Next Generation Hardware Monitoring Infrastructure for Multi-core Resource Auditing

by

Jie Chen

A Dissertation submitted to

The Faculty of

The School of Engineering and Applied Science
of the George Washington University in partial satisfaction
of the requirements for the degree of Doctor of Philosophy

Oct, 2014

Dissertation Advisor:
Dr. Guru Prasadh Venkataramani
Assistant Professor of Engineering and Applied Science

Abstract

Thesis Statement: Performance counters in hardware have been very successful in providing feedback about application performance to programmers and compilers. With the growing relevance of understanding power, energy and security (information leakage), we envision that the next generation hardware monitoring infrastructure will support these features. In this work, we study the design and implementation of such hardware monitors.

Continuous advances in semiconductor technologies have enabled the integration of billions of transistors in modern multicore processors. This offers software applications with abundant hardware resources to use. To realize more parallelism and higher performance, software developers are concerned about characterizing and optimizing their applications over the usage of hardware resources. At the same time, hardware monitoring infrastructure in most current processors offers a collection of hardware counters for auditing architectural events on hardware units. Such counters can be used by programmers and performance analysts for auditing performance bottlenecks and consequently, optimizing application performance. In recent years, there is a surging demand for improving application power, energy and information leakage beyond performance, due to increasing complexity in power delivery within the chip and vast amount of shared hardware resources between applications running on the processors.

As power budget is limited, it becomes necessary to audit software power usages and look for power optimizations *at all levels* to better utilize the limited power. Optimizing the applications for energy is necessary due to the impact that they have on system operation and costs. Inefficient software has been often cited as a major reason for wasteful energy consumption in computing systems. It is essential for programmers to audit the energy usage of their program code and apply code optimizations for better energy. While power and energy are already among the top issues that need to be solved, a fast growing concern is information leakage using shared resource in multi-core hardware. Over the past years, it has been shown many times that multicore hardware resources are vulnerable and can easily be exploited as covert timing channels to leak sensitive information at very high bandwidth. Unfortunately, there is no existing hardware-supported auditing for such information leakage.

As these set of factors such as power, energy and information leakage are becoming more critical, software developers and system administrators are urgently looking for appropriate tools to address such challenges, just like they used to rely on performance counters for performance tuning. The next generation hardware monitoring infrastructure should take users' need into consideration, and provide sufficient and convenient resource auditing support *beyond just performance*. It will not only enable the programmers to improve the scalability of their softwares by better utilizing power budget and to reduce cost by improving the energy efficiency of their program code, but also help system administrators enhance the level of trust of their systems by tracking and removing information leakage sources.

In this dissertation, we proposed and explored the design of three novel resource auditing techniques as part of the next generation hardware monitoring infrastructure, namely, application power auditing, application energy auditing, and covert timing channel auditing. The design methodologies of these three techniques share the same goal of leveraging hardware support to enable the gathering of advance resource usage information in an efficient and cost-effective manner. These hardware support are also equipped with lightweight software support to maximize flexibility in usage. Overall, our contributions push the hardware monitoring support to the next new level, and enable the programmer to efficiently address a spectrum of existing and emerging system issues.

Contents

Abstract **i**

Contents **iii**

List of Figures **v**

List of Tables **viii**

1 Introduction **2**

1.1 Need for Next Generation Hardware Monitoring 2

1.2 Multicore Resource Usage Problems– Power, Energy, and Information Leakage 3

1.3 Overview 6

1.4 Scope of this Dissertation 7

2 Background **8**

2.1 Application Power Auditing 8

2.2 Application Energy Auditing 9

2.3 Information Leakage Auditing 10

3 Watts-inside: Application Power Auditing **13**

3.1 Motivation – Understanding Multicore Power 13

3.2 Design Overview 15

3.2.1 Hardware Support 15
3.2.2 Software Support 19
3.3 Implementation 22
3.3.1 Hardware Support 22
3.3.2 Software Support 24
3.4 Experimental Setup 24
3.5 Evaluation 26
3.5.1 Adaptive Filter vs. Ideal Filter 26
3.5.2 Sampling 27
3.5.3 Scalability of CSPPV Memory Log 28
3.5.4 Area, Power and Latency of Watts-inside Hardware 28
3.5.5 Case study – Improving Load/Store Queue power consumption 29
3.6 Related Work 33
3.7 Summary 34

4 enDebug: Application Energy Auditing 35

4.1 Understanding Application Energy 35
4.2 Fine-grain Energy Profiling 37
4.2.1 Energy Model Using Performance Counters 38
4.2.2 Attributing Energy to Program Functions 39
4.3 Automated Recommendation System for Energy Optimization 41
4.3.1 Basics of Genetic Programming 42
4.3.2 Artificial Selection Genetic Programming 43
4.3.3 ASGP Algorithm 43
4.3.4 Neutral Transforms 47
4.3.5 Genetic operators and Triggering heuristics 48
4.3.6 System Support for Automated Energy Optimization 53
4.4 enDebug Evaluation 54

4.4.1 Validation on a real system 58
4.5 Related Work 60
4.5.1 Energy Estimation 60
4.5.2 Energy Optimization 61
4.6 Summary 62

5 CC-Hunter: Information Leakage Auditing 64
5.1 Understanding Covert Timing Channels 64
5.2 Threat Model and Assumptions 65
5.3 Design Overview 66
5.3.1 Covert Timing Channels on Combinational hardware 67
5.3.2 Recurrent Burst Pattern Detection 69
5.3.3 Covert Timing Channel on Memory 73
5.3.4 Oscillatory Pattern Detection 74
5.4 Implementation 77
5.4.1 Hardware Support 77
5.4.2 Software Support 80
5.5 Evaluation and Sensitivity Study 81
5.5.1 Varying Bandwidth Rates 81
5.5.2 Encoded Message patterns 84
5.5.3 Varying Cache Channel Implementations 84
5.5.4 Testing for False Alarms 84
5.6 Related Work 86
5.7 Summary 88

6 Conclusions and Future Work 90

List of Figures

2.1 A Covert Timing Channel using timing modulation on a shared resource to divulge secrets . . . 11

3.1 Dynamic Power traces for four-threaded applications measured during their execution . . . 14

3.2 Design overview of the Watts-inside framework . . . 16

3.3 Code Sequence Power Profile Vector (CSPPV) . . . 18

3.4 Hardware modifications needed for Watts-inside framework . . . 23

3.5 Ideal Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads. . . . 25

3.6 Adaptive Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads. 25

3.7 Average relative error in the mean and standard deviation of power consumption by code sequences (in Splash-2 and Parsec-1.0 applications) due to sampling, relative to a baseline execution without sampling. . . . 27

3.8 Average and worst-case CSPPV Memory log requirements for capture ratio=0.25 for different numbers of cores . . . 28

3.9 Code snippets from ocean and streamcluster benchmarks where store-to-load dependencies result in high power . . . 30

3.10 Modified code snippets from ocean and streamcluster benchmarks that no longer have store-to-load dependencies 31

3.11 Power reduction across the entire chip, scheduler and load/store queue after removing store-to-load dependencies 32

4.1 Recording performance counters and PCs at function calls and returns . . . 40

4.2 Storing performance counters and PCs to the hardware buffer 40

4.3 Examples of Genetic operator Applications on CDDG 49

4.4 Fmm code snippet 55

4.5 Ocean code snippet 56

4.6 Cholesky code snippet 57

4.7 Water-sp/Water-n2 code snippet 57

4.8 Fluidanimate code snippet 58

4.9 Streamcluster code snippet 58

5.1 Average latency per memory access (in CPU cycles) in Memory Bus Covert Channel 68

5.2 Average loop execution time (in CPU cycles) in Integer Divider Covert Channel 69

5.3 Event Train plots for Memory Bus and Integer Divider showing burst patterns. 70

5.4 Illustration of Event Train and its corresponding Event Density Histogram. The distibution is compared against the Poisson Distribution shown by the dotted line to detect presence of burst patterns. 71

5.5 Event Density Histograms for Covert Timing Channels using Memory Bus and Integer Divider. 71

5.6 Ratios of cache access times between G_1 and G_0 cache sets in Cache Covert Channel 74

5.7 Oscillatory pattern of cache conflict misses between trojan and spy. An auto-correlogram is shown for the conflict miss event train. 75

5.8 Hardware Implementation and integration of CC-Hunter into processor hardware 77
5.9 Bandwidth test using Memory Bus, Integer Divider and Cache Covert Channels 82
5.10 Autocorrelograms for 0.1 bps cache covert channels at reduced observation window sizes 83
5.11 Test with 256 randomly generated 64-bit messages on Memory Bus, Integer Divider and Cache covert channels. Black (thick) bars are the means, and the red (annotations) arrows above them show the range (min, max). 83
5.12 Autocorrelograms for cache covert channel with varying numbers of cache sets for communication. 83
5.13 Event Density Histograms and Autocorrelograms in pair combinations of SPEC2k6, Stream & Filebench 87

List of Tables

3.1 Performance and Dynamic Power imbalance in SPLASH-2 and PARSEC-1.0 benchmarks with four threads 15

3.2 Area, Power and Latency Estimates of Watts-inside hardware 29

4.1 Energy and performance profile of functions in Splash-2 and PARSEC-1.0 benchmarks with 8 threads 35

4.2 Processor Configuration and Power Model 36

4.3 Triggering heuristics for genetic operators 50

4.4 Number of mutants for the un-guided natural selection GP algorithm 51

4.5 Number of mutant for our ASGP algorithm with triggering heuristics 51

4.6 Execution time overhead comparison between the natural selection GP algorithm and our heuristic-triggered ASGP algorithm 52

4.7 Energy reduction for our benchmark applications in simulator (Baseline: GCC -O3) 59

4.8 Energy reduction on a real system (Intel Core i7) 60

5.1 Area, Power and Latency Estimates of CC-Auditor 80

Chapter 1

Introduction

1.1 Need for Next Generation Hardware Monitoring

As software is getting more complex, efficiently characterizing and optimizing applications has posed great challenges for programmers and system administrators. Hardware monitoring has become a very popular option for them to address such challenges. Hardware monitoring infrastructure provides collections of highly efficient on-chip performance counters that can gather critical architectural statistics at application runtime. These profile-based data can be used for analyzing program behaviors, identifying performance bottlenecks, and ultimately, improving the overall system performance. The importance and usefulness of hardware monitors have already been justified by the popularity of software tools that access hardware counters for profiling, such as PAPI [1] and Likwid [2]. In the future, hardware monitoring infrastructure will continue to play important roles, and that is why major processor companies are actively improving their on-chip hardware monitors to enhance user experience.

Designing hardware monitors to improve application performance is important, but we should also notice that there is significant demand for improving application power, energy and information leakage, as hardware platforms and software paradigms have rapidly changed in the past years. Today, programmers and system administrators not only care about performance, but also care about application power consumption, financial cost incurred by application energy consumption, and whether their applications run in a trustable system environment. Unfortunately, since existing hardware monitors were originally designed for just performance, they do not provide direct auditing support to the users about which parts of the program exhibit high power utilization, which program function consumes a lot of energy, and what process is leaking information over shared hardware resources. In response to such growing concerns, modern high performance processor architectures have begun integrating new counters for gathering processor power and energy usages. For example, starting from Sandy Bridge, Intel provides a driver interface called RAPL (Running Average Power Limit) that can let programmers periodically sample processor power/energy usually at the granularity of a few milliseconds of program execution time. While such counters could be used to obtain application power trace or total energy consumption, they do not provide feedback at a granularity that relates the processor power/energy consumption back to the program source code.

Therefore, we envision that the next generation multicore hardware monitoring infrastructure should provide better auditing support for application power, energy, and information leakage in an cost-effective manner, and ultimately, help programmers and system administrators to make actionable improvements.

1.2 Multicore Resource Usage Problems– Power, Energy, and Information Leakage

While the need for the next generation hardware monitoring infrastructure becomes clear, power, energy, and information leakage that resulted from applications' usage of hardware resources need to be understood better. In this section, we will review each resource usage problem.

Power has long been a design constraint in processors. Generations of processors used to follow Dennard scaling which states that, as transistors get smaller their power density stays constant, so that power stays constant as long as the dies size remains the same. While Dennard scaling starts to break down around 2005, the trend of integrating more number of cores has not been stopped. The consequence is that, due to limited power budget, only part of chip transistors can switch at full frequency, and many of them have to be powered off. As a result, software applications are forced to use only a portion of all the multi-core resources, for example, Intel Turbo Boost Mode runs half of the cores at full frequency, while shutting off the rest cores. Not appropriate usage of hardware resources may also trigger other power issues. As instances– (1) An increase in the chip power density could affect the reliability of operation and result in decreased lifetime of processors [3], (2) Uneven consumption of power by different cores could complicate heat dissipation and electric supply in multicores [4]. As processor demands rapidly change the electric current consumption over a short timeframe, supply voltage perturbations could occur, resulting in power-delivery subsystem having large parasitic inductance and voltage ripples on the chip's supply lines. When voltage ripples exceed a certain tolerance range, CPUs may begin to malfunction [5]. Such unwanted consequences in applications can be mitigated by optimizing them for reduced peak power consumption, balancing power across different cores, and by minimizing the power density that can lead to thermal issues.

Energy is different from power in that it is the integral of power over time. Whenever a hardware resource is used, it just adds up to the energy cost. That being said, even after power has been optimized, application energy consumption, which directly translates to financial cost, does not get automatically reduced. Energy costs incurred by software applications have grown rapidly in recent years. In 2014, energy is projected to be a major expense for most major data centers, and is estimated to account for up to 3% of worlds total energy consumption [6]. However, not all applications are using hardware resources in an energy-efficient way. As instances– (1) unnecessary branches use branch units but the branch outcome never affects program execution. (2) redundant code repeatedly uses the processor pipeline but only for computing the same value. Because of that, Inefficient software has often been cited as a major reason for wasteful energy consumption in computing systems. To highlight the importance of application energy and its impact on data service efficiency, companies like eBay have online dashboards for the users to track energy consumption per transaction in their servers [7]. Such efforts clearly illustrate the motivation of major software giants in removing the inefficiencies in their applications.

While power and energy are two of the top issues in multi-core resource usages, there is a growing concern about information leakage on shared hardware resources. As processor core counts are increasing, more shared hardware resources are being integrated into the chip. For example, compute logics and execution engines are shared among virtual cores in processors that support SMT (Simultaneous Multithreading); the last level cache and the memory bus is shared among physical cores, etc. In recent years, it has been shown many times these shared resources are vulnerable one type of information leakage, namely, covert timing channels attacks. The inside malicious users can construct covert timing channels by intentionally modulating the timing of events on certain shared hardware resources to illegitimately reveal sensitive information to the outside world. Compared to software-based information leakage, hardware based approaches are mostly hidden and much harder to audit. As confinement mechanisms in software improve, such hardware-based information leakage are more likely to be exploited, and hence put the trust of systems at risk.

Currently, solving these resource usage problems on multicore hardware are of paramount importance. But, not having the right tools to address such problems just makes such tasks harder. Therefore, having the next generation hardware monitoring infrastructure to support auditing application power, energy and information leakage will greatly improve the productivity of programmers and system administers in addressing these new challenges.

1.3 Overview

This dissertation contains six chapters.

- Chapter 1 motivates the need for next generation hardware monitoring infrastructure for resource auditing, and states the reasons why the existing hardware infrastructure is not sufficient. It also briefly reviews power, energy, and information leakage problems resulted from the usage of processor hardware resources.

- Chapter 2 provides necessary background for this dissertation. We introduce specific problems in individual domains of application energy auditing, application power auditing, and covert timing channel auditing.

- Chapter 3 describes Watts-inside, a hardware-software cooperative approach for application power auditing. Watts-inside replies on the efficiency of hardware support to accurately gather application power profiles, and utilizes software support and causation principles for a more comprehensive understanding of application power.

- Chapter 4 describes enDebug, a hardware software cooperative framework for application energy auditing. EnDebug attributes energy consumption by applications to fine grain regions of program code (say functions), and utilizes an automated recommendation system to explore energy improvements in the program code.

- Chapter 5 describes CC-Hunter, a novel framework for covert timing channel auditing on shared hardware resources. CC-Hunter detects the presence of covert timing channels by dynamically tracking conflict patterns on shared processor hardware. It offers low-cost hardware support that gathers data on certain key indicator events during program execution, and provides software support to compute the likelihood of covert timing channels on a specific shared hardware.

- Chapter 6 presents conclusion of this work.

1.4 Scope of this Dissertation

The main goal of this work is to motivate the need for the next generation hardware infrastructure that is able to audit multicore resource usage problems, such as power, energy and information leakage. We describe our proposed solutions and the necessary hardware modifications to modern processors. We quantitatively evaluate our schemes either through simulators that model the modified processor platforms, or directly through real hardware machines where possible. For application power and energy auditing, we are interested in optimizing power or energy through code modifications. For information leakage auditing, we are interested in detecting covert timing channel activities on the shared hardware resources, rather than showing how to robustly construct these channels. Our work shows promising results that can be achieved from using application power, energy and information leakage auditing. We envision that this dissertation will stimulate more researches on improving the next generation hardware monitoring infrastructure to have further enhanced auditing capabilities that optimize applications for factors beyond just performance.

Chapter 2

Background

2.1 Application Power Auditing

Multicore processors have permeated the computing domains with their promise of higher performance and power efficiency over single core processors. With the limitations posed by Dennard scaling, power-related issues will be significant in future multicore chip designs and limit the scalability of multicore computing [8, 9, 10].

Conventional power saving strategies utilize dynamic, hardware-based solutions such as Dynamic Voltage-Frequency Scaling, Power and Clock Gating. Recent proposals have studied power management platforms that are either built as part of the the microprocessor circuitry [11, 12] or via tight interfacing with the external microcontrollers [13]. Unfortunately, such mechanisms are limited to dynamic power optimizations and can be cost-ineffective on applications that are *not statically tuned for power*. On the other hand, software-only power profiling tools are mostly disadvantaged by their limited knowledge of the underlying hardware parameters and complex interactions occurring between various functional units in multicore processors. Therefore, a more effective strategy is to combine the hardware efficiency in providing an accurate view of the program behavior with the software flexibility to effect low-cost, program-level power optimizations, ultimately improving the power consumption of multicore applications.

Our Watts-inside framework for power auditing combines the best of both hardware and software. The hardware performs power estimation and identifies the functional unit that was responsible for higher power in the captured code sequence. The software then does further analysis with finer grain code blocks (for instance, basic blocks) to figure out which region of program code was responsible for higher power.

Having information about specific code regions together with the functional units that are responsible for higher power, offers meaningful feedback to the users about how to apply power optimizations to the application. Such feedback can especially be more critical to heterogeneous multicore environment where the individual cores and the threads running on them have to be carefully tailored to maximize performance while reducing power. Our integrated approach to understanding multicore power can immensely aid in improving *application-level power awareness* and not just having to simply rely on the hardware-only mechanisms for power optimizations.

2.2 Application Energy Auditing

A plethora of solutions ranging from virtualization to application aware power management have been proposed to reduce system energy footprint. While such techniques are useful, a more effective solution is to incorporate energy smartness into the software itself such that the application performance can be aligned more closely with revenue. Being able to find the high energy consuming code region, and consequently, automate the process of energy optimization process would be vital to the future of energy-aware software development.

Conventionally, execution time of applications is a commonly adopted proxy measure for software developers to identify the energy bottlenecks in their program code. Recent studies by Hao et al [14] have shown that the execution time and energy consumption do not have a strong correlation because of several factors such as (1) multiple power states- at two different frequencies f_1 and f_2, even if the execution times are the same, the energy drawn will be different, (2) asynchronous design of system and API calls- when the application sends data over the network, the data is handled by the OS which results in the corresponding data-sending application not being charged for the data transmission time. These necessitate a dedicated framework for energy auditing.

Further, the application energy profile and optimizations are often specific to the processor architecture and hardware configurations. That is, a set of code optimizations that improve energy in one processor configuration does not necessarily improve the energy in other platforms. Without having a sound understanding of the underlying hardware details, many software-level energy models often resort to instruction-based or a specific hardware event-based (say, cache misses) energy accounting to derive application energy. Such strategies ignore the numerous and complex interactions between shared hardware resources and overlapped instruction execution that occur during application runtime. To overcome such problems, it is critical to debug the energy consumption of applications through profiles generated on the target architecture.

Our solution, enDebug, audits energy consumption by applications to fine grain regions of program code (say functions), and utilizes an automated recommendation system to explore energy improvements in the program code. In doing so, we enable the participation of software developers and toolchains (such as compilers and runtime) in energy-aware software development without necessarily having them to rely on expensive runtime energy saving strategies.

2.3 Information Leakage Auditing

Covert timing channels are information leakage channels where a trojan process intentionally modulates the timing of events on certain shared system resources to illegitimately reveal sensitive information to a spy process. Note that the trojan and the spy do not communicate explicity through send/receive or shared memory, but via covertly modulating certain events (Figure 2.1). In contrast to side channels where a process unintentionally leaks information to a spy process, covert timing channels have an insider trojan process (with higher privileges) that intentionally colludes with a spy process (with lower privileges) to exfiltrate the system secrets.

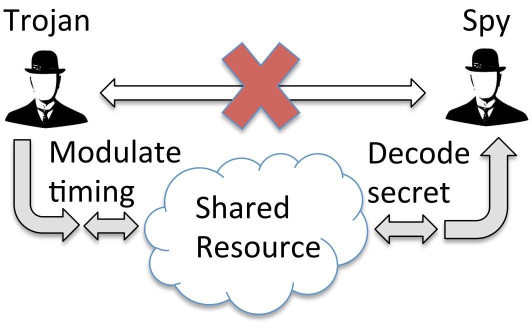


Figure 2.1: A Covert Timing Channel using timing modulation on a shared resource to divulge secrets

To achieve covert timing based communication on shared processor hardware, a fundamental strategy used by the trojan process is modulating the timing of events through intentionally creating conflicts[1]*. The spy process deciphers the secrets by observing the differences in resource access times.* On hardware units such as compute logic and wires (buses/interconnects), the trojan creates conflicts by introducing distinguishable contention patterns on the shared resource. On caches, memory and disks, the trojan creates conflicts by intentionally replacing certain memory blocks such that the spy can decipher the message bits based on the memory hit/miss latencies. This basic strategy of creating conflicts for timing modulation has been observed in numerous covert timing channel implementations [15, 16, 17, 18, 19, 20, 21, 22, 23].

Recent studies [19, 22] show how popular computing environments like cloud computing are vulnerable to covert timing channels. Static techniques to eliminate timing channel attacks such as program code analyses are virtually impractical to enforce on every third-

[1] We use "conflict" to collectively denote methods that alter either the latency of a single event or the inter-event intervals.

party software, especially when most of these applications are available only as binaries. Also, adopting strict system usage policies (such as minimizing system-wide resource sharing or fuzzing the system clock to reduce the possibility of covert timing channels) could adversely affect the overall system performance. Our framework, CC-Hunter, audits the presence of covert timing channels by dynamically tracking conflict patterns on shared processor hardware. CC-Hunter offers low-cost hardware support that gathers data on certain key indicator events during program execution, and provides software support to compute the likelihood of covert timing channels on a specific shared hardware. CC-Hunter's dynamic detection is a desirable first step before adopting damage control strategies like limiting resource sharing or bandwidth reduction. As a lightweight detection framework that audits on-chip hardware resources, our framework can be extremely beneficial to users as we transition to an era of running our applications on remote servers that host programs from many different users.

Chapter 3

Watts-inside: Application Power Auditing

3.1 Motivation – Understanding Multicore Power

To understand the power consumption behavior of applications, we perform experiments that characterize their power when executing on symmetric multicore processors. We note that more complex multicore environments that are asymmetric or heterogeneous can present even further challenges. In our studies, we run four-threaded applications on four core processors without placing any specific constraints on power consumption or voltage-frequency settings, i.e., the settings are assumed to result in the best possible execution time. We measure chipwide power during intervals of 10,000 cycles by running our benchmarks on SESC [24], a cycle-accurate architecture simulator with an integrated dynamic power model that uses Wattch [25] and Cacti [26] for power estimation. 32 nm technology is assumed in all of our experiments. Figure 3.1 shows dynamic power traces for a representative subset of our benchmark applications when executing on four-core processors. Our results indicate that different multicore applications can exhibit different characteristics during the various phases of their execution– (1) monotonically increasing power, e.g., cholesky, (2) phases of high and low power, e.g., ocean, (3) occasional peaks of high power, e.g., volrend, and (4) almost uniform power, e.g., fluidanimate.

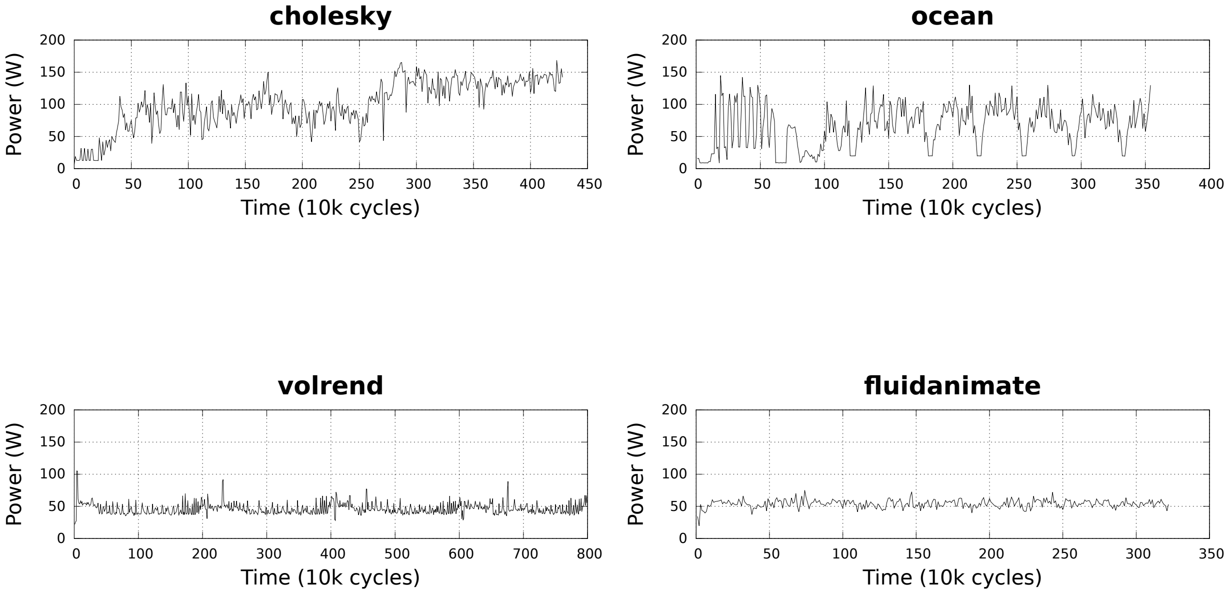


Figure 3.1: Dynamic Power traces for four-threaded applications measured during their execution

Even for applications that have been thoroughly debugged for performance and load balanced, our studies show that the parallel sections of multicore applications could still suffer from uneven power consumption between multiple cores. Table 3.1 shows parallel sections in some of the well-known Splash-2 and Parsec-1.0 applications that are running on four cores with four threads and shows the average imbalance in performance and power across several dynamic instances of the parallel section. Note that average power (or performance) imbalance in an application's parallel section is measured as the average difference between the threads having the highest and lowest power (or execution time). Despite almost perfect performance balance that can be achieved through hardware optimizations like out-of-order execution and prefetching, we see significant power imbalance (up to 31.7% in cholesky) across the different cores because power consumption by the functional units are still deter-

Table 3.1: Performance and Dynamic Power imbalance in SPLASH-2 and PARSEC-1.0 benchmarks with four threads

Application	Parallel Section (File/Function)	Num. of Dynamic Instances	% of Application Execution Time	Average Performance Imbalance	Average Power Imbalance
Volrend (SPLASH-2)	adaptive.c/ray_trace(...)	3	44.33%	0.001%	10.21%
Barnes (SPLASH-2)	load.C/maketree(...)	4	72.47%	0.90%	8.89%
Cholesky (SPLASH-2)	solve.C/Go(...)	1	28.09%	2.82%	31.74%
Bodytrack (PARSEC-1.0)	WorkerGroup.cpp/ WorkerGroup::Run()	82	50.06%	0.47%	4.32%

mined by the amount of work to be done. These results are consistent with a recent survey by Chen et al [27] and *show the necessity to understand the application's power characteristics in greater detail in order to accurately effect changes that improve power consumption.*

3.2 Design Overview

3.2.1 Hardware Support

To improve power, the user (programmer, compiler or the hardware) should first understand which parts of the program code suffer from power-related issues and what functional units are responsible for this effect. We design hardware support that estimates dynamic power for a string of N consecutively executing basic blocks (which we call Code Sequence), and log its power information in memory for further analysis. A code sequence is chosen as a granularity in our hardware design to capture meaningful power information that is relatable back to program code, while minimizing the hardware implementation complexity. In our

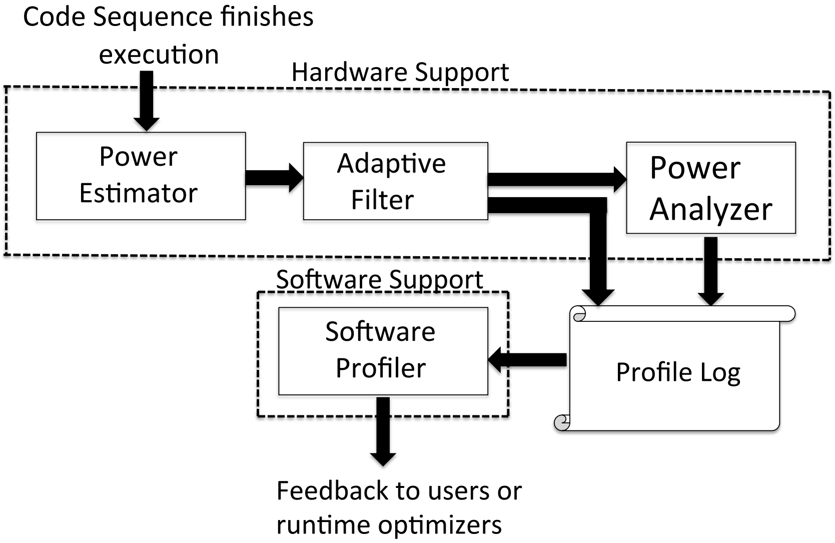


Figure 3.2: Design overview of the Watts-inside framework

experiments, we assume N=5 because it offers a nice trade-off between capturing power information at finer granularity and accuracy of power measurement on overlapped instructions. Sometimes, a code sequence can contain fewer than N basic blocks in cases of a function call/return and exceptions; we terminate such code sequences prematurely to prevent them from straddling program function boundaries and exceptions.

Figure 3.2 depicts an overview of our hardware-software design for Watts-inside framework. Conceptually, we divide the hardware support for Watts-inside into three stages, namely,

Power Estimator

This module is responsible for computing (or estimating) the power consumption of code sequences. The processor chip is embedded with activity sense points inside various functional units which are monitored by a power estimator unit. In our design, this module is conceptually similar to the IBM Power7's power proxy module that has specifically architected and programmably weighted counter-based architecture to keep track of activities and form an aggregate value denoting power [13].

Adaptive Filter

This module is responsible for filtering code sequences that are essentially 'uninteresting' with respect to power and do not warrant a *second hardware-level* analysis for functional unit-specific power information. Note that, when needed, the software profiler has the capability to analyze all code sequences regardless of filtering.

The adaptive filter has two active parameters– (1) maximum power so far (observed from the start of the application execution), and (2) capture ratio (C), a *user-defined* parameter that specifies the threshold for code sequences whose average power fall within the top C% of highest power (e.g., if the capture ratio is 10% and the highest power for any code sequence so far is 50 W, then the filter forwards all of the code sequences whose power consumption is at least 45 W.). *In the remaining sections of this Chapter, we refer to high power code sequences as ones within the capture ratio, and the remaining code sequences as low power (or NOT high power) for simplicity.*

To detect the high power sequences, the filter checks whether the code sequence (Q's) power falls within or exceeds the high power range. If true, then the filter forwards Q to the Power Analyzer for further processing. Whenever Q's power exceeds the maximum power observed thus far in the application, maximum power and threshold are updated. We note that after the maximum power reaches a stable value (i.e., after the highest power consuming sequence has executed), updates are no longer necessary.

Power Analyzer

This module is responsible for estimating the contribution of individual microarchitectural (or functional) units for high power code sequences, and then determining the functional unit that was responsible for the highest amount of power. We forward the output of this stage to a log that can be further analyzed by software profilers.

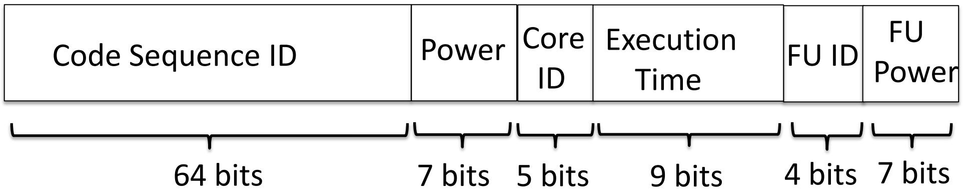


Figure 3.3: Code Sequence Power Profile Vector (CSPPV)

For design efficiency, we adopt common activity based component power estimation that can estimate power for a large number of functional units using just a few generic performance counters [28]. We identify fourteen functional units (Instruction and Data Translation Lookaside Buffers, Instruction and Data Level-1 caches, Branch Predictor, Rename logic, Reorder Buffer, Register File, Scheduler, Integer ALU, Float ALU, Level-2 cache, Level-3 cache and Load Store Queue) to study the power breakdown by individual units . We chose these fourteen units based on our analysis of functional unit-level power consumption across our benchmark suites.

Figure 3.3 shows the output of the Watts-inside hardware. For each code sequence, we construct a 96-bit long Code Sequence Power Profile Vector (CSPPV) that includes:

- Code Sequence ID: The power estimator generates a unique 64-bit identifier for every code sequence by folding the 32-bit address of the first basic block, and then concatenating lower order bits of other constituent basic blocks within the code sequence.

- Code Sequence Power: The power estimator uses 7 bits to store the code sequence power.

- Core ID: 5 bits are used for the core ID where the code sequence executed, filled by power estimator.

- Execution time: The power estimator uses 9 bits to store the execution time of the code sequence. This can be later used for: (1) computing energy, and (2) ranking code sequences to prioritize longer running blocks.

- FU ID: The power analyzer uses 4 bits to uniquely identify the one of the fourteen functional units that consumes the most power.

- FU Power: The power analyzer uses 7 bits to show power consumed by the highest power consuming functional unit.

The power analyzer module records the CSPPV into a memory log that can later be utilized by software profilers.

3.2.2 Software Support

Causation probability

To help programmers and compilers apply targeted power-related optimizations to program code, feedback must be given at the level of fine-grain code blocks (say, a few instructions within a basic block). Toward this goal, we develop a causation probability model to determine whether an individual basic block within a code sequence could cause higher power.

Watts-inside quantifies the impact of a certain basic block B on the power of the code sequence Q using three probability metrics:

- Probability of Sufficiency (PS): If B is present, then Q consumes high power. A higher range of PS values indicate that the presence of B is a sufficient cause for Q's high power consumption.

- Probability of Necessity (PN): Among Qs that consume high power, if B were not present, then Q would have not consumed high power. A higher range of PN values indicate that the absence of B would have caused Q to lower its power.

- Probability of Necessity and Sufficiency (PNS): B's presence is both sufficient and necessary to infer that Q consumes high power. Higher values of PNS prove that B's likeliness to be the reason behind Q's higher power.

To compute the boundaries of PS, PN and PNS, we define the following additional probability terms:

Let b be the event that a basic block B occurs in a code sequence, and h be the event that the code sequence consumes high power. $P(h_b)$ denotes counterfactual relationship between b and h, i.e., the probability that if b had occurred, h would have been true.

$$P(h) = (\#HighPowerSeq)/(\#Seq) \tag{3.1}$$

$$P(b,h) = (\#HighPowerSeqWithB)/(\#Seq) \tag{3.2}$$

$$P(b^{'},h^{'}) = (\#LowPowerSeqWithoutB)/(\#Seq) \tag{3.3}$$

$$P(h_b) = (\#HighPowerSeqWithB)/(\#SeqWithB) \tag{3.4}$$

$$P(h_{b'}) = (\#HighPowerSeqWithoutB)/(\#SeqWithoutB) \tag{3.5}$$

$$P({h^{'}}_{b'}) = (\#LowPowerSeqWithoutB)/(\#SeqWithoutB) \tag{3.6}$$

The boundary values for PS, PN and PNS are defined below:

$$max\left\{0, \frac{P(h_b) - P(h)}{P(b^{'},h^{'})}\right\} \leq PS \leq min\left\{1, \frac{P(h_b) - P(b,h)}{P(b^{'},h^{'})}\right\} \tag{3.7}$$

$$max\left\{0, \frac{P(h) - P(h_{b'})}{P(b,h)}\right\} \leq PN \leq min\left\{1, \frac{P({h^{'}}_{b'}) - P(b^{'},h^{'})}{P(b,h)}\right\} \tag{3.8}$$

$$PNS \geq max\left\{\begin{array}{l} 0, P(h_b) - P(h_{b'}), \\ P(h) - P(h_{b'}), P(h_b) - P(h) \end{array}\right\} \tag{3.9}$$

$$PNS \leq min\left\{\begin{array}{l} P(h_b), P({h^{'}}_{b'}), P(b,h) + P(b^{'},h^{'}), \\ P(h_b) - P(h_{b'}) + P(b,h^{'}) + P(b^{'},h) \end{array}\right\} \tag{3.10}$$

By using the boundary equations 3.7- 3.10, we present a few test cases below to verify our causation model:

- If a basic block B appears frequently in high power code sequences and sparsely in low power sequences, both PS and PN boundary values are very high (closer to 1.0). Consequently, PNS values are also very high. Such blocks *are certainly* candidates for power optimization. For example, if there are 1000 code sequences, of which 200 are classified as high power (via capture ratio C). Let us assume that B_1 appears in 100 of the high power code sequences, and does not appear in any low power sequences. Using the boundary equations, we find that $1 \leq PS \leq 1$ and $0.9 \leq PN \leq 1$. These high PS and PN values show that B_1 is certainly a candidate for power optimizations.

- If a basic block B appears sparsely in high power code sequences, both PS boundary values are closer to 0.0, and the PN boundary values are either a widely varying range or are closer to 0.0. Such blocks *cannot* be good candidates for power improvement considerations. Using the same example above, let us assume that the block B_2 appears in 5 of the 200 high power code sequences and B_2 appears in 95 of the 800 low power (NOT high power) code sequences. Using the boundary equations, we find that $0 \leq PS \leq 0.06$ and $0 \leq PN \leq 1$. Low PS values combined with practically unbounded PN values indicate that B_2 cannot be a good candidate for power improvement.

- If a basic block B appears $L\%$ of the time in high power code sequences and $M\%$ of the time in low power sequences (where L and M are non-trivial), PNS boundary values determine the degree to which B's likeliness in causing higher power in the corresponding program code sequences. Therefore, higher ranges of PNS values for B indicates higher benefit in applying power-related optimizations to B. Using the example described above, let us consider two blocks B_3 and B_4– (1) B_3 appears in 40 of the 200 high power code sequences and in 200 of the 800 low power code sequences, where $0 \leq PNS \leq 0.167$, and (2) B_4 appears in 35 of the 200 high power sequences and 20 of the 800 low power sequences, where $0.462 \leq PNS \leq 0.636$. Even though B_3 appears more frequently in high power code sequences than B_4, there is higher benefit to optimizing B_4 because of its *larger* high power causation probability.

We find that this approach mathematically helps us to quantify the degree to which a specific set of instructions result in higher power consumption.

Code Sequences with varying power consumption between cores

Our software support can improve the quality of feedback information via two mechanisms – (1) Use clustering algorithms (e.g., k-means) to cluster sequences based on the degree of power variation, i.e., code sequences that show higher power variation are clustered separately from the ones that have lower power variation. This can aid runtime systems to do better scheduling of threads and map them on to cores that satisfy their power needs. (2) Identify the cause for power variation using the CSPPVs. Since the vector contains information on functional unit consuming the highest power, it can facilitate targeted optimizations including code changes and dynamic recompilation.

Predicting potential for Thermal Hotspots

By monitoring a contiguous stream of code sequences executing on the same core where a functional unit repeatedly contributes to the highest portion of power, we could predict parts of the chip where thermal hotspots could develop. Also, by having information on the physical chip floorplan, we can even detect local thermal hotspots resulting out of continuously high activity in adjacent functional units. Such analysis can effectively help temperature-aware software development of multicore applications.

3.3 Implementation

In this section, we show how our framework can be integrated with a modern multicore architecture.

3.3.1 Hardware Support

Figure 5.8 shows the hardware modifications needed to implement Watts-inside framework. We include the power estimator (similar to the modules found in modern processors like Intel Sandybridge, IBM Power7) and adaptive filter logic locally in every core. After power

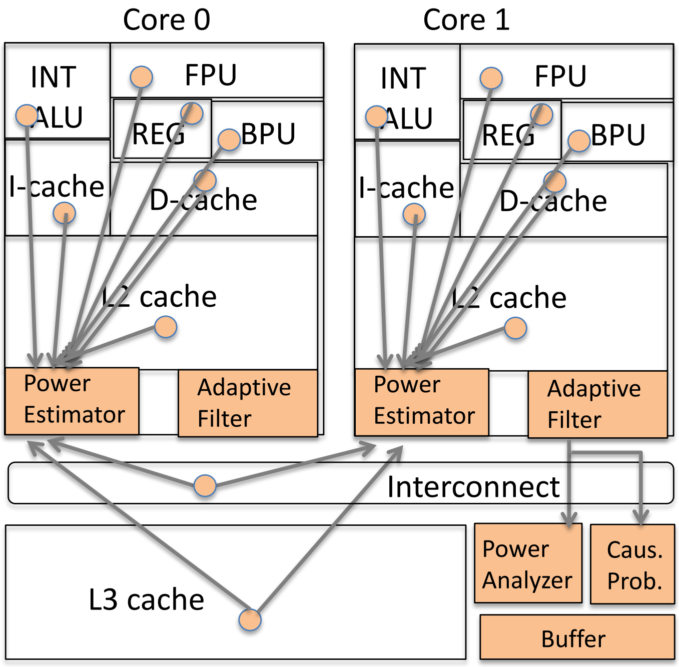


Figure 3.4: Hardware modifications needed for Watts-inside framework

estimation, our adaptive filter determines whether this block warrants further processing. To do this, there are two special registers- a programmable register to store the user-desired capture ratio, and an internal register to hold maximum power observed so far.

We implement the power analyzer module as a centralized resource that is shared by all cores within a multicore chip. The adaptive filters inside the cores forward only the high power code sequences to the power analyzer.

To reduce the performance impact of hardware profiling, we consider two more optimizations– (1) hardware buffer to accumulate the CSPPVs and update memory when the bus is idle, and (2) sampling of code sequences to minimize the impact on multicore performance.

Additionally, we implement an online hardware causation probability module and a watch register (that can be programmed by the user with a specific basic block address). This is conceptually similar to setting watchpoints in program debuggers. The adaptive filter forwards all of the code sequences that contains the *basic block address under watch* to the hardware causation probability module, that in turn computes the PS, PN and PNS values. We believe that such a feature shall aid runtime systems like dynamic recompilation or adaptive schedulers to optimize specific code regions during program execution.

3.3.2 Software Support

We run the software profiler as a separate privileged process in the kernel mode. The profiler supports APIs for functions such as (1) querying which basic blocks have high power causation probability (note: this offline software implementation is more comprehensive and separate from the online hardware causation probability module in Section 3.3.1), (2) automatically mining the CSPPVs for basic blocks that cause higher power. This software profiler gets its input from the CSPPV log created by the power analyzer. The memory pages belonging to CSPPV log are managed by the Operating System and are allocated on demand. If the OS senses that memory demands of CSPPV log interferes with the performance of regular applications, the OS pre-emptively deallocates certain memory pages and/or alter the sampling rate of code sequences to minimize the memory demands of CSPPV log. Also, we use Lempel-Ziv coding to compress and decompress CSPPV logs [29], that helps us to reduce memory footprint sizes.

3.4 Experimental Setup

We use SESC, a cycle accurate architectural multicore simulator [24] that has an integrated power model. We model a four-core Intel Core i7-like processor [30] running at 3 GHz, 4-way, out-of-order core, each with a private 32 KB, 8-way associative Level 1 cache and a private 256 KB, 8-way associative Level 2 cache. All cores share a 16 MB, 16-way set-associative Level 3 cache. The Level 2 caches are kept coherent using the MESI protocol. The block size is assumed to be 64 Bytes in all caches. We use parallel applications from Splash-2 [31] and PARSEC-1.0 [32] that were compiled by gcc with -O3 flag, and run four-threaded version on

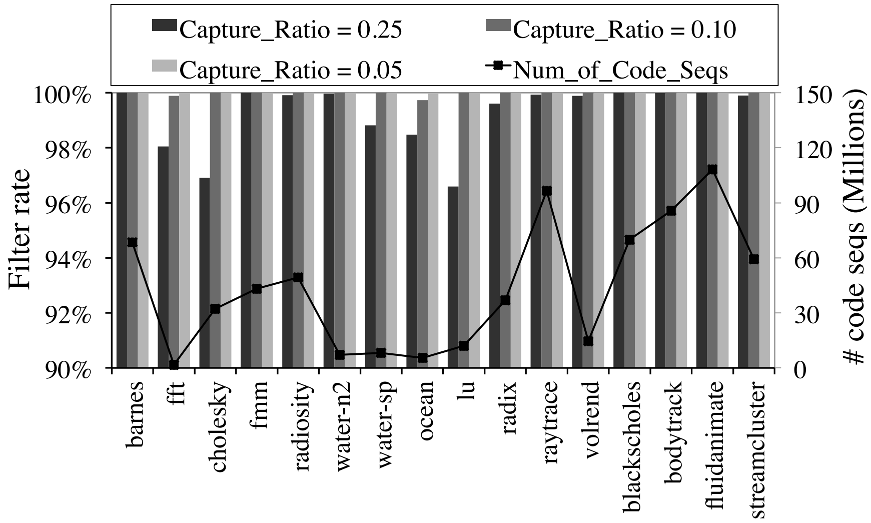


Figure 3.5: Ideal Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads.

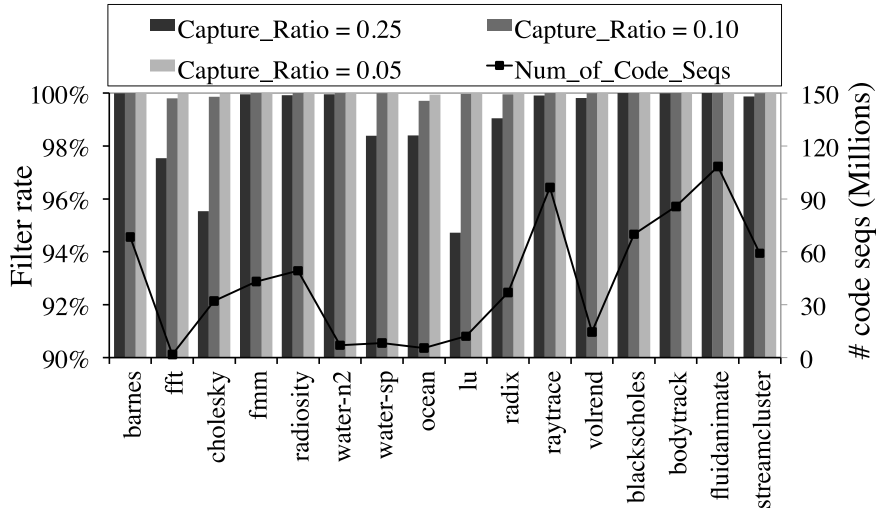


Figure 3.6: Adaptive Filter rate for capture ratios of 0.25, 0.1 and 0.05. The right axis shows the number of code sequences that are executed across all four threads.

four cores.

3.5 Evaluation

3.5.1 Adaptive Filter vs. Ideal Filter

In this experiment, we compare the effectiveness of our adaptive filter (that *adjusts its threshold dynamically* to filter code sequences based on the maximum power seen thus far and the capture ratio) against an ideal filter (that *does not need to adjust the threshold dynamically* because it has prior knowledge of the maximum power consumed by any code sequence in the multicore application and the capture ratio). Figures 3.5 and 3.6 show the results of our experiments. For each benchmark, we show the percentage of code sequences that are filtered for three separate capture ratios namely 0.25 (or code sequences within top 25% of maximum power), 0.10 and 0.05 respectively. On the right axis, we show the total number of code sequences that are executed by each application. As an example, cholesky benchmark executes a total of 32.13 million code sequences; at a capture ratio of 0.25, 96.9% of the code sequences are filtered by ideal filter and 95.5% of the code sequences are filtered by adaptive filter.

Based on our experiments, we notice that in a majority of benchmarks, except fft, cholesky and lu, our adaptive filter successfully filters above 99% of the code sequences (for all three capture ratios) and sends only ≤1% code sequences to the power analyzer module for further analysis. These filter rates are nearly same as that of the ideal filter. In fluidanimate that has a large number of code sequences, our adaptive filter performs nearly equal to the ideal filter in minimizing the number of sequences that are sent to the Power Analyzer. In certain benchmarks like lu, our adaptive design filters up to 3.8% less than an ideal filter, especially for capture ratio of 0.25. However, lu has fewer than 12 million code sequences and the absolute numbers of code sequences that reach power analysis stage are still far fewer than the benchmarks with hundreds of millions of code sequences. Therefore, we conclude that

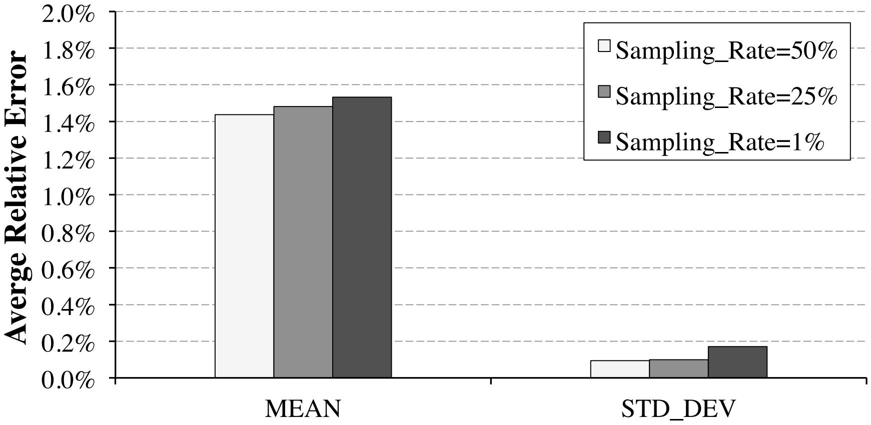


Figure 3.7: Average relative error in the mean and standard deviation of power consumption by code sequences (in Splash-2 and Parsec-1.0 applications) due to sampling, relative to a baseline execution without sampling.

our adaptive filter design proves effective and is able to perform very close to an ideal filter.

3.5.2 Sampling

Even after filtering, for certain applications, the number of CSPPVs might still be high enough to cause significant performance overheads. To minimize the traffic of code sequences that reach the power analyzer module from various cores, we perform periodic sampling, i.e, one out of every N code sequences is chosen by Watts-inside framework for power estimation and analysis. Figure 3.7 shows the results of our experiments when we sample code sequences at the rates of 50%, 25% and 1%, and compare the observed mean and standard deviation of code sequence power with the baseline execution where we do not have sampling. At 50%, we note that periodic sampling introduces fairly low relative error of about 1.4% on mean code sequence power and approximately 0.10% on standard deviation; at lower sampling rates, these relative errors are slightly worse. *One caveat with aggressive sampling (such as 1%) is that we might only see fewer CSPPV samples, that may result in inability to accurately assess PNS, PS and PN probability values for certain basic blocks that are omitted due to*

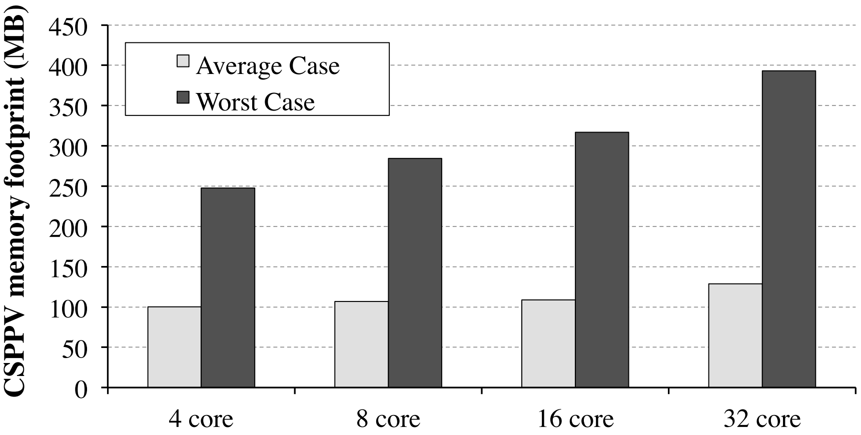


Figure 3.8: Average and worst-case CSPPV Memory log requirements for capture ratio=0.25 for different numbers of cores

sampling.

3.5.3 Scalability of CSPPV Memory Log

We study the average and worst-case CSPPV memory footprint sizes for different numbers of cores after applying Lempel-Ziv compression. In other words, we measure the total memory needed for all of the CSPPVs from start to end of application execution. We note that the OS does not need to store the entire log, and could minimize the log size by periodically deallocating the memory pages that have already been processed by the software profiler. Figure 3.8 shows that the average-case memory requirements for many of our benchmarks are between 100 MB and 125 MB due to the efficiency of LZ compression (that offers up to 70% compression ratio). The worst case memory requirements are observed in fluidanimate benchmark which needs between 250 MB (4 cores) and 395 MB (32 cores).

3.5.4 Area, Power and Latency of Watts-inside Hardware

To obtain the area, power and latency of Watts-inside hardware, we create a Verilog-based RTL model of the power estimator, power analyzer, and hardware causation probability mod-

Table 3.2: Area, Power and Latency Estimates of Watts-inside hardware

	Power Estimator (×4)	Power Analyzer	Causation Prob. module	Buffer (4 KB)
Area(mm^2)	0.18	0.11	0.09	0.03
Power(mW)	9.08	8.72	4.53	49.70
Latency† (CPU cycles)	24	38‡	26‡	NA^*

†Based on instruction latencies of Intel Core i7 [30]

‡Not significant at runtime due to efficient filtering of code sequences

*Accessed when memory bus is free

ules. We use Synopsys Design Compiler (ver G-2012.06) [33] and FreePDK 45nm standard cell library [34] to synthesize each module. Table 3.2 shows the results of our experiments. We note that the area requirements for Watts-inside are modest and are about 0.2% of total onchip area of 4-core Intel Core i7 processor (263mm^2) [30]. Power requirements are less than 0.06% of 130 W peak power. *Since our hardware is designed to be off the critical path of the processor pipeline, we did not observe any significant performance impact in applications.*

3.5.5 Case study – Improving Load/Store Queue power consumption

In this subsection, we show how our Watts-inside framework offers a hardware-software cooperative solution in identifying and analyzing program code, and eventually improving the power consumption of the processor. Specifically, we pick two benchmark applications (ocean and streamcluster) that suffer from high load/store queue power.

We first modify SESC simulator to implement our framework. We then run our two case-study benchmarks (4-threaded version), and identify the portions of program code (shown in Figure 3.9) that consume high power. In ocean, the measured chip-level power is 73 W, and the instructions within the identified loop have $0.972 \leq PS \leq 1.0$, $0.70 \leq PNS \leq 0.72$. In streamcluster, the measured chip-wide power is 58 W, and the loop instructions under study have $0.968 \leq PS \leq 1.0$, $0.76 \leq PNS \leq 0.78$. In other words, for both benchmarks, Watts-inside indicate that the instructions within the loops have very high probabilities of sufficiency for high power within their corresponding code sequences.

```
//Ocean(Splash-2):main.cpp:337
...
for (i=numlev-2;i>=0;i--) {
      imx[i] = ((imx[i+1] - 2) / 2) + 2;
      jmx[i] = ((jmx[i+1] - 2) / 2) + 2;
      lev_res[i] = lev_res[i+1] * 2;
}
...

------------------------------------

//Streamcluster(Parsec-1.0):streamcluster.cpp: 657
...
accumweight= (float*)malloc(sizeof(float)*points->num);
accumweight[0] = points->p[0].weight;
totalweight=0;
for( int i = 1; i < points->num; i++ ) {
     accumweight[i] = accumweight[i-1]+points->p[i].weight;
}
...
```

Figure 3.9: Code snippets from ocean and streamcluster benchmarks where store-to-load dependencies result in high power

In both of the code sections, we observe a store-to-load dependency that results in a forwarding operation in load/store queue between the array elements across two iterations of the loop, i.e, the element that is stored in the previous iteration of the loop is loaded in the next iteration again. To reduce this unnecessary forwarding between the two iterations,

```
//Ocean(Splash-2):main.cpp: 337
//**No more Store-to-Load Dependencies**
...
t1 = imx[numlev-1];
t2 = jmx[numlev-1];
t3 = lev_res[numlev-1];
for (i=numlev-2;i>=0;i--) {
      imx[i] = ((t1 - 2) / 2) + 2;
      jmx[i] = ((t2 - 2) / 2) + 2;
      lev_res[i] = t3 * 2;
      //Storing array elements in temp
      t1 = imx[i];
      t2 = jmx[i];
      t3 = lev_res[i];
}
...

-----------------------------------

//Streamcluster(Parsec-1.0):streamcluster.cpp: 657
//**No more Store-to-Load Dependencies**
...
accumweight= (float*)malloc(sizeof(float)*points->num);
accumweight[0] = points->p[0].weight;
totalweight=0;
t = accumweight[0];
for( int i = 1; i < points->num; i++ ) {
     accumweight[i] = t+points->p[i].weight;
     //Storing array elements in temp
     t = accumweight[i];
}
...
```

Figure 3.10: Modified code snippets from ocean and streamcluster benchmarks that no longer have store-to-load dependencies

we modify the code to include temporary variables that store the value from previous iteration and supply this value to the next iteration [35]. The code modifications are shown in Figure 3.10.

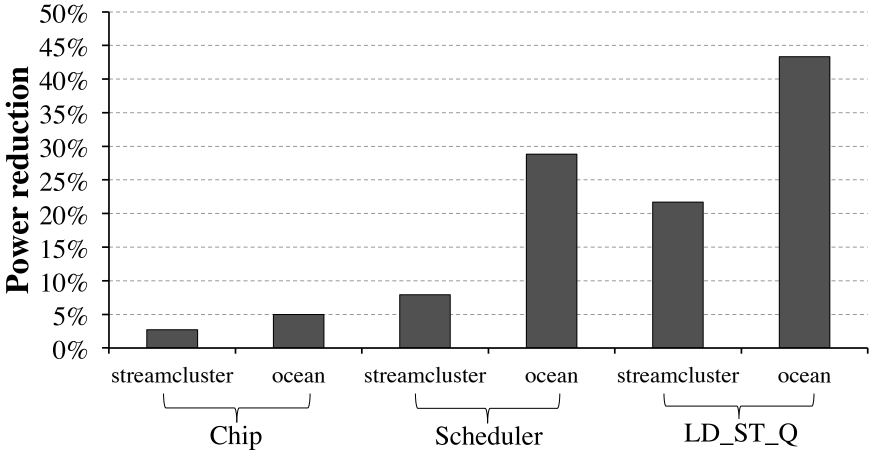


Figure 3.11: Power reduction across the entire chip, scheduler and load/store queue after removing store-to-load dependencies

As a result of this code optimization, we find an improvement in chip-wide power consumption in both benchmarks. An interesting side-effect of our code modification was the reduction in the number of memory load instructions in each loop iteration due to replacement of memory load with operations on temporary registers, that consequently showed reduction in scheduler power. Figure 3.11 shows the results of our experiments. In streamcluster, we observe an average savings of 2.72% for chip power (and up to 21% reduction in load/store queue and 7% savings in scheduler power consumption) with a slight 0.25% speedup in execution time; In ocean, we get an average savings of 4.96% in chip power (and up to 43% reduction in load/store queue power and 28% savings in scheduler power consumption) with a slight 0.19% speedup in execution time.

From this case study, we observe the usefulness of understanding application power and how the feedback information can be utilized in meaningful ways to improve power behavior of multicore applications. We note that, in this particular case study of removing store-to-load dependencies, many compilers typically are unable to optimize code in a way that avoids store-to-load-dependency [35]. In some cases, the language definition prohibits the compiler from using code transformations that might remove store-to-load dependency. Therefore, a framework like Watts-inside, that offers a hardware-software cooperative solution to understanding and improving multicore power, can be an invaluable tool for multicore software developers.

3.6 Related Work

Several prior works have proposed techniques to reduce power consumption of processor components through hardware optimizations. These studies include caches with variable associativity or number of banks [36, 37], bypassing expensive tag checks [38], trading cache capacity for low supply voltage [39], dynamically adjusting issue or load/store queue sizes and avoiding wasteful wake-up/select checks [40, 41, 42]. Recent works have considered mapping program computation structures onto dynamically synthesized energy-efficient hardware accelerators [43, 44], as well as designing entire pipelines for low power [45, 46]. Prior works have employed DVFS-based power optimizations by taking advantage of slack time available for threads [4, 47, 48] or using compiler-assisted profiling techniques [49, 50, 51]. Our Watts-inside framework can synergetically work with the above prior techniques to improve power based on the observed application behavior.

Tiwari et al. [52] developed an instruction-level power model which attributes an energy number to every program instruction. Such methods are difficult to use in multicore processors that have complex interactions between functional units during instruction execution. Profiling tools utilize hardware program counters to provide procedure-level feedback [53, 54, 55]. Cycle-accurate simulation tools for energy consumption have been proposed [56, 57]. The disadvantages with the software-only profiling tools and simulators are: (1) Simulators and software profiling tools can be prohibitively slow for analyzing real-world applications, (2) Building software simulators that accurately capture all microarchitectural aspects is nearly impossible, especially without fully understanding of the underlying hardware. In contrast, our Watts-inside proposes a hardware-software cooperative solution, where hardware provides reliable information of program execution and the software offers flexible platform to analyze program power.

Isci et al. [58, 59] have proposed runtime power monitoring techniques for core and functional units. CAMP [28] and early stage power modeling [60] show how to use limited number of hardware statistics for power model. Unlike these prior techniques whose primary goal is developing accurate power models for core and the functional units, our Watts-inside framework offers application-level feedback to improve power.

3.7 Summary

In this chapter, we showed the necessity to gather fine-grain information about program code to better characterize application power and effect improvements. We proposed Watts-inside, a hardware-software cooperative framework that relies on the efficiency of hardware support to accurately gather application power profiles, and utilizes software support and causation principles for a more comprehensive understanding of application power. We presented a case study using two real applications, namely ocean (Splash-2) and streamcluster (Parsec-1.0) where, with the help of feedback from Watts-inside, we performed relatively straightforward code changes and realized up to 5% reduction in chip power and slight improvement ($\leq$ 0.25%) in execution time.

Chapter 4

enDebug: Application Energy Auditing

4.1 Understanding Application Energy

Table 4.1: Energy and performance profile of functions in Splash-2 and PARSEC-1.0 benchmarks with 8 threads

Ocean			Radiosity			Bodytrack		
Function	% of Energy	% of Time	Function	% of Energy	% of Time	Function	% of Energy	% of Time
relax	30.31%	15.02%	v_intersect	11.06%	6.60%	InsideError	25.38%	9.12%
slave2	18.98%	30.30%	compute_diff_disc_formfactor	7.39%	14.07%	Exec	19.20%	4.34%
jacobcal2	14.47%	12.22%	traverse_bsp	4.83%	5.53%	EdgeError	18.53%	6.77%
laplacalc	12.68%	9.85%	four_center_points	3.03%	5.06%	ImageProjection	10.66%	3.67%

With increasingly complex interactions between instruction execution and the associated timing in functional units (due to parallelism and pipelining), execution time can no longer be considered a good proxy for energy measurement. To illustrate this effect, we conduct experiments on several real-world applications from Splash-2 [31] and PARSEC-1.0 [32] benchmark suites, where the energy consumption characteristics of individual functions drastically differ

Table 4.2: Processor Configuration and Power Model

Processor	3 GHz, 8-core CMP; 4-wide issue/retire, out-of-order execution; 4096-entry BTB; hybrid branch Predictor; 8-entry instruction queue; 176-entry ROB; 96 int registers; 90 fp registers; 64-entry LD/ST queues; 48-entry scheduler
Memory Sub-system	32KB, 4-way, I-cache; 32KB, 4-way, D-cache; 256KB, 8-way, private L2 cache; 16MB, 16-way, shared L3 cache; 64-entry ITLB/DTLB
Interconnect	shared bus below private L2 caches
Power Model	McPAT, 32 nm, $V_{dd} = 1.25$ V

from their corresponding execution time profiles.

Table 4.1 shows the energy and execution time profiles on several applications with 8 threads running on 8 cores. All of our experiments were done using SESC [24], a cycle-accurate, multi-core architecture simulator that is integrated with McPAT power model [61]. Table 4.1 shows the processor configuration details input to the McPAT power model.

1. **Ocean**: relax() consumes 30.31% of the total energy but only accounts for 15.02% of the total execution time. On the other hand, slave2() accounts for 30.30% of execution time, but only consumes 18.98% energy. Upon further examination, we observed *highly overlapped execution* of double-word arithmetic instructions in relax() led to higher energy with lower execution time. However, slave2() had higher numbers of branch and load/store instructions leading to longer execution time despite consuming lower energy than relax().

2. **Radiosity**: v_intersect() consumes 11.06% of total energy with only 6.60% of the total execution time, while compute_diff_disc_formfactor() has 14.07% of the total execution time with only 7.39% of the total energy. On a closer review, we found that v_intersect() heavily used complex instructions like madd.d (that perform multiply-add of double word values) leading to higher energy, while compute_diff_disc_formfactor() had a lot of load operations leading to higher execution time despite consuming lower energy than v_intersect().

3. **Bodytrack**: The top four energy consuming functions account for 74% of the total en-

ergy, but only account for about 24% of the total execution time. About 64% execution time is actually spent on lock and barrier synchronizations implemented by pthread_cond_wait() that actually puts threads into sleep without consuming much energy.

The examples above clearly show that an energy auditing framework is necessary to better understand the energy profile of applications beyond just performance, and use this profile information for energy optimization.

Are Current Hardware Energy Meters Sufficient? Modern high performance processor architectures [62, 11] have begun integrating hardware energy meters that can be read through software driver interfaces. For example, starting from Sandy Bridge, Intel provides a driver interface called RAPL (Running Average Power Limit) that can let programmers periodically sample processor energy usually at the granularity of a few milliseconds of program execution time. While this is going to be a useful first step toward helping programmers to understand the processor energy consumption, it is still far from providing them with a more practical feedback at a granularity that relates the processor energy consumption back to the program source code.

4.2 Fine-grain Energy Profiling

While our power auditor captures short code sequences that cause high power, our energy auditor cares more about the amount of accumulated energy consumption in code regions. Due to such different auditing goals, we need to design a separate energy profiler, and based on that, look for energy optimizations.

To help compilers, runtime optimizers or even programmers effectively apply the energy optimizations to the right code regions, energy profile information is needed at the level of fine-grain code, say functions or certain critical loop structures. We note that the current hardware energy profiling infrastructure such as RAPL interface [62] provide energy information at a granularity of several microseconds to a few milliseconds. To bridge the gap between the current hardware support and the fine-grain energy profile (needed to attribute energy back to application source code), we undertake a two step strategy– *First*, we build a energy model (regression) using certain well-known hardware performance counters. *Second*, we show how fine-grain energy for functions can be obtained using simple hardware support and the energy regression function from our first step. Essentially, our solution is able to estimate energy and attribute them at the granularity of program functions without requiring extensive hardware support.

We note that the first step (building the energy model) can be done on many current processor platforms that have support for hardware energy and performance counters (which can be read using lightweight tools such as likwid [2]) . However, the second step of fine grain profiling and attribution does not exist in any modern processor platform to the best of our knowledge. Therefore, to provide an accurate view of our energy profiler, we implement the entire fine-grain energy profiling framework using SESC [24], an cycle-accurate, out-of-order issue, multi-core processor simulator with an integrated McPAT energy model [61].

4.2.1 Energy Model Using Performance Counters

Current hardware support to measure energy and power consumption [62] are at a very coarse granularity that cannot be directly used by code optimizers. Therefore, to attribute energy consumption to fine grain regions of code (say functions), we first build an energy model that estimates energy consumption through regression on a set of hardware performance counters.

Prior works [63, 58, 59, 28, 60] have shown that linear regression techniques can be effective in estimating processor energy with high accuracy. The energy model builds a relationship between the processor energy, denoted as E, and a set of key performance metrics, denoted as P. Most modern processors have hardware performance counters that can dynamically capture many critical performance metrics made available through the performance monitoring infrastructure. With this existing hardware support, we build our energy model as follows: We choose a subset of benchmarks from Splash-2 [31] and PARSEC-1.0 [32] to train our energy model. In each benchmark run, for every 1 million cycle windows, we gather CPU clock cycles, Instruction count, L1 Data cache access count, L2 cache access count, Floating point operation count through hardware performance counters. We also tabulate the corresponding energy consumption within the observation period from the energy counters. Using the measured features, the energy consumption E (in nanojoules) is estimated as a function of the key performance metrics.

$$\begin{aligned} E =& 1.347 * CPUCycles + 0.484 * InstCount + \\ & 0.867 * L1DCacheAccess + 1.097 * L2CacheAccess + \\ & 0.104 * FloatOps \end{aligned}$$

A separate set of Splash-2 and PARSEC-1.0 benchmarks (non-overlapping with the training set) are used for testing and validation. We calculate the relative error between the estimated core energy and the ground-truth energy numbers for the core using McPAT [61]. We adopt Ten-fold Cross-validation [64] method where 90% of the samples are used as the training set and the remaining 10% of them are used for validation. This step is repeated ten times where a different validation set is selected during each time. In our experiments, we found that the average cross-validation errors is 2.13%, and the worst-case absolute error to be less than 12%.

4.2.2 Attributing Energy to Program Functions

To attribute energy back to the program source code, an important question that needs to be answered is the granularity at which energy should be attributed. Note that the program code can be analyzed at different granularities, for example, instructions, basic blocks, functions, or whole programs. Attributing energy to individual instructions and basic blocks are practically very difficult because modern superscalar processors can execute multiple instructions (and basic blocks) in an overlapped fashion. This makes it very hard to accurately attribute energy back to each of such entities. On the other hand, with the energy profile information at larger granularities such as whole program level, we may not have meaningful feedback to effect code changes. Therefore, we attribute energy to program functions, that already have programmer-defined boundaries and have a more bounded scope than the entire program.

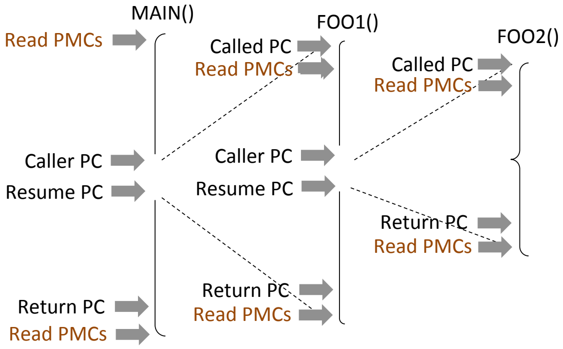


Figure 4.1: Recording performance counters and PCs at function calls and returns

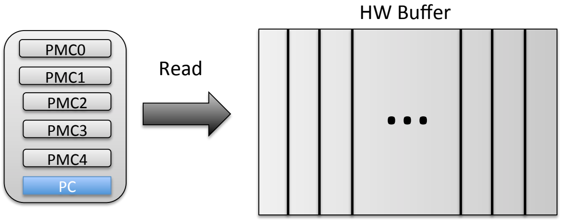


Figure 4.2: Storing performance counters and PCs to the hardware buffer

To attribute the energy profiles to the corresponding functions, during every function call, we record the (1) performance counters from Section 4.2.1, (2)*Program Counter* (PC) at the caller site and the (3) called function address (that can be later attributed back to program source code using well known tools such as GNU addr2line utility [65]). Similarly, during function return, we also record (1) the performance counters from Section 4.2.1, (2) the PC at the return site and (3) the PC at the caller function after return. At the end of the program execution, the corresponding energy profiles of function fragments (due to possibly multiple calls and returns between caller/callee pairs) are accumulated and attributed back to individual functions in the program code.

We use a 16-entry hardware buffer to record the performance counters and the PCs. The hardware buffer values are periodically logged into a file that can later be analyzed further by software modules. In particular, the logged values and the regression coefficients from Section 4.2.1 can be used to determine the energy consumption by individual functions during program execution. Since the hardware buffer is off the critical path, and has access latency of 0.18ns using Cacti 5.1 [26] (less than one CPU clock cycle even on a 3 GHz processor), it is unlikely to adversely impact the processor performance. We note that not all function calls are executed through *call* instructions, for example, compilers may generate function calls using *jmp* instructions, e.g., virtual functions are implemented using jump table in C++. In such cases, we can still correctly identify functions by monitoring the execution of *push* instructions (stack pushing operations) before *jmp*.

4.3 Automated Recommendation System for Energy Optimization

Once the profiler identifies the application code regions (functions) that are behind high energy consumption, the next logical step in energy debugging is to explore automated ways to optimize energy. Frequently, this step lends substantial benefits in large scale software development that involves a huge code base and numerous input test samples. To automatically modify program code, Genetic programing that mutates program code is one possible option. However, without guidance, regular genetic programming would randomly mutate program, and this process normally takes hours to complete. Fortunately, our fine-grained energy profiler has narrowed down the code optimization regions to only a few top energy consuming functions. As a result, genetic programming can leverage this information to guide program code mutation only inside these energy-hungry functions, which in turn, will gain huge performance advantages over the un-guided approach.

In summary, our application energy auditor applies genetic programming-based algorithm to automate the process of energy optimization. This algorithm uses *artificial selection* or guided mutation strategy to create energy-improving program code mutants based on the observed runtime energy profile.

In this section, we will first briefly describe the basics of genetic programming, and then show the design of our algorithm explaining why artificial selection is necessary.

4.3.1 Basics of Genetic Programming

Genetic programming (GP) [66] is the process of evolving computer program code to inductively find preferable program output given an input program. The process involves a series of mutations (or code transformations) to the input program such that we eventually arrive at a preferred output. Similar to animal and plant species evolving through natural selection process, GP applies a set of powerful genetic operators to *randomly* alter the "gene" (or code structure) in the program. These genetic operators range from simple replication of the program code to random variations on the inner parts of a program. Once the randomly selected genetic operators are applied, GP breeds new mutants that may or may not resemble the original program from which they are mutated. GP usually tests if these new mutants are qualified to remain in the evolving population using a fitness test. The fitness test is defined by a function F which measures the error distance between the mutant program output and the preferred outcome. Examples of fitness tests include testing for functional correctness (i.e., the mutant has the exact same functional behavior as the original), performance improvement, power reduction, or combinations of one or more of such factors. If a mutant is *deemed fit*, it is usually retained and allowed to join the evolution population (i.e., allowed to participate in the next round/generation of mutations). The premise is that, by breeding new programs from this dynamic pool, GP will find program mutants that will ultimately have the preferred program output.

Note that fitness test is usually the most time consuming step in the entire process of GP evolution. For example, if the fitness test involves checking for functional correctness, a representative range of possible input values have to supplied to the program to make sure that the original program and its mutant versions behave alike on all of such representative inputs. As a result, natural selection or unguided mutation could take several hours to produce useful mutants, and is popularly dubbed as *overnight optimization* [67].

4.3.2 Artificial Selection Genetic Programming

To overcome the challenges associated with natural (or un-guided, random) selection based approach, we propose artificial (or guided) selection for genetic operator selection. In biological terms, artificial selection (also known as Selective Breeding) refers to the process of human-controlled mutation and breeding that accelerates the evolution of preferable traits, instead of relying on slow natural selection. In our approach, we apply a set of code transformations guided by heuristics derived from program code structure or runtime profile. Our fitness function expects to see lesser energy consumption in the offspring than the parent, while making sure that functional correctness is preserved on a range of inputs. In this vein, we name our algorithm as Artificial Selection Genetic Programming (ASGP, for short).

4.3.3 ASGP Algorithm

As a first step, a control data dependency graph (or CDDG) is used to represent the profiler-identified high energy program functions. Algorithm 1 shows how our CDDGs are constructed from the program assembly code. Our graph construction algorithm goes through every constituent basic block and builds a data dependency subgraph for each of them starting from the very last instruction (branch) and proceeding backwards until the first instruction in the block by tracing the dependency of data values. A directed edge from node A to node B means that node A depends on node Bs output to compute its own output. After constructing data dependency subgraphs, our graph construction algorithm will connect all of the subgraphs based on control dependencies, that is, each basic block's branch node will be connected to all its target nodes in other subgraphs.

To further assist the ASGP algorithm, certain nodes in the CDDG are annotated with runtime profile information. For example, the branch nodes are annotated with the taken frequency (from which branch taken vs. non-taken ratio can be derived). We note that these annotations based on runtime information are simply to guide our ASGP algorithm to make decisions in an informed manner rather than arbitrary, random choices.

We note that instruction operands can be immediate, register, or memory. Among these, memory-type operands need special attention due to unknown memory dependences at compile time. In several popular ISAs such as x86 , data is read or stored into a particular memory address by using different memory addressing modes, such as direct displacement, register indirect, base indexed. Note that registers are specifically used as base or index registers as part of the calculation of memory address where data is read or written. In order for CDDG to track the true dependencies between two memory-type operands, we monitor for changes to the registers that are used as base and/or index in memory addressing. For example, we mark that a memory-type Operand_dst node from instruction A depends on a memory-type Operand_src node from instruction B only if all the memory address-related registers used in both A and B do not get updated by any other instructions between A and B. In other words, if there is any update to a register used as an index/base in another instruction's operand, we do not establish dependency between the instructions A and B. In addressing modes, such as memory indirect, where it is more complex to resolve memory dependencies prior to runtime, we chose to represent these operands as two separate nodes in CDDG. To address situations where multiple operands might write to the same memory location using different base and index registers (memory aliasing problem that are usually unresolved prior to runtime), our CDDG algorithm flags all of the instructions that have memory operands as destinations. Our genetic programming algorithm, ASGP, can use this as a guide to prevent

Algorithm 1: CDDG construction algorithm

```
input  : Assembly code of the profiler-identified function
output: A Graph G that shows data and control dependencies
//Construct data dependency subgraphs for every basic block;
foreach basic block in the identified function do
    Create an empty set S;
    //Go through instructions in reverse order;
    foreach instruction in the basic block do
        //This if-else only applies to instructions that has destination operand;
        if Operand_dst is not in S then
            Create a new node with label Operand_dst;
            Add Operand_dst to S;
        else
            Delete Operand_dst from the S;
            Delete memory operands derived from Operand_dst from S;
        end
        Include Opcode in the node labeled with Operand_dst;
        foreach  Operand_src in the instruction do
            if Operand_src is not in S then
                Create a new node with label Operand_src;
                Add Operand_src to S;
            end
            Create a directed data dependency edge from Operand_dst to Operand_src;
        end
    end
end
//Connecting sub-graphs through branch nodes;
foreach branch node (with branch Opcode) do
    Locate all target nodes in the graph;
    Create a control dependency edge from target nodes to this branch node;
end
```

reordering of instructions past these instruction boundaries and potentially avoid memory conflicts. Note that our fitness test acts as a safety net, and will fail if ASGP unintentionally created any memory conflicts.

Based on the information contained in the CDDG, the first generation mutants are evolved using *neutrally transformed* program mutants from CDDG. Neutrally transformed mutant refer to a program version that preserves the functional equivalence derived either by applying algebraic rules or via merging opcodes already available from the ISA (see Section 4.3.4). This step helps initialize the evolution population with multiple mutant samples (with functionally equivalent versions to the original program) such that the ASGP algorithm can do a more effective exploration by applying its genetic operators on this population. Successive generations of mutants are created by applying genetic operators on the evolution population guided by triggering heuristics (see Section 4.3.5). The fitness function tests whether the new mutant preserves the functional equivalency on all of the available input sets while potentially optimizing energy consumption over its parent mutant, and then is added to the evolution population. In other words, if the mutant is not functionally equivalent to the original program or does not show the potential to improve energy over its parent, the mutant is discarded from further consideration. We determine functional equivalency by comparing all of the registers (data, flags and program counters) and the set of all memory locations modified by the original program and the mutant in the *region of interest* (i.e., where the mutation operation was applied). The *potential for energy optimization* is defined as the mutant that already lowers energy consumption of the parent or is within 5% above the parent's energy consumption. We allow this slight increase in energy consumption among the mutants to avoid being stuck in local minimas and being unable to find more energy optimizing mutants in the future without exploring slightly more energy-expensive mutants. However, to avoid the mutant population from being polluted with higher energy consuming variations, we periodically (after every five generations in our implementation) remove the mutants that consume energy higher than the original input program.

After reaching a maximum number of generations starting from all of the initial mutants, the algorithm stops. The maximum number of generations is input by the user based on the number of tries (or cost) that she is willing to pay for the ASGP algorithm to explore the program mutants.

Algorithm 2: Artificial Selection Genetic Programming Algorithm

```
input  : Original program (assembly)
output: Lesser energy consuming program mutants

Initialize the population with the original program code and mutants derived using
neutral transforms;
Run the initial mutants and obtain their energy consumption;
repeat
    Randomly select one mutant from the evolution population as parent;
    Mutate using one or more genetic operators with triggering heuristics;
    Apply fitness test on the evolved mutant code;
    if the mutant passes the fitness test then
        Replace its parent in the evolution population;
    else
        Discard the mutant from the evolution population;
    end
    At every fifth generation, discard mutants with energy greater than original program;
until maximum number of generations is reached from all of the initial mutants;
Output the mutant that consumes the least energy in the evolution population;
```

4.3.4 Neutral Transforms

If unguided or natural selection process is used to create initial mutants, most of them will likely not be functionally equivalent to the original program, thus failing the fitness test. As a result, a significant amount of computational time is wasted in evolving these random programs. To avoid wasting time on evolving such "pure random" programs, our current implementation of enDebug uses the following set of *neutral transforms* on the original CDDG to populate the set of initial program mutants, and hence increase the efficiency of our approach.

1. **Sign Conversion**– This helps generate complements of numbers that can be primarily exploited to cluster certain types of operations and reduce energy if such operations are supported by the underlying ISA. For example, $a - b + c + d$ can also be represented $a + (-b) + c + d$ that can be utilized by vector operations to cluster operands with the same operator.

2. **Commutativity**– Commutativity applies on arithmetic operations such as add and multiply nodes, for example, $a+b = b+a$, $a \times b = b \times a$. This can be used to group clusters of operands in nearby memory locations and potentially improve spatial locality to save energy.

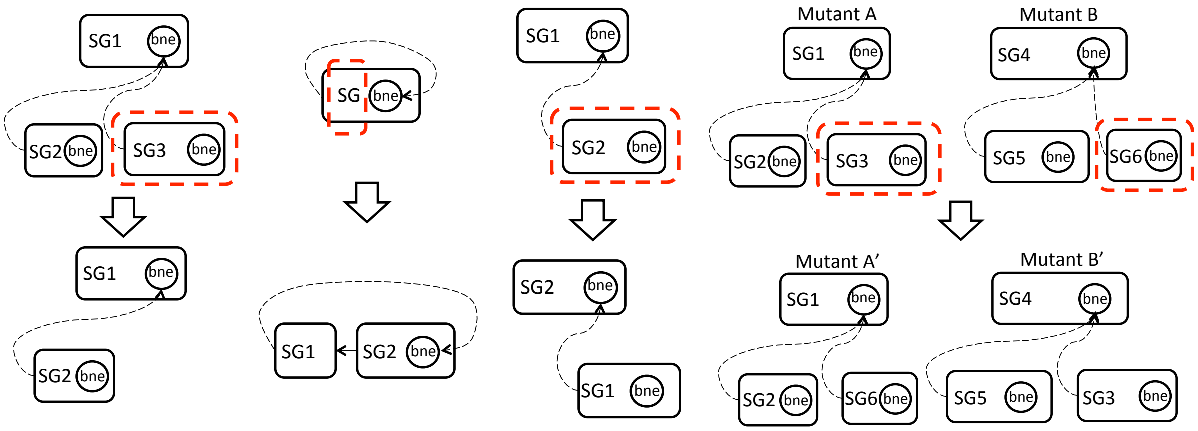
3. **Distributivity**– Distributivity also applies on arithmetic operations that helps to reduce the number of multiplication/division operations in the program. For example, $a \times b + a \times c = a \times (b + c)$, $a/b + c/b = (a + c)/b$, $(a/b)/(c/d) = (a \times d)/(b \times c)$, $(a/b)/c = a/(b \times c)$.

4. **Merge**– This serves to optimize energy consumption by combining certain operations into ISA-supported more complex types. Energy savings can be had from lesser number of instruction fetches and execution. Examples of merge operation include: 1. when nearby nodes in CDDG have mul.d and add.d, the ASGP algorithm can consider replacing the node with madd.d in certain ISAs, 2. when nearby nodes in the CDDG have load and address increment/decrement, a vector load can be applied.

4.3.5 Genetic operators and Triggering heuristics

We first describe the four standard genetic operators [66] that are used in our genetic algorithm, and then summarize the heuristics that guide the usage of such operators on program mutants.

1. **Delete**–Delete operator serves to eliminate the CDDG nodes that unnecessarily in-



(a) Delete (b) Copy (c) Swap (d) Crossover

Figure 4.3: Examples of Genetic operator Applications on CDDG

crease the program energy consumption. As examples, 1. when a branch is always not taken for all inputs, the branch instruction and all instructions that compute the condition value for the branch can be deleted (see Figure 4.3a), 2. certain subexpressions or instructions can be removed that may be eventually moved to another place in the program code.

2. **Copy**– Copy operator works either on the block level or individual instruction level. At the block level, copy operator can potentially increase the ratio of useful code while reducing energy spent on meta code and the related control transfer instructions. When copy operator is applied, the subgraphs need to be properly replicated (similar to loop unrolling) - memory indices should be correctly adjusted, common registers should be renamed, and the dependency edges be connected to the right node. Potential example applications of this operator include the loop structures that were not able to be unrolled by the optimizing compiler at static time; at the instruction level, copy operator helps achieve physically moving an instruction to another place. This movement of instruction requires copy to be jointly used with delete operator.

3. **Swap**– Swap operator swaps the positions of two nodes (instructions) or subgraphs (blocks). This can be used to accomplish useful transforms such as code reordering. As

examples: 1. If consecutive nodes exhibit long latencies due to stalls in integer execution unit (resource contention), swapping this node with an another node (that has a different set of operations not competing for the integer execution unit) would alleviate the energy wasted over stall time. 2. In a if..elsif..else code, let us say that the percentage of execution of if{}, elsif{} and else{} blocks are 10%, 10% and 80% respectively. A swap of the else{} and if{} blocks would save the unnecessary control transfer through two basic blocks a majority of the time.

4. **Crossover**– Crossover operator takes subgraphs from two parent mutants, and creates two new offsprings. For example, between two parent mutants A and B, exchanging subgraphs SG3 and SG6 results in an improved offspring, A (see Figure 4.3d). Specifically, we look for mutants that show reduced energy consumption compared to their parents, and mark the mutated subgraphs (the portion of program code that was mutated in the parent). Then a crossover operation is performed to check if two mutants with having marked subgraphs from different program locations will create a new mutant that combines the energy savings from both of its parental mutants.

Table 4.3: Triggering heuristics for genetic operators

Operator/Transform	Triggering Heuristics
Delete	Always taken/not taken branches, redundant instruction
Copy	High branch to non-branch instruction ratios, Known iterator count values
Swap	Skewed frequencies of execution in branch chains, cluster of instructions of similar type
Crossover	Energy optimized mutants in two separate subgraphs among two different mutants

Table 4.3 summarizes some of the possible heuristics that were used by our ASGP algorithm to find program mutants that can lead to better energy consumption. We note that

this table shows a list of heuristics used in our current implementation rather than being exhaustive.

Table 4.4: Number of mutants for the un-guided natural selection GP algorithm

Application	# discarded mutants	# kept mutants
Fmm	558	15
Ocean	440	20
Cholesky	40	19
Water-sp	290	33
Water-n2	290	33
Fluidanimate	275	11
Streamcluster	486	14

Table 4.5: Number of mutant for our ASGP algorithm with triggering heuristics

Application	# discarded mutants	# kept mutants
Fmm	13	11
Ocean	14	17
Cholesky	0	14
Water-sp	31	8
Water-n2	31	9
Fluidanimate	11	19
Streamcluster	5	7

Tables 4.4 and 4.5 show the number of distinct mutants generated by the unguided (natural selection) and guided (artificial selection) algorithms assuming that the user has no limit on the maximum number of generations. We found that the energy-optimized mutants found by both the GP and the ASGP algorithms were identical for each of our benchmark.

In each experiment, the mutants that are not deemed fit by the fitness function either due to functional incorrectness or not exhibiting the capability to reduce energy are considered as *discarded mutants* (refer Section 4.3.3). Functional correctness was determined by comparing the outputs of the original and the mutant codes on all of the input sets made available by the benchmark developers. All of the mutants that pass the fitness test are *kept mutants*, i.e., considered part of the evolution population. As seen in our experiments, the ASGP reduces the total number of mutants anywhere from 4× (in cholesky) to over 40× (in streamcluster). This helps to speedup the process of choosing energy optimized mutants of the program code.

Table 4.6: Execution time overhead comparison between the natural selection GP algorithm and our heuristic-triggered ASGP algorithm

Application	GP (secs)	ASGP (secs)	Speedup (times)
Fmm	701.6	28.1	25.1×
Ocean	705.1	47.6	14.8×
Cholesky	19.5	4.7	4.1×
Water-sp	145.7	17.6	8.3×
Water-n2	936.6	116.3	8.1×
Fluidanimate	1354.9	142.1	9.5×
Streamcluster	2881.5	69.2	41.7×

Table 4.6 shows the execution time (in seconds) for a un-guided natural selection GP algorithm and our guided, artificial selection GP algorithm. We note that the execution time includes the time to run the algorithm and fitness test for each of the benchmarks. In every benchmark, functional correctness was determined through multiple runs of all of the input sets made available by the benchmark developers. We observe a speedup of 4.1× (in cholesky) and up to 41.7× (in streamcluster).

4.3.6 System Support for Automated Energy Optimization

The energy regression function can derive its inputs from the already available hardware performance counters. To attribute the energy back to functions, we need to log information about the performance counters and the corresponding program counters in a separate hardware buffer.

Our ASGP algorithm can be implemented in software as a standalone post-compile optimizer. On the hardware side, the algorithm needs support for three modules, energy meter and functional correctness (fitness test) and heuristics. For energy metering, an off-the-shelf energy measurement interface such as Intels RAPL would suffice since the objective is to reduce overall program energy. However, for verifying functional equivalence, we need system support that will compare the values produced by mutated regions of program code and the original code. This requires hardware support to track all of the registers (including data, flags and Program Counter) and the set of memory locations altered by the program in the mutated region of program code. To this end, we design a small hardware structure, *hardware shadow buffer*, that maintains a shadow copy of the registers and the modified memory locations. Empirically, we determined that the maximum number of memory locations that were modified in energy critical code sequences were typically less than 4 KB. Therefore, we design our shadow buffer to store up to 4 KB memory locations. Using Cacti 5.3, we found that the area overhead was $0.31mm^2$ and the shadow buffers per-access latency was $0.26ns$. *Start_log* and *Stop_log* instructions are added to the ISA for recording values. Finally, most of our heuristics can be derived in a relatively straightforward manner using existing performance counters in most modern architectures such as branch taken/not taken profile, resource stalls [68].

4.4 enDebug Evaluation

We evaluate over 20 Splash-2 and PARSEC-1.0 applications, and summarize the energy-related code optimizations that we found using our enDebug framework on six such representative cases below. We perform our first round of experiments on SESC simulator with an integrated McPAT model. This is due to ease of verifying the functional correctness of the mutants through comparing all of the register and memory values produced by the mutant with the original code. We were unable to turn on profile guided optimizations (PGO) on our GCC cross compiler infrastructure for MIPS. Table 4.4 summarizes the observed energy savings on two different reference input sets with -O3 settings without PGO.

1. **Fmm**– Figure 4.4 shows the high energy consuming code region found by our energy profiler. Using our profiler-guided heuristics, the ASGP algorithm applied the *copy* operator based on high branch to non-branch instruction ratio inside the tight loop. After a few rounds of delete and some swap operations among instructions, the algorithm output a mutant with least amount of energy consumption. We found that the final mutant produced by ASGP algorithm did not have the if..else.. control part, and the energy consumption had dropped at least 3.7% during our fitness test. This code is functionally equivalent since the COMPLEX_SUB works on odd j, and COMPLEX_ADD on even j values that is replaced by j and $j+1$ during *copy* operation.

2. **Ocean**– Figure 4.5 shows the high energy consuming code region found by our energy profiler. Within this dense loop, eight local variables ($f1$ to $f8$) are calculated, and then all of them are added together to find t1c[iindex]. ASGP found that there was a single node (load of $t1a[iindex]$) that was connected to another 8 nodes. After applying a few neutral transforms, deletions and a crossover operation, the mutated program reduced the instruction count by 13, and consumed up to 4.3% less energy than the original program.

3. **Cholesky**– Figure 4.6 shows the high energy consuming code region found by our energy profiler. In this case, our ASGP algorithm was able to make effective use of merge (a

```
//Fmm (Splash−2): interaction.C
for (j = 1; j < Expansion_Terms; j++) {
    temp.r = C[i + j − 1][j − 1];
    temp.i = (real) 0.0;
    COMPLEX_MUL(temp, temp, temp_exp[j]);
    if ((j & 0x1) == 0x0) {
        COMPLEX_ADD(result_exp, result_exp, temp);
    } else {
        COMPLEX_SUB(result_exp, result_exp, temp);
    }
}
```

Figure 4.4: Fmm code snippet

neutral transform) and delete operators to find energy-optimized mutant. The original code was written in a way that prevented the compiler to generate more compact instructions like msub.d (in MIPS). After applying algebraic laws on the original CDDG, the resulting neutral transforms were more conducive to merging the adjacent nodes and forming madd.d. ASGP was able to generate a program mutant that primary used madd.d instructions, and reduced the overall program energy consumption by slightly over 1%.

4. **Water-sp and Water-n2**– Figure 4.7 shows the high energy consuming code regions found by our energy profiler. Our runtime profile data indicated that two branches in this code region (see *sign* macro in the source code line#2) were never taken for all of the available program inputs. This was because the values of a were always positive in all of the input sets. Based on the branch frequency heuristic, the ASGP algorithm generated two separate mutants by deleting the blocks corresponding to the first two branches, and then a crossover from these two mutant versions resulted in an energy optimized code. The overall energy savings in water benchmarks was around 2%.

5. **Fluidanimate**– Figure 4.8 shows the high energy consuming code region found by

```
//Ocean (Splash-2): jacobcalc.C
for (iindex=firstcol;iindex<=lastcol;iindex++) {
    indexp1 = iindex+1;
    indexm1 = iindex-1;
    f1 = (t1b[indexm1]+t1d[indexm1]-t1b[indexp1]-t1d[indexp1])
        *(t1f[iindex]-t1a[iindex]);
    f2 = (t1e[indexm1]+t1b[indexm1]-t1e[indexp1]-t1b[indexp1])
        *(t1a[iindex]-t1g[iindex]);
    f3 = (t1d[iindex]+t1d[indexp1]-t1e[iindex]-t1e[indexp1])
        *(t1a[indexp1]-t1a[iindex]);
    f4 = (t1d[indexm1]+t1d[iindex]-t1e[indexm1]-t1e[iindex])
        *(t1a[iindex]-t1a[indexm1]);
    f5 = (t1d[iindex]-t1b[indexp1])*(t1f[indexp1]-t1a[iindex]);
    f6 = (t1b[indexm1]-t1e[iindex])*(t1a[iindex]-t1g[indexm1]);
    f7 = (t1b[indexp1]-t1e[iindex])*(t1g[indexp1]-t1a[iindex]);
    f8 = (t1d[iindex]-t1b[indexm1])*(t1a[iindex]-t1f[indexm1]);

    t1c[iindex] = factjacob*(f1+f2+f3+f4+f5+f6+f7+f8);
}
```

Figure 4.5: Ocean code snippet

our energy profiler. This loop contains very intensive arithmetic operations. Note that Vec3 is a structure that has three double elements, for which $+=$ and $/=$ have been overridden. The energy reduction came from the initial neutral transform and delete operators. Our ASGP moved the $/=$ around through commutative and distributive rules, which deleted certain expensive (long pipeline cycles) division instructions. Our experiments show up to 4.3% energy savings.

6. **Streamcluster**– Figure 4.9 shows the high energy consuming code region found by our

```
//Cholesky (Splash-2): numLL.C
while (srcNZ0 != last) {
    d0 = *dest0; d1 = *dest1;
    tmp0 = *srcNZ0++; d0 -= ljk0_0*tmp0; d1 -= ljk0_1*tmp0;
    tmp1 = *srcNZ1++; d0 -= ljk1_0*tmp1; d1 -= ljk1_1*tmp1;
    tmp0 = *srcNZ2++; d0 -= ljk2_0*tmp0; d1 -= ljk2_1*tmp0;
    tmp1 = *srcNZ3++; d0 -= ljk3_0*tmp1; d1 -= ljk3_1*tmp1;
    *dest0++ = d0; *dest1++ = d1;
}
```

Figure 4.6: Cholesky code snippet

```
//Water-sp/Water-n2 (Splash-2): cshift.C
#define sign(a,b) (b < 0 ) ? ( (a < 0) ? a : -a) : ( (a < 0) ? -a : a)
for (I = 0; I < 14; I++) {
    /* if the value is greater than the cutoff radius */
    if (fabs(XL[I]) > BOXH) {
        XL[I] = XL[I] -(sign(BOXL,XL[I]));
    }
}
```

Figure 4.7: Water-sp/Water-n2 code snippet

energy profiler. Based on the runtime profile data, the ASGP algorithm chose to apply *copy* genetic operator. Even at O3 compiler optimization level, this loop could not be unrolled by the compiler since the range or actual values of iteration count (function parameter *dim* is unknown at compile time. We found that *dim* is a fixed number during program execution. This mutant version improved the energy savings by about 18% since this loop was a prominent kernel in streamcluster benchmark.

```
//Fluidanimate (Parsec-1): parallel.cpp
for(int iparNeigh = 0; iparNeigh < numNeighPars; ++iparNeigh) {
...
   Vec3 acc = disp * pressureCoeff * (hmr*hmr/dist)
                     * (cell.density[j]+neigh.density[iparNeigh] - doubleRestDensity);
   acc += (neigh.v[iparNeigh] - cell.v[j]) * viscosityCoeff * hmr;
   acc /= cell.density[j] * neigh.density[iparNeigh];
...
}
```

Figure 4.8: Fluidanimate code snippet

```
//Streamcluster (Parsec-1): streamcluster.cpp
float dist(Point p1, Point p2, int dim)
{
    int i;
    float result=0.0;
    for (i=0; i<dim; i++)
        result += (p1.coord[i]-p2.coord[i])*(p1.coord[i]-p2.coord[i]);
    return(result);
}
```

Figure 4.9: Streamcluster code snippet

4.4.1 Validation on a real system

By conducting experiments on a real system, we validate the energy-optimized mutant programs generated by our prototype enDebug framework implemented on SESC simulator with McPAT power model. We note that these energy-optimized mutant versions have already passed the fitness test for both functional accuracy and energy optimization. Our goal is to

Table 4.7: Energy reduction for our benchmark applications in simulator (Baseline: GCC -O3)

Application	ASGP-evolved code region	# code modifications	Energy reduction (with input ref1)	Energy reduction (with input ref2)
Fmm	interaction.C: line 398	16	3.7%	4.1%
Ocean	jacobcalc.C: line 310	15	2.2%	4.3%
Cholesky	numLL.C: line 436 & line 473	44	1.2%	1.4%
Water-sp	cshift.C: line 58	34	1.9%	2.7%
Water-n2	cshift.C: line 54	34	2.3%	1.3%
Fluidanimate	parallel.cpp: line 689	12	4.3%	4.2%
Streamcluster	streamcluster.cpp: line 159	31	18.0%	15.5%

see if similar energy savings can be observed on a real-world system. Each benchmark is run using the largest inputs (native inputs in PARSEC-1.0 and reference inputs in Splash-2). Our test environment is a Dell Latitude E6520 workstation laptop with 8GB memory, and 4-core Intel(R) Core(TM) i7-2720QM processor (Sandy Bridge) run at 2.2 GHz with turbo mode at 3.3 GHz using RAPL interface.

Table 4.8 shows the energy savings in our real system experiments on two baseline settings: 1. with -O3 optimizations, 2. with both -O3 and GCC's PGO. To enable PGO, we use -fprofile-generate and -fprofile-use flags. Note that -fprofile-generate enables -fprofile-arcs, -fprofile-values and -fvpt flags that include value profile transformations and program flow arcs. The flag -fprofile-use enables -fbranch-probabilities, -funroll-loops, -fpeel-loops, -ftracer that include tracking probabilities of branches and removal of small loops with constant number of iterations. The results show that, at -O3, the energy savings trend are fairly similar to our experiments on SESC (Table 4.4). With PGO enabled, we find that ASGP is still able to achieve fairly high energy savings (above 5%) in benchmarks such as fluidanimate, water-sp and ocean due to its ability to apply mutations such as deletion, swap and crossover that the compiler normally does not handle. In other benchmarks such as streamcluster, the savings

Table 4.8: Energy reduction on a real system (Intel Core i7)

Application	Energy reduction (Baseline: GCC -O3)	Energy reduction (Baseline: GCC -O3 and PGO)
Fmm	7.1%	4.6%
Ocean	4.0%	5.7%
Cholesky	N/A	N/A
Water-sp	5.1%	5.7%
Water-n2	2.1%	2.2%
Fluidanimate	10.4%	7.0%
Streamcluster	17.6%	1.6%

are diminished due to the advanced optimization settings in the compiler that can achieve a similar outcome as our ASGP algorithm. Overall, we find that our enDebug framework can be extremely useful in finding energy-optimized mutants of program code even after extensive optimizations by the compiler.

4.5 Related Work

We discuss prior works in two broad categories- energy estimation and optimization.

4.5.1 Energy Estimation

Isci et al [59] propose runtime power monitoring techniques for processor core and functional units. Some recent works demonstrate the feasibility of using a limited set of of metrics to estimate processor component power [28, 60, 69]. Zhai et al [70] have developed hyper-thread aware power model that extends the functionality of the current Intel RAPL interface. These works do not explicitly address how to attribute energy back to the program code. In contrast to these prior schemes, we investigate ways to attribute energy to fine-grain code.

Tiwari et al [52] developed an energy model using instruction counts, and assume a predetermined cost for various instruction types. This ignores dynamic hardware effects such as parallelism and interference that occur in most modern architectures. Alternative strategies that use a specific set of hardware events (such as cache misses) [63, 71] for energy estimation often fail to include a comprehensive view of application execution and ignore system-level effects and interactions with other instructions (e.g., pipeline stalls, pipeline flushes due to mispredicted instructions). In contrast to these prior approaches, our methodology uses largely existing hardware (with simple modifications) to accurately gather the total energy consumed by fine grain sections of code (functions).

4.5.2 Energy Optimization

Microarchitecture– Hardware modifications have been proposed to minimize energy consumption by the microarchitectural units. For instance, to reduce cache energy, prior works have investigated reconfigurable cache with variable associativity or number of banks [36, 37], bypassing expensive tag check operations [38], trading cache capacity for low supply voltage [39]. For issue logic and load/store queues that rely on energy-expensive structures such as Content Address Memory (CAM), researchers have studied dynamically adjusting issue or load/store queue sizes and avoiding wasteful wake-up checks [40, 41, 42, 72]. Recent works have started investigating custom hardware accelerators for specific types of applications [43, 44, 73]. Prior works also consider optimizing processor pipelines for low power and energy [45, 46, 74].

Dynamic voltage and frequency scaling (DVFS) Prior works have shown how to exploit slack time in the running thread to put the processor core in a sleep state [4], or run at a low voltage/frequency level [47, 48]. Other works have investigated into finding long running code regions, and then instrumenting these regions with favorable voltage/frequency settings to reduce energy [75, 49, 50]. DVFS has also been demonstrated to do system-level power and energy management [76, 77]. Since our approach is complementary to this line of work, we may gain even further energy savings by applying DVFS to our energy-optimized program mutants.

Automated Code Optimization– As software inefficiency increasing becomes the dominant source of wasteful energy consumption, there is a strong urge to develop energy-aware program code. Recently, Schulte et al [67] proposed a post-compiler software optimization technique. In this work, authors designed a genetic optimization algorithm that exhaustively searches for program variant with functional correctness and lesser energy consumption. Although both of our works broadly use genetic programming for mutating program code, our enDebug differs from [67] in the following ways: 1. enDebug incorporates a fine grain energy profiler in hardware that is estimates energy for program functions to apply targeted optimizations, while [67] uses an energy model that estimates energy at the process level. 2. enDebug applies mutant operations only on the most energy consuming code region, while [67] approach variate the entire program, which may make the search space become prohibitively expensive for large scale programs 3. enDebug adopts a guided (artificial selection) approach by taking advantage of heuristics derived from program code structure or runtime profile, while [67] selects mutation operators randomly which is not performance friendly as shown in Table 4.6.

4.6 Summary

In this chapter, we showed the need to understand application energy profile. We show the design our solution approach, enDebug, that has two major components- 1. energy profiler, and 2. automated recommendation system for program energy optimization.

We designed an energy profiler using largely existing hardware support that measures the energy of program functions. We explored an automated recommendation system that incorporates artificial selection genetic programming algorithm to generate program mutants that minimize energy consumption. In contrast to prior approaches, we adopt a guided approach to mutant generation that reduces the search space and quickens the time (4× - 41×) to arrive at energy-optimized program mutants. We show case studies from several Splash-2 and PARSEC-1.0 benchmarks, and demonstrate energy savings of up to 7% beyond the highest compiler optimization settings when tested on real-world Intel Core i7 processors.

Chapter 5

CC-Hunter: Information Leakage Auditing

5.1 Understanding Covert Timing Channels

Trusted Computer System Evaluation Criteria (or TCSEC, The Orange Book) [78] defines covert channel as *any communication channel that can be exploited by a process to transfer information in a manner that violates the system's security policy.* In particular, covert timing channels are those that would allow one process to signal information to another process by modulating its own use of system resources in such a way that the change in response time observed by the second process would provide information.

Note that, between the trojan and the spy, the task of constructing a reliable covert channel is not very simple. Covert timing channels implemented on real systems take significant amounts of synchronization, confirmation and transmission time even for relatively short-length messages. As examples, (1) Okamura et al. [17] construct a memory load-based covert channel on a real system, and show that it takes 131.5 seconds just to covertly communicate 64 bits in a reliable manner achieving a bandwidth rate of 0.49 bits per second; (2) Ristenpart et al. [19] demonstrate a memory-based covert channel that achieves a bandwidth of 0.2 bits per second. This shows that the covert channels create non-negligible amounts of traffic on shared resources to accomplish their intended tasks.

TCSEC points out that a covert channel bandwidth exceeding a rate of one hundred (100) bits per second is considered “high” based on the observed data transfer rates between several kinds of computer systems. In any computer system, there are a number of relatively low-bandwidth covert channels whose existence is deeply ingrained in the system design. If bandwidth-reduction strategy to prevent covert timing channels were to be applied to all of them, it becomes an impractical task. Therefore, TCSEC points out that channels with maximum bandwidths of less than 0.1 bit per second are generally not considered to be very feasible covert timing channels. This does not mean that it is impossible to construct very low bandwidth covert timing channel, just that it becomes very expensive and difficult for the adversary (spy) to extract any meaningful information out of the system.

5.2 Threat Model and Assumptions

Our threat model assumes that the trojan wants to *intentionally* communicate the secret information to the spy covertly by modulating timing on certain hardware. We assume that the spy is able to seek the services of a compromised trojan that has sufficient privileges to run inside the target system. As confinement mechanisms in software improve, hardware-based covert timing channels will become more important. So, we limit the scope of our work to shared processor hardware.

A hardware-based covert timing channel could have noise due to two factors- (1) processes other than trojan/spy using the shared resource *frequently*, (2) trojan artificially inflating the patterns of random conflicts to evade detection by CC-Hunter. In both cases, the reliability of covert communication is severely affected resulting in loss of data for the spy as evidenced by many prior studies [79, 80, 23]. For example, on a cache-based covert timing channel, Xu et al. [23] find that the covert transmission error rate is at least 20% when 64 concurrent users share the same processor with trojan/spy. Therefore, we point out that it is impossible for a covert timing channel to just randomly inflate conflict events or operate in noisy environments simply to evade detection. In light of these prior findings, we model moderate amounts of interference by running a few other (at least three) active processes alongside the trojan/spy processes in our experiments.

In this work, our focus is on the detection of covert timing channels rather than showing how to actually construct or prevent them. We do not evaluate the robustness of covert communication itself that has been demonstrated adequately by prior works [19, 22, 23].

We assume that covert timing based communication happen through recurrent patterns of conflicts over non-trivial intervals of time. CC-Hunter cannot detect the covert timing attacks that happen instantly where the spy has the ability to gain sensitive information in one pass. Also, covert timing channels that employ sophisticated combinations of timing and storage channels at both hardware and software layers are not considered in this work. Finally, we assume that the system software modules (including the operating system kernel and security enforcing layers) are trusted.

5.3 Design Overview

From the perspective of covert timing channels that exploit shared processor hardware, there are two broad categories–

(1) Combinational structures such as logic and wires, relying on patterns of high and low contention to communicate on the corresponding shared resource. Consequently, a recurrent (yet sometimes irregular) pattern of contention (conflicts) shall be observed in the corresponding event time series when the trojan covertly communicates the message bits to the spy.

(2) Memory structures, such as caches, DRAM and disks, using intentional replacement of memory blocks (previously owned by the spy process) to create misses. As a result, we observe an recurrent pattern of cache conflict misses between the trojan and the spy.

We design pattern detection algorithms to identify the recurrence patterns in the corresponding event time series[1]. Our algorithms look for patterns of conflicts, a fundamental

[1] Our solution is inspired from studies in neuroscience that analyze patterns of neuronal activity to un-

property of covert timing channels. Hence, even if the trojan/spy dynamically change their communication protocol, CC-Hunter should still be able to detect them based on conflict patterns.

To demonstrate our framework's effectiveness, we use three realistic covert timing channel implementations, two of which (shared caches [23], memory bus [22]) have been demonstrated successfully on Amazon EC2 cloud servers. We evaluate using a full system environment by booting marss [82] with Ubuntu 11.04. The simulator models a quad core processor running at 2.5 GHz, each core with two hyperthreads, and has a few (at least three) other active processes to create real system interference effects. We model a private 32 KB L1 and 256 KB L2 caches. Prior to conducting our experiments, we validated the timing behavior of our covert timing channel implementations on marss against measurements in a real system environment (dual-socket Dell T7500 server with Intel 4-core Xeon E5540 processors running at 2.5 GHz, Ubuntu 11.04). Note that the three covert timing channels described below are randomly picked to test our detection framework. CC-Hunter is neither limited to nor derived from their specific implementations, and can be used to detect covert timing channels on all shared processor hardware using recurrent patterns of conflicts for covert communication.

5.3.1 Covert Timing Channels on Combinational hardware

To illustrate the covert timing channels that occur on combinational structures and their associated indicator events, we choose memory bus and integer divider unit (a similar implementation was shown using multipliers by Wang et al [20]).

In case of the memory bus covert channel, when the trojan wants to transmit a '1' to the spy, it intentionally performs an atomic unaligned memory access spanning two cache lines. This action triggers a memory bus lock in the system, and puts the memory bus in contended state for most modern generations of processors including Intel Nehalem and AMD K10 family. The trojan repeats the atomic unaligned memory access pattern for a

derstand the physiological mechanisms associated with behavioral changes [81].

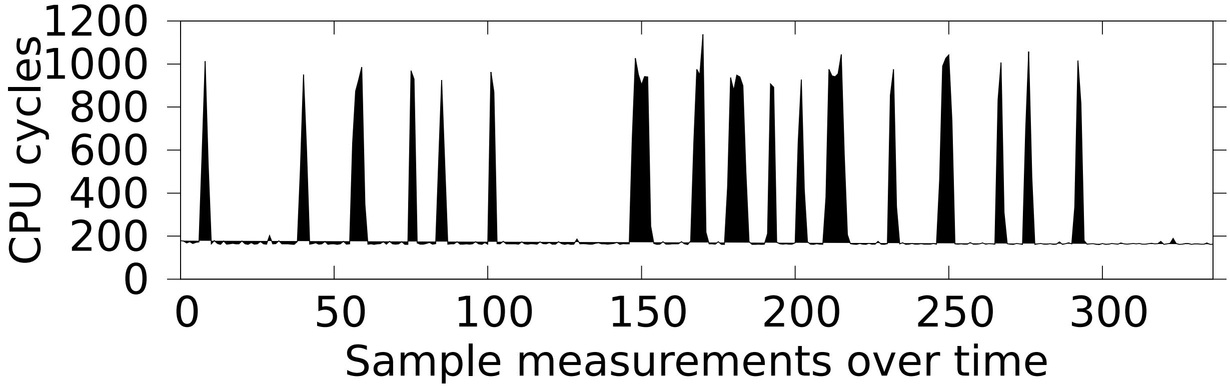


Figure 5.1: Average latency per memory access (in CPU cycles) in Memory Bus Covert Channel

number of times to sufficiently alter the memory bus access timing for the spy to take note of the ‘1’ value transmission. Even on x86 platforms that have recently replaced the shared memory bus with QuickPath Interconnect (QPI), the bus locking behavior is still emulated for atomic unaligned memory transactions spanning multiple cache lines [83]. Consequently, delayed interconnect access is still observable in QPI-based architectures. To communicate a ‘0’, the trojan simply puts the memory bus in un-contended state. The spy deciphers the transmitted bits by accessing the memory bus intentionally through creating cache misses. It times its memory accesses and detects the memory bus contention state by measuring the average latency. The spy accumulates a number of memory latency samples to infer the transmitted bit. Figure 5.1 shows the average loop execution time observed by the spy for a randomly-chosen 64-bit credit card number. A contended bus increases the memory latency making the spy to infer ‘1’, and an un-contended bus helps the spy to infer ‘0’.

For integer division unit, both the trojan and the spy processes are run on the same core as hyperthreads. The trojan communicates ‘1’ by creating a contention on all of the division units by executing a fixed number of instructions. To transmit a ‘0’, the trojan puts all of the division units in an un-contended state by simply executing an empty loop. The spy covertly listens to the transmission by executing loop iterations with a constant number of

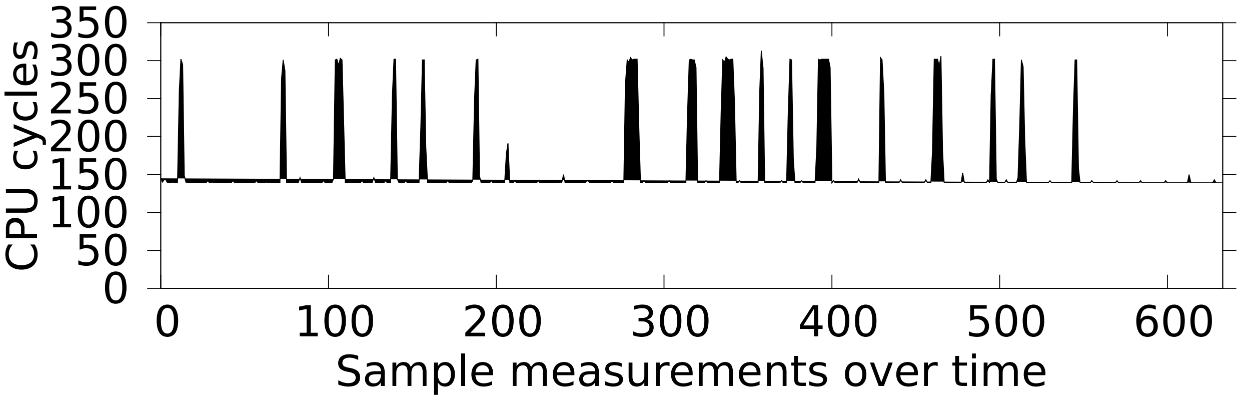


Figure 5.2: Average loop execution time (in CPU cycles) in Integer Divider Covert Channel

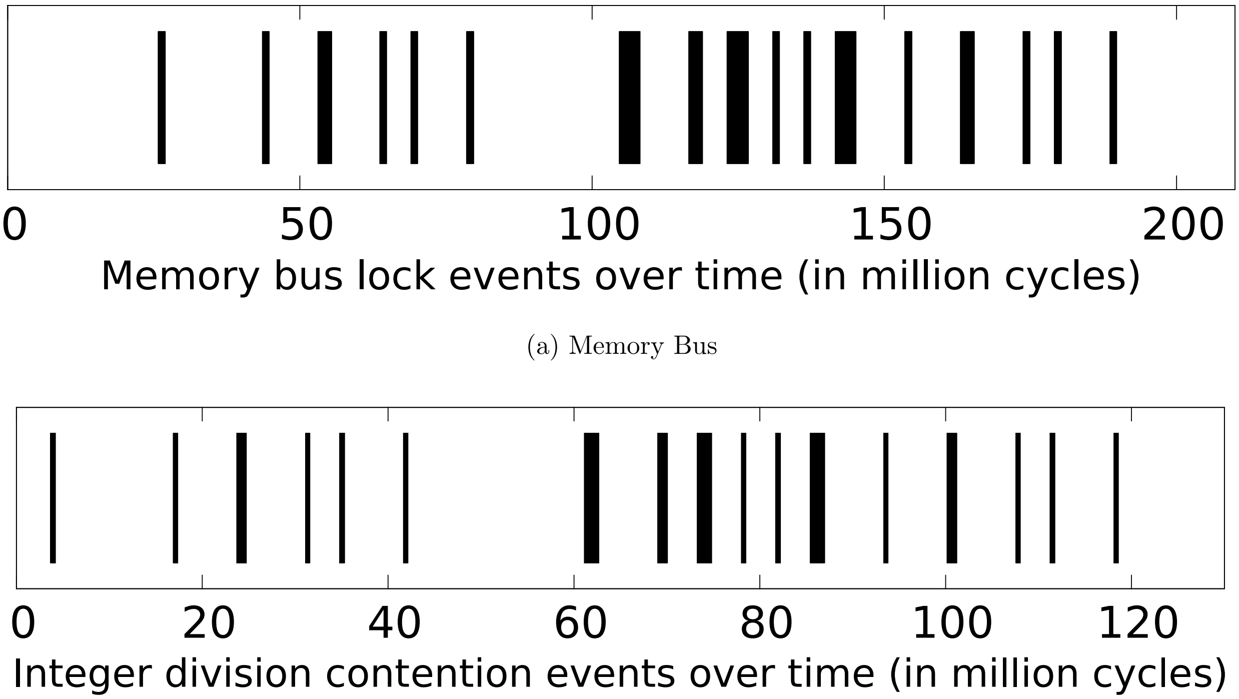
integer division operations and timing them. A '1' is inferred on the spy side using iterations that take longer amounts of time (due to contentions on the divider unit created by the trojan), and '0' is inferred when the iterations consume shorter time. Figure 5.2 shows the average latency per loop iteration as observed by the spy for the same 64-bit credit card number chosen for memory bus covert channel. We observe that the loop latency is high for '1' transmission and remains low for '0' transmission.

5.3.2 Recurrent Burst Pattern Detection

The *first* step in detecting covert timing channels is to identify the event that is behind the hardware resource contention. In the case of memory bus covert channel, the event to be monitored is memory bus lock operation. In the case of integer division covert channel, the event to be monitored is the number of times a division instruction from one process waits on a *busy* divider occupied by an instruction from another process. Note that not all division operations fall in this category.

The *second* step is to create an *Event Train*, i.e., a uni-dimensional time series showing the occurrence of events (see figures 5.3a and 5.3b). We notice a large number of thick bands (or bursty patterns of events) whenever the trojan intends to covertly communicate a '1'.

As the *third* step, we analyze the event train using our recurrent burst pattern detec-



(a) Memory Bus

(b) Integer Divider

Figure 5.3: Event Train plots for Memory Bus and Integer Divider showing burst patterns.

tion algorithm. This step consists of two parts: (1) check whether the the event train has significant contention clusters (bursts), and (2) determine if the time series pattern exhibits recurrent patterns of bursts.

Our algorithm is as follows:

1. *Determine the interval (Δt) for a given event train to calculate event density.* Δt is the product of the inverse of average event rate and α, an empirical constant determined using the maximum and minimum achievable covert timing channel bandwidth rates on a given shared hardware. In simple terms, Δt is the observation window to count the number of event occurrences within that interval. The value of Δt can be picked from a wide range, and is tempered by α factor which ensures that Δt is neither too low (when the probability of a certain number of events within Δt follows Poisson distribution) nor too high (when the probability of a certain number of events within Δt follows normal distribution). For covert

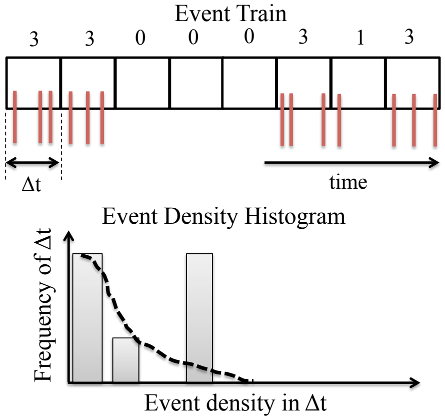
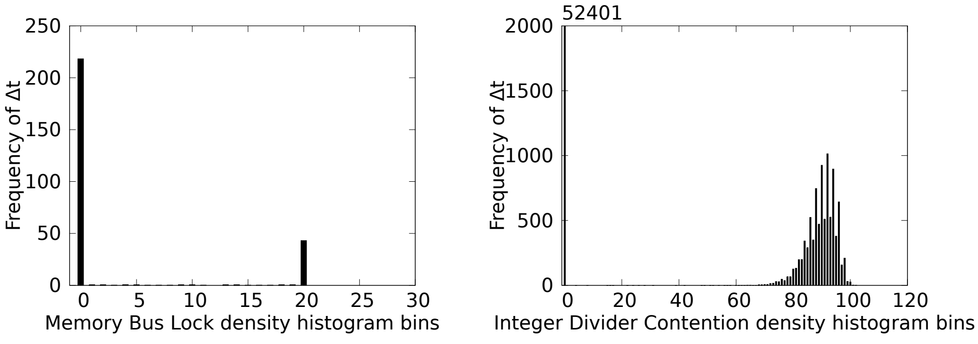


Figure 5.4: Illustration of Event Train and its corresponding Event Density Histogram. The distibution is compared against the Poisson Distribution shown by the dotted line to detect presence of burst patterns.



(a) Memory Bus

(b) Integer Division Unit

Figure 5.5: Event Density Histograms for Covert Timing Channels using Memory Bus and Integer Divider.

timing channel with memory bus, Δt is determined as 100,000 CPU cycles (or 40 μs), and in the case of covert timing channel with integer divisions, Δt is determined as 500 CPU cycles (or 200 ns).

2. *Construct the event density histogram using* Δt. For each interval of Δt, the number of events are computed, and an event density histogram is constructed to subsequently estimate the probability distribution of event density. An illustration is shown in Figure 5.4. The x-axis in the histogram plot shows the range of Δt bins that have a certain number of events. Low density bins are to the left, and as we move right, we see the bins with higher numbers of events. The y-axis shows the number of Δt's within each bin.

3. *Detect burst patterns.* From left to right in the histogram, threshold density is the first bin which is smaller than the preceding bin, and equal or smaller than the next bin. If there is no such bin, then the bin at which the slope of fitted curve becomes gentle is considered as the threshold density. If the event train has burst patterns, there will be two distinct distributions- (1) one where the mean is below 1.0 showing the non-bursty periods, and (2) one where the mean is above 1.0 showing the bursty periods present in the right tail of the event density histogram. Figure 5.5 shows the event density histogram distributions for covert timing channels involving bursty contention patterns on memory bus and integer division unit. For both timing channels, we see significant non-burst patterns in the histogram bin#0. In case of memory bus channel, we see significant bursty pattern at histogram bin#20. For Integer Division, we see a very prominent second distribution (burst pattern) between bins#84 and #105 with peak around bin#96.

4. *Identify significant burst patterns (contention clusters) and filter noise.* To estimate the significance of burst distribution and filter random (noise) distributions, we compute the likelihood ratio[2] of the second distribution. Empirically, based on observing realistic covert timing channels [85, 22], we find that the likelihood ratio of the burst pattern distribution

[2] Likelihood ratio is defined as the number of samples in the identified distribution divided by the total number of samples in the population [84]. We omit bin#0 from this computation since it does not contribute to any contention.

tends to be at least 0.9 (even on very low bandwidth covert channels such as 0.1 bps). On the flipside, we observe this likelihood ratio to be less than 0.25 among regular programs that have no known covert timing channels despite having some bursty access patterns. We set a conservative threshold for likelihood ratio at 0.5, i.e., all event density histograms with likelihood ratios above 0.5 are considered for further analysis.

5. *Determine the recurrence of burst patterns.* Once the presence of significant burst patterns are identified in the event series, the next step is to check for recurrent patterns of bursts. We limit the *window of observation* to 512 OS time quanta (or 51.2 secs, assuming a time quantum of 0.1 secs), to avoid diluting the significance of event density histograms involved in covert timing channels. We develop a pattern clustering algorithm that performs two basic steps- (1) discretize the event density histograms into strings, and (2) use k-means clustering to aggregate similar strings. By analyzing the clusters that represent event density histograms with significant bursts, we can find the extent to which burst patterns recur, and hence detect the possible presence of covert timing channel. Our algorithm can detect covert timing channels *regardless* of burst intervals (i.e., even on low-bandwidth irregular bursts or random noise due to interference from the system environment), since it uses clustering to extract recurring burst patterns.

5.3.3 Covert Timing Channel on Memory

We utilize L2 cache-based timing channel demonstrated by Xu et al [23]. To transmit a '1', the trojan visits the a dynamically determined group of cache sets (G_1)[3] and replaces all of the constituent cache blocks, and for a '0' it visits another dynamically determined group of cache sets (G_0) and replaces all of the constituent cache blocks. The spy infers the transmitted bits as follows: It replaces all of the cache blocks in G_1 and G_0, and times the accesses to the G_1 and G_0 sets separately. If the accesses to G_1 sets take longer than the G_0

[3] The conflict miss causing cache sets participating in covert transmission are chosen during covert channel synchronization phase.

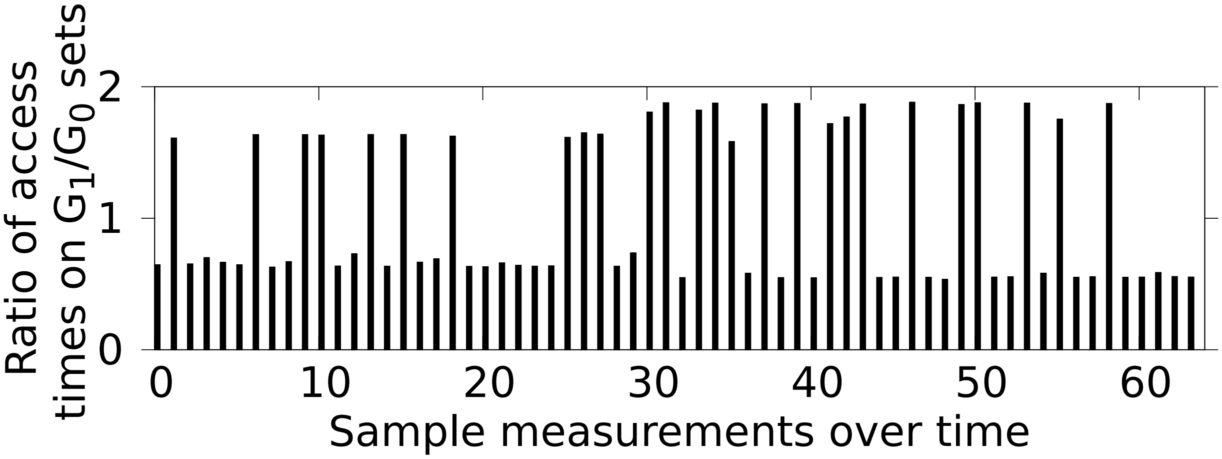
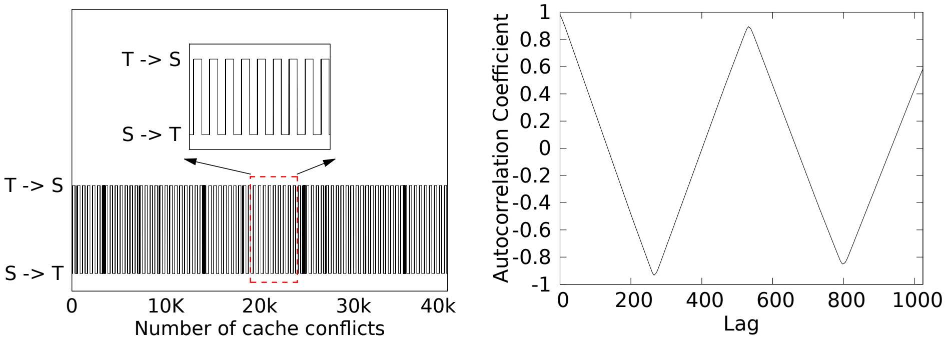


Figure 5.6: Ratios of cache access times between G_1 and G_0 cache sets in Cache Covert Channel

sets (that is, all of the G_1 sets resulted in cache misses and G_0 sets were cache hits), then the spy infers '1'. Otherwise, if the accesses to G_0 sets take longer than the G_1 sets (that is, all of the G_0 sets resulted in cache misses and G_1 sets were cache hits), then the spy infers a '0'. Figure 5.6 shows the ratio of the average cache access latencies between G_1 and G_0 cache set blocks observed by the spy for the same 64-bit randomly generated credit card number. A '1' is inferred for ratios greater than 1 (i.e., G_1 set access latencies are higher than G_0 set access latencies) and a '0' is inferred for ratios less than 1 (i.e., G_1 set access latencies are lower than G_0 set access latencies).

5.3.4 Oscillatory Pattern Detection

Unlike combinational structures where timing modulation is performed by varying the inter-event intervals (observed as bursts and non-bursts), cache based covert timing channels rely on latency of events to perform timing modulation. To transmit a '1' or a '0', the trojan and the spy create a sufficient number of conflict events (cache misses) alternatively among each other that lets the the spy decipher the transmitted bit based on the average memory access times (hit/miss). Note that this leads to oscillatory patterns of conflicts between the



(a) Event Train (T->S: Trojan's(S) conflict misses with Spy(S) and vice versa)

(b) Autocorrelogram

Figure 5.7: Oscillatory pattern of cache conflict misses between trojan and spy. An autocorrelogram is shown for the conflict miss event train.

trojan and spy processes.

Oscillation is defined as a property of periodicity in an event train. This is different from bursts that are specific periods of high frequency event occurrences in the event train. Oscillation of an event train is detected by measuring its autocorrelation [86]. Autocorrelation is defined as the correlation coefficient of the signal with a time-lagged version of itself, i.e., it is the correlation coefficient between two values of the same variable, X_i and X_{i+p} separated by a lag p.

In general, given the measurements of a variable X, (X_1, X_2, ..., X_N) at time instances of t (t_1, t_2, ..., t_N), autocorrelation coefficient r_p at a time lag of p and mean of $\bar{X}$ is defined as,

$$r_p = \frac{\sum_{i=1}^{n-p}(X_i - \bar{X}).(X_{i+p} - \bar{X})}{\sum_{i=1}^{n}(X_i - \bar{X})^2}$$

The autocorrelation function is primarily used for two purposes– (1) detecting non-randomness in data, (2) identifying an appropriate time series model if the data values are not random [86]. To satisfy #1, computing the autocorrelation coefficient for a lag value of 1 (r_1) is sufficient. To satisfy #2 and detect oscillation, autocorrelation coefficients for a sequence of lag values exhibit significant periodicity.

An autocorrelogram is a chart showing the autocorrelation coefficient values for a sequence of lag values. An oscillation pattern is inferred when the autocorrelation coefficient shows significant periodicity with peaks sufficiently high for certain lag values (i.e., the values of X correlates highly with itself at a lag distances of k_1, k_2 etc.).

Figure 5.7 shows the oscillation detection method for the covert timing channel on shared cache. In particular, Figure 5.7a shows the event train (cache conflict misses) annotated by whether the conflicts happen due to the trojan replacing spy's cache sets, or vice versa. "T->S" denotes Trojan (T) replacing Spy's(S) blocks because the spy had previously displaced those same blocks previously owned by the trojan. Since the conflict miss train shows dense cluttered pattern, we show a legible version of this event train in the inset of Figure 5.7a.

The conflict misses that are observed within each observation window (typically one OS time quantum) are used to construct conflict miss event train plot. Every conflict miss in the event train is denoted by an identifier based on the replacer process and the victim process. Note that every ordered pair of processes have unique identifiers. For example "S->T" is assigned '0' and "T->S" is assigned "1". Autocorrelation function is computed on this conflict miss event train. Figure 5.7b shows the autocorrelogram of the event train. A total of 512 cache sets were used in G_1 and G_0 for transmission of 1/0 bit values. We observe that, at a lag value of 533 (that is very close to the actual number of conflicting sets in the shared cache, 512), the autocorrelation value is highest at about 0.893. The slight offset from the actual number of conflicting sets was observed due to random conflict misses in the surrounding code and the interference from conflict misses due to other active processes sharing the cache. At a lag value of 512, the autocorrelation coefficient value was also high ($\approx$0.85). To evade detection, the trojan/spy may (with some effort) introduce pseudo-random delays between cache conflicts. This could lead to potentially lower autocorrelation coefficients, but we note that the trojan/spy may face a much bigger problem in reliable

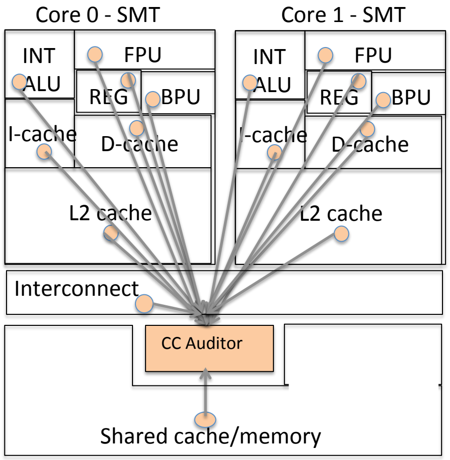


Figure 5.8: Hardware Implementation and integration of CC-Hunter into processor hardware

transmission due to higher variability in cache access latencies.

5.4 Implementation

In this section, we show the hardware modifications and software support to implement our CC-Hunter framework.

5.4.1 Hardware Support

In current microprocessor architectures, we note that most hardware units are shared by multiple threads, especially with the widespread adoption of Simultaneous Multi-Threading (SMT). Therefore, all of the microarchitectual units are potential candidates for covert timing channel mediums.

The Instruction Set is augmented with a special instruction that lets the user to program the CC-auditor and choose the certain hardware units to audit. This special instruction shall be a privileged instruction that only a subset of system users (usually the system administrator) can utilize for system monitoring. The hardware units have a programmable bit, which when set, places the hardware unit under audit for covert timing channels. The hardware units are wired to fire a signal to the CC-auditor on the occurrence of certain key indicator events seen in covert timing channels (Figure 5.8).

In super-secure environments, where performance constraints can be ignored, CC-auditor hardware can be enabled to monitor all shared hardware structures. However, this would incur unacceptable performance overheads in most real system environments. Therefore, to minimize CC-Hunter implementation complexity, we design CC-auditor with the capability to monitor up to *two* different hardware units *at any given time.* The user (system administrator) is responsible for choosing the two shared hardware units to monitor based on her knowledge of the current system jobs. We believe that this hardware design tradeoff is necessary to prevent unnecessary overheads on most regular user applications.

For most of the core components such as execution clusters and logic, the indicator events are conflicts detected by a hardware context when another context is already using them. On certain uncore components such as memory bus, conflicts are created using special events such as bus locks.

To accumulate the event signals arriving from the hardware units, the CC-Auditor contains (1) two 32-bit count-down registers initialized to the computed values of Δt based on the two microarchitecture units under monitor, (2) two 16-bit registers to accumulate the number of event occurences within Δt, and (3) two histogram buffers with 128 entries (each entry is 16-bits long) to record the event density histograms. Whenever the event signal arrives from the unit under audit, the accumulator register is incremented by one. At the end of each Δt, the corresponding 16-bit accumulator value is updated against its entry in the histogram buffer, and the count-down register is reset to Δt. At the end of OS time quantum, the histogram buffers are recorded by the software module.

For memory structures such as caches, conflicts are created using conflict-based replacement misses. Ideally, to detect conflict misses, we need a fully-associative LRU (Least Recently Used policy) stack modeling total cache capacity that tracks the recency of access and cache block replacements. Our practical scheme approximates the LRU stack through maintaining a number of generations (four in our implementation) that are ordered by age [87]. Cache blocks in the youngest generation correspond to the entries at the top of the LRU stack and the oldest generation corresponds to the blocks at the bottom of the LRU stack. Note that the blocks within a generation itself are unordered. Each cache block metadata field is augmented with four bits to track accesses during each of the four generations, and two more bits are added to track the current owner core (assuming four cores). The youngest generation bit in the cache block metadata is set upon block access, and during block replacement, the replaced cache tags are placed in a compact three-hash bloom filter corresponding to the youngest generation. A new generation is started when the number of recently accessed blocks reaches a threshold, T ($\#totalcacheblocks/\#generations$). At this time, the oldest generation is discarded by flash clearing the corresponding generation column in the cache metadata and the respective bloom filter. This action represents the removal of entries from the bottom of the LRU stack. The generation tracking bits are reused similar to circular buffers, and a separate two bit register (per cache) holds the ID of the current youngest generation. *A conflict miss is detected when the incoming cache tag is found in one of the bloom filters*, which implies that the block was replaced before reaching the cache capacity. Inside our CC-auditor, we maintain two alternating 128-byte vector registers that, *upon every conflict miss*, records the two-bit core IDs of the replacer (core requesting the cache block) and victim (current owner core in the cache block metadata). When one vector register is full, the other vector register begins to record the data. Meanwhile, the software module records the vector contents in the background (to prevent process stalling), and then clears the vector register for future use.

Area, Latency and Power Estimates of CC-auditor

We use Cacti 5.3 [26] to estimate the area, latency and power needed for our CC-auditor hardware. Table 5.1 show the results of our experiments. For the two histogram buffer, we model 128-entries that are each 16-bits long. For registers, we model two 128-byte vector

Table 5.1: Area, Power and Latency Estimates of CC-Auditor

	Histogram Buffers	Registers	Conflict Miss Detector
Area(mm^2)	0.0028	0.0011	0.004
Power(mW)	2.8	0.8	5.4
Latency(ns)	0.17	0.17	0.12

registers, two 16-bit accumulators, and two 4-byte countdown registers. For conflict miss detector, we model 4 three-hash bloom filters with $(4 * \#totalcacheblocks)$ bits, six metadata bits per cache block (four generation bits plus two bits of owner core). Our experimental results show that the CC-Hunter hardware area overheads are insignificant compared to the total chip area (e.g., 263 mm^2 for Intel i7 processors [30]). The CC-auditor hardware has latencies that are less than processor clock cycle time (0.33 ns for 3 GHz). Given that the CC-auditor hardware is accessed only when conflicts happen, it is unlikely that the CC-auditor hardware would extend the clock cycle period. Similarly, the dynamic power drawn by CC-auditor hardware are in the order of a few milliwatts compared to 130 W peak in Intel i7 processors [30].

5.4.2 Software Support

In order to place a microarchitectural unit under audit, the user requests the CC-auditor through a special software API exported by the OS, where the OS performs privilege checks. This is to prevent the sensitive system activity information from being exploited by attackers.

A separate daemon process (part of CC-Hunter software support) accumulates the data points by recording the histogram buffer contents at each OS time quantum (for contention-based channels) or 128-byte vector register (for oscillation-based channels). Lightweight code is carefully added to avoid perturbing the system state, and to record performance counters as accurately as possible [88]. To further reduce perturbation effects, the OS scheduler schedules the CC-Hunter monitors on (currently) un-audited cores.

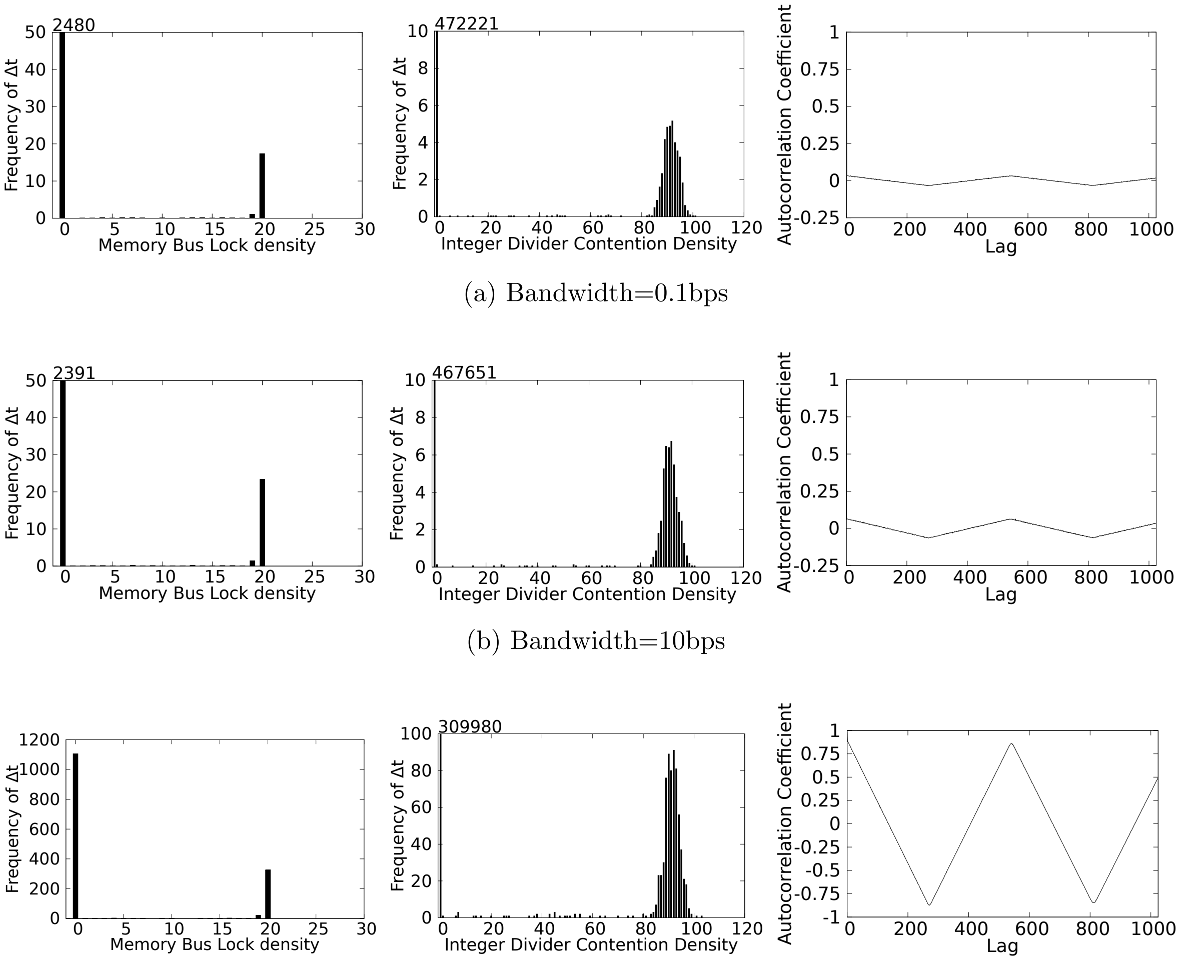
Since our analysis algorithms are run as background processes, they incur minimal effect on system performance. Our pattern clustering algorithm is invoked every 51.2 secs (Section 5.3.2) and takes 0.25 secs (worst case) per computation. We note that further optimizations such as feature dimension reduction reduces the clustering computation time to 0.02 secs (worst case). Our autocorrelation analysis is invoked at the end of every OS time quantum (0.1 secs) and takes 0.001 secs seconds (worst case) per computation.

5.5 Evaluation and Sensitivity Study

5.5.1 Varying Bandwidth Rates

We conduct experiments by altering the bandwidth rates of three different covert timing channels from 0.1 bps to 1000 bps. The results (observed over a window of OS time quantum, 0.1 secs) are shown in Figure 5.9. While the magnitudes of Δt frequencies decrease for lower bandwidth contention-based channels, the likelihood ratios for second (burst) distribution are still significant (higher than 0.9)[4]. On low-bandwidth cache covert channels such as 0.1 bps, despite observing periodicity in autocorrelation values, we note that their magnitudes do not show significant strength. We conduct additional experiments by decreasing the windows of observation to fractions of OS time quantum on 0.1 bps channel. This fine grain analysis is especially useful for lower-bandwidth channels that create a certain number of conflicts per second (needed to reliably transmit a bit) frequently followed by longer periods

[4] The histogram bins for second distribution (covert transmission) are determined by the number of successive conflicts needed to reliably transmit a bit and the timing characteristics of the specific hardware resource. For example, Δt for memory bus channel is 100,000 cycles and minimum inter-access interval between successive conflicts is 5,000 CPU cycles. Therefore, the second distribution is clustered around bin#20.



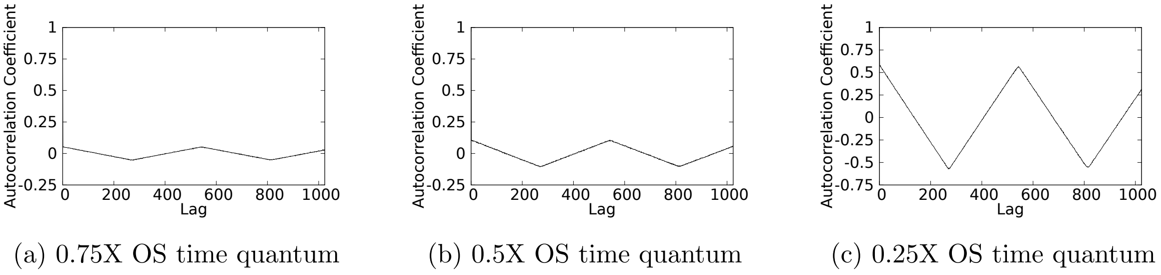
(a) Bandwidth=0.1bps

(b) Bandwidth=10bps

(c) Bandwidth=1000bps

Figure 5.9: Bandwidth test using Memory Bus, Integer Divider and Cache Covert Channels

of dormancy. Figure 5.10 shows that, as we reduce the sizes of observation window, the autocorrelograms show significant repetitive peaks for 0.1 bps channel. Our experiments suggest that autocorrelation analysis should be periodically done at higher frequency (i.e., finer granularity observation windows) to increase the probability of detection for lower-bandwidth timing channels.



(a) 0.75X OS time quantum (b) 0.5X OS time quantum (c) 0.25X OS time quantum

Figure 5.10: Autocorrelograms for 0.1 bps cache covert channels at reduced observation window sizes

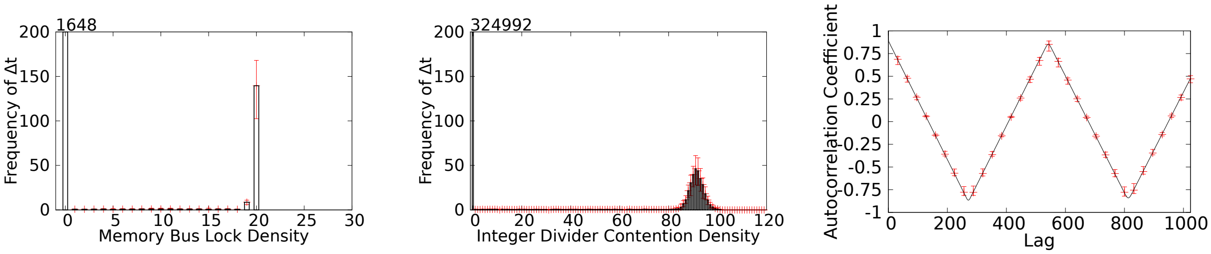
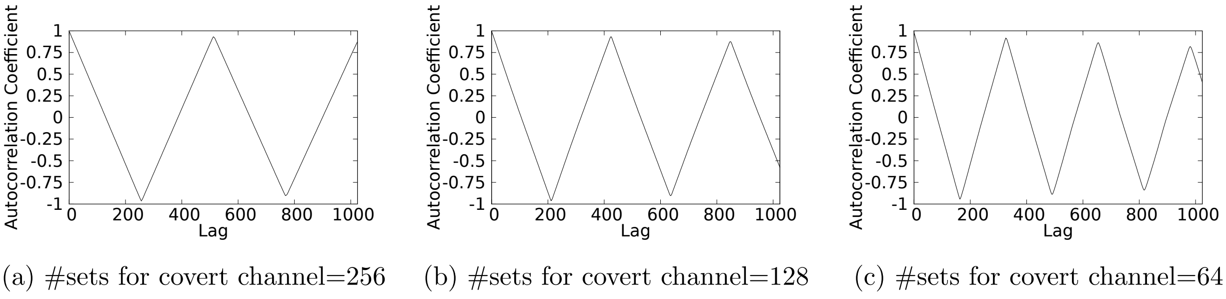


Figure 5.11: Test with 256 randomly generated 64-bit messages on Memory Bus, Integer Divider and Cache covert channels. Black (thick) bars are the means, and the red (annotations) arrows above them show the range (min, max).



(a) #sets for covert channel=256 (b) #sets for covert channel=128 (c) #sets for covert channel=64

Figure 5.12: Autocorrelograms for cache covert channel with varying numbers of cache sets for communication.

5.5.2 Encoded Message patterns

To simulate encoded message patterns that the trojan may use to transmit messages, we generate 256 random 64-bit combinations, and using them as inputs to the covert timing channels. Our experimental results are shown in Figure 5.11. Mean values of histogram bins are shown by dark bars that are annotated by the range (maximum, minimum) of bin values observed across the 128 runs. Despite variations in peak magnitudes of Δt frequencies (especially in integer divider), we notice that our algorithm still shows significant second distributions with likelihood ratios above 0.9. For autocorrelograms in cache covert channels, we notice insignificant deviations in autocorrelation coefficients despite varying bit combinations.

5.5.3 Varying Cache Channel Implementations

In Figure 5.12, we implement the cache covert timing channels by varying the number of cache sets used for bit transmission from 64 to 512. We find that the autocorrelograms in all of the cases show significant periodicity in autocorrelation with maximum peak correlation values of around 0.95, a key characteristic observed in covert timing channels. For covert channels that use smaller number of cache sets, note that random conflict misses occurring on other cache sets and interference from other active processes increase the wavelength of the autocorrelogram curve beyond the expected values (typically the number of cache sets used in covert communication).

5.5.4 Testing for False Alarms

We test our recurrent burst and oscillation pattern algorithms on 128 pair-wise combinations of several standard SPEC2006, Stream and Filebench benchmarks run simultaneously on the same physical core as hyperthreads. We pick two different types of servers from Filebench- (1) webserver, that emulates web-server I/O activity producing a sequence of open-read-close on multiple files in a directory tree plus a log file append (100 threads are used by default), (2) mailserver, that stores each e-mail in a separate file consisting of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single directory (16 threads are used by default). The individual benchmarks are chosen based on their CPU-intensive (SPEC2006) and memory- and I/O-intensive (Stream and Filebench) behavior, and are paired such that we maximize the chances of them creating conflicts on a particular microarchitectural unit. As examples, (1) both gobmk and sjeng have numerous repeated accesses to the memory bus, (2) both bzip2 and h264ref have a significant number of integer divisions. The goal of our experiments is to study whether these benchmark pairs create similar levels of recurrent bursts or oscillatory patterns of conflicts that were observed in realistic covert channel implementations (which, if true, could potentially lead to a false alarm). Despite having some regular bursts and conflict cache misses, all of the benchmark pairs are known to not have any covert timing channels. Figure 5.13 presents a representative subset of our experiments[5]. We observe that most of the benchmark pairs have either zero or random burst patterns for both memory bus lock (first column) and integer division contention (second column) events. The only exception is mailserver pairs, where we observe a second distribution with bursty patterns between histogram bins#5 and #8. Upon further examination, we find that the likelihood ratios for these distributions was less than 0.25 (which is significantly less than the ratios seen in all of our covert timing channel experiments). In almost all of the autocorrelograms (third column), we observe that the autocorrelation coefficients do not exhibit any noticeable periodicity typically expected of cache covert timing channels. The only exception was webserver where we see a very brief period of periodicity between lag values 120 and 180, but becomes non-periodic beyond lag values of 180. Therefore, we did not observe any false alarms. Also, regardless of the "cover" programs that embed the trojan/spy, CC-Hunter is designed to catch the covert transmission program phases that should be already synchronized between trojan/spy to achieve covert

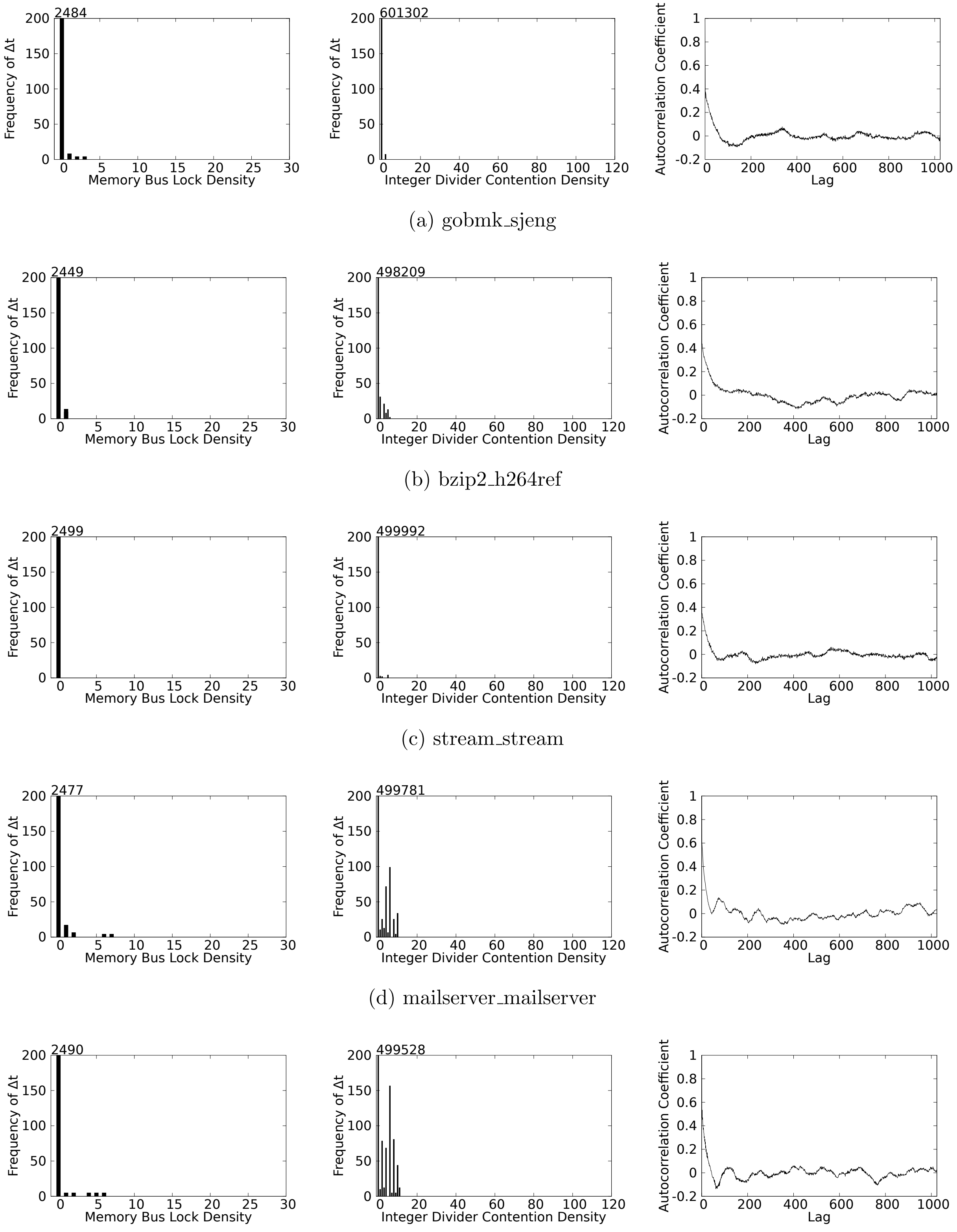
[5] Due to space constraints, we are unable to show all of our experimental results.

communication. Hence, we do not believe that the cover program characteristics could lead to any false negatives.

5.6 Related Work

The notion of covert channel was first introduced by Lampson et al [89]. Hu et al [16] proposed fuzzing system clock by randomizing interrupt timer period between 1ms and 19 ms. Unfortunately, this approach could significantly affect system's normal bandwidth and performance in the absence of any covert timing channel activity. Recent works have primarily focused on covert information transfer through network channels [90, 91] and mitigation techniques [92, 93, 94]. Among studies that consider processor-based covert timing channels, Wang et al. [20] identify two new covert channels using exceptions on speculative load (*ld.s*) instructions and SMT/multiplier unit. Wu et al. [22] present a high-bandwidth and reliable covert channel attack that is based on QPI lock mechanism where they demonstrate their results on Amazon's EC2 virtualized environment. Ristenpart et al. [19] present a method of creating cross-VM covert channel by exploiting the L2 cache, which adopts the Prime+Trigger+Probe [95] to measure the timing difference in accessing two pre-selected cache sets and decipher the covert bit. Xu et al. [23] construct a quantitative study over cross-VM L2 cache covert channels and assess their harm of data exfiltration. Our framework is tested using the examples derived from such prior covert timing channel implementations on shared hardware.

To detect and prevent covert timing channels, Kemmerer et al. [96] proposed a shared matrix methodology to *statically* check whether potential covert communications could happen using shared resources. Wang et al [97] propose a covert channel model for an abstract system specification. Unfortunately, such static code-level or abstract model analyses are impractical on every single third-party application executing on a variety of machine configurations in today's computing environments, especially when most of these applications are available in binary-only format.



(a) gobmk_sjeng

(b) bzip2_h264ref

(c) stream_stream

(d) mailserver_mailserver

(e) webserver_webserver

Figure 5.13: Event Density Histograms and Autocorrelograms in pair combinations of SPEC2k6, Stream & Filebench

Other works have studied side channels and solutions to minimize information leakage. Side channels are information leakage mechanisms where a certain malware secretly profiles a legitimate application (via differential power, intentional fault injection etc.) to obtain sensitive information. Wang et al. [98, 99] propose two special hardware cache designs, Partition-Locking (PL), Random Permutation (RP) and New cache to defend against cache-based side channel attacks. Kong et al. [100] show how secure software can use the PL cache. Martin et al. [101] propose changes to the infrastructure (timekeeping and performance counters) typically used by side channels such that it becomes difficult for the attackers to derive meaningful clues from architectural events. Demme et al. [102] introduce a metric called Side Channel Vulnerability Factor (SVF) to quantify the level of difficulty for exploiting a particular system to gain side channel information. Many of the above preventative techniques complement CC-Hunter by serving to provide enhanced security to the system.

Demme et al [103] explore simple performance counters for malware analysis. This strategy is not applicable for a number of covert channels because they use specific timing events to modulate hardware resources that may not be measurable through the current performance counter infrastructure. For instance, the integer divider channel should track cycles where one thread waits for another (unsupported by current hardware). Using simple performance counters as alternatives will only lead to high number of false positives. Second, our algorithm understands the time modulation characteristics in covert timing channel implementations. Using machine learning classifiers without considering covert timing channel-specific behavior could result in high number of false alarms.

5.7 Summary

In this chapter, we describe *CC-Hunter*, a new microarchitecture-level framework to detect the possible presence of covert timing channels on shared processor hardware. Our algorithm works by detecting recurrent burst and oscillation patterns on certain key indicator events associated with the covert timing channels. We test the efficacy of our solution using example covert timing channels on three different types of processor hardware- wires (memory bus/QPI), logic (integer divider) and memory (caches). We conduct sensitivity studies by altering the bandwidth rates, message bit combinations and number of cache sets. Our results show that, at low bandwidths, more frequent analysis (at finer grain windows of observation) may be necessary to improve the probability of detection. Through experiments on I/O, memory, CPU-intensive benchmarks such as Filebench [104], SPEC2006 [105] and Stream [106] that are known to have no covert channels, we show that our framework does not have any false alarms.

Chapter 6

Conclusions and Future Work

With the shifting of software landscape and advances of hardware platforms, software developers and system administrators are facing new challenges, namely, power, energy and information leakage, beyond just performance. While there are mature and readily available hardware monitors for optimizing performance, direct hardware monitoring support for auditing power, energy, and information leakage, and enabling actionable optimizations is still lacking.

In this dissertation, we proposed and explored the design of three novel resource auditing techniques to be implemented as part the next generation hardware monitoring infrastructure, namely, application power auditing, application energy auditing, and covert timing channel auditing. Watts-inside provides efficient hardware support to accurately audit application power, and utilizes software support and causation principles for a more comprehensive understanding of application power. EnDebug audits application by attributing energy to fine grain regions of program code, and utilizes an automated recommendation system to explore energy improvements. CC-Hunter uses low-cost hardware support for auditing and detecting the presence of covert timing channels by dynamically tracking conflict patterns on shared processor hardware.

Using the next generation hardware monitors, we can provide valuable feedbacks to the programmers, and enable them to apply static code optimizations for improving power and energy. While hardware monitors offer runtime auditing capabilities, a natural next step is to explore runtime optimization opportunities with hardware monitors. Just-in-time or JIT compiling is commonly used for runtime optimizations. At runtime, JIT engine may gather statistics from hardware performance counters. If the profile data reveals performance bottleneck in a code region, JIT engine may choose to recompile the code to gain better performance. While our next generation hardware monitors offers beyond just performance auditing, as future work, one attractive extension is to combine power and energy auditing with runtime systems, such as JIT engines, and use auditing feedbacks to dynamically optimize application code for better power or energy.

In fact, our information leakage auditor is already capable of performing runtime confining (instead of ”optimizing” as in the case of energy) toward covert timing channels. As future work, with simple extensions, our information leakage auditor, together with the operating system, can either migrate the detected processes to an isolated core for further forensic analyses, or directly kill these processes for immediate damage control. It is also interesting to study other types of information leakage using hardware resources, and explore the design of hardware monitors for them.

Bibliography

[1] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra, “Using papi for hardware performance monitoring on linux systems,” in *Conference on Linux Clusters: The HPC Revolution*, vol. 5, 2001.

[2] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ser. ICPPW ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 207–216. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2010.38

[3] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan, “Temperature-aware microarchitecture: Modeling and implementation,” *ACM Trans. Archit. Code Optim.*, 2004.

[4] J. Li, J. F. Martinez, and M. C. Huang, “The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors,” in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2004.

[5] M. S. Gupta, J. L. Oatley, R. Joseph, G. Wei, and D. M. Brooks, “Understanding voltage variations in chip multiprocessors using a distributed power-delivery network,” in *Proceedings of Design, Automation and Test in Europe (DATE)*, 2007.

[6] A. Rallo, "Data center efficiency trends for 2014," *http://www.energymanagertoday.com/data-center-efficiency-trends-for-2014-097779/*, 2013.

[7] eBay Inc., "Digital service efficiency," *http://dse.ebay.com/*, 2013.

[8] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Proceedings of 38th Annual International Symposium on Computer Architecture*, 2011.

[9] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.209

[10] D. H. Woo and H.-H. S. Lee, "Extending amdahl's law for energy-efficient computing in the many-core era," *Computer*, vol. 41, no. 12, pp. 24–31, Dec. 2008. [Online]. Available: http://dx.doi.org/10.1109/MC.2008.494

[11] R. Jotwani, S. Sundaram, S. Kosonocky, A. Schaefer, V. Andrade, G. Constant, A. Novak, and S. Naffziger, "An x86-64 core implemented in 32nm soi cmos," in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, Feb 2010, pp. 106–107.

[12] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and S. Troyer, "A 65 nm 2-billion transistor quad-core itanium processor," *Solid-State Circuits, IEEE Journal of*, 2009.

[13] M. Floyd, M. Allen-Ware, K. Rajamani, B. Brock, C. Lefurgy, A. Drake, L. Pesantez, T. Gloekler, J. Tierno, P. Bose, and A. Buyuktosunoglu, "Introducing the adaptive energy management features of the power7 chip," *Micro, IEEE*, 2011.

[14] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 92–101. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486801

[15] J. Gray III, "On introducing noise into the bus-contention channel," in *IEEE Computer Society Symposium on Security and Privacy*, 1993.

[16] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3, pp. 233–254, 1992.

[17] K. Okamura and Y. Oyama, "Load-based covert channels between xen virtual machines," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, 2010.

[18] C. Percival, "Cache missing for fun and profit," 2005.

[19] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security.* ACM, 2009, pp. 199–212.

[20] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Annual Computer Security Applications Conference.* IEEE, 2006, pp. 473–482.

[21] J. C. Wray, "An analysis of covert timing channels," *Journal of Computer Security*, vol. 1, no. 3, pp. 219–232, 1992.

[22] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: high-speed covert channel attacks in the cloud," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12, 2012.

[23] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting, “An exploration of L2 cache covert channels in virtualized environments,” in *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, ser. CCSW ’11, 2011.

[24] J. Renau *et al.*, “SESC,” *http://sesc.sourceforge.net*, 2006.

[25] D. Brooks, V. Tiwari, and M. Martonosi, “Wattch: a framework for architectural-level power analysis and optimizations,” in *Proceedings of 27th Annual International Symposium on Computer Architecture*, 2000.

[26] N. P. Jouppi *et al.*, “Cacti 5.1,” *http://quid.hpl.hp.com:9081/cacti/*, 2008.

[27] J. Chen, G. Venkataramani, and G. Parmer, “The need for power debugging in the multi-core environment,” *IEEE Computer Architecture Letters*, 2012.

[28] M. D. Powell, A. Biswas, J. Emer, S. Mukherjee, B. Sheikh, and S. Yardi, “Camp: A technique to estimate per-structure power at run-time using a few simple parameters,” in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2009.

[29] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Trans. Inf. Theor.*, vol. 23, no. 3, Sep. 2006.

[30] Intel Corporation, “Intel core i7-920 processor,” *http://ark.intel.com/Product.aspx?id=37147*, 2010.

[31] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: characterization and methodological considerations,” in *Proceedings of 22th Annual International Symposium on Computer Architecture*, 1995.

[32] C. Bienia, S. Kumar, J. Singh, and K. Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications,” *Princeton University Technical Report TR-811-08*, January 2008.

[33] Synopsys Inc., "Design Compiler User Guide," *http://www.synopsys.com*, 2012.

[34] J. E. Stine, I. Castellanos, M. Wood, J. Henson, F. Love, W. R. Davis, P. D. Franzon, M. Bucher, S. Basavarajaiah, J. Oh, and R. Jenkal, "Freepdk: An open-source variation-aware design kit," in *Proceedings of MSE*, ser. MSE '07, 2007.

[35] AMD, "Software optimization guide for the amd64 processors," *http://support.amd.com/us/Processor_TechDocs/25112.PDF*, 2008.

[36] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," in *Proceedings of 30th Annual International Symposium on Computer Architecture*, 2003.

[37] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Reducing set-associative cache energy via way-prediction and selective direct-mapping," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2001.

[38] E. Witchel, C. S. Larsen, S.and Ananian, and K. Asanović, "Direct addressed caches for reduced power consumption," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2001.

[39] C. Wilkerson, H. Gao, A. R. Alameldeen, Z. Chishti, M. Khellah, and S. Lu, "Trading off cache capacity for low-voltage operation," *IEEE Micro*, 2009.

[40] D. Folegnani and A. González, "Energy-effective issue logic," in *Proceedings of 28th Annual International Symposium on Computer Architecture*, 2001.

[41] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, and P. Bose, "Energy efficient co-adaptive instruction fetch and issue," in *Proceedings of 30th Annual International Symposium on Computer Architecture*, 2003.

[42] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2001.

[43] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: reducing the energy of mature computations," in *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[44] V. Govindaraju, C. H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '11, 2011.

[45] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: A low-power pipeline based on circuit-level timing speculation," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2003.

[46] J. Sartori, B. Ahrens, and R. Kumar, "Power balanced pipelines," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2012.

[47] K. K. Rangan, G. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proceedings of 36th Annual International Symposium on Computer Architecture*, 2009.

[48] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *Proceedings of 36th Annual International Symposium on Computer Architecture*, 2009.

[49] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho, "Profile-based dynamic voltage and frequency scaling for a multiple clock domain processor," in *Proceedings of 30th Annual International Symposium on Computer Architecture*, 2003.

[50] Q. Wu, D. W. Martonosi, M.and Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee, and D. Brooks, "A dynamic compilation framework for controlling microprocessor energy and performance," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2005.

[51] Y. Zhu, G. Magklis, M. L. Scott, C. Ding, and D. H. Albonesi, "The energy impact of aggressive loop fusion," in *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[52] V. Tiwari, S. Malik, A. Wolfe, and M. T. Lee, "Instruction level power analysis and optimization of software," *J. VLSI Signal Process. Syst.*, vol. 13, no. 2-3, Aug. 1996.

[53] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, ser. WMCSA '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 2–. [Online]. Available: http://dl.acm.org/citation.cfm?id=520551.837522

[54] F. Wolf, , J. Kruse, and R. Ernst, "Segment-wise timing and power measurement in software emulation," in *Proceedings of Design, Automation and Test in Europe (DATE) , Designers' Forum*, 2001.

[55] F. Chang, K. I. Farkas, and P. Ranganathan, "Energy-driven statistical sampling: detecting software hotspots," in *Proceedings of the 2Nd International Conference on Power-aware Computer Systems*, 2003.

[56] T. Simunic, L. Benini, and G. De Micheli, "Energy-efficient design of battery-powered embedded systems," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, 1999.

[57] T. Šimunić, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proceedings of the 13th International Symposium on System Synthesis*, 2000.

[58] C. Isci and M. Martonosi, "Runtime power monitoring in high-end processors: Methodology and empirical data," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2003.

[59] C. Isci, G. Contreras, and M. Martonosi, "Live, runtime phase monitoring and prediction on real systems with application to dynamic power management," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2006.

[60] H. Jacobson, A. Buyuktosunoglu, P. Bose, E. Acar, and R. Eickemeyer, "Abstraction and microarchitecture scaling in early-stage power modeling," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2011.

[61] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the annual ACM/IEEE international symposium on Microarchitecture*, 2009.

[62] E. Rotem, A. Naveh, D. Rajwan, A. Ananthakrishnan, and E. Weissmann, "Power-management architecture of the intel microarchitecture code-named sandy bridge," *Micro, IEEE*, 2012.

[63] F. Bellosa, "The benefits of event: Driven energy accounting in power-sensitive systems," in *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, ser. EW 9, 2000.

[64] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Proceedings of the 14th International Joint Conference*

on Artificial Intelligence - Volume 2, ser. IJCAI'95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143. [Online]. Available: http://dl.acm.org/citation.cfm?id=1643031.1643047

[65] Free Software Foundation, Inc, "GCC, the GNU Compiler Collection," *http://gcc.gnu.org*, 2012.

[66] M. Brameier and W. Banzhaf, *Linear Genetic Programming.* Springer, 2007.

[67] E. Schulte, J. Dorn, S. Harding, S. Forrest, and W. Weimer, "Post-compiler software optimization for reducing energy," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[68] Intel Corporation, *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, March 2009, no. 248966-018.

[69] W. Bircher and L. John, "Complete system power estimation using processor performance events," *Computers, IEEE*, 2012.

[70] Y. Zhai, X. Zhang, S. Eranian, L. Tang, and J. Mars, "Happy: Hyperthread-aware power profiling dynamically," in *In Proc. of Usenix Annual Technical Conference*, 2014.

[71] K. Shen, A. Shriraman, S. Dwarkadas, X. Zhang, and Z. Chen, "Power containers: An os facility for fine-grained power and energy management on multicore servers," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13, 2013.

[72] T. M. Jones, M. F. P. O'Boyle, J. Abella, and A. Gonzalez, "Software directed issue queue power reduction," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, 2005.

[73] V. J. R. Yuhazo Zhu, "Webcore: Architectural support for mobile web browsing," in *Proceedings of 40th Annual International Symposium on Computer Architecture*, 2014.

[74] M. Hayenga, V. Reddy, and M. H. Lipasti, “Revolver: Processor architecture for power efficient loop execution,” in *In Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA 2014)*, 2014.

[75] C.-H. Hsu and U. Kremer, “The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03. New York, NY, USA: ACM, 2003, pp. 38–48. [Online]. Available: http://doi.acm.org/10.1145/781131.781137

[76] W. Kim, M. Gupta, G.-Y. Wei, and D. Brooks, “System level analysis of fast, per-core dvfs using on-chip switching regulators,” in *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, 2008.

[77] K. Ma, X. Li, M. Chen, and X. Wang, “Scalable power control for many-core architectures running multi-threaded applications,” in *Proceedings of 38th Annual International Symposium on Computer Architecture*, 2011.

[78] Department of Defense Standard, *Trusted Computer System Evaluation Criteria DoD 5200.28-STD.* US Department of Defense, 1983.

[79] H. Okhravi, S. Bak, and S. King, “Design, implementation and evaluation of covert channel attacks,” in *2010 IEEE International Conference on Technologies for Homeland Security (HST)*, 2010.

[80] N. E. Proctor and P. G. Neumann, “Architectural implications of covert channels,” in *Proceedings of the Fifteenth National Computer Security Conference*, vol. 13, 1992, pp. 28–43.

[81] Y. Kaneoke and J. Vitek, “Burst and oscillation as disparate neuronal properties,” *Journal of neuroscience methods*, vol. 68, no. 2, pp. 211–223, 1996.

[82] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011 (DAC'11)*, 2011.

[83] Intel Corporation, "Intel 7500 chipset," *Datasheet*, 2010.

[84] National Institute of Standards and Technology, "Maximum Likelihood," 2013. [Online]. Available: http://www.itl.nist.gov/div898/handbook/eda/section3/eda3652.htm

[85] B. Saltaformaggio, D. Xu, and X. Zhang, "Busmonitor: A hypervisor-based solution for memory bus covert channels," in *Proceedings of European Workshop on System Security*, 2013.

[86] G. E. Box, G. M. Jenkins, and G. C. Reinsel, *Time series analysis: forecasting and control.* Wiley, 2011, vol. 734.

[87] G. P. V. Venkataramani, "Low-cost and efficient architectural support for correctness and performance debugging," *Ph.D. Dissertation, Georgia Institute of Technology*, 2009.

[88] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proceedings of 38th Annual International Symposium on Computer Architecture.* IEEE, 2011, pp. 353–364.

[89] B. W. Lampson, "A note on the confinement problem," *Commun. ACM*, vol. 16, no. 10, Oct. 1973.

[90] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia, "Model-based covert timing channels: Automated modeling and evasion," in *Recent Advances in Intrusion Detection.* Springer, 2008, pp. 211–230.

[91] K. Kothari and M. Wright, "Mimic: An active covert channel that evades regularity-based detection," *Comput. Netw.*, vol. 57, no. 3, pp. 647–657, Feb. 2013.

[92] S. Cabuk, C. E. Brodley, and C. Shields, "Ip covert channel detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 4, p. 22, 2009.

[93] S. Gianvecchio and H. Wang, "Detecting covert timing channels: an entropy-based approach," in *Proceedings of the 14th ACM conference on Computer and communications security*, ser. CCS '07, 2007.

[94] A. Shabtai, Y. Elovici, and L. Rokach, *A survey of data leakage detection and prevention solutions*. Springer, 2012.

[95] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *J. Cryptol.*, vol. 23, no. 2, Jan. 2010.

[96] R. A. Kemmerer, "Shared resource matrix methodology: An approach to identifying storage and timing channels," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 3, pp. 256–277, 1983.

[97] Z. Wang and R. B. Lee, "New constructive approach to covert channel modeling and channel capacity estimation," in *Proceedings of the 8th International Conference on Information Security*, ser. ISC'05, 2005.

[98] ——, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07, 2007.

[99] Z. Wang and R. Lee, "A novel cache architecture with enhanced performance and security," in *41st IEEE/ACM International Symposium on Microarchitecture*, 2008.

[100] J. Kong, O. Aciicmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *IEEE 15th International Symposium on High Performance Computer Architecture*, 2009.

[101] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *Proceedings of the 39th International Symposium on Computer Architecture*, 2012.

[102] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *Proceedings of 39th Annual International Symposium on Computer Architecture*, 2012.

[103] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013.

[104] File system and Storage Lab, "Filebench," *http://sourceforge.net/apps/mediawiki/filebench*, 2011.

[105] Standard Performance Evaluation Corporation, "Spec 2006 benchmark suite," *http://www.spec.org*, 2006.

[106] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.