterogeneity, where only a few heterogeneous classes of threads can be configured and all the threads in the system have to belong to one of these few heterogeneous classes. The threads within a class are homogeneous. This controls the state space explosion and we will also be able to mimic the communication characteristics of the real parallel applications. We have found that a rea-

sonable number for heterogeneous classes is four, up to which the state space is tractable and also allows to control power for the major power consuming components.

## 5.2 Genetic Algorithm

The machine learning approach we use in our framework is popularly called the Genetic Algorithm (GA) [63]. Among the various machine learning techniques, Genetic Algorithm (GA) is known to be very effective with respect to global optimization problems. GA is a particular class of search heuristics that use techniques like mutation, crossover, inheritance and selection to solve optimization problems. GA is a search technique inspired by evolutionary biology where problem solutions are encoded as chromosomes and these chromosomes are mutated and recombined to form newer chromosomes. A population in the genetic space is defined as a set of chromosomes or possible outcomes of the problem under investigation. The algorithm proceeds by first choosing a set of D random chromosomes as the initial population, where D is the deme size or the population size used in the algorithm. These D random chromosomes (multithreaded synthetics) form the first generation of individuals for the algorithm to get started. These individuals of the first generation are evaluated for their fitness and the fitness values represent the quality of these individuals in the population and are fed back to the GA. Based on the fitness values of these synthetics, there are different operators that are applied on them like mutation, crossover and elite reproduction to produce the chro-

mosomes of the individuals of the next generation, which are again evaluated for their fitness and fed back to the GA. This evolutionary process continues until the Genetic Algorithm converges with the same value for each of the different dimensions and is repeated by seeding the GA with different random seeds to make sure that the results are robust. Though one may argue that this process of GA does not necessarily guarantee to achieve the best theoretically maximum power virus as it is still a heuristic based global optimization technique, by seeding the GA with different starting points and running it until convergence does guarantee a tight upper-bound for the maximum power for practical purposes.

The GA tool set, IBM SNAP [64] [65] takes in the description of the search space in terms of the bounds for the various parameters in the abstract workload model given as input by the user. For our power virus search, a chromosome will refer to the set of parameters in the abstract workload space for a candidate multithreaded synthetic workload. SNAP initializes the individuals of the first population with random workload characteristics and these individuals are crossed over or randomly mutated to form a new population for the next generation. After the workload parameters of the individuals for the next generation are constructed, they are fed to the code generator to generate the synthetic clone. This synthetic clone is automatically compiled and run on the corresponding processor/full-system simulator to evaluate the fitness, which is the power consumption for the design under study. These power consumption values are used as feedback to generate the next generation of individuals and

this process of evolution continues until each characteristic of the workloads converge to the maximum power consuming synthetic workload.

The most significant operators used in GA are mutation and crossover. Mutation operator probabilistically chooses parts of the chromosome and modifies them to form new chromosomes. In our case, the specifications of the multithreaded synthetics in terms of abstract workload parameters are modified randomly to form new multithreaded synthetics. The crossover operator recombines parts of two chosen chromosomes in some way to form a new chromosome for the offspring. The specifications of two chosen multithreaded synthetics are combined in a meaningful way to form the specifications of the new multithreaded synthetic offspring. SNAP provides the following parameters to control how the individuals are chosen for the next generation, i) Mutation rate: number of individuals that should be probabilistically chosen to mutate ii) Reproduction rate: number of individuals that should be probabilistically chosen to copy into new population iii) Elite reproduction rate: number of fittest individual of previous generations that should be copied into new generation iv) Crossover rate: number of individuals probabilistically chosen to serve as parents for point crossover, where a crossover point within a parent is selected and then interchange the two parent chromosomes at this point to produce two new offsprings. v) Uniform crossover rate: number of individuals probabilistically chosen to serve as parents for uniform crossover. Uniform crossover is the process in which individual bits in the chromosome are compared between two parents and are swapped with a fixed probability of 0.5.

The values used for these GA parameters like the mutation rate, crossover rate and reproduction rate for our power virus search problem will be explained in next section.

In this Subsection, we further explain why we chose GA over other search techniques. Firstly, as a general rule of thumb, a directed search technique like Genetic Algorithm (GA) is more efficient than a random search technique or a brute force methodology. Through various experiments, we have found that the crossover operator employed in GA is very effective when searching through the workload space for a power virus. This is because when we cross over two good solutions in our space, the characteristics of the parents can be very meaningfully merged and hence the offspring is also usually good, when compared to a random sample in the same space. In the rest of this Section, we elaborate on the abstract workload model that is employed and the process of code generation for the multithreaded synthetic workload.

## 5.3 Simulation Infrastructure

To evaluate the power consumption of a synthetic workload on a given design, the Virtutech Simics full system simulator is used along with Wisconsin Multifacet GEMS [66] to evaluate the power consumption of the multithreaded synthetic workloads for the SPARC ISA using Solaris 10 operating system. The cycle accurate out-of-order processor simulator Opal, the detailed memory simulator Ruby and the network simulator Garnet, all of which are a part of GEMS was used to model a typical Chip-MultiProcessor (CMP). The power

consumption in the core is evaluated using the power models provided by Wattch [60] for the most aggressive clock gating 'cc3' in Wattch. The power consumption of the shared L2 cache and the directory is modeled with help of the latest power models for caches using CACTI [67].

The power consumption of the network was evaluated using the network power model Orion [68]. The DRAMsim [69] is integrated with Ruby to accurately simulate the memory controller and a DDR2 DRAM and also provides power consumption estimates. The power models used for all the components of the CMP are for a 90nm technology. It is to be noted that this power virus generation methodology aims to help a system designer in the design stage of a system, when only the simulators will be available than real hardware. The GNU gcc compiler for SPARC ISA with the optimization level of O2 is used for compiling the synthetics and an optimization level of O3 for compiling other workloads. For the experiments using the Alpha ISA, the Sim-wattch [60] simulator built on the SimpleScalar [59] simulation framework is used to evaluate the CPU power consumption and the workloads are compiled on an alpha machine running the Tru64 UNIX operating system using gcc 4.2 with an optimization level of -O2.

## 5.4 State-of-the-art Power viruses

There have been many industry efforts towards writing power viruses and stress benchmarks. Among them, *MPrime* [25], *CPUburn-in* [17], *CPUburn* [18] are the most popular benchmarks.

***MPrime*** [25] is a BSD software application that searches for a Mersenne prime number using an efficient Fast Fourier Transform (FFT) algorithm. For the past few years, *MPrime* has been popularly called the torture test and has been used for testing the stability of a computer system by overclockers, PC enthusiasts and the processor design industry. This is because of the fact that this program is designed to subject the processor and memory to an incredibly intense workload resulting in errors. The amount of time a processor remains successfully stable while executing this workload is used as a measure of that system's stability by a typical overclocker. *MPrime* has been used in testing the CPU, memory, L1 and L2 caches, CPU cooling, and case cooling efficiencies.

***CPUburn-in*** [17] is advertised as an ultimate stability testing tool written by Michal Mienik, which is also written for overclockers. This program attempts to heat up any x86 processor to the maximum possible operating temperature. It allows the user to adjust the CPU frequency to the practical maximum while still being sure that stability is achieved even under the most stressful conditions. The program continuously monitors for erroneous calculations ensuring the CPU does not generate errors during calculations. It employs FPU intensive functions to heat up the CPU.

***CPUburn*** [18] is a power virus suite written in assembly language, copyrighted but freely licensed under the GNU Public License by Robert Redelmeier. The purpose of these programs is also to heat up x86 CPUs as much as possible. Unlike *CPUburn-in*, they are specifically optimized for different

| Power Virus | Optimized to stress | Language |
|---|---|---|
| MPrime | All CPUs, all ISAs | 'C' |
| CPUburn-in | X86 machines | x86 assembly |
| BurnP5 | Intel Pentium w&w/o MMX processors | x86 assembly |
| BurnP6 | Intel PentiumPro, Pentium II, Pentium III and Celeron CPUs | x86 assembly |
| BurnK6 | AMD K6 processors | x86 assembly |
| BurnK7 | AMD Athlon/Duron processors | x86 assembly |
| BurnMMX | cache/memory interfaces on all CPUs with MMX | x86 assembly |

Figure 5.3: Single-threaded power viruses widely used in the industry

processors. FPU and ALU instructions are coded at the assembly level into an infinite loop. The goal has been to maximize CPU temperature, stressing the cooling system, motherboard and power supply. The programs are *BurnP5, BurnP6, BurnK6, BurnK7, BurnMMX*. The description of each of the power viruses are given in Figure 5.3.

But, all these power viruses are single-threaded benchmarks targeting single-core systems. Due to the unavailability of any power viruses for multi-core systems, as a first step, our power virus generation framework is evaluated by generating single-threaded power viruses and compare them with the aforementioned state-of-the-art viruses. The effectiveness of the power viruses generated using our singl-core power virus generation framework SYstem-level Max-POwer (SYMPO) are validated on the SPARC and the Alpha ISAs by comparing with the industry grade power virus *MPrime* torture test along

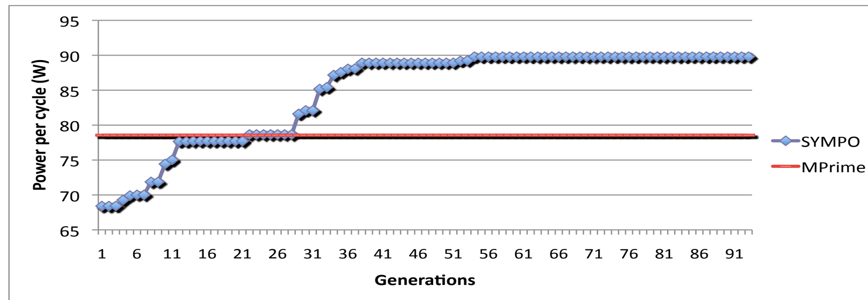
with measurements on the instrumented quadcore AMD system. Among the different industry grade power viruses discussed previously, *MPrime* is the only benchmark for which the source code is available. Most of the other power viruses discussed were handcrafted using x86 assembly and can only be used on x86 machines. Due to this limitation, we compare the power consumption of the SYMPO viruses only with that of *MPrime* on SPARC and Alpha ISAs. But, on x86 ISA, we use all the industry grade power viruses for comparison.

## 5.5 SYstem-level Max POwer (SYMPO) - Power Viruses for Single-core systems

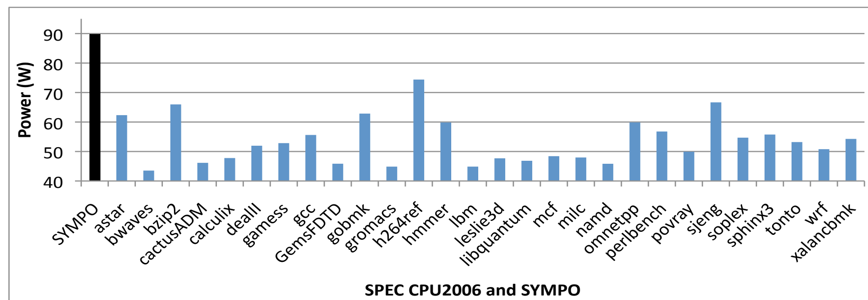
In the process of generation of power viruses for single-core systems, we only use a subset of the knob of the abstract workload model that are only relevant to single-core systems excluding the knobs falling under the category thread level parallelism, shared memory access patterns and communication characteristics.

### 5.5.1 Results on SPARC ISA

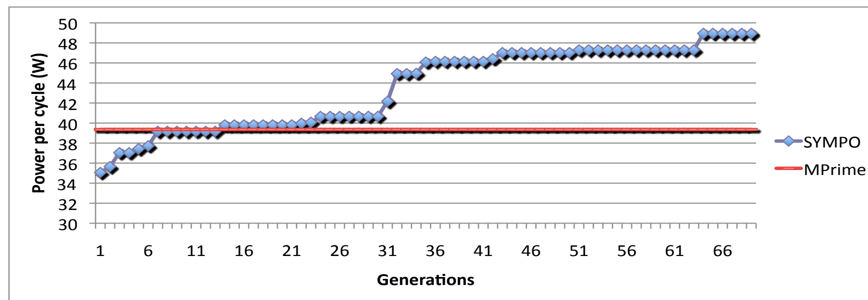
To see the efficacy of using SYMPO to find power viruses, we compare the power consumption of the industry grade MPrime torture test with that of the individuals chosen by SYMPO for a set of 3 entirely different microarchitecture configurations using the GEMS full system processor simulator. The 3 microarchitecture configurations used are given in Figure 5.10. The microarchitectures differ in terms of number of functional units, cache sizes, instruction



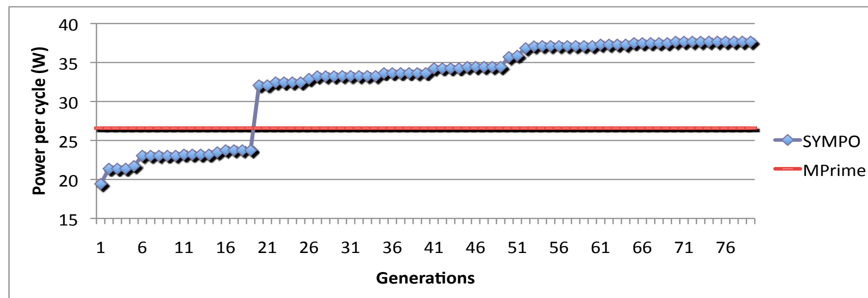

(a) Config 1




(b) Comparison with SPEC CPU2006




(c) Config 2




(d) Config 3

Figure 5.4: Evaluation of SYMPO on SPARC ISA for single-core systems

| | Config 1 | Config 2 | Config 3 |
|---|---|---|---|
| ALU | 4 Int, 4 FP | 3 Int, 2 FP | 1 Int, 1 FP |
| L1 I- & D-Cache | 64 KB, 256 MSHR | 32 KB, 128 MSHR | 16 KB, 64 MSHR |
| L1 Cache Latency | 3 cycles | 2 cycles | 1 cycles |
| L2 Cache | 4 MB, 128 MSHR | 2 MB, 64 MSHR | 512 KB, 32 MSHR |
| DRAM | 8 GB | 4 GB | 2 GB |
| ROB | 256 entries | 128 entries | 32 entries |
| Machine Width | 8 | 4 | 2 |

(a) SPARC ISA

| | Config 1 | Config 2 | Config 3 |
|---|---|---|---|
| ALU | 8 Int, 4 FP | 4 Int, 2 FP | 2 Int, 1 FP |
| L1 Cache | 64 KB, 4-way | 32 KB, 4-way | 16 KB, 2-way |
| L2 Cache | 4 MB, 8-way | 4 MB, 8-way | 256 KB, 4-way |
| ROB / LSQ | 256/128 | 128/64 | 16/8 |
| Machine Width | 8 | 4 | 2 |
| Branch predictor | Hybrid 4 KB | Hybrid 4 KB | 2-level |
| Mem access time | 150 cycles | 150 cycles | 40 cycles |

(b) Alpha ISA

Figure 5.5: Single-core machine configurations used to evaluate SYMPO

window size, DRAM size and the machine width. Figures 5.4(a), 5.4(c) and 5.4(d) show the increase in the power consumption of the best power virus as SYMPO progresses with each generation for each of the 3 microarchitectures respectively. The same figures also show the power consumption of MPrime torture test for comparison. The power viruses generated by SYMPO consume 14%, 24% and 41% more power than MPrime for microarchitectures 1, 2 and 3 respectively. For the above results, the genetic algorithm was seeded with random workloads and run for 91, 69 and 79 generations for each of the microarchitectures resulting in 728, 552 and 632 simulations. The number of dynamic instructions in the power viruses were set to be 10 million. It is to be noted that the caches get warmed up in just a few thousand instructions in the synthetic and the power consumption converges to steady state in not more

than 10 million dynamic instructions. Since these simulations were done on a Xeon parallel machine, the fitness evaluation for the individuals in a generation were let to run in parallel resulting in an efficient exploration consuming a total simulation time of 15 hours, 11 hours and 13 hours for SYMPO to generate the viruses for the machine configurations 1, 2 and 3 respectively. The genetic algorithm parameters that were used and found to be well suited to explore this particular search problem are a mutation rate of 0.03, reproduction rate of 0.01, elite reproduction rate of 0.125, crossover rate of 0.825 and a uniform crossover rate of 0.01. Since many parameters in our search space are correlated with each other, having a higher non-disruptive point crossover rate performs better than having higher disruptive uniform crossover rate.

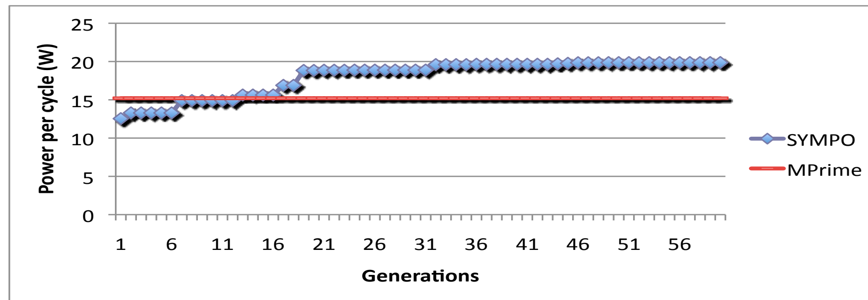
To further compare the power consumption of the generated power virus to that of the real world applications, the SPEC CPU2006 workloads were simulated on our full system simulation infrastructure for 1 billion dynamic instructions after fast forwarding for 2 billion instructions on machine configuration 1. Figure 5.4(b) shows the power consumption of the SYMPO virus compared to real world SPEC workloads. The SPEC workloads have an average power consumption of 53.4 Watts compared to 89.8 Watts consumed by the SYMPO virus.

It is to be noted that the power viruses generated for each of the configurations are different. For instance the characteristics of the power virus generated for machine configuration 1 are a basic block size of 10 instructions, 200 static basic blocks, the memory pointers are reset to beginning every 200

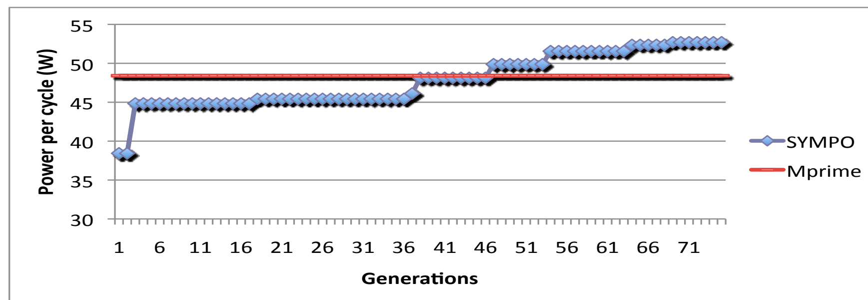
iterations, a branch transition rate of 0.98, 10% of the memory access instructions having a stride of 12 and 90% of the memory access instructions have a stride of zero, a memory level parallelism of 1. The instruction mix of this power virus was int ALU-19.5%, int mul-6.5%, int div-19.5%, FP mov-19.5%, load-6.5%, store-19.5% and branch-10%. The characteristics of the virus generated for machine configuration 3 was significantly different in its instruction mix compared to that generated for machine configuration 1. The instruction mix of the power virus generated for machine configuration 3 was int ALU-18%, load-36.3%, store-36.3% and branch-10%. It is very hard to make general inferences about the importance of the characteristics of the synthetics for the various hot cases as they vary extensively based on the targeted machine configuration. The main aim of using micro-architecture independent characteristics along with machine learning for this problem is to be able to have a black box approach towards the generation of the power virus and avoid making models/inferences about how the power virus should be designed given a machine configuration.

### 5.5.2 Results on Alpha ISA

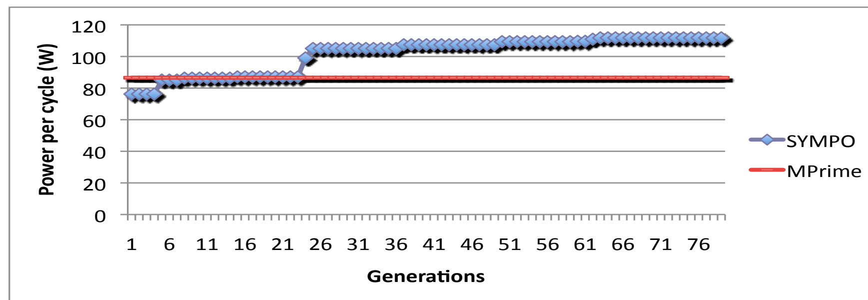
Figures 5.6(a) 5.6(b) 5.6(c) show the results of using SYMPO for generating power viruses in the Alpha ISA to maximize the power consumption in the processor core for configurations 1, 2 and 3 as given in Figure 5.5(b). The machine configurations used for the experiments on the Alpha ISA are the same as used in the previous work by Joshi et. al [22] to enable us to



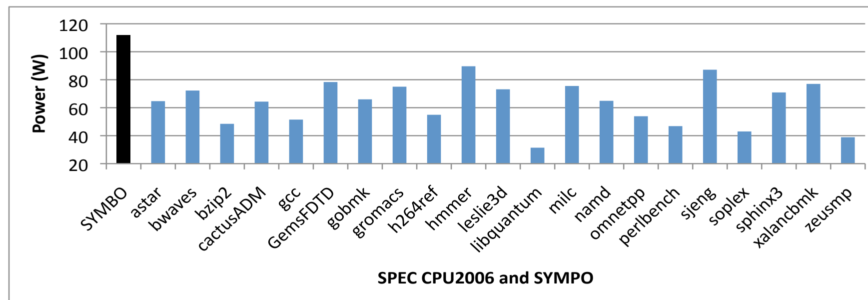

(a)




(b)




(c)




(d) Comparison with SPEC CPU2006

Figure 5.6: Evaluation of SYMPO on Alpha ISA using SimpleScalar for single-core systems

do a direct comparison of the power consumption of the generated viruses. The power virus generated using SYMPO consumes 30%, 7.5% and 29% more power in the processor core than MPrime torture test on Alpha ISA. To be able to make a fair comparison to the stressmarks of the previous approach [22] we compare only the CPU power for all the experiments in the Alpha ISA. It is to be noted that the power viruses generated using SYMPO consume 15%, 9%, and 24% more power than the stressmarks generated for the same set of machine configurations using the Sim-Wattch simulator by the previous approach. This improvement in the power consumption is attributed to the fact that we model the instruction mix at a finer granularity and we also model the memory level parallelism in the synthetic. The memory level parallelism of a workload is shown to be a very significant factor when modeling the performance and power consumption of a workload even at the core level by Ganesan et al [56]. Ganesan et al show an improvement of 12.5% in the accuracy of the workload model when including the Memory Level Parallelism. It should also be noted that our genetic algorithm framework (SNAP) is more sophisticated, enabling us to explore a larger search space than what was used in [22].

Figure 5.6(d) shows the power consumption of the SYMPO virus compared to real world SPEC workloads. The SPEC workloads have an average power consumption of 63.22 Watts compared to 111.79 Watts consumed by the SYMPO virus. The sum of the power consumption numbers of all the units present in a machine defines the 'theoretical maximum' for this max-power search problem. Since all the units of a machine cannot be kept busy all the

time by any practical real world workload, reaching this theoretical maximum is almost an impossible event. For example, the theoretical maximum for the machine configuration 3 is 220 Watts and the power virus generated for this configuration consumes a sustainable average power of 112 Watts. Designing a system with a worst case power behavior equal to that of the theoretical maximum can result in highly wasteful over provisioning. This further motivates the necessity towards using an automatic search to be able to design a system for a reasonable worst case behavior.

### 5.5.3 Suitability of Genetic Algorithm for SYMPO

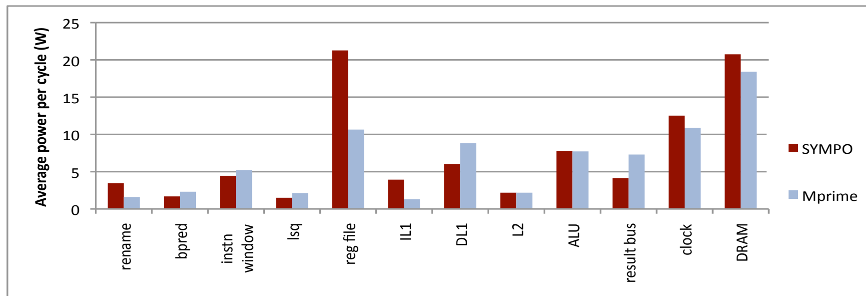
Figures 5.7(a) and 5.7(b) show the break down of the power consumption in each individual component of the system including the DRAM for the SPARC configurations 1 and 3 respectively. From this breakdown, It can be observed that SYMPO leverages the power consumption in the DRAM to maximize the overall system level power consumption. This shows that a power virus generated specifically for the CPU may not be the best power virus at the system level. It is to be noted that the virus generated by SYMPO does not always consume more power than MPrime in various components. This is due to the reason that the fitness function that the search algorithm targets to maximize is the total power consumption of the system. The same framework can be used to generate different types of stressmarks as in [22] by changing the fitness function evaluation.

For the Alpha ISA, Figures 5.7(c) and 5.7(d) show the components of the power consumption in the various parts of the CPU. These power consumption breakdown results shows that each of the synthetic workloads have interacted with the different microarchitectures in a unique fashion. This shows that how non-trivial it is to hand craft a max-power virus by speculating about this complex interaction, thus emphasizing the need for an automatic search methodology. Machine learning based approaches are considered to be more efficient than most of the brute force searches and genetic algorithm has proven to be a promising solution to this problem. In the machine configurations, SPARC config-3 has a lesser access latency for L1 cache than config-1 and one can observe that SYMPO is aware of this and generates a power virus that stresses the L1 cache effectively.

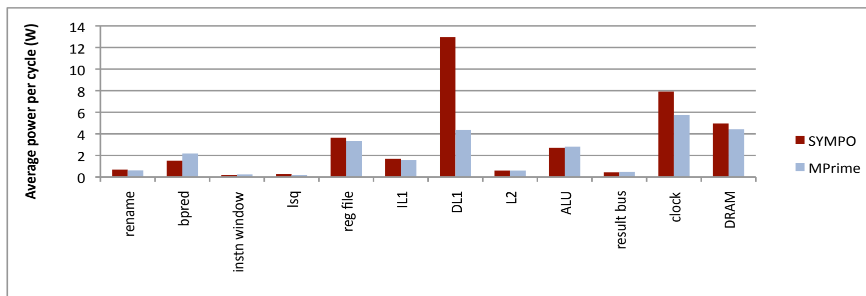
These breakdown results show that how non-trivial it is to hand craft a max-power virus by speculating about these complex interactions, thus emphasizing the need for an automatic search methodology.

### 5.5.4 Validation of SYMPO using measurement on instrumented real hardware

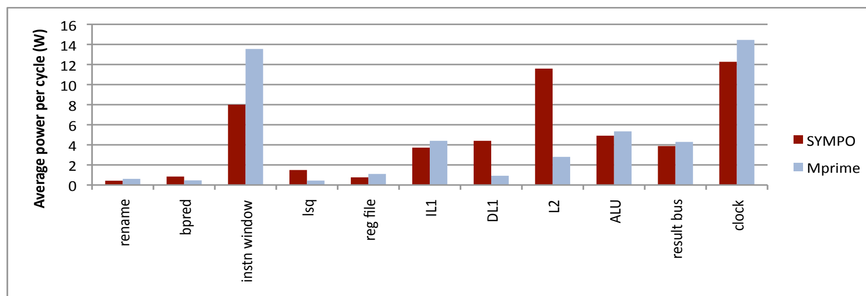
Having validated the power consumption of the viruses generated using SYMPO on 2 simulators and 2 ISAs, the next step is to measure the power consumption of the viruses on real hardware. To see the effectiveness of these power viruses on real hardware, we measure their power and thermal characteristics on the AMD Phenom II X4 Processor Model 945 system. Figure 5.8



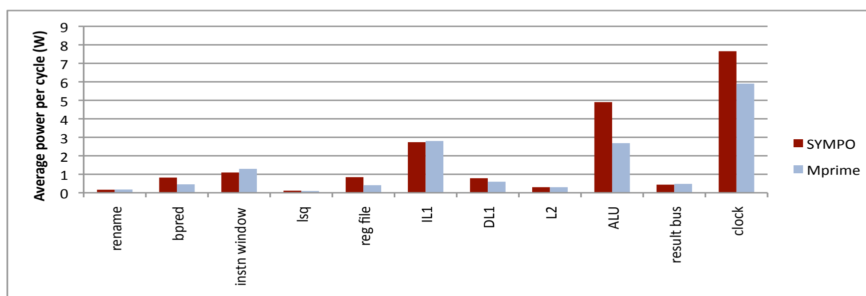

(a) SPARC config 1




(b) SPARC config 3




(c) Alpha config 2




(d) Alpha config 3

Figure 5.7: Breakdown of power consumption of SYMPO and MPrime for single-core systems on SPARC and Alpha ISAs

| Cores | Quad core |
| --- | --- |
| Clock | 3.0 GHz |
| Technology | 45 nm |
| IL1, DL1 Cache | 2-way, 64 B line, 64 KB per core |
| L2 Cache | 16-way, 64B line, 512 KB per core |
| L3 Cache | 32-way, 64B line, 6MB shared |
| Fetch size | 32 bytes |
| Branch Predictor | 12 global history, 16 K bi-mod predictor<br>2 K target buffer |
| ROB | 72 entries in 3 micro-op groups |
| ALU | 3 symmetry Int ALUs, 3 FP ALUs |

Figure 5.8: Machine configuration of AMD Phenom II

shows the configuration of this system. The CPU core power of this system is measured using in-system instrumentation. A specialized AMD-designed system board is used which provides fine-grain power instrumentation for all power rails, including CPU core. Each high-power rail, such as CPU core, contains a Hall-Effect current sensor connected at its origin. The sensor provides a 0-5V signal that is linearly proportional to the power flowing into the rail. The 5V signal is measured by a National Instruments PCI-6255 data logger. The data logger attaches to the current sensor through a small twisted pair conductor. The data logger samples current and voltage applied to each rail at a rate of 10KHz. Since the voltage cannot be assumed to be a constant due to droops, spikes and drifts, we measure both voltage and current to calculate power. Using the data logs, application power is calculated off-line with post-processing software.

The power measurements of the various power viruses are shown in

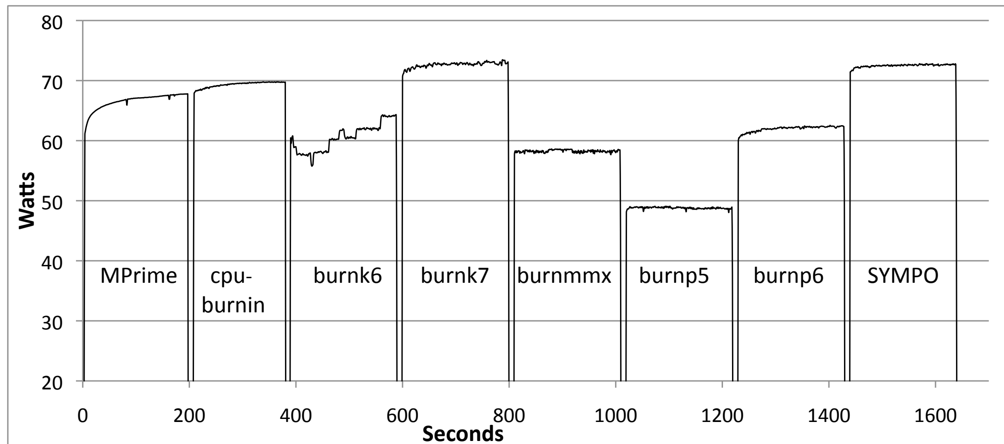



Figure 5.9: Power measurement on quad-core AMD Phenom II

Figure 5.9. Four copies of these benchmarks were run on the quad-core hardware until the power consumption reached a stable state, which was around 200 seconds. Among the industry standard power viruses, it can be noted that burnk7 consumes the maximum power on this hardware of 72.1 Watts after reaching a steady state. Even the maximum power consuming two SPEC CPU2006 workloads, 416.gamess and 453.povray consume only 63.1 and 59.6 Watts respectively. Burnk7 consuming high power on this hardware can be attributed to the fact that the machine configurations of the AMD Phenom II (K10) and K7 are to some extent similar to each other. It can be observed that the power viruses generated for other machines like the burnp5, burnp6 do not consume as much power as burnk7, again showing the importance of developing a specialized power virus for each of the microarchitectures.

Since our code generator was not equipped to generate code using CISC ISAs, we constructed a microarchitecturally equivalent system for the instru-

mented AMD Phenom II system on GEMS full system infrastructure and first generated power viruses in SPARC ISA. These viruses were ported to x86 ISA with the help of the LLVM [70] compiler infrastructure and the power consumption was measured on real hardware. These indirectly generated virus has a power consumption of 72.5 Watts on the cores, which is more than all the other hand crafted power viruses that were executed on this hardware, viz., CPUburnin-68.42 W, MPrime-68.1 W, CPUburnK6-64.2 W, CPUburnK7-72.1 W, CPUburnMMX-58.4 W, CPUburnP5-48.7 W and CPUburnP6-62.4 W as in Figure 5.9. The difference in the way the power viruses were generated for the SPARC/Alpha ISAs and the x86 ISA is that the automatic feed back loop was complete in the case of SPARC/Alpha ISAs due to the usage of the cycle accurate simulators and there was no automatic feedback to SYMPO in the case of x86. The hardware power readings were manually fed to the genetic algorithm and since this process was too tedious, the genetic algorithm was run only for 3 generations. If the feedback loop could have been automated in x86, the generated power virus is expected to consume much high amounts of power. This shows the importance of automating the process of a max-power virus generation as compared to the usage of enormous human effort.

## 5.6 MAximum Multicore POwer (MAMPO) - Power Viruses for Multicores

The power virus generation framework targeting multicores is called MAximum Multicore POwer (MAMPO) and we use all the dimensions spec-

| Parameter | System – I | System – II | System – III |
|---|---|---|---|
| No. of cores | 4 | 8 | 16 |
| DRAM | 4 GB | 8 GB | 16 GB |
| L1 cache | 64 KB, 4 way, 2 cycles | 32 KB, 4 way, 1 cycle | 16 KB, 2 way, 1 cycle |
| L2 cache | 4 MB, 4 way, 4 banks | 4 MB, 8 way, 8 banks | 8 MB, 16 way, 16 banks |
| L1, L2 MSHRs | 48 | 32 | 24 |
| ROB | 128 | 64 | 32 |
| Mach-width | 8 | 4 | 2 |
| Branch pred. | YAGS, 12 bit PHT | YAGS, 11 bit PHT | YAGS, 10 bit PHT |
| BTB size | 1024 | 512 | 256 |
| Int ALUs | 4 ALU, 2 Int div | 3 ALU, 1 Int div | 2 ALU, 1 Int div |
| Topology | Crossbar | Hierarchical switch | File-specified |
| FP ALUs | 2 ALU, 2 Mul, 2 div | 2 ALU, 1 Mul, 1 div | 1 ALU, 1 Mul, 1 div |

Figure 5.10: Multicore system configurations for which power viruses are generated to evaluate the efficacy of MAMPO on SPARC ISA

ified in the abstract workload model other than the ones related to synchronization. Since having locks, mutex in the code will only slow down the applications resulting in lesser power consumption, we do not include these knobs in the power virus search.

### 5.6.1 Experimental Setup

The Figure 5.10 shows the three multicore system configurations that are used to evaluate the efficacy of MAMPO. Figure 5.11 shows the various interconnection networks used in these multicore systems. We use the most popular MOESI cache coherence protocol for all our experiments, which has the states Modified, Owned, Exclusive, Shared and Invalid for every cache block. We use a multibanked shared L2 cache and a Non-Uniform Memory Access protocol with a directory size of 1 MB. Our power models were val-

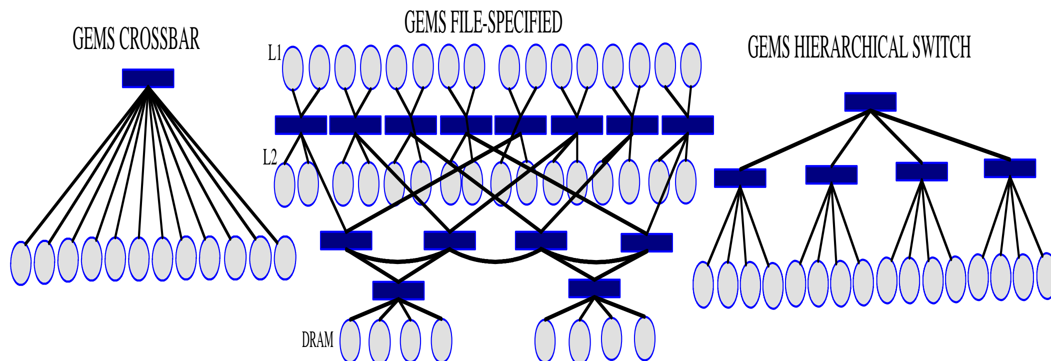



Figure 5.11: Interconnection networks used in the multicore system configurations for evaluating the efficacy MAMPO

idated against published power numbers for the Sun Microsystem's Niagara and the Rock systems by constructing an equivalent system using our infrastructure. For the machine learning, we use IBM's Genetic Algorithm toolset called SNAP [65] [71]. We have used a mutation rate of 0.05, crossover rate of 0.85 and a reproduction rate of 0.10. A population size of 48 individuals per generation was found to be the most optimal deme size for this problem. Increasing it beyond 48 does not help as the execution time of each generation becomes high due to the increased number of chromosomes to evaluate and when the deme size is smaller than 48, the population size is not big enough to search such a large abstract workload space in the same time.

We compare the power consumption of the generated MAMPO virus with that of the power consumption of the PARSEC workloads. In the multithreaded synthetic, we use a feature called MAGIC instruction in Simics to be able to perform detailed simulation for only the core part of the synthetic

code. We start the detailed simulation after all the threads have reached the barrier after the initial memory allocation and processor bind system calls. The first thread that reaches the end of it's execution signals Simics to stop the simulation and the profiled data is used to calculate the power consumption using the power models. Typically the number of dynamic instructions in the multithreaded synthetic is around a few million instructions per thread. For PARSEC workloads, we use the input set provided for detailed microarchitectural simulations called 'simsmall'.

### 5.6.2 Results and Analysis

Figure 5.12(a) shows the power consumption of the best power virus at the end of each generation for approximately 30 generations, after which there is negligible increase in power consumption due to the convergence of the GA. It is to be noted that there are not any known power viruses targeting multicores and so we compare our generated viruses against running multiple copies of single-core power viruses. MPrime [25], which is popularly called the torture test is one of the system-level industry grade power viruses for single-core systems. SYMPO [71] is the most recent previous work by Ganesan et. al to generate a max-power virus for a given single-core system. We have implemented the SYMPO framework to enable us to generate SYMPO viruses for each of our configurations and compare the overall power consumption of running multiple copies of SYMPO viruses, one on each core, with that of MAMPO viruses. Other than these power viruses, we also compare our power

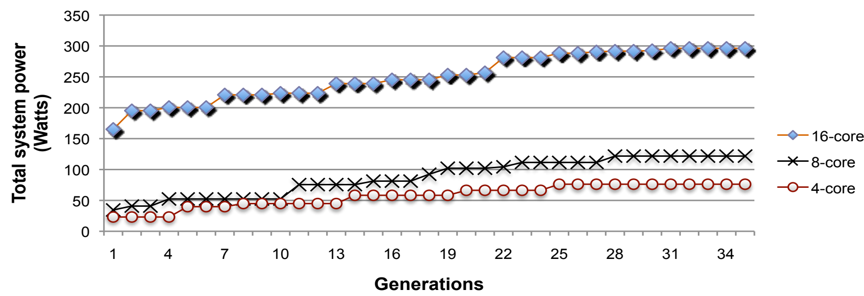
viruses with that of the commercial Java benchmark SPECjbb. The number of threads in SPECjbb was set to be equal to the number of processors in the system configuration.

Figures 5.12(b), 5.12(c) and 5.12(d) show the comparison of the power consumption of MAMPO viruses with that of the power consumption of the workloads in the PARSEC benchmark suite, MPrime, SYMPO, multithreaded Linpack and that of SPECjbb for the three machine configurations as in Figure 5.10. It can be noted that the MAMPO viruses consume 45%, 52% and 98% more power than the average power consumption of the workloads in the PARSEC suite. The MAMPO viruses consume 63%, 72% and 89% more power than that of MPrime and 40%, 49% and 69% more power than that of the SYMPO virus for the three machine configurations respectively, clearly bringing out the importance of such a multithreaded synthetic power virus generation framework compared to running multiple single-core power viruses. The MAMPO viruses consume 41%, 48% and 56% more power than that of the SPECjbb. The MAMPO viruses consume 68%, 76% and 85% more power than that of the Linpack. From these results, it can be observed that the MAMPO virus outperforms the other workloads as the number of cores increases due to the reason that MAMPO is very effective in stressing the interconnection network. It is to be noted that the energy spent in terms of data transfer through the interconnection network is predicted to increase many folds [72] due to global wire scaling problems compared to the energy spent in computation bringing out the significance of their contribution to the power consumption of future

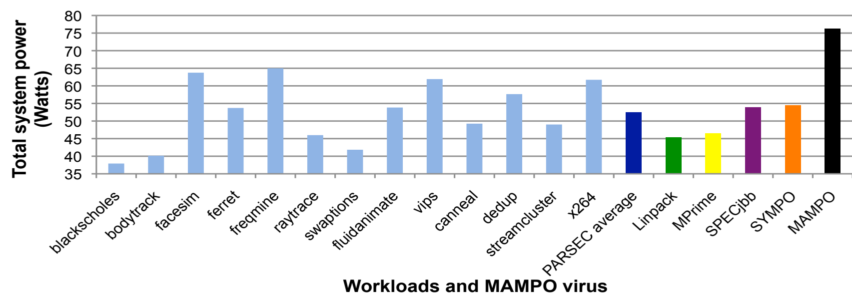
systems.

Since the fitness evaluation of the individuals in a generation is independent of each other, they can be run in parallel. Thus, when we use a modern parallel system with many cores, this process of finding a power virus can be done with a good amount of parallelism resulting in a quicker convergence of the GA. The time taken for MAMPO to generate these power viruses for the three system configurations range between 8 to 12 hours on a 3.4 GHz Intel Xeon system with 16 cores. Though we use a full system simulator with cycle accurate models to evaluate the power consumption, the total number of dynamic instructions in the synthetic is restricted to be less than 16 million instructions, to enable this search happen within a reasonable time frame. Rather, to find the same virus manually, a system architect will have to typically spend a few weeks of manpower and can still not be sure if it is a good power virus or not.

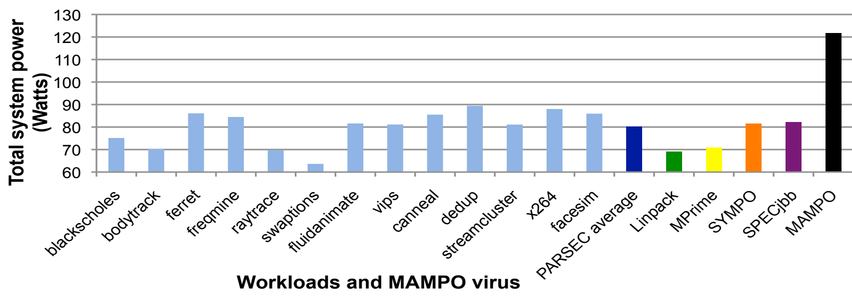
The power viruses generated for each of these configurations are found to be having exactly the same number of threads as that of the number of processors. For example, a four-threaded workload is found to be a more suitable candidate for a quad-core system than an eight or sixteen threaded workload. This can be attributed to the fact that the time taken for even a DRAM access in our framework is not enough to force a context switch in the thread scheduler used in Solaris 10. But, a knob like number of threads may be utilized more effectively when a hard disk access is also modeled, where the access latency could force the scheduler to do a context switch. We do not



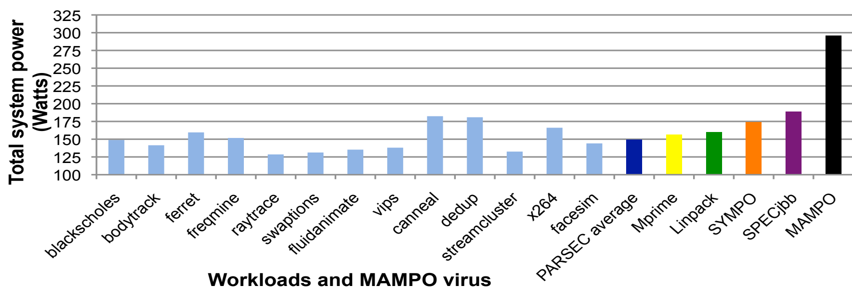

(a) Genetic algorithm convergence




(b) 4-core system power




(c) 8-core system power




(d) 16-core system power

Figure 5.12: MAMPO virus generation and evaluation on multicore systems on SPARC ISA

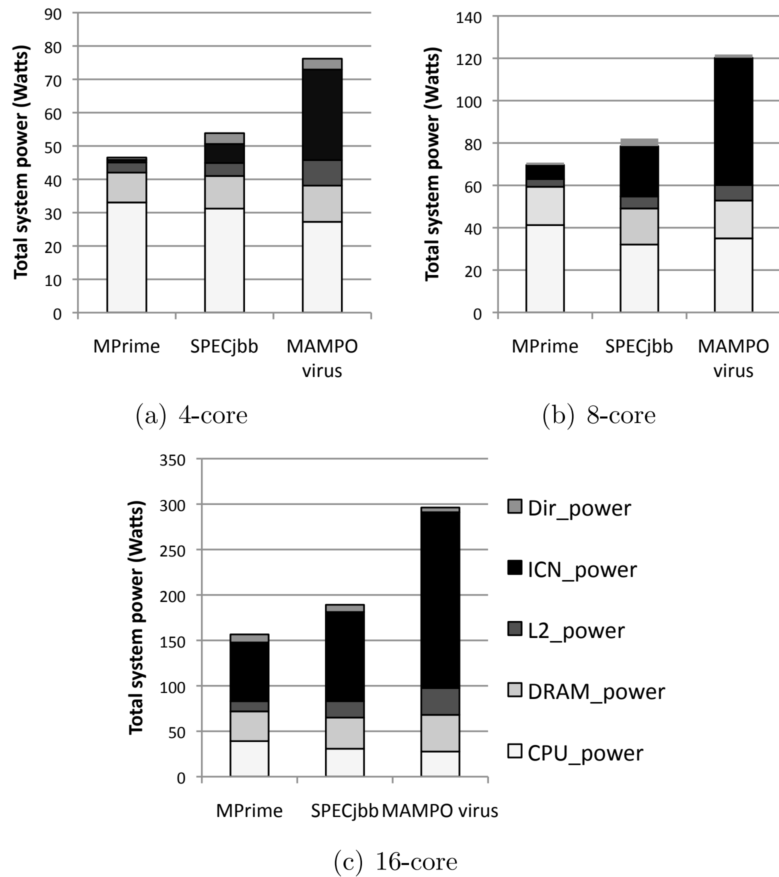



Figure 5.13: Breakdown of power consumption of MAMPO virus for various multicore system configurations and comparison to MPrime on SPARC ISA

model the components like the chipset and the disk subsystem in this study due to the reason that they have nearly constant power consumption over various range of workloads [73].

It would be interesting to see how the characteristics of the finally generated power viruses vary across the different system configurations. Figures 5.13(a), 5.13(b) and 5.13(c) show the breakdown of the power consumption

of the MAMPO viruses, SYMPO, MPrime and SPECjbb in various parts of the system. It can be noted that the single-core power viruses SYMPO and MPrime consume maximum power in the cores, rather the MAMPO viruses stress different parts of the system in such a way that the total power is maximized. Some common characteristics of these power viruses are that they have 10-20% of the memory accesses to shared data and they try to move as much data as possible through the interconnection network, besides making sure that the slowdown caused to the CPUs due to this is minimum. The maximum power achieved by our tool is still ‘realistically attainable‘ as the characteristics of the power viruses still map to the range for the abstract workload model parameters of realistic workloads. It is to be noted that the power viruses for each of these systems configurations have different settings for most of the knobs other than the aforementioned ones and it is wasteful to analyze this further due to their sensitivity to the microarchitecture changes and the aim of this whole machine learning based framework is to make this power virus generation a completely automated black-box approach to avoid the need to model the complex interactions involved in the execution of a workload within a multicore system.

# Chapter 6

# Conclusions and Future Research

Automating computer system design has been a major area of research in the past few decades due to the ever increasing complexity involved in designing modern systems. This has resulted in the usage of simulation models at varying levels of abstraction to characterize the performance and power consumption of designs. Most of these simulation models are slower than real hardware by several orders of magnitude and running a typical user workload on them completely has become almost impossible due to longer run times. Generating synthetic benchmark clones for target applications has proven to be a very good solution to this prohibitive run time problem. Todays ever increasing number of applications and a need to design processors tailored to a particular class of applications along with a faster time-to-market necessitates the need for a framework to automate the process of generating synthetic benchmark clones for the target workloads. Such a framework will enable architects to be up-to-date with their applications, as they keep evolving and also have proxies for futuristic applications generated. Such a framework can also be a valuable tool to exercise different execution behaviors to specifically test and understand the working of different subsystems. For example, with our synthetic benchmark generation framework, one can generate various types

of Inter-Connection Network (ICN) traffic, which can be very valuable in ICN design. Also,

Modern computer systems are limited by power delivery and cooling costs than critical path delay, bringing out the importance of design parameters like the Thermal Design Power (TDP). One of the applications of the synthetic benchmark generation framework proposed in this dissertation is to simplify the effort to find the practically attainable maximum power for a design. Though this dissertation is limited to finding power viruses that have maximum sustained power, a small change to the way in which the fitness function is evaluation for the power virus search can open doors to exercise many other types of workload behaviors. Some examples are to generating dI/dT stressmarks that will create cycles of maximum and minimum power droops, causing ripples in the power delivery lines. One can also use such a framework to cause hotspots in the chip to test various heat sinks and cooling features.

Thus, in this dissertation, I propose a system level synthetic benchmark generation framework targeting single core and multicore systems. This framework is evaluated for it's efficacy for two major applications namely, workload cloning and max-power stressmark generation. Each of these contributions are elaborated further in this section.

## 6.1 Workload Cloning

A characterization of the SPEC CPU2006 workloads mostly based on microarchitecture-independent characteristics have been provided and miniaturized synthetic clones [74] for these workloads have been formulated and provided to aid in accelerating architecture simulations with simulation speedups of up to 6 orders of magnitude. Along with that, the absolute and the relative accuracies of these synthetics in predicting the performance and the power consumption of various microarchitectures is provided. The proposed MLP aware synthetic benchmark generation methodology is compared with previous approaches [20] [21] and is shown that the synthetic benchmarks generated using this proposed methodology have 12.5% more accuracy in terms of IPC in the representativeness of the synthetics to that of the original workloads. The synthetics generated using this methodology have a correlation coefficient of 0.95 and 0.98 for IPC and power-per-cycle for the sensitivity to changes in microarchitecture. The availability of the provided synthetic clones will enable computer architects to use these latest workloads instead of the older SPEC suites for future studies. The futuristic workloads to be used in bio-implantable systems have also been characterized and the clones are provided.

A characterization of the PARSEC workloads have been provided and miniaturized clones for these workloads have been generated and provided to help solve the prohibitive runtime problem of multithreaded applications. These clones have been validated by assessing their performance in comparison to the original applications on a 8-core typical modern system configuration.

The average error in the IPC for these workloads is 4.87% and maximum error is 10.8% for Raytrace in comparison to the original workloads. Similarly, the average errors in the L1 cache hitrates and branch prediction rates are 0.67% and 0.53% respectively. It is also shown that the generated synthetic clones also have very similar power consumption to that of the original workloads, opening the doors for using these synthetic clones for power modeling. The average error in the power-per-cycle metric is 2.73% with a maximum of 5.5% when compared to original workloads. To further show how faithfully the synthetic benchmark follows the execution behavior of the original workloads in various system components, the breakdown of the power consumption of the synthetic is graphically compared with that of the original workloads. The representativeness of the synthetic clones to that of the original workloads in terms of their sensitivity to design changes is also shown to be quite good by finding the correlation coefficient between the trends followed by the synthetic and the original for design changes. The correlation coefficient is 0.92 for performance. Finally, the speedup achieved by using the synthetic proxies instead of the original workloads is shown to be around 4 orders of magnitude and up to 6 orders of magnitude for some specific workloads. A small amount of manual intervention in terms of tuning the code generator was done during generation of clones. It would be a good extension to this work if this tuning process can be automated using machine learning.

| | Sparc | | | Alpha | | | X86 |
|---|---|---|---|---|---|---|---|
| | Config 1 | Config 2 | Config 3 | Config 1 | Config 2 | Config 3 | |
| SYMPO | 89.8 W | 48.95 W | 37.68 W | 19.86 W | 52.70 W | 111.8 W | 72.5 W |
| MPrime | 78.56 W | 39.36 W | 26.58 W | 15.20 W | 48.41 W | 86.58 W | 68.1 W |
| % increase | 14.3 % | 24.36 % | 41.78 % | 30.6 % | 8.80% | 29.1 % | 2.3 % |

Figure 6.1: Summary of the power consumption of the single-threaded power virus generated by SYMPO in comparison to *MPrime* on Alpha, SPARC and x86 ISAs

## 6.2 Power Viruses for Single-core Systems

In this dissertation I have proposed the usage of SYMPO, a framework to automatically generate system level max-power viruses for a given machine configuration. I have shown that with the proposed workload space along with the machine learning based search, I can automatically generate reasonably good power viruses for any given microarchitecture within a few hours. I have shown the efficacy of the power viruses by comparing their power consumption with that of *MPrime* for various microarchitectures in the SPARC, Alpha and x86 ISAs. A summary of these results are shown in Figure 6.1. These results clearly show that SYMPO is very effective in leveraging the power consumption on the SPARC, Alpha ISAs compared to the x86 ISA. If the feedback loop could have been completed in x86, the generated power virus might have consumed much high amounts of power. This shows the importance of automating the process of a max-power virus generation as compared to the usage of time consuming human effort.

I also show that the power viruses generated by SYMPO are superior compared to the automatically generated power viruses using the previously

proposed methodology as in [22]. I measure the power consumption of the various industry grade hand crafted power viruses on an instrumented AMD Phenom II system and compare it with that of an x86 power virus generated by SYMPO. I also provide a detailed analysis of these various industry grade hand crafted power viruses and the x86 virus generated by SYMPO based on a set of microarchitecture independent characteristics.

## 6.3 Power Viruses for Multicore Systems

In this dissertation, I proposed the usage of MAMPO, which is a multithreaded synthetic power virus generation framework targeting multicore processors. I validate the efficacy of MAMPO by comparing the power consumption of the generated virus with that of the workloads in PARSEC for three different multicore system configurations and show that the MAMPO virus consumes 45%, 52% and 98% more power than the average power consumption of the PARSEC workloads. I also show that the single core power viruses, when run on muticore systems do not serve the purpose as a multicore system virus by comparing the power consumption of the MAMPO virus with that of the previously proposed SYMPO viruses and the well known power virus MPrime. The MAMPO virus consumes 40% to 89% more power than running multiple copies of single-core viruses in parallel. I also provide a comparison of the power consumption of the MAMPO virus with that of SPECjbb and show that the MAMPO virus consumes 41%, 48% and 56% more power than that of SPECjbb. Though the power viruses generated by

MAMPO cannot theoretically guarantee to be the absolute worst-case, based on the convergence of the Genetic Algorithm run with multiple seeds, we can be sure that the generated power viruses will serve as a tight upper-bound for the maximum power for all practical purposes and such a framework will be a very useful tool for the system designers.

# Bibliography


[1] Karthik Ganesan, Deepak Panwar, and Lizy K John. Generation,Validation and Analysis of SPEC CPU2006 Simulation Points Based on Branch, Memory, and TLB Characteristics. *SPEC Benchmark Workshop 2009, Austin, TX, Lecture Notes in Computer Science 5419 Springer pages 121-137, January 2009.*

[2] SPEC. Standard performance evaluation corporation. http://www.spec.org.

[3] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *In Proceedings of the 22nd International Symposium on Computer Architecture, pages 24-36, Santa Margherita Ligure, Italy*, June 1995.

[4] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *In Computer Architecture News, vol. 20, no. 1, pages 5-44.*

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[6] Zhanpeng Jin and Allen C. Cheng. ImplantBench: Characterizing and Projecting Representative Benchmarks for Emerging Bio-Implantable Computing. *IEEE Micro (IEEE Micro), 28(4):71-91*, July/August 2008.

[7] Greg Hamerly, Erez Perelman, and Brad Calder. How to Use SimPoint to Pick Simulation Points. *ACM SIGMETRICS Performance Evaluation Review*, March 2004.

[8] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. *Proceedings of the International Symposium on Computer Architecture, (ISCA 2003), p. 84 - 95.*

[9] M. Van Biesbrouck, T. Sherwood, and B. Calder. A co-phase matrix to guide simultaneous multithreading simulation. *In Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS04)*, March 2004.

[10] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of Redundancy and Application Balance in the SPEC CPU2006 Benchmark Suite. *The 34th International Symposium on Computer Architecture (ISCA 2007)*, June 2007.

[11] Stuart Berke, David Moss, and Randy Randall. Understanding the challenges of delivering cost-effective, high- efficiency power supplies. *http://www.dell.com/downloads/global/power/ps2q07-20070270-PowerTCO.pdf*, May 2007.

[12] Xiao Ping Wu, Masataka Mochizuki, Koichi Mashiko, Thang Nguyen, Vijit Wuttijumnong, Gerald Cabsao, and Aliakbar Akbarzadeh Randeep Singh. Energy conservation approach for data center cooling using heat pipe based cold energy storage system. *26th Annual IEEE Semiconductor Thermal Measurement and Management Symposium, 2010. SEMI-THERM 2010 Page(s): 115 - 122*, March 2010.

[13] Michael K Patterson. The Effect of Data Center Temperature on Energy Efficiency. *11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008 Page(s): 1167 - 1174*, May 2008.

[14] Amip Shah, Chandrakant Patel, Cullen Bash, Ratnesh Sharma, and Rocky Shih. Impact of rack-level compaction on the data center cooling ensemble. *11th Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems, 2008. ITHERM 2008 Page(s): 1175 - 1182*, May 2008.

[15] The industry changing impact of accelerated computing `http://sites.amd.com/us/Documents/AMD_fusion_Whitepaper.pdf`.

[16] Stephen L. Smith. Intel roadmap overview. intel developer's forum 2009 san francisco, ca. `http://download.intel.com/pressroom/kits/events/idffall_2009/pdfs/IDF_SSmith_Briefing.pdf`. September 2009.

[17] `http://www.softpedia.com/get/System/Benchmarks/CPU-Burnin.shtml`.

[18] `http://pages.sbcglobal.net/redelm`.

[19] Private Communication with Advanced Micro Devices (AMD) Design Engineer.

[20] Ajay Joshi, Lieven Eeckhout, Robert H. Bell Jr., and Lizy K. John. Performance Cloning: A Technique for Disseminating Proprietary Applications as Benchmarks. *International Symposium on Workload Characterization*, October 2006.

[21] Jr Robert H. Bell, Rajiv R. Bhatia, Lizy K. John, Jeff Stuecheli, John Griswell, Paul Tu, Louis Capps, Anton Blanchard, and Ravel Thai. Automatic Testcase Synthesis and Performance Model Validation for High Performance PowerPC Processors. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006)*, March 2006.

[22] Ajay Joshi, Lieven Eeckhout, Lizy K. John, and Ciji Isen. Automated microprocessor stressmark generation. *The 14th International Symposium on High Performance Computer Architecture (HPCA)*, February 2008.

[23] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy management for commercial servers. *IEEE Computer, 36(12):3948*, December 2003.

[24] J. Laudon. UltraSPARC T1: Architecture and Physical Design of a 32-threaded General Purpose CPU. *Proceedings of the ISSCC Multi-Core*

*Architectures, Designs, and Implementation Challenges Forum*, 2006.

[25] `http://www.mersenne.org/freesoft`.

[26] Mark Oskin, Frederic T. Chong, and Matthew Farrens. HLS: Combining Statistical and Symbolic Simulation to Guide Microprocessor Design. *In Proceedings of the International Symposium on Computer Architecture (ISCA 2000)*, 2000.

[27] Sebastien Nussbaum and James E. Smith. Modeling Superscalar Processors via Statistical Simulation. *International Conference on Parallel Architectures and Compilation Techniques (PACT 2001)*, 2001.

[28] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. *Proceedings. 31st Annual International Symposium on Computer Architecture, (ISCA 2004)*, 2004.

[29] Wing Shing Wong and Robert J. T. Morris. Benchmark Synthesis Using the LRU Cache Hit Function. *IEEE Transactions on Computers*, 1988.

[30] Cheng-Ta Hsieh and M. Pedram. Microprocessor power estimation using profile-driven program synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,*, November 1998.

[31] E.S. Sorenson and J.K. Flanagan. Evaluating synthetic trace models using locality surfaces. *2002. WWC-5. 2002 IEEE International Workshop on Workload Characterization*, November 2002.

[32] T. Sherwood and B. Calder. Time varying behavior of programs. *Technical Report UCSD-CS99-630, UC San Diego,*, August 1999.

[33] Jeremy Lau, Jack Sampson, Erez Perelman, Greg Hamerly, and Brad Calder. The Strong Correlation Between Code Signatures and Performance. *IEEE International Symposium on Performance Analysis of Systems and Software*, March 2005.

[34] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[35] E. Perelman, T. Sherwood, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[36] Erez Perelman, Greg Hamerly, and Brad Calder. Picking Statistically Valid and Early Simulation Points. *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[37] T. Chou and K. Roy. Accurate power estimation of cmos sequential circuits. *IEEE Transactions on VLSI Systems,*, 1996.

[38] C. Lim, W. Daasch, and G. Cai. A thermal-aware superscalar microprocessor. *ISQED*, 2002.

[39] W. Felter and T. Keller. Power measurement on the apple power mac g5. *IBM Tech Report RC23276*, 2004.

[40] M. Gowan, L. Biro, and D. Jackson. Power considerations in the design of the alpha 21264 microprocessor. *Design Automation Conference*, 1998.

[41] R. Vishwanath, V. Wakharkar, A. Watwe, and V.Lebonheur. Thermal performance challenges from silicon to systems. *Intel Technology Journal*, 2000.

[42] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. *High Performance Computer Architectures*, 2003.

[43] F. Najm, S. Goel, and I. Hajj. Power estimation in sequential circuits. *Design Automation Conference*, 1995.

[44] K. Lee, K. Skadron, and W. Huang. Analytical model for sensor placement on microprocessors. *ICCD*, 2005.

[45] Eron Jokipii. Jobe - The Java obfuscator - `http://www.primenet.com/~ej/index.html`. 1996.

[46] John J. Marciniak. Encyclopedia of Software Engineering. *chapter Reverse Engineering, pp 1077-1084. John Wiley & Sons, Inc, 1994. ISBN 0-471-54004-8.*

[47] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems, vol. 5, no. 4, pp 371-393, November 1987.*

[48] Rex Jaeschke. Encrypting C source for distribution. *Journal of C Language Translation, vol. 2, no. 1, 1990.*

[49] Ajay Joshi, Lieven Eeckhout, Jr. Robert H. Bell, and Lizy K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO 2008)*, August 2008.

[50] Robert H Bell and Lizy K John. Improved Automatic Test Case Synthesis For Performance Model Validation. *Proceedings of the International Conference on Supercomputing 111-120*, 2005.

[51] Nick Barrow-Williams, Christian Fensch, and Simon Moore. A communication Characterization of Splash-2 and Parsec. *IEEE International Symposium on Workload Characterization*, October 2009.

[52] Michael C. Huang Hemayet Hossain, Sandhya Dwarkadas. Improving support for Locality and fine-grain sharing in chip multiprocessors. *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, October 2008.

[53] Liqun Cheng, John B. Carter, and Donglai Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. *IEEE 13th*

*International Symposium on High Performance Computer Architecture, 2007. HPCA 2007*, February 2007.

[54] Umakishore Ramachandran, Gautam Shah, Anand Sivasubramaniam, Aman Singla, and Ivan Yanasak. Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors. *Proceedings of the IEEE/ACM Supercomputing Conference*, 1995.

[55] Guhan Viswanathan and James R. Larus. Compiler-directed Shared-Memory Communication for Iterative Parallel Applications. *Proceedings of the ACM/IEEE Conference on Supercomputing*, 1996.

[56] Karthik Ganesan, Jungho Jo, and Lizy K John. Synthesizing Memory-Level Parallelism Aware Miniature Clones for SPEC CPU2006 and ImplantBench Workloads. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2010.

[57] Stijn Eyerman and Lieven Eeckhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. *IEEE 13th International Symposium on High Performance Computer Architecture, 2007. (HPCA 2007)*, February 2007.

[58] Haungs M, Sallee P, and Farrens M. Branch transition rate: a new metric for improved branchclassification analysis. *Sixth International Symposium on High-Performance Computer Architecture (HPCA 2000), Volume , Issue , 2000 Page(s):241 - 25*, January 2000.

[59] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342. *University of Wisconsin, Madison*, June 1997.

[60] Margaret Martonosi, Vivek Tiwari, and David Brooks. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *isca, pp.83, 27th Annual International Symposium on Computer Architecture (ISCA 2000)*.

[61] Engin Ipek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.

[62] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, Oct 2006.

[63] L D Davis and Melanie Mitchel. Handbook of genetic algorithms. *Van Nostrand Reinhold*, 1991.

[64] Private Communication with Jason F Cantin, IBM.

[65] Sameh Sharkawi, Don Desota, Raj P, Rajeev Indukuru, Stephen Stevens, and Valerie Taylor. Performance Projection of HPC Applications Using

SPEC CFP2006 Benchmarks. *IEEE International Parallel & Distributed Processing Symp.*, May 2009.

[66] Milo M.K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, , and David A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News (CAN),*, September 2005.

[67] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches With CACTI 6.0. *Proc. 40th Annual IEEE/ACM Intl Symp. on Microarchitecture (MICRO 07), IEEE CS Press pp. 3-14.*, December 2007.

[68] Hangsheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. Orion: A Power-Performance Simulator for Interconnection Networks. *In Proceedings of MICRO 35, Istanbul, Turkey*, November 2002.

[69] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Katie Baynes, Aamer Jaleel, and Bruce Jacob. DRAMsim: A memory-system simulator. *Computer Arch. News, vol. 33, no. 4, pp. 100-107*, Sep 2005.

[70] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California*, March 2004.

[71] Karthik Ganesan, Jungho Jo, W. Lloyd Bircher, Dimitris Kaseridis, Zhibin Yu, and Lizy K. John. System-level Max Power (SYMPO) - A systematic approach for escalating system-level power consumption using synthetic benchmarks. *In the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, Austria*, September 2010.

[72] Michele Petracca, Benjamin G. Lee, Keren Bergman, and Luca P. Carloni. Design Exploration of Optical Interconnection Networks for Chip Multiprocessors. *IEEE Symposium on High Performance Interconnects pages:31-40*, September 2008.

[73] W. Lloyd Bircher and Lizy K.John. Complete System Power Estimation: A Trickle-Down Approach Based on Performance Events. *International Symposium on Performance Analysis of Systems and Software*, April 2007.

[74] `http://users.ece.utexas.edu/~kganesan/snth.tgz`.

[75] Valentina Salapura, Karthik Ganesan, Alan Gara, Michael Gschwind James C. Sexton, and Robert E. Walkup. Next-Generation Performance Counters: Monitoring Over Thousand Concurrent Events. *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium, pages 139-146*, April 2008.

[76] AJ KleinOsowski and David Lilja. MinneSPEC: A New SPEC bench-

mark Workload Simulation-Based Computer Architecture Research. *Computer Architecture Letters, vol. 1*, June 2002.

[77] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro Hot Tutorials, (Vol. 27, No. 3) pp. 63-72*, May/June 2007.

[78] R. Desikan, D. Burger, and S. Keckler. Measuring experimental error in microprocessor simulation. *ISCA*, 2001.

[79] P. Bose and J. Abraham. Performance and functional verification of microprocessors. *In the IEEE VLSI Design Conference*, 2000.

[80] P. Bose. Performance test case generation for microprocessor. *In the IEEE VLSI Test Symposium*, 1998.

[81] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, Wallace S. and Vijay Janapa Reddi Lowney, G., and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation.

# Vita

Karthik Ganesan was born in Chennai, India. He did his schooling at A.V. Meiyappan Matriculation Higher Secondary School. He received his Bachelor of Engineering from Anna University, Chennai, India in May 2006 and Master of Science in Engineering in 2008 from the University of Texas at Austin. During his undergraduate study, he was also working as a part time research trainee at Waran Research Foundation (2004 - 2006) in the high performance computer architecture group directed towards designing the Memory in Processor Super Computer On Chip architecture. He joined the graduate program of the ECE department at the University of Texas at Austin in Fall 2006. He was working as a Research Assistant in the Laboratory for Computer Architecture directed by Dr. Lizy K John. In Summer 2007, he interned at the T. J. Watson research labs of IBM in the BlueGene/P design team. In Summer 2008, he interned at IBM Austin, where he was involved in improving the open source tool Performance Inspector from IBM. In Summer 2011, he was a performance architect intern at ARM Inc., Austin, working on multicore ARM processor designs to aid in identifying performance bottlenecks. He is currently working in the performance team at Oracle Austin.

Email address: kganesan@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.