AN INFRASTRUCTURE FOR PROGRAM POWER BEHAVIOR CHARACTERIZATION AND OPTIMIZATION EVALUATION

BY CHUNLING HU

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Prof. Daniel A. Jiménez **Prof. Ulrich Kremer**

and approved by

_____

_____

_____

_____

New Brunswick, New Jersey

January, 2008

ABSTRACT OF THE DISSERTATION

An Infrastructure for Program Power Behavior Characterization and Optimization Evaluation

by Chunling Hu

Dissertation Director: Prof. Daniel A. Jiménez Prof. Ulrich Kremer

Fine-grained program power behavior is useful in both evaluating power optimizations and observing power optimization opportunities. Detailed power simulation is time consuming and often inaccurate. Physical power measurement is faster and objective. However, fine-grained measurement generates enormous amounts of data in which locating important features is difficult, while coarse-grained measurement sacrifices important detail.

This thesis presents a program power behavior characterization infrastructure that identifies program phases, selects a representative interval of execution for each phase, and instruments the program to enable precise and objective power measurement of these intervals to get their time-dependent power behavior. This infrastructure is constituted of three components for instrumentation, phase classification and power measurement, respectively. The *Camino* compiler, a GCC post-processor, is used to instrument the assembly code of a program on various levels. A phase classification algorithm using infrequent basic blocks and the combination of control-flow information and runtime event counts finds the representative intervals in terms of time-dependent power behavior. These selected intervals accurately characterize the fine-grained time-dependent behavior of the program, as well as accurately estimate the total energy consumption of a program. The power measurement method enables users to measure any specified region of program execution. A two-level profiling method implemented in this

infrastructure maps the measured detailed power behavior back to source code. The accuracy of this infrastructure is validated on a StrongARM SA110 through simulation, and on an Intel Pentium 4 system through physical power measurement.

This thesis also presents the uses of this infrastructure in understanding the power behavior of program components, such as procedures or loops, in finding good threshold for metrics used in dynamic voltage and frequency scaling, and in scheduling simultaneous multi-threaded programs for peak power optimization.

Acknowledgements

I am deeply grateful to my advisers, Professor Daniel A. Jiménez and Professor Ulrich Kremer. I was lucky to be supervised by them and it was a great pleasure to work with them. Their knowledge, wisdom, and enthusiasm for research guided me into this thesis work and played very important roles in the past 4 years. Their kindness, patience, humor, and understanding made my Ph.D life very happy. This thesis is not possible without their support and help. I thank my thesis defense committee member, Professor Ricardo Bianchini and Dr. Jack Liu, for their review of my thesis and their comments.

I want to thank my labmates, Yang Ni, Jerry Hom, and John McCabe, for their kind help in my research and life. I also want to thank many friends around for all the happy time we had together.

I owe my parents. I thank them and my brother for their generous love and support, without which my Ph.D study would not have been possible. Finally, I thank my dear husband, Weilei Zhang, who always stands beside me, listens to me , and encourages me in my study and life.

Dedication

To my parents and Weilei.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Increased transistor density has supported performance improvement of computing systems, but power has also emerged as a challenge for device scaling. Most of the power dissipation of CMOS microprocessors comes from the switching power of transistors, which can be calculated as

$$P = f * C * V_{dd}^2 \tag{1.1}$$

Here, f is the switching frequency, C is the load capacitance of the transistor and V_{dd} is the voltage. Frequency increases due to increasing pipeline depths. The increases in the number of transistors and frequency dominate the decreases in load capacity and voltage, resulting in power dissipation growth. Power is likely to become the major limitation in the development of computer architecture [35]. Increasingly popular and capable hand-held devices give us convenience, but the short battery life limits their usefulness. Increased energy consumption is also a big environmental problem due to its requirement of cooling systems and its pressure on resources. Low power and energy consumption is no longer a by-product of performance improvements yielded by computer architecture, compiler, and operating system optimizations.

Research in power and energy optimizations focuses not only on reducing overall program energy consumption, but also on improving time-dependent power behavior. Evaluating such optimizations requires both accurate total energy consumption estimation and precise detailed time-dependent power behavior. Simulators are often used for power and performance evaluation, but detailed power simulation is very time-consuming and often inaccurate. While

physical measurement is much faster, fine-grained power measurement requires proper measurement equipment, a large amount of space to store measurement results, and a method to extract useful information from the results. This thesis introduces a new strategy to enable efficient time-dependent power behavior characterization based on physical measurements.

1.1.1 Detailed Time Dependent Power Behavior

Some power optimizations can not be evaluated through only simulating the program to get the total energy consumption. Detailed power behavior is desired in many cases. An example optimization that requires fine-grained, time-dependent power behavior information for its experimental evaluation is instruction scheduling for peak power and step power (dI/dt problem) reduction, for instance in the context of VLIW architectures [70, 64, 62]. This previous work relies on simulation to evaluate the impact of the proposed optimizations. The dI/dt problem is caused by large variations of current in a short time. Large variations in CPU current requires the power distribution network have sufficient capacitance, and small enough inductance and resistance, to maintain the supply voltage to be stable. So such variations in CPU current may cause undesired oscillation in CPU supply voltage, which may results in timing problems and incorrect calculations [20]. In mobile applications, minimizing peak power can help reduce the physical battery size [49]. The size of the battery depends on the storage capacity required, the maximum discharge rate, the maximum charge rate, and the minimum temperature at which the batteries will be used. When intermittent operation is added, it is the peak power rather than the average power that determines capacity. There are various ways to save energy consumption, but minimizing the energy consumption of a battery-powered system is not equivalent to maximizing its battery life [51].

Figure 1.1 shows two power curves that have the same energy consumption but different time-dependent power behavior. Estimating total energy consumption is not sufficient to evaluate peak power optimization.

Another example is peak power reduction on hyper-threading and multi-core systems. In a hyper-threading or multi-core system, multiple threads run simultaneously to improve overall efficiency of CPU(s). During program execution, CPU-intensive regions and memory intensive regions result in different CPU power, i.e. CPU current. When the high-power region of two

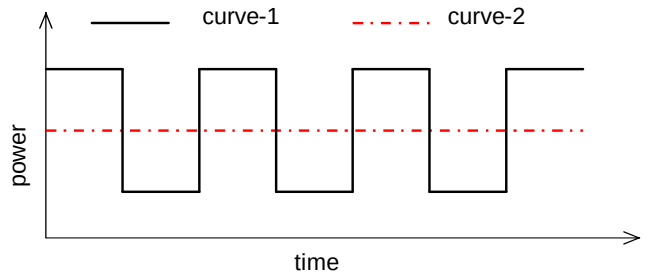


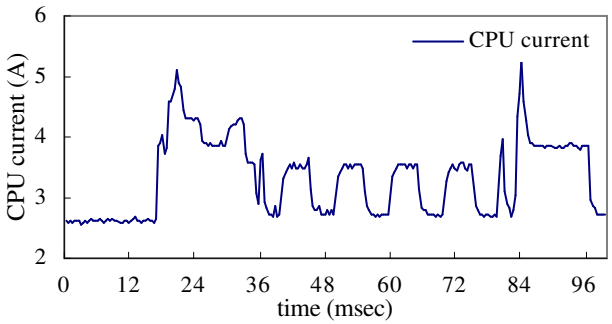
Figure 1.1: Power curves with the same energy consumption but different time-dependent power behavior.

threads are executed simultaneously, the total CPU current is increased. Figure 1.1.1(a) is the power curve of a simple benchmark running on a single CPU of a hyper-threading machine. There are high-power and low-power regions during program execution. When two of such benchmark run on the two virtual processors of a hyper-threading machine, the measured power curve is shown in Figure 1.1.1(b). Since the two programs are in their high-power region simultaneously, the measured CPU peak power is much higher than the peak power when only one program is running. They are both in very low power region in between the high-power regions.

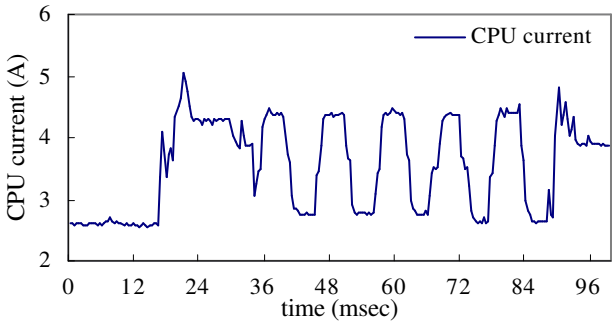
Figure 1.1.1(c) shows the measured power curve the two simultaneous programs when they are synchronized to avoid high peak-power. The CPU peak power is much lower than the one shown in Figure 1.1.1(b). The power behavior of the regions other than the loop is not changed because we only synchronized the iterations of the loop. The dotted line is the trigger signal used to identify the power behavior of the four high-power regions in Figure 1.1.1(a).

If we can characterized the detailed time-dependent power behavior of program execution, we can find its high-power regions and perform synchronization to avoid high peak-power.

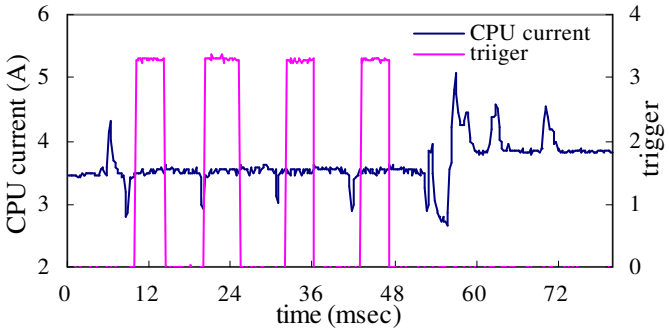
Furthermore, detailed power information can expose more power and energy optimization opportunities.



(a) Measured CPU power of a small benchmark.



(b) Measured CPU power when two copies of the same benchmark are running on two virtual machines. The two programs are not synchronized.



(c) Measured CPU power when two copies of the same benchmark are running on two virtual machines. The two programs are synchronized.

1.1.2 Insufficiency of Several Efficient Simulation Methods in Characterizing Power Behavior

Although detailed power simulation is useful, its cost in terms of time and space prevents it from being used to collect the profile of a long-running program. Some power simulators are built on SimpleScalar, a cycle-level simulator [1]. Many of the SPEC2000 programs have more than 300 billion or more instructions. Simulation of such a benchmark by SimpleScalar takes approximately 1 month of CPU time at a simulation rate of 400 million instructions per hour [48]. Simulating power takes longer time due to the calculation of power consumption of each modeled component in each step. Recording the power dissipation value for each cycle of the execution of a long program requires even more time and a huge amount of space.

Researchers have developed many ways to handle the time cost problem, such as statistical sampling [14] [68], smaller input set [36], and SimPoint [57]. These methods reduce simulation time while keeping high accuracy, they are useful in estimation of the overall characteristics of the simulated program, such as instructions-per-cycle (IPC), cache miss rate, branch misprediction rate, etc. But it is not always true for time-dependent behavior. Figure 1.2 shows the power behavior of two intervals from the same phase of *jpegencode*, one of the MediaBench benchmarks [39], after the phase classification performed by SimPoint. Intervals in the same phase are expected to have similar behavior. Due to the large number of cycles for an interval, only a segment of the interval power behavior is shown here. The power behavior is from the middle of the intervals, totally 25000 cycles, not the whole interval. Granularity is 50 cycle/point.

There is apparent difference between the two intervals in terms of power behavior. Interval-1 has a larger power range and more dramatic fluctuation in power dissipation than interval-2. Both have repeating power behavior, but interval-1 has a longer period. Furthermore, the two intervals have different execution cycles and total power dissipation, which is not shown in Figure 1.2. No matter which interval is selected as the simpoint for this cluster, it can not represent the two intervals in power behavior.

It is even harder for statistical sampling to get the representative slices for power behavior. To efficiently and accurately characterize program power behavior, we need another phase

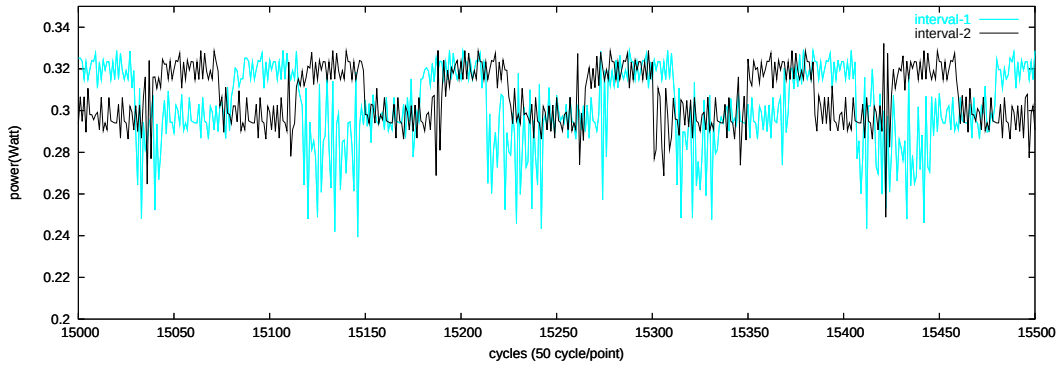


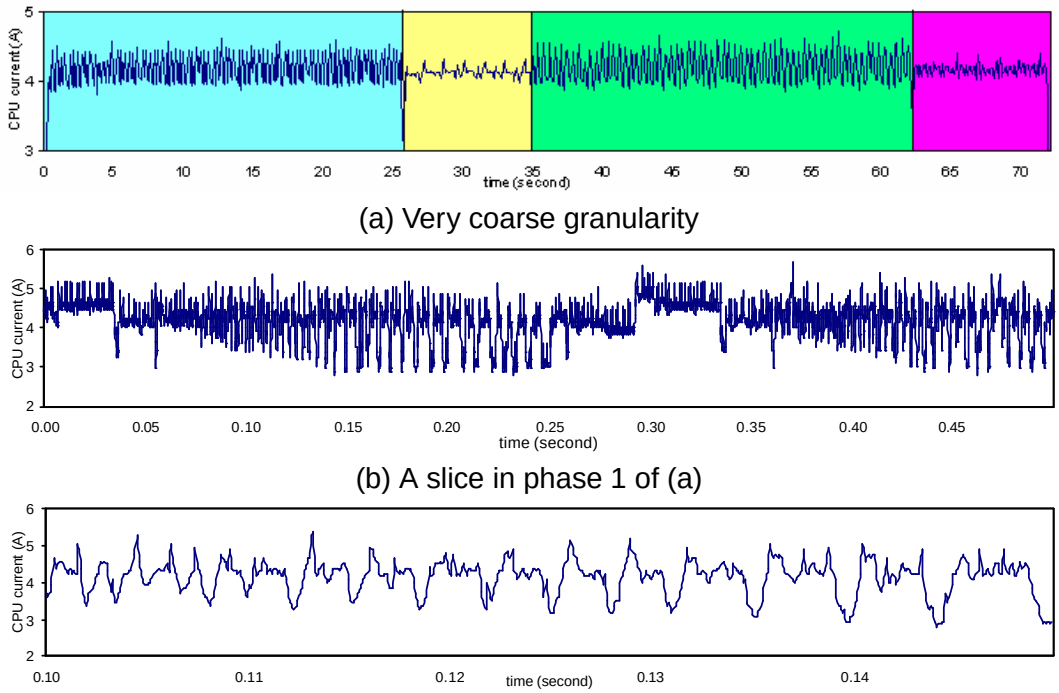
Figure 1.2: Power dissipation of two intervals from the same cluster of *jpegencode*.

classification method to find out the representative intervals for power simulation.

Another disadvantage of these methods is that they can not provide the relation between the sampled power behavior and the corresponding code, while this relation can help in better understanding of program behavior and the impact of some specific optimization, and sometimes inspire programmer or compiler researchers to do power optimizations.

1.1.3 Illustrating Time-Dependent Power Behavior

Figure 1.3 shows the measured CPU current of *256.bzip2* from SPEC CPU 2000 measured using an oscilloscope. Figure 1.3(a) shows that the program execution can be roughly partitioned into 4 phases based on its power behavior. One representative slice from each phase can be measured to characterize the detailed power behavior of the benchmark. Figure 1.3(b) is the measured power behavior of half of a second in the first phase with a resolution that is 100 times higher than the one used for Figure 1.3(a). There is a repeated power behavior period of 300 milliseconds. Figure 1.3(c) shows the detailed power behavior of a piece of 0.05 second, from 0.1 second to 0.15 second in Figure 1.3(b). It shows repeated power behavior periods of less than 5 milliseconds, indicating possible finer phase classification than Figure 1.3(b). Also, finer measurement gives more information of time-dependent CPU power due to the resolution of the oscilloscope used for power measurement. The oscilloscope reports the average power for a given time granularity. This is the reason why the difference between the observed peak power (peak current) in Figure 1.3(a) and (c) is almost 6 Watts (0.5 amperes).



(a) Very coarse granularity

(b) A slice in phase 1 of (a)

(c) Detailed CPU power behavior of a small slice in (b)

Figure 1.3: Measured power behavior of bzip2 with different granularity.

1.2 An Infrastructure for Characterizing Time-Dependent Power Behavior

Thesis: a phase classification method based on the combination of control-flow information and runtime events accurately and efficiently characterize program time-dependent power behavior; furthermore, a good interval partitioning method enables objective program power behavior characterization on real systems.

This thesis presents our infrastructure for program time-dependent power behavior characterization and optimization evaluation. It has three components for instrumentation, phase classification, and physical power measurement, respectively. A two-level profiling method implemented in this infrastructure enables the mapping between the measured power behavior and the corresponding source code level structures, such as a procedure or loop.

Our *Camino* compiler statically instruments the assembly code of a program generated by GCC. Instrumentation can be performed on various levels, from instruction-level to procedure-level. Basic block-level instrumentation is needed for *basic block vector* (BBV) or *edge vector* (EV) profiling and infrequent basic block filtering. Procedure-level and loop-level instrumentation supports a two-level profiling to setup semantic connection between physical measurement result and source code.

The phase classification component partitions program execution into intervals, which are demarcated by infrequent basic blocks. Unlike the phase classification methods based on fixed length intervals, control-flow information, or runtime events, EV and event counters are combined to be used as the fingerprint of each interval in our infrastructure. Phase classification is based on both control-flow information and runtime event counts, and representative intervals are selected for measurement. Since our objective is to characterize detailed time-dependent power behavior of program execution, the selected intervals should be representative in terms of not only energy consumption, but also time-dependent power behavior. That is, the power curve of two intervals from the same phase should be similar to each other. We use the FFT result of power curves to evaluate the similarity between two intervals. Experimental result shows that our phase classification method can find out representative intervals in terms of time-dependent power behavior, such that we can use this infrastructure for efficient power behavior characterization.

The power measurement component measure the CPU current and voltage of any specified program region or the whole program.

All of the operations in the three components are implemented as automatic processes. The threshold for determining whether a basic block is infrequent, the minimum number of instructions in each interval, and the number of phases are the input to this process. The implementation of each step will be presented in the following chapters.

We show that our method enables us to do power measurement for simpoints with very low interference to program execution. To demonstrate the improved accuracy of using edge vectors, instead of basic block vectors, for classification, we show that our infrastructure estimates the total energy of a program with an average error of 7.8%, compared with 12.0% using basic block vectors, an improvement of 35%. More importantly, in addition to chracterizing overall metrics as IPC and total energy consumption, we want to find representative intervals that represent the fine-grained time-dependent power profile of a phase. We develop a metric for measuring the accuracy of estimating a power profile and show that using edge vectors with event counter information improves accuracy by 22%.

This infrastructure can be used to efficiently characterize whole program power behavior and evaluate optimizations for energy consumption or time-dependent power behavior, for example, the impact on power behavior of pipeline gating or dynamic voltage/frequency scaling [44, 22].

1.3 Challenges

Characterizing the time-dependent power behavior of whole program execution requires accuracy and efficiency. When phase classification is used to find representative intervals, these intervals should be really representative in terms of runtime time-dependent power behavior, as well as energy consumption and performance. A method is needed to evaluate power behavior similarity among the intervals of the same phase. Without loss in accuracy, the number of the selected intervals should be as few as possible to save simulation time or simplify analysis on measurement result.

Control-flow graph constructing and runtime event counter profiling need instrumentation

on various levels of a program. The instrumentation should identify each traversed edge during program execution and identify each interval for event counter profiling with low overhead.

The measurement result of a program region should be as close as possible to the real power behavior of the region. That is, the region should be identified precisely during program execution and the instrumentation overhead should have negligible impact on the measured power behavior.

Measured power behavior is just raw data. Power behavior of the representative intervals can be used to characterize the behavior of the whole program execution. However, without information of the source code, it is hard to figure out the reason for some interesting behavior and use the observed power behavior to direct compiler-level power optimization.

Validation of the infrastructure on real systems requires system-level support for event counter access and processor voltage/frequency scaling.

1.4 Contribution

This thesis describes an infrastructure for program power behavior characterization and optimization evaluation, as well as its uses in understanding program power behavior and power optimizations. It makes the following contributions:

1. It demonstrates that there are phases in the power behavior of program execution and validates the phase behavior through power physical measurement on real systems.

2. It uses infrequent basic block to demarcate intervals during program execution, which significantly reduces instrumentation overhead for dynamic interval identification. This low instrumentation overhead enables objective power behavior measurement of intervals. It is crucial for the application of phase classification on program power behavior on real systems.

3. It shows that using edge vectors significantly improves accuracy over using basic block vectors for estimating total program energy as well as fine-grained power behavior. Combining control-flow information such as edge vectors with runtime event counts can further improve the phase classification accuracy. Distance between the Fast Fourier Transform (FFT) results of the two power curves can be used to evaluate the similarity between two intervals in terms of time-dependent power behavior, given that the two intervals have similar edge vectors.

4. It presents the *Camino* compiler infrastructure, a GCC post-processor. *Camino* enables instrumentation on various levels for profiling or power/performance measurement.

5. It proposes a power physical measurement method that can be used to measure the power behavior of any interval of program execution. By using infrequent basic blocks to demarcate intervals, we efficiently measure the power profile of an interval with minimal perturbation to the running program.

6. It implements a two-level profiling method to solve the problem of lack of semantic meaning in power measurement result. This also provides possible feedback to compiler researchers for further power optimizations.

7. It compares the benefit from using different metrics in Dynamic Voltage and Frequency Scaling and shows the use of this infrastructure in power optimizations.

1.5 Organization of this thesis

The rest of this thesis is organized as the follows:

Chapter 2 describes previous work related to this thesis. Chapter 3 introduces SimPoint [57] and demonstrates that there are phases in program power behavior. Chapter 4 describes our Camino compiler, including its structure, supported program control-flow and runtime-event profiling, and various levels of instrumentation function. A Linux device driver developed for event counter profiling and dynamic voltage/frequency scaling is also shown in this chapter. Chapter 5 shows the physical power measurement setup used in the experiments of this thesis. Measurement is performed on a StrongARM SA110-based Skiff board, an Intel

Pentium 4 machine, and an Intel Conroe E6600. We put the measurement description before the details of our phase classification method for better understanding of the experimental results in the following chapters. Chapter 6 analyzes the correlation between power dissipation and runtime event counts, presents a new phase classification method using the combination of control-flow information and runtime event counts, and shows its improvement in power phase classification accuracy. Chapter 7 presents a new interval partitioning method that using infrequently executed basic blocks to demarcate intervals. This method significantly decreases the instrumentation interference due to dynamic interval identification during program execution and makes it possible to validate program power phase behavior through physical power measurement. Experimental result shows that this method has negligible instrumentation overhead with no loss in phase classification accuracy. Chapter 8 presents the current state of our infrastructure. Edge vectors, instead of Basic Block Vectors are used in control-flow-based phase classification and instruction-per-cycle is used as runtime event. Through experiments on a real system and power behavior similarity evaluation, we show that this infrastructure can find representative intervals in terms of time-dependent power behavior. Chapter 9 shows the application of this infrastructure in peak power optimization, dynamic voltage/frequency scaling metric observation, and program power behavior understanding. Chapter 10 concludes this thesis.

Chapter 2

Related Work

2.1 Power Evaluation Techniques

In simulation-based power evaluation methods, the system is abstracted into various components and the energy consumption of a program is estimated as the sum of the energy consumption of all the components during the program execution. Simulators can be classified into different levels based on their levels of abstraction. They target different levels of detail and make different trade-offs between simulation time and accuracy. Most simulators are parameterized so they can be used to estimate the energy consumption systems with different configurations. Simulators are very important in the early stage of architecture design and evaluation. Furthermore, many simulators can give details at a very fine level of semantic granularity.

2.1.1 Transistor-level

These simulators characterize models of transistors and estimate voltage and current behavior over time [53]. Power dissipation of transistors comes from three sources: switching power, short-circuit power, and leakage power. Such simulation is time-consuming but useful in integrated circuit design. Transistor-level simulators are not suitable in evaluating power consumption of large programs on complex systems.

2.1.2 Cycle-accurate Microarchitecture-level

A microarchitecture-level simulator simulates the execution of a program and estimates the energy consumption in each cycle. Cycle-level microarchitectural simulators can provide power behavior changing with the execution of the program. They are suitable for simulations of

modern superscalar processors. Three examples of cycle-level simulator are Wattch [15], SimplePower [46] and Sim-Panalyzer [2]. Sim-Panalyzer is used in this thesis work to estimate the power dissipation of the benchmarks.

2.1.3 Instruction-level

Instruction-level simulators provide coarser power behavior than the above two. The simulation is based on the instruction-level energy profiling of the instruction set of the target processors and the assumption that the energy consumption of an instruction is mostly independent of the addressing mode or operands. Instruction-level simulators are normally faster than cycle-level simulators and useful when only total energy consumption is needed. One instruction-level simulator is JouleTrack [5].

2.1.4 System-level

Hardware component system-level simulators characterize the energy consumption of each system component in different states. The simulator records the transitions between states and the time each component spends in each state during the simulation of the program execution and calculates the energy consumption of the whole program. Such simulators do not provide detailed power behavior of a program, but they are useful for component selection and system partitioning phase. An example is the simulator from Duke University [65]. It is an extension of POSE, a palm OS Simulator.

There are also some software component system-level evaluators. PowerScope [17] is a time-driven statistical sampler that uses samples from a digital multimeter.An energy-driven statistical sampler, energy profiling, from Compaq is similar to PowerScope except that the sampling period is determined by energy quanta.

SoftWatt [63] is the first simulator to target the complete system power profile of high-end system. It extends SimOS with validated analytical energy models for hardware components. This simulator identifies power hotspots in system components, and captures the relative contribution of the power profile to the user and kernel code and identifies power-hungry OS services.

ECOSystem [21] is a modified Linux that manages energy as an OS resource. Parameters

of its "currency model" can be changed to support different platforms.

Isci *et al.* [10] propose a coordinated measurement approach that combines real total power measurement with performance-counter-based per-unit power estimation. This is useful for dynamic power/energy management but it is not suitable for power measurement of small programs. This thesis work is different from this approach because our objective is an evaluation infrastructure for OS/compiler optimizations. Our infrastructure can get the power measurement of any small region of a program as well as the power behavior of a long program. Furthermore, even through this previous work provides power breakdown for CPU components, there is no semantic connection between the measurement result and the measured program, which is important for observing power/energy optimization opportunities and will be an important contribution of our infrastructure.

2.2 Disadvantages of Energy Simulators

The above simulators have some common features. Energy models for various components are characterized before the evaluation and energy consumption evaluations are done through looking up values in many tables by the simulator. The higher the precision is, the larger the tables are. So speed is usually decreased with the increase in precision. Simulator are valuable for power and energy estimation of unavailable architectures. For OS and compiler level power optimization on available architectures, physical measurement can be used for evaluation.

Performance modeling is subject to many sources of error [6]. *Modeling* errors are from the incorrect coding of the desired functionality. Desikan *et al.* measured the experimental error in microprocessor simulation and showed that the error in common simulators is often larger than the performance gains yielded by new architecture ideas reported in the literature [50]. From the construction of power simulators, we can see that power simulators are also subject to errors [6]. Some tables are usually simplified to accelerate simulation. There may be mismatches between reality and the simulation of the program execution. The effect of the OS is not considered in many simulators. All of these issues make accuracy a problem of simulators. Ghiasi *et al.* compared two architectural power models, the Cai-Lim power model and Wattch, and found that these models disagree on the efficacy of the design choices in each experiment and do not

always produce statistically significant results [58].

The disadvantages of power simulators and physical measurement show that we need a faster, more precise power and energy evaluation infrastructure to correctly reflect the power behavior of a program and evaluate the benefit of an optimization. This is the motivation of our research.

2.3 Program Behavior Profiling Tools

There is a long history of program profiling tools, including static instrumentation tools, dynamic instrumentation tools, simulators, and built-in hardware monitors.

Static instrumentation tools modify a program prior to its execution with the purpose of monitoring the behavior of the program during execution. Instrumentation can be done on source code, assembly code, or binary code. ATOM is a commonly used static binary instrumentation tool [61]. An instrumentation file and an analysis file are needed to instrument a program through ATOM. Due to its dependence on the huge gap between data segment and code segment in memory address space, ATOM is not very portable. Another static binary instrumentation tool is FIT [8], which has better portability. As an instrumentation tool, our infrastructure, Camino, instruments assembly code.

Dynamic instrumentation tools insert profiling code in executable image during program execution. They can instrument dynamic generated code, which is impossible for static instrumentation tools. Changes in the profiling method does not require recompiling the instrumented program. Possible disadvantages are imprecise mapping between the profiling result and the instrumented program, and high instrumentation overhead. Pin uses a just-in-time compiler for dynamic instrumentation and does not change code and data addresses when instrumenting a program [52]. It enables users to observe runtime processor state. DynamoRIO is another powerful dynamic code modification infrastructure capable of running existing binaries [7]. Since runtime power behavior is time-dependent and sensitive to instrumentation interference, we do not want to use dynamic instrumentation for power behavior measurement of specific intervals.

Simulators are widely used in research for nonexistent architectures. Simulation is like

dynamic instrumentation, but much slower. It provides precise mapping between the profiling result and the simulated program, but the difference between a real system and the modeled one makes it infeasible in evaluating some low-level optimizations.

Hardware monitor measures resource utilization during program execution. It has separate pieces of equipment that are attached to the system component being monitored. It does not consume system sources and has low overhead. Hardware monitors include probes, performance counters, and logic elements.

2.4 Program Phase Behavior and Phase Classification

Execution of a program tends to fall into repeating behaviors called *phases*. The behavior of a phase can be characterized by simulating or measuring a representative slice of this phase. Various phase classification methods have been proposed to identify phases. Program execution is partitioned into intervals, which are classified into phases. Some of them use control-flow information [56, 3, 55, 32, 37], such as the executed instructions, basic blocks, loops, or functions, as the fingerprint of program execution. This fingerprint depends on the executed source code. Some methods depend on run-time event counters or other metrics [13, 16, 60, 30], such as IPC, power, cache misses rate and branch misprediction, to identify phases. Research shows that phase behavior also exists in Java applications [47].

SimPoint is a tool developed by a lab in UCSD [3, 56]. It partitions a program's execution into intervals, clusters the intervals into phases based on the similarity of their Basic Block Vectors (BBV), and selects a representative interval for each phase, called a simpoint. It is independent of the underlying microarchitecture and provides an idea to estimate whole program metrics from the behavior of the simpoints. Sherwood *et al.* show the use of this idea in estimating instructions per cycle (IPC), branch prediction, instruction cache, data cache, and unified L2 cache miss rates of the SPEC 2000 benchmarks [3]. We show that SimPoint can also be used to estimate the energy consumption of a program through simulation its simpoints [23]. We implement a method in Camino to profile BBVs during program execution, do phase classification, and select the simpoints. Physical CPU power measurement is performed for the

simpoints on a real system. Also, we propose a SimPoint-like method that does phase classification and selects representative intervals in terms of time-dependent power behavior, which results in very low instrumentation overhead in power measurement.

Lau *et al.* compare different architecture-independent structures used for phase classification [38], including basic blocks, loop branches, procedures, opcodes, register usage, and memory address. They used cycle-per-instructions (CPI) as a metric to evaluate their ability to create homogeneous phases and the accuracy of using these structures to pick simpoints. BBVs perform almost the best among the structures in terms of CPI coefficient of variation and calculated CPI error. Our work in this thesis compares BBVs and Edge Vectors (EV) by calculating the error rate in energy consumption estimation. EVs perform better than BBVs. The low coefficient of variation is important in power behavior characterization from the measured result of selected intervals.

Shen *et al.* proposed a data locality phase identification method for run-time data locality phase prediction [55]. They use variable distance sampling, wavelet filtering and optimal phase partitioning in analyzing data accesses to identify locality phases. A basic block that is always executed at the beginning of a phase is identified as the marker block of this phase. This results in variable interval lengths. They use phase hierarchy to identify composite phases. We also use variable interval lengths, but the basic block that marks a phase is not necessary to uniquely mark the phase. It might be the mark for other phases. The infrequent basic blocks are selected first, and the intervals are demarcated by these basic blocks, but limited by a pre-defined length. Phases are identified by the execution times of the infrequent basic blocks that demarcate the intervals, such that we implement precise physical measurement.

A new version of SimPoint supports variable length intervals. Lau *et al.* shows a hierarchy of phase behavior in programs and the feasibility of variable length intervals in program phase classification [37]. They break up variable length intervals based on procedure call and loop boundaries. Experiments in this thesis show that the power behavior of the same procedure or loop varies due to runtime events. We use basic blocks that are infrequently executed to break up intervals and at the same time use a pre-defined length to avoid too long or too short intervals. This satisfies our requirement for low-overhead instrumentation and accurate power behavior measurement. Besides phase classification, we also generate statically instrumented

executables for physical power measurement of simpoints.

Isci *et al.* proposed a coordinated measurement approach to monitor runtime power behavior of a real architecture [11]. They showed that program power behavior also fell into phases. We proposed using SimPoint to find representative program execution slices to simplify power behavior characterization, and we validated the feasibility of SimPoint in power consumption estimation through power simulation of some MediaBench benchmarks [23]. Isci *et al.* compared two techniques of phase characterization for power and demonstrated that the event-counter-based technique offers a lower average power phase classification errors of 1.9% for SPEC benchmarks than the control-flow-based technique, which offers an average classification error of 2.9% for SPEC benchmarks [31]. Our work is different from this one because our objective is to characterize the time-dependent power behavior of programs and map the observed behavior back to source code. We want to measure the fine-grained power behavior of the representative intervals, so the result is very sensitive to instrumentation overhead. The new interval demarcation method and the instrumentation and measurement infrastructure proposed in this thesis causes negligible overhead for identification of an interval during program execution, and the measurement result is very close to the real time-dependent power behavior of the interval.

2.5 Dynamic Voltage and Frequency Scaling

Dynamic Voltage and Frequency Scaling (DVFS) has emerged as an important technique for runtime power optimization and is supported by more and more modern processors. DVFS is applied in different levels that include architectural level, OS level, and static or dynamic compilation level.

In the architectural level [45], hardware monitors certain system statistics, such as Instruction Per Cycle (IPC), cache misses or issue queue occupancy, in fixed time intervals to direct frequency and voltage scaling.

In the operating system (OS) level, interval-based or task-based algorithms use heuristic scheduling to perform dynamic voltage and frequency scaling. Grunwald et al. investigated a number of OS-level clock scaling algorithms through implementation and measurement on

a real system [18, 43, 54]. They concluded that the investigated algorithms failed to achieve their goal of saving power with little impact on program behavior. This is consistent to our observations in the experiments on an Intel Conroe machine.

Both the architectural level and the OS level approaches use heuristics to predict the processor utilization and scale voltage or frequency according to the predicted results. The advantage of these approaches is the global view of resource usage of the whole system. The disadvantage is that they have no detailed information provided by the running applications, thus they cannot explore the characteristics of the running programs to improve the accuracy of processor utilization prediction.

Compilers can analyze program structure and thus are capable to perform DVFS at a finer granularity level. Static compilation level DVFS approaches collect profiles before programs execution and decide when and where to insert instructions to perform voltage scaling [22]. The DVFS decisions are dependent on the input and configuration used for profiling. The generated executables are not portable to other architectures. Dynamic compilation level approaches seem to be more reasonable and more flexible [69, 67, 29]. They consider the runtime information and scale voltage or frequency based on both program structure and runtime system statistics. Wu *et al* implemented DVFS in a dynamic compilation system and performed DVFS based on the profiled relative CPU slack time of procedures and loops [67]. Isci *al* predict the power requirement of each time-based interval of program execution using branch prediction-like method and dynamically change the voltage and frequency of the processor according to the prediction result [29].

A virtual machines (VM) creates a virtualized environment between the computer platform and its operating system. Due to its knowledge of both the running applications and independence of microarchitecture and operating system (OS), VM-level DVFS is also employed, as shown in [19, 24].

In this thesis, our infrastructure is used to find good metrics and thresholds. Some DVFS policies are implemented and measured for evaluation.

Chapter 3

Program Power Phase Behavior

As we mentioned before, sometimes it is necessary to get the power measurement of the whole program in fine precision. But it is hard to get exact power behavior measurement if the execution time is longer than the length of a record. Even though we can run the program many times to measure the power behavior of one small slice in each running and combine the results from the slices to get the final answer, it is time consuming and dealing with the overlap between two slices is not trivial. In order to simplify the measurement work, we use off-line phase classification from SimPoint [57] to find representative intervals to simulate/measure.

Figure 1.3(a) shows the measured CPU current of *256.bzip2* from SPEC CPU 2000. We can see obvious phase behavior in the current curve, which can be roughly classified into 4 phases. In each phase, the program power behavior is consistent. If we can find a representative interval for each phase in terms of power behavior, we can characterize the detailed power behavior of a long-running program through simulating or measuring just small slices of the program execution. In the first step of validating the feasibility of the SimPoint-like idea in power behavior estimation, experiments are performed on a StrongARM SA110 board through simulating 10 MediaBench [39] benchmarks.

3.1 Off-line Phase Clustering Analysis

The selection of simpoints includes the following steps:

1. **Basic Block Profiling** Code profiling is performed on a given program/input pair to get the basic block vector(BBV) for each interval with fixed number of instructions.

2. **BBV Dimension Reduction** The dimension of a BBV from code profiling depends on the source code. It can be as large as several thousands. It is time-consuming to do

comparison (get the distance) between two vectors of such dimension during clustering. Random projection is used to reduce the dimension to 15 to speedup the third step.

3. **Phase Classification Through K-Means Clustering** In order to find the phases, the intervals should be clustered into groups. K-means clustering algorithm is used for classification [57]. This is a recursive process and *k* is a parameter of the algorithm. Sherwood *et al.* tried several k values for one program/behavior to find the best one [57]. K-means clustering algorithm works as following:

   - choose *k* intervals randomly as the initial centers for the *k* clusters
   - for each interval, compare its distance to all the *k* cluster centers(centroid). Put it into the "nearest" cluster
   - recalculate the centroid of each cluster based on its current members
   - repeat the above three steps until the clusters are stable After the process, *k* clusters are formed.

     In our experiments, for each interval size, several *k* values are tried. So we do comparisons in both horizontal and vertical directions to get the best *k*/interval size pair.

4. **Picking Simulation/Measurement Points** After getting the best *k*/interval size pair, we can pick 1 interval from each cluster to simulate or measure. *Perelman et. al* proposed a method to pick early points to save fast-forwarding time [48]. Weighted simulation results of the picked points are summed to generate the final estimated program behavior.

3.2 Feasibility Validation of the SimPoint Idea

In order to do validation of the points selected by the above method, we did some experiment on StrongARM SA110. We chose 10 benchmarks from MediaBench [39] and compiled them into ARM executables. The description of the benchmarks is shown in Table 3.1. The SimPoint release [3] provides an extended sim-fast.c based on the sim-fast.c of SimpleScalar, but sim-fast doesn't support ARM code. We extended sim-outorder of SimpleScalar to do BBV profiling on ARM benchmarks. Also, we extended the sim-panalyzer.c program of sim-panalyzer and

made it record the power dissipation of each interval. The original SimPoint algorithm was also extended to support fixed number of simpoints instead of choose the best one among several numbers, so that we could investigate the impact of the number of simpoints on the error rate of power/energy estimation. Table 3.2 shows the baseline configuration of StrongARM SA110 used in our experiment.

Table 3.1: MediaBench benchmarks used in experiment.

Benchmark	Description
adpcmencode,adpcmdecode	Adaptive differential pulse code modulation.
epic,unepic	Experimental image compression utility.
g721encode,g721decode	Reference implementations of the CCITT (International Telegraph and Telephone Consultative Committee) G.711, G.721 and G.723 voice compressions.
jpegencode,jpegdecode	Standardized compression method for full-color and gray-scale images.
mpeg2encode,mpeg2decode	High-quality digital video transmission.

Then we explore the error rate of each benchmark in a 3-Dimension space, interval size, number of simpoints and error rate.

For each benchmark, the following steps were performed:

1. Compile the benchmark locally on the skiff board with extended options -static and -msoft-float. StrongARM has no Floating-point Unit(FPU). FP instructions are emulated in the kernel with integer instructions. The compile output has library calls for floating point instructions. In order to do future comparison between simulated and measured results, we did not use cross-compiler.

2. Run the modified sim-outorder on the benchmark to get BBVs

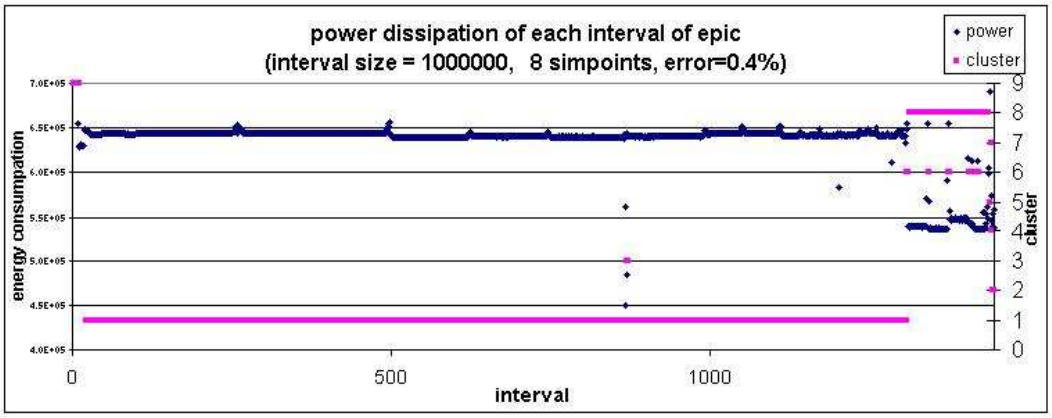
3. Run the BBV analysis program on the BBVs from step 2 to get the simpoints and their corresponding weights. We modified the BBV analysis program to get different number of intervals for each benchmark to see the change of the error rate with the increase in interval number.

4. Run sim-panalyzer to get the power consumption of each selected simpoints.

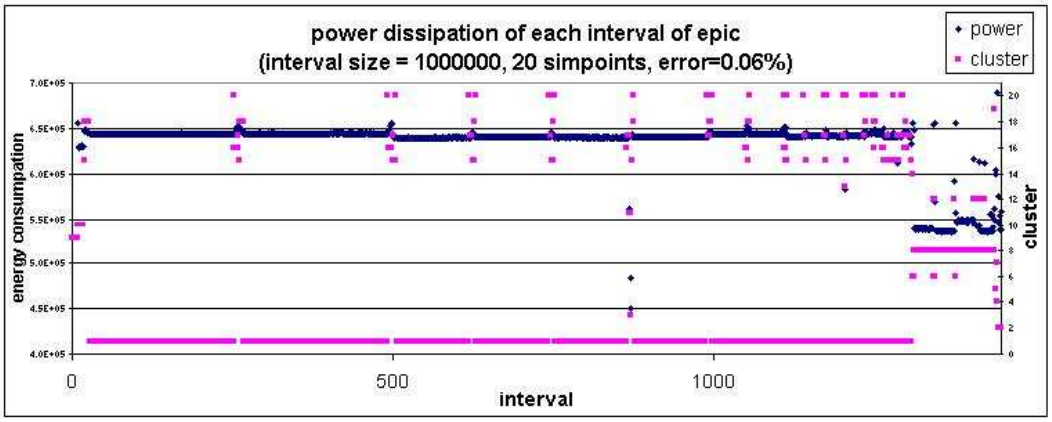
Table 3.2: Baseline configuration of Skiff board

Technology/V_{dd}/Frequency	0.35um/2.0v/233Mhz
Instruction Fetch Queue(IFQ) size	2 instructions
Extra Branch Mis-prediction latency	3
Branch Predictor Type	nottaken
Instruction Decode Bandwidth	1
Instruction Issue Bandwidth	1
Run Pipeline with In-order Issue	true
Instruction Commit Bandwidth	1
L1 Data Cache	16KB, 32-way(RR), 32B blocks, 1 cycle latency
L1 Instruction Cache	16KB, 32-way(RR), 32B blocks, 1 cycle latency
L2 Cache	none
Register Update Unit (RUU) size	16
Load/Store Queue (LSQ) Size	8
Flush Caches on System Calls	false
Memory	16MB
Memory Access Latency ($\langle$ first_chunk $\rangle$ $\langle$ inter_chunk $\rangle$)	12 4
Memory Access Bus Width	4B
Memory Accesses	fully pipelined
Instruction TLB	4KB, Fully-associative(RR), 32B blocks
Data TLB	4KB, Fully-associative(RR), 32B blocks
Instruction/Data TLB Miss Latency	10 cycles
Total Number of Integer ALU's	1
Total Number of Integer Multiplier/Dividers	1
Total Number of Memory System Ports(to CPU)	2
Total Number of FP ALU's	0
Total Number of FP Multiplier/Dividers	0

5. Calculate whole-program power estimation based on the power values from step4 and the weights from step 2



(a) 8 clusters are formed.



(b) 20 clusters are formed.

Figure 3.1: Energy consumption of each interval and phase clustering of *epic*. Interval size=1million instruction.

Figure 3.1 shows the energy consumption of each interval and the clusters obtained from the off-line phase clustering. From Figure 3.1(a), we can see that intervals with similar energy consumption are clustered into the same cluster. In Figure 3.1(b), 20 simpoints are used and we got better clustering result than when 8 simpoints are used.

Figure 3.2 shows the ipc and power error rates of the simulated benchmarks and the average error rate. For each benchmark, the error rates of ipc and power are similar and the average error rate is below 2%.

Figure 3.1 and Figure 3.2 show the simulation-validated feasibility of the SimPoint idea

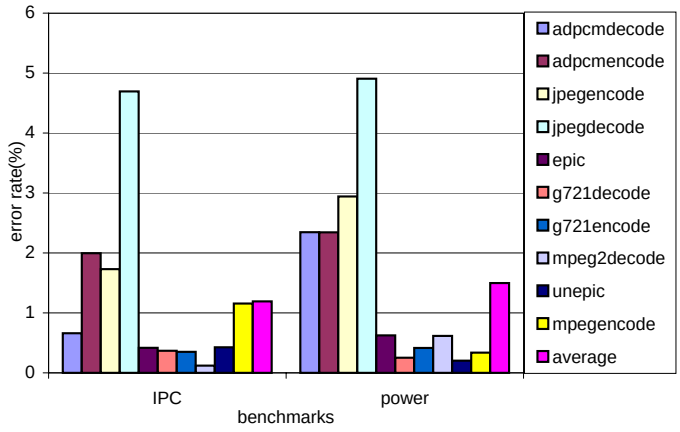


Figure 3.2: Error rates of ipc and power for each benchmark and their average error rates.

in total energy consumption evaluation. This is the first step of our infrastructure for efficient program power behavior characterization.

Chapter 4

The Camino Compiler Infrastructure

This chapter introduces our program analysis and instrumentation tool, the Camino Compiler Infrastructure [25]. The goal of Camino is to serve as a testbed for various low-level optimizations. It is currently used to study performance optimizations as well as power and energy optimizations. Camino supports the x86 instruction set. Support for ARM is in our ongoing work. *Camino* is the Spanish word for "path," representing our lab's focus on control-flow-oriented optimizations.

Camino can be used as a static instrumentation tool for profiling. It parses the assembly code generated by GCC and distinguishes data and code, thus bypassing one of the serious disadvantages of static binary instrumentation [41]. Camino implements several types of profiling, including basic block counts, edge profiling, and interprocedural path profiling. It also includes a special technique that allows using a SimPoint-like methodology to efficiently characterize the fine-grained power behavior of an application with very low overhead by sampling specially chosen intervals [3].

Camino also supports a growing set of code placement optimizations such as branch alignment [9] and pattern history table partitioning [34]. Branch alignment improves instruction fetch bandwidth by reordering code such that most conditional branches to be not taken and thus do not incur a discontinuous fetch penalty. Pattern history table partitioning improves branch prediction accuracy through a feedback-directed placement of conditional branches that reduces the likelihood that they will interfere destructively with one another in branch prediction tables.

A two-level profiling is implemented in Camino to support semantic connection between the measured detailed power behavior of selected intervals and program source code.

The Camino infrastructure is currently implemented as a post-processor to GCC. It can theoretically handle programs in any language that can be compiled by GCC. Currently it supports compiling C, C++, and FORTRAN 77.

4.1 Camino Overview

This section describes the basic organization and operation of the Camino Compiler Infrastructure. Figure 4.1 explains the compilation of a program using Camino.

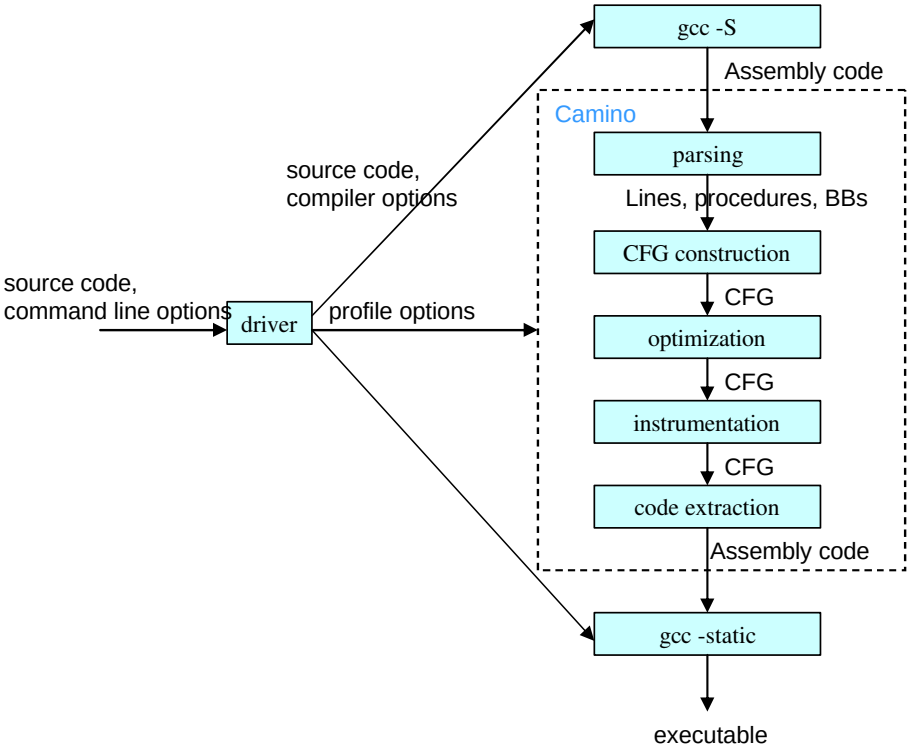


Figure 4.1: Compilation using Camino.

4.1.1 Using Camino

Camino provides three drivers, *ccc*, *ccpp*, and *cf77*, for the compilation of programs in C, C++, and FORTRAN 77, respectively. The driver wraps GCC compilation and the post-processing provided by Camino. A user can invoke the driver with arguments at the command line to start compiling a program. The driver invokes the corresponding front-end from GCC with the `-S`

option to generate assembly language output from the source file. It then invokes the Camino post-processor, described later, on the assembly language file. The driver parses the command line arguments, passing Camino-specific arguments to Camino and other options to the GCC front-end program. These specific arguments control what kind of profiling will be inserted, as well as which optimizations will be applied to the assembly code.

4.1.2 Internal Representation

The Camino post-processor is a C++ program that reads the assembly code generated by GCC and the arguments passed by the driver. It first parses the assembly code into three basic abstractions: procedures, basic blocks, and lines.

Functions implemented in an assembly program are parsed into a Standard Template Library (STL) list of procedures. For each procedure, the control-flow is analyzed and a control-flow graph (CFG) is maintained with STL lists of basic blocks with pointers. There are two distinguished basic blocks: *entry* and *exit* nodes. *entry* is the first executable basic block in the procedure.

The basic block is probably the most important abstraction in the compiler. The intraprocedural structure of the code is completely represented within basic blocks. Each CFG node is the data structure of a basic block. It has a lists of lines and a number of pointers to predecessor and successor nodes, the targets of a conditional branch at the end of the basic block, basic blocks that this block dominates and postdominates, and the list of loops in which this basic block appears. Also, each basic block has an edge profile and a list of path profiles that may be read from a path profile file generated by certain profiling.

Camino distinguishes data and code from directives in the assembly output. Non-text items such as string constants and other data are kept in special basic blocks that are included in the nearest procedure but are not part of any CFG. Each executable basic block is assigned a hash value calculated based on the name of the program, the name of the procedure this basic block belongs to, and its sequence number in this procedure. This value is used by the profiling instrumentation for reference. This value may be considered unique for most purposes. Although collisions are possible, our tests show that they occur very infrequently with no impact on the quality of profiling.

Lines in a basic block are represented with a list of *line* objects. Each line consists of an optional label, an optional x86 opcode or assembler pseudo-op, and an optional set of operands. Camino also has the ability to determine the byte offset of any given instruction in the final executable modulo a moderate power of two. This capability is useful in certain code placement optimizations [34] where knowing the lower bits of the address of an instruction is important in predicting how the microarchitecture will treat this instruction. We currently use this information for a branch prediction optimization, but it could also be useful for instruction cache optimizations.

In this internal representation, instrumentation and/or optimizations may be performed on various level, from procedure to instruction. Since each basic block has a "unique" reference value and the lines of a basic block are stored, a user can instrument or optimize only the basic blocks that satisfy some special condition, instead of all basic blocks. This is also true for instructions. Instrumentation using Camino is very simple. Only two routines are required, an instrumentation routine and an analysis routine. The instrumentation routine inserts a call to the analysis routine at proper positions in each basic block. The analysis routine is normally implemented as a library function linked to the instrumented program at the last step of the compilation. This sort of instrumentation is used to implement various types of profiling as well as triggering power and energy measurement by an external device.

4.1.3 Output

Once Camino is finished with its transformations, it extracts the modified assembly code from the internal representation and overwrites the original assembly file. Then the driver calls the appropriate GCC component to complete the process of assembling the compilation unit and possibly linking the program.

4.2 Program Profiling Supported in Camino

Camino currently implements four types of profiling: 1) basic block and edge counts, 2) interprocedural path profiling, and 3) profiling in support of obtaining basic block vectors and

edge vectors for SimPoint-like clustering, 4) two-level profiling for procedure/loop power behavior characterization. Because of its clear internal representation on multiple levels, it is easy for a user to insert instrumentation at proper positions, or just change the analysis routine, to implement a new type of profiling.

4.2.1 Basic Block and Edge Counts

This type of profiling combines basic block counts and edge counts. For each basic block not containing a conditional branch, a record is kept for the number of times it is encountered. For basic blocks ending in a conditional branch, a basic block count is kept as well as a count of the number of times the branch was taken. When this information is read back into the compiler later to recompile the program with the guide of the profiling result, it is converted into counts of the number of times CFG edges were traversed through conditional branches. The analysis code also has the ability to simulate a simple branch predictor and record the number of times the branch was correctly predicted; however, this option is usually turned off for efficiency.

4.2.2 Interprocedural Path Profi ling

Many branch predictors use history tables. Camino implements a special form of path profiling whose goal is to determine the path corresponding to the global history used by certain types of branch predictors.

Interprocedural path profiling is implemented through the instrumentation of branch instructions. The analysis routine invoked during program execution maintains a record for a global path of a given fixed length. The frequency with which this history is encountered, the frequency with which this path is taken, and a sequence of branches identifiers along this path are recorded. The taken and not taken information of a branch is stored in the profile data structure of the basic block corresponding to this branch. Each time a branch is executed, the analysis routine updates this information and determines if this branch should be shifted into the global path record. If the frequency of a global path is higher than a given threshold, the profile of this path is temporarily stored in some table. At the end of program execution, all of the recorded path profiles are written to an output file. Camino provides a method for reading in the output file for use in path-based optimizations.

4.2.3 Basic Block Vector and Edge Vector Profiling

As described in Section 4.1.2, instrumentation at the basic block (BB) level using Camino is very simple. We instrument each basic block before its first instruction for BBV profiling. During program execution, the analysis routine is invoked for each BB, computing the execution frequency of the BB in the current interval. The hash value of a BB is used as its identification in BBVs. The BBV for each interval is output into a file.

Edge Vector profiling is similar to BBV profiling except that the taken frequency of each edge is recorded. Each BB is still instrumented. The analysis routine identifies each edge based on the previous BB and the current BB. In our implementation, we only record the edges comming out of conditional branches, which cuts the vector size by almost a half compared to when all edges are recorded. EVs are also written to a file for future analysis.

4.2.4 Event Counter Profiling

We developed a Device Driver as a Linux kernel module (LKM) to provide interfaces for model-specific register (MSR) access and control. In Camino, routines are implemented to open the device and read or write performance counters. This device driver supports Linux 2.6.9 on Intel Pentium 4 and AMD Athlon 64, and Linux 2.6.18 on an Intel Conroe E6600.

The Pentium 4 supports 48 event detectors and 18 event counters [59]. Up to 18 performance count events can be concurrently collected. The event detectors form 4 groups, each of which consists of event detectors and a block of counters. Each event detector contains an event select control register (ESCR). Each counter contains a counter configuration control register (CCCR). An ESCR selectes the desired event. It can qualify event detection by privilege mode and thread ID. A CCCR choose the event detector output that the counter should use. It configures the selected event detector to support threshold comparison, edge detection, thread mode qualification, or performance monitor interrupts generation on counter overflow.

AMD Athlon 64 provides 4 48-bit performance counters [4]. Each counter monitors a different event. Each counter has a corresponding Performance Event-Select register (PerfEvtSel*i*), which specifies the event counted by this counter and controls other aspects of its operation. Up to 4 events can be concurrently monitored.

Intel Conroe E6600 is based on Intel Core microarchitecture [27]. It is much simpler performance counter control compared to Pentium 4. It has 3 fixed function performance counters, each of which is dedicated to count a pre-defined performance mornitoring event. A control MSR enables that fixed function performance counters to use. Two other counters are controlled by three MSRs. Event selection for each counter is performed using one of the two IA32_PERFEVTSELx MSRS.

In addition to hardware event monitoring, our device driver also supports changing the value of some MSRs. Voltage and frequency scaling is implemented through writing the target voltage/frequecy configuration to a performance control MSR.

4.2.5 Two-level Profiling

We profile the Interval Vector (IV) for each invocation of procedures and loops when BBVs or EVs are profiled. After phase classification, each interval has a phase number. Each interval number in IV is replaced with its phase number. Now an IV is like a BBV or EV, but each element is a phase number. Through classifying the IVs for a procedure or loop, we can find its representative invocation and the variance in different invocations. Based on the measured power behavior of representative intervals, the detailed time-dependent power behavior of a procedure or loop can be characterized. In our current implementation, only natural loops are considered. A natural loop is defined by a back-edge. Back-edge (*n*,*d*) is a control-flow edge from node *n* to node *d* such that *d* dominates *n*. A natural loop of back-edge (*n*,*d*) has only one loop header, *d*. The set of nodes of this loop are dominated by *d* and there is a path from any node of this set to *n* that does not contain *d*. A node can the header of multiple natural loops. An example of natural loops is shown in Figure 4.2.

To profile IVs of procedures, we maintain a call stack during program execution. At the entrance of a procedure, the hash value of procedure, the current interval number and the number of executed instructions in the current interval are pushed onto the stack as a record. At the exit of the procedure, its record is popped off the stack and the number of executed instructions and interval information for this invocation are put into a linked list for this procedure. We set thresholds to avoid profiling too small procedures and loops. For example, a size threshold of an interval size is used and a frequency threshold of 100 is used. After a procedure or loop

is invoked 100 times, we calculate its average size, that is, the average number of executed instructions. If its smaller than the predefined threshold, we mark the procedure or loop as "not profiled" and its invocations are no longer recorded. The stacks are checked at the end of program execution. If the loop stack is not empty, or the procedure stack has more than 1 element, the profiling result is incorrect.

Sometimes a procedure call is compiled into a "jump" to the callee, in which case the exit of the caller can not be detected. In the instrumentation, we also pass the procedure call type to the analysis routine, e.g. whether this call is a "jump" to the callee. When a record is popped off the procedure call stack and this procedure is called through a "jump", the next record on the stack is also popped off the stack. This operation is repeated until a popped procedure is not called through "jump". This solutions enables us to accurately keep track of procedure calls.

Loop IV profiling is similar to procedure IV profiling, but the information for each invocation of a loop, instead of each iteration of a loop is recorded. Loops are identified during CFG construction. The unique hash value of the first BB of a loop is used as the identification of this loop. In addition, the sequence number of a loop in the source code of a procedure is also recorded for easy mapping between the profiling result and the corresponding source code. The format is *procname_no*. The exit edges of each loop are written to a file for future use in profiling. At the entrance of a loop, if the top of the call stack is this loop, the analysis routine does nothing; otherwise, the identification of the loop, the current interval number and the number of executed instructions in the current interval are pushed onto the stack as a record. When an exit edge of a loop is taken, its record is popped off the stack and the number of executed instructions and interval information for this invocation are put into a linked list for this loop. In our current implementation, all of the natural loops that have the same loop head are treated as the same loop during profiling. In Figure 4.2, the loop with backedge DA and the one with backedge EA are treated as the same loop, but the one with backedge CB is treated as a different loop.

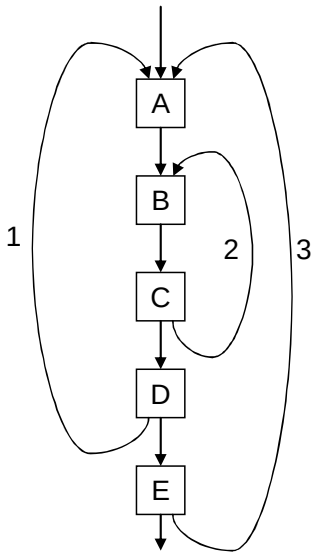


Figure 4.2: Natural loops identified by Camino.

Chapter 5

Power Measurement Infrastructure

For a physical power measurement-based infrastructure, the measurement result should be precisely the power curve of the measured program region. The measurement method should be able to handle program regions with any execution time. In our infrastructure, we use a TDS3014 oscilloscope to measure the current and voltage of a system component. Measurement experiments performed on a StrongARM SA110 based Skiff board show how fine-granulated power measurement helps in evaluating optimizations. All of experiments performed on a Pentium 4 and a Conroe E6600 validate the accuracy of our power phase classification on real systems and show the uses of our infrastructure in program power behavior characterization, power optimization evaluation and DVFS metric selection.

Most current computer architectures have cycle count registers and clocks that can be accessed by users or system programs. Physical execution time measurements are therefore easy to obtain, for instance through the UNIX *time* command. This is not true for power and energy measurements. In order to measure the power dissipation of a system component such as the CPU, the supply current and voltage for the component needs to be measured. This requires a printed circuit board (PCB) design that has separate power planes for each component, and each such power plane has an access point that allows a voltage and current measurement probe. When separate power planes are not available, power and energy measurements can only be done for the entire system. In this case, micro-benchmarking will be used that stress individual system components while keeping the activity levels in other system components the same. The variations in overall power dissipation can then be attributed to the single component.

5.1 Usage and Measured Parameters of Oscilloscope

The TDS3014 oscilloscope has four channels. The longest record length for each channel is 10,000 samples. Users can program to control its acquisition mode, record length, trigger mode, data encoding and other configurations. Data collection can be done on one channel or several channels in turn.

Record length is the number of points that comprise a complete waveform record. Record length determines the amount of data that can be captured with each channel. TDS3014 has two record length options: 500 samples and 10k samples. Since an oscilloscope can store only a limited number of samples, the waveform duration (time) is inversely proportional to the oscilloscope's sample rate.

TDS3014 has two sample acquisition modes. We use different record length options in the two modes due to the limit on the communication between the oscilloscope and the data acquisition machine.

- Normal : record length=10k points, rate is up to 450 waveforms/s
- Fast trigger: 500 points, rate is up to 30000 waveforms/s

In our first step, we used fast trigger mode to collect the power behavior of the whole measured program. But there are two problems:

- The oscilloscope keeps acquiring samples all the time, so it is hard to know the beginning and the end of the measured program.
- The communication cost to collect 500 samples from the oscilloscope to the data-acquisition machine is much longer than the time used to generate these samples when high resolution is used.

In our experiments, the communication cost is usually about 135ms, whereas the sample generation time depends on sampling rate. Each time the data acquisition machine wants to gather samples from the oscilloscope, a session should be set up between the two. If we adjust the oscilloscope to generate 500 samples in 135ms, the resolution is 270us/sample. For a 233Mhz machine, each sample covers about 62910 cycles. Obviously, this resolution is not

enough if we want to have a close look at the power behavior of the measured program. If we use higher resolution, some samples are lost because of the overwriting mechanism of the sample buffers.

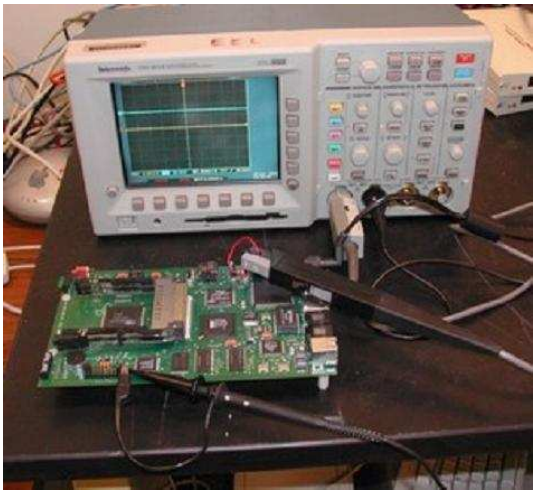
To solve the first problem, we used the normal acquisition mode by using the trigger module of the oscilloscope. In this mode, the oscilloscope starts sample acquisition only after a trigger event happens. The acquisition is stopped after a whole record is obtained. Since TDS3014 has no external trigger input, we used a dedicated channel to monitor the trigger event. Trigger of starting measurement is generated by generating a signal, for example, setting some voltage to a high value. At the end of the measurement, another signal is generated. Although normal trigger mode can help us identify the beginning and the end of the measured program, it is still hard to get the power behavior in high precision for programs with long execution time due to the high communication cost. Larger memory for oscilloscope can help, but the problem is still there if we want higher precision. This is why we plan to use the SimPoint-like idea to find representative slices of a program and get power evaluation for the whole program based on the physical measurement of the selected slices.

In our infrastructure, users can specify which region of the program to measure by adding some comments before and after the region. trigger generation code is inserted into the source code through MACRO extension and the generated executables will trigger the oscilloscope at exactly the beginning of the measured region. By setting the trigger pin to low voltage at the end of the region, we can get the precise mapping between the measurement result and the source code. Since there is no interrupt during the execution of the measured region, we can achieve *non-intrusive* power/energy measurement.

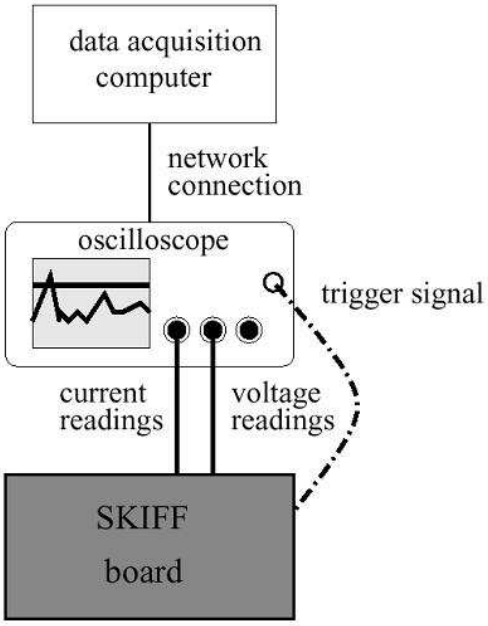
5.2 Fine-granulated Power Measurement on a StrongARM Board

5.2.1 Measurement Setup

As depicted in Figure 5.1 (a), our evaluation infrastructure for ARM has three components: a Skiff board(a Compaq Personal Server PCB Board with a StrongARM SA110 CPU and a 32MB SDRAM) [28], a Tektronix TDS3014 DPO oscilloscope with TDS3TRG advanced trigger module and a data-acquisition machine, which is not shown in the picture. The Skiff



(a) measurement infrastructure



(b) communication among the three measurement components

Figure 5.1: Prototype power measurement infrastructure for the StrongARM based Skiff board.

board has separate power planes and current measurement points for CPU and memory. Trigger signals are generated by setting a pin on the skiff board to a high or low voltage. We call this pin “trigger pin”. Through running some specially designed benchmarks, we got the delays of setting the high voltage and low voltage are **20ns** and **47ns**, respectively. The data-acquisition machine is an Intel P4 2.8GHz Linux machine.

Figure 5.1(b) depicts the communication among the three components. The measured program runs on the skiff board. Physical measurement is performed by the oscilloscope, which measures the current or voltage of the components(CPU, memory) of the skiff board or the whole board. A data-collecting machine communicates with the oscilloscope to gather data and does offline analysis. Sampling is done by the oscilloscope and the data collecting machine only communicates with the oscilloscope, so there is no interference between the measured program, sampling and data collecting, correctness of the result is improved.

5.2.2 Effect of Loop Unrolling and Instruction Scheduling

Using this measurement setup, we can have a close look at the power behavior of a program. Figure 5.2 shows a very simple program with a loop. The part in the gray lox is the measured part. After the loop was unrolled eight times, we got a new version, **version A**, which generates a basic block with 16 loads followed by 16 additions. By hand, the instructions of the assembly of **version A** were reordered to get two other versions, **version B** and **version C**Table 5.1 shows the instruction order for each version.

Table 5.1: Instruction order of each version of the loop.

Version	Instruction Order
Original	2 loads followed by 2 additions in each loop, but 8 times of loops compared to other versions
Version A	16 loads followed by 16 additions
Version B	2 loads, followed by 2 additions, followed by 14 loads, followed by 14 additions
Version C	alternating groups of 2 loads, followed by 2 additions

Figure 5.3 shows the measured CPU current for the StrongARM SA110 processor (2.0V, 233MHz). The line marked “trigger” represents the trigger signal for the oscilloscope. At

the beginning and end of the program region of interest, the trigger pin is set to high and low voltage, respectively.

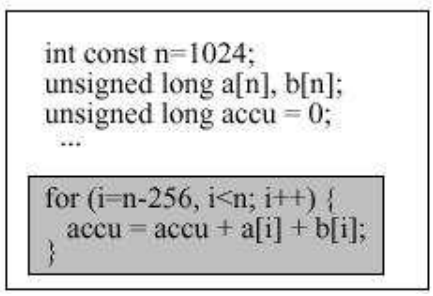


Figure 5.2: A simple program with loop.

In Figure 5.3 shows, we see what we expect for the current behavior of the program, one cache miss every 8 iterations of the original version. **Version C**, the alternating schedule of memory and CPU instructions, leads to the shortest execution time, the lowest energy consumption and the smoothest power dissipation profile. Choosing this schedule over the alternatives will lead to a fast program with low peak power dissipation and small variations in power dissipation.

As comparison, we also simulated the same program using Sim-Panalyer,a cycle-accurate architecture-level ARM power simulator. Since it is hard to identify the loop exactly in simulation, we simulated the power dissipation of the whole program. Only the loop is different in different simulations, so we can say that the difference among simulation results are from our modification to the loop. Table 3.2 gives the configuration of the StrongArm SA110 for our simulation experiments. Most of the architectural configuration values are from the StrongARM v4 data-sheet. Others are from the default configuration provided by Sim-Panalyzer. We used the power configuration file provided by Sim-Panalyzer but changed the frequencies to 233Mhz.

Table 5.2 gives the simulation results for the four versions of the program. Both the power dissipation and simulated cycles are normalized by the results of the original version.

From Table 5.2, we can see that **version A, B** and **C** all brought better results compared to the **original version**, no matter in power dissipation or execution cycles. But based on

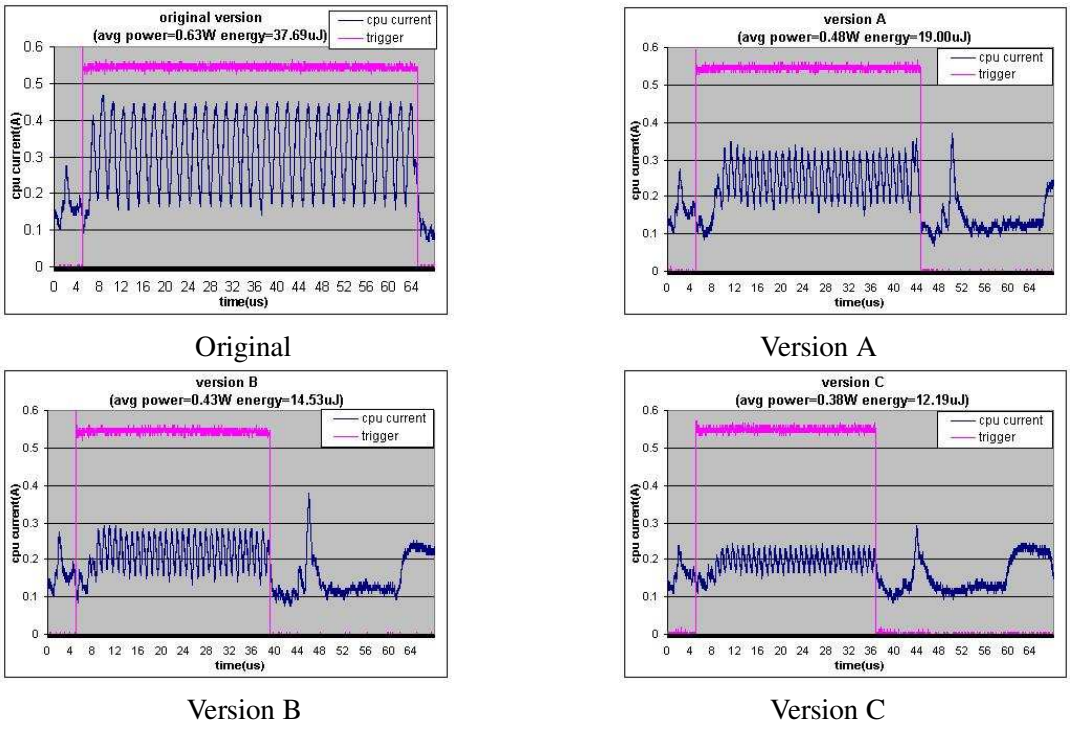


Figure 5.3: Physical measurement results for the four versions in Table 5.1.

Table 5.2: Simulated power and cycles for the unrolled loop in Figure 5.2.

Version	power/cycles
Original	1.000/1.000
Version A	0.653/0.644
Version B	0.668/0.684
Version C	0.667/0.683

the simulation result, **Version A** is the best of the four versions. This is different from the observation we got from the measurement result. Furthermore, even through we simulated the whole program, the power consumption of the loop is a big part of that of the whole program based on the comparison of the simulation results of the original version and **version A**. But we can not see significant difference between the simulation results of **version B** and **C**, which also disagree with the physical measurement result. We can not always trust the simulation result.

5.3 CPU Power Measurement on Pentium 4 and Conroe

We validated our infrastructure on a Pentium 4 machine through measuring the current of the CPU package. This machine runs Linux 2.6.9, GCC 3.4.2 and GCC 2.95.4. Pentium 4 has a separate power cable for the CPU, and its voltage is 12V. We measure the current on this cable using a Tektronix TCP202 DC current probe, which is connected to a Tektronix TDS3014 oscilloscope. The experiment setup is shown in Figure 5.4. The data acquisition machine is a Pentium 4 Linux machine that reads data from the oscilloscope when benchmark execution time is larger than the window size of the oscilloscope and the measurement for the whole benchmark execution is needed. Simultaneous benchmark execution and power data acquisition on different machines eliminates the interference to the measured benchmark. The picture on the right of Figure 5.4 is our experimental setup, data acquisition machine is not shown in the picture.

5.3.1 Precise Power Measurement

The oscilloscope has a TDS3TRG advanced trigger module. When it is in trigger mode, it accepts trigger signals from one of its four channels. We use its edge trigger. It starts measurement only after the voltage or current on the trigger channel increases to some predefined threshold and stops when its window fills to its capacity of 10,000 data points. The data points stay in the buffer until the next trigger signal comes. We generate the trigger signal through controlling the *numlock* LED on the keyboard. A voltage probe is connected to the circuit of the keyboard to measure the voltage on the led, as shown in Figure 5.4. The voltage difference

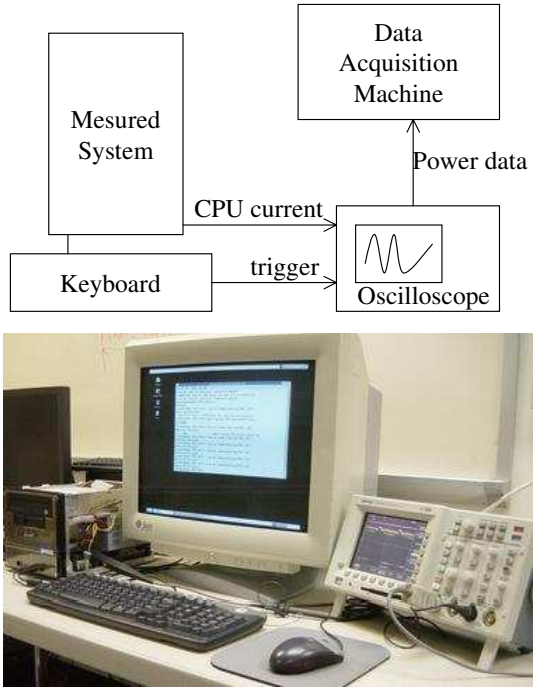


Figure 5.4: The physical measurement infrastructure used in the experiments.

between when the light is on and off is more than 3.0V, which is big enough to trigger the oscilloscope. The delay to set the trigger signal is small and does not affect our measurement result.

The voltage on the trigger channel is set to high to trigger the oscilloscope at the beginning of the program execution slice to measure. This voltage is consistently high until when it is set to low at the end of this slice. Figure 5.5 shows the measurement result using trigger signals. It is easy to identify the power behavior of the measured slice in the measurement result.

Data acquisition for a short program execution slice is easy since we can read the power samples after the execution of this slice without loss of sample. When the oscilloscope is in its trigger mode, the samples are stored in the oscilloscope until the next trigger signal is sent to the oscilloscope.

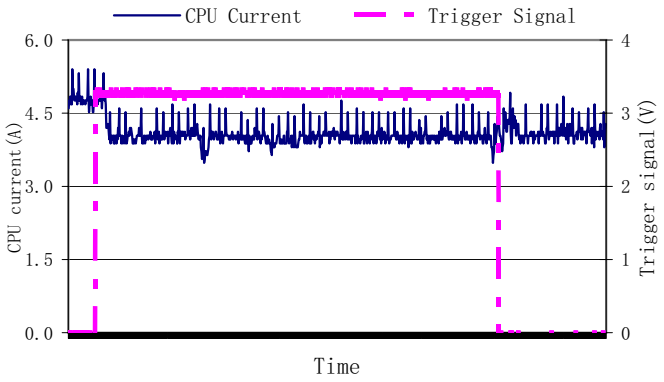


Figure 5.5: The display window of the oscilloscope after the execution of a simpoint. The dotted line is the trigger signal. The power curve for the measured simpoint is surrounded by the trigger signal.

5.3.2 Measuring Whole Program Energy Consumption

The execution time of a benchmark is often much longer than the maximum measurement record size in trigger mode of the oscilloscope, which is 100 seconds. We cannot cover the power curve of the benchmark using trigger mode, so we use its auto mode to measure the power behavior of the whole benchmark execution and still identify the exact power data points for the benchmark by setting the voltage on the trigger channel to high and low before and after the execution of each benchmark. But no instrumentation is needed to generate signals during program execution. In auto mode, the oscilloscope records power data points continuously, the data acquisition program is adjusted to read the data in each window without losing data points or reading duplicated data points, due to too long or too short data reading period respectively. This is validated through the comparison of the real benchmark execution time and the one obtained from the measurement result.

Power measurement on the Conroe machine is similar to that on the Pentium 4 machine. The power cable to the CPU package is measured and the keyboard is modified to generate trigger signals.

Chapter 6

Power Phase Classification Using Combination of Control-flow and Event Count

Experiments in Chapter 3 show that phase classification based on BBVs results in low error rate in energy consumption estimation. However, two intervals that execute the same basic blocks may generate different time-dependent power behavior due to run-time events, such as cache misses and branch mispredictions. Phase classification only based on control flow information can give us low error rate in estimating average metrics, but cannot precisely differentiate these intervals, so the resulting simpoints may not really be representative in terms of power behavior. Figure 1.2 in Chapter 1 shows the power curve of two intervals from the same phase classified based on BBV. In order to accurately characterize the time-dependent power behavior of a long-running program through simulating/measuring the selected simpoints, our phase classification should be able to differentiate intervals as shown in Figure 1.2 and clssify them into different phases. Through investigating the correlation between IPC and power dissipation, we propose a two-stage phase clssification method, which uses IPC to refine the phase classification result based on control-flow information and validate this method on StrongARM SA110 by simulation.

6.1 Correlation between IPC and Power Dissipation

From the results of profiling both IPC and power dissipation for each interval, we find a direct correlation between these two. [12], [66] and [40] also mentioned similar correlation between IPC and power. Power dissipation in each cycle depends on the work done in that cycle. Dynamic power behavior of an interval depends on execution cycles and power per cycle. Both are proportional to IPC. If two intervals with similar executed basic blocks have different IPC, we can say that they have different time-dependent power behavior since both power per cycle

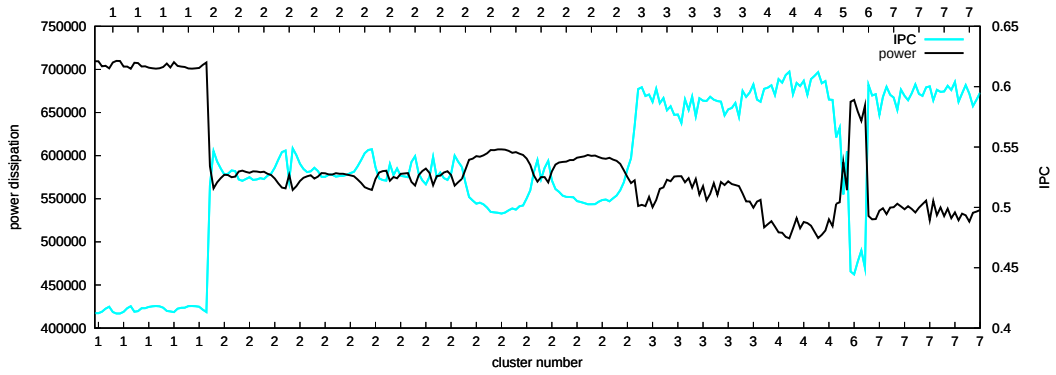


Figure 6.1: Correlation between power and IPC for intervals of *jpegencode*.

and total execution cycles are different. Figure 6.1 shows this correlation for *jpegencode*. Cluster numbers are the result of SimPoint phase classification. Not all of the cluster numbers are shown here due to the limited space on *x* axes. They are shown every 7 clusters from the first one on the primary *x* axes, and from the fifth cluster on the secondary *x* axes.

There are totally 247 intervals, each has 1 million instructions. The *x* axis is the cluster that each interval belongs to. The primary *y* axis on left is the power dissipation of each interval. The secondary *y* axis on right is the profiled IPC for each interval. Intervals are shown in the order of their cluster number and their appearance in program execution, so that we can see the difference of power dissipation and IPC among all the intervals from the same cluster generated by SimPoint. Within each cluster, the correlation between power and IPC is obvious. For instance, in cluster 2, the same basic blocks are executed, high IPC means low power dissipation and vice versa. Similar correlation exists in the other benchmarks in our experiments. If we can classify the intervals from the same BBV-based cluster based on IPC, intervals from a new cluster will have higher similarity.

6.2 A two-stage Program Power Phase Classification

Based on the obaervation in 6.1, a new phase classification method, which combines BBV and IPC to classify intervals, can be used to get the representative intervals for power behavior characterization. By this method, intervals with similar BBVs but different IPCs are classified into different phases. The difference level between IPCs is adjustable to balance the accuracy

and time cost.

An objective of this infrastructure is to find representative slices for the power behavior of a program. Figure 6.2 shows the process used to characterize program dynamic power behavior. The simulator used in step 1 is *sim-outorder* for ARM from SimpleScalar V4.0. The power simulator used in step 3 is *sim-panalyzer* [2]. Sim-outorder is modified to apply BBV and IPC profiling. Sim-panalyzer is modified to get the power dissipation of the specified piece of code and record power values based on the given granularity.

The input to this process includes the ARM binary code of the program compiled with the *-static* option, a script file showing the interval size and how to run the program, and the granularity for recording the power values during simulation. Profiling of basic block vectors and IPC is performed first to get the basic block vector and IPC value for each interval. The the intervals are classified into different clusters and then a representative interval, called*ppoint*. is selected from each cluster and simulated for detailed power information. Finally, the power behavior of each ppoint is expressed graphically. The output of the tool includes the *ppoints*, the power dissipation and number of occurrence for each *ppoint*, total power dissipation estimation, sequence of ppoints in the occurrence order of their corresponding cluster, and the graph for each ppoint. From the output of this tool, users can easily derive the overall power behavior of the program, without high cost in time and space.

6.2.1 Using IPC to Refine Control-folw-based Phase Classification

Figure 6.2 illustrates the two-stage classification used to get the ppoints. Phase classification 1 and 2 are independent. Phase classification 1 generates clusters based on the similarity between basic block vectors. The intervals clustered into the same phase execute similar binary code. Phase classification 2 generates clusters based on similarity between IPCs. After the two-stage classification, intervals from the same phase have similar IPC values. Intervals from different phases must have different power behavior whether they execute the similar basic blocks or not, since they have either different basic block vectors or different IPC.

The assumption of this power phase classification is that, if different executions of the same basic blocks have different power dissipation, they have different power behavior. Our tool combines the clusters from these two phase classification methods to do clustering. The

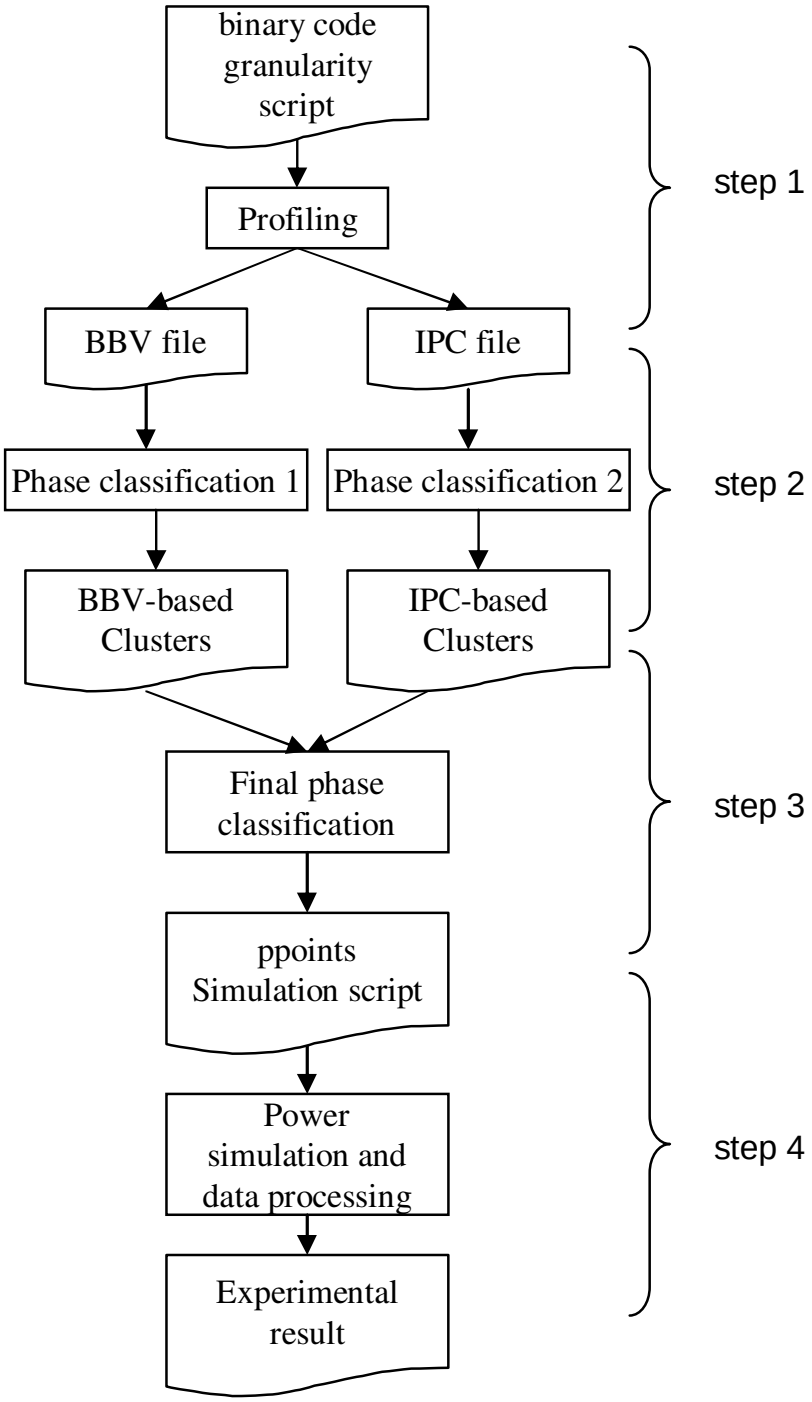


Figure 6.2: Dynamic power behavior characterization process

correlation between IPC and power dissipation ensures that the intervals of the same cluster of the BBV-based classification, but with different power consumption, are partitioned into different clusters in step 3 of Figure 6.2.

In the current implementation of our tool, the number of phases identified in classification 1 is no more than 10. It is exactly the same phase classification as in SimPoint. The best number between 1 and 10 is selected during the classification procedure. The number of phases identified in classification 2 is 10.

In step 3 of Figure 6.2, the final clusters and *ppoints* are generated. Suppose the number of clusters from classification 1 and classification 2 are N_{bbv} and N_{ipc} respectively. The initial number of final clusters is $N_{bbv} \times N_{ipc}$. Let C_{bbv} be the cluster ID of an interval in classification 1 and C_{ipc} be the one in classification 2. The cluster of an interval C is determined by:

$$C = (C_{bbv} - 1) \times N_{ipc} + (C_{ipc} - 1)) + 1 \tag{6.1}$$

where C_{bbv} ranges from 1 to N_{bbv} and C_{ipc} ranges from 1 to N_{ipc}. It is unlikely that the IPC values of every cluster in classification 1 fall into N_{ipc} clusters in classification 2, so some clusters are empty. These clusters are pruned and cluster IDs are adjusted to get the final classification. Later experimental results show that the real number of final clusters is much smaller than $N_{bbv} \times N_{ipc}$ and 10 is too large in classification 2 for some benchmarks.

6.2.2 Controlling Unnecessarily Fine Phase Classification

Using a constant K value for the IPC-based phase classification of all programs results in unnecessarily fine partitioning and more simpoints to simulate or measure when the IPC values of the intervals in the same phase are already very close to each other. We control the number of resulting phases based on IPC in two steps.

The first step controls the selection of the initial centers based on the maximum and minimum IPC of the program. A percentage of the minimum IPC value is used as the distance d between the initial centers. This ensures that intervals with very close IPCs need no further partitioning and the final number of clusters does not explode with little benefit. This percentage

is adjustable in our infrastructure. The maximum value is divided by d. The value of quotient plus 1 is then compared with the given k. The smaller one is used as number of clusters. This value may be 1, meaning that the IPC values of all of the intervals are very close and no finer partitioning is necessary.

The second step maintains the distance between centers during the initialization of the centers in case there is a IPC much higher than others, but there are only two different IPC values during program execution. The first step does not know this and the number of clusters will be k which results in unnecessarily more cluaters. This step is similar to the construction of a minimum spanning tree except that we use the largest values in each step to choose the next initial center. The first initial center is selected randomly. During the generation of the other initial centers, each time the value with largest distance to the existing centers is the candidate. If this distance value is less than half of d, no more initial centers are generated. This prevents intervals with the similar BBVs and very close IPCs from being partitioned into different clusters.

6.3 Experimental Results

6.3.1 Benchmarks and Experimental Setup

10 benchmarks from MediaBench [39] are used for the verification of this new phase classification method. The description of the benchmarks and the baseline configuration for simulation are shown in Table 3.1 and Table 3.2, respectively in Chapter 3. The power configuration is almost the same as the default configuration provided with Sim-panalyzer.

We enlarge the input to *adpcmencode, adpcmdecode, unepic and jpegencode* to make them run longer, so that a uniform interval size, 1 million, can be used. *jpegencode* does not work on the large input and is removed from the final experimental results. These 10 benchmarks are used because of hardware condition. These 10 benchmarks are the only ones that can be compiled successfully and run on our Skiff board.

Each benchmark is compiled with `-static` and `-msoft-float` options by gcc2.95.2 on the Skiff board. Then the binary code, a script file, and a granularity of 10, are input to the tool. After the process in Figure 6.2, the *ppoints* and other output files are generated.

This phase classificaiton method is also evaluated through experiments on a real system in Chapter 8.

6.3.2 Experimental Results

To illustrate the improvement of our tool in finding representative points for power behavior, we simulate the power dissipation of each interval for each benchmark and use the average relative standard deviation(RSD) in power dissipation of each benchmark to evaluate the benefit from our classification method. Higher average RSD means the classification method is worse. We divide the standard deviation of each cluster by the average interval power dissipation of the cluster to get the RSD of each cluster. Then the average of the RSDs of all of the clusters is the average RSD for the benchmark.We compare three methods of getting representative intervals.

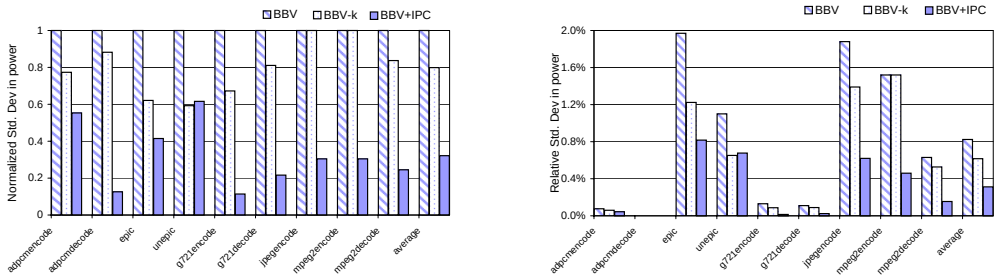
1. **BBV** the original SimPoint method. No more than 10 clusters are generated.
2. **BBV-k** the same classification method as in SimPoint, but is given the same number of clusters as generated by the **BBV+IPC** method. The best number is selected by the classification procedure.
3. **BBV+IPC** our new phase classification. Both the old method with fixed number of clusters and the new one with flexible clusters are considered here.

In the following figures, There are three columns for each benchmark, corresponding to the three methods, left to right.

BBV+IPC Method without Finer Classification Control

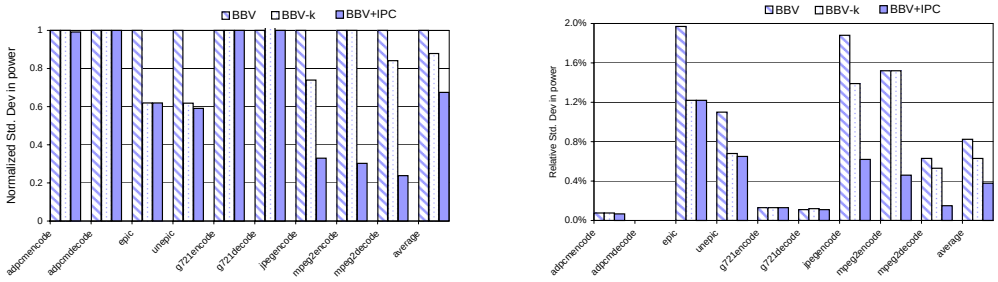
Figure 6.3(a) shows the decrease in normalized average relative standard deviation(RSD) of the clusters for each benchmark. Figure 6.3 (b) shows the decrease in average RSD of the clusters for each benchmark. The BBV+IPC method in Figure 6.3 is the old one with fixed number of clusters. That is, there are 10 clusters based on IPC.

In Figure 6.3(a), our old **IPC+BBV** method aggressively decreases the RSD for most of the benchmarks relative to the RSD of the **BBV** method. The average relative decrease is 68%. The average numbr of *ppoints* to simulate is 20.55, about twice of that generated in



(a) (b)

Figure 6.3: Decrease in RSD. The BBV+IPC method is the old one with fixed number of clusters.



(a) (b)

Figure 6.4: Decrease in RSD. The BBV+IPC method is the new one with flexible number of clusters

the BBV method. For the old **IPC+BBV** method, finer classification can be obtained through increasing the number of clusters based on IPC. But there is a trade-off between the granularity and simulation time.

For some benchmarks, such as *adpcmencode, adpcmdecode, g721encode* and *g721decode*, although there is a big relative decrease in RSD in In Figure 6.3(a), Figure 6.3(b) shows that the absolute decrease in RSD is very small. For example, **BBV+IPC** decreases the average RSD of *g721encode* by almost 90%, but the change in absolute RSD is from 0.13% to 0.01%. The benefit can not offset the extra time and space consumption of the unnecessary *ppoints*. Restriction on number of IPC-based clusters is needed to control unnecessary finer classifications.

BBV+IPC Method with Finer Classification Control

Figure 6.4 shows the relative and absolute decrease in average RSD. Through the restriction on phase classification based on IPC described in section 3, there is no finer classification for the 4 benchmarks with already very low average RSD. The average number of *ppoints* is reduced from 20.55 to 14.55, by about 30%. The average relative decrease by **BBV+IPC** in Figure 6.4(a) is 38%, much less than the relative decrease of the old **BBV+IPC**. But the relative improvement to the 6 benchmarks that have a high average RSD in SimPoint is 56%, comparable to the 64% relative improvement for the same benchmarks by the old **BBV+IPC**.

Figure 6.4 (b) shows that when the refined phase classification is used, five out of the six benchmarks with high RSD show the same improvement as when the **BBV+IPC** without finer classification control is used. *epic* gets the same improvement from **BBV-k**. As to averaged decrease in RSD for all of the benchmarks, the new **BBV+IPC** gets a decrease from 0.82% to 0.38%, while the old method gets a decrease from 0.82% to 0.31%. The difference is less than 10%. This is also acceptable considering the 30% reduction in number of *ppoints*. With the increase in number of IPC-based clusters, the benefit from the unnecessary finer classificaiton gets larger. The trade-off between the number of *ppoints* and the accuracy depends on the requirement of the user.

For the two intervals in Figure 1.2, our **BBV+IPC** classifies them into two different phases. They are identified to have different power behavior. Figure 6.5 shows the power and IPC for each interval of *jpegdecode*. The same method is used to show the cluster numbers on the two *x* axes as in Figure 6.1.We can see the power line for each cluster is smoother than in Figure 6.1. The decrease in average RSD is 67% for *jpegdecode*.

BBV-based Classification with Larger *K*

The columns for the **BBV-k** method in Figure 6.3 and Figure 6.4 show the insufficiency of SimPoint in find representative intervals for power behavior. Even though it is given the same number of initial clusters as in **BBV+IPC**, its improvement in RSD is much less. Since the phase classification of SimPoint is based on BBVs, it generates fewer clusters than the given number. That is, only this number of phases can be identified. The improvement from **BBV-k**

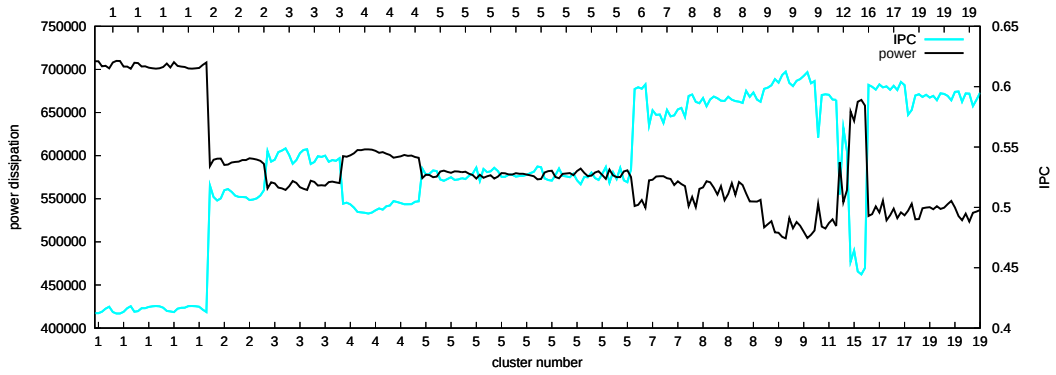


Figure 6.5: Power and IPC for each cluster of *jpegencode* after the **BBV+IPC** classification.

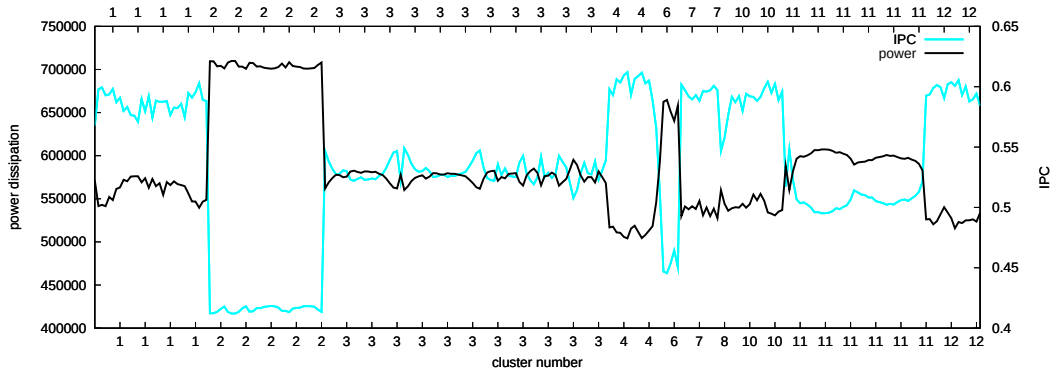


Figure 6.6: Power and IPC for each cluster of *jpegencode* after the **BBV-k** classification.

remains the same in Figure 6.3 and Figure 6.4, although the given number of initial IPC centers is changed by 30%. This again shows the limitation of SimPoint in find representative intervals for power behavior characterization.

Figure 6.6 shows the power and IPC for the intervals of the clusters generated by the **BBV-k** method. The same method is used to show the cluster numbers on the two *x* axes as in Figure 6.1. This is different from Figure 6.1 because of the larger number of initial centers. This K is the same as the number of clusters in Figure 6.5. Even when a k value of 19 is given to **BBV-k**, it generates only 12 clusters. That is, only 12 phases are identified. The difference of power and IPC among intervals from the same cluster is still large, e.g. cluster 1 and 11. The two intervals in Figure 1.2 are still in the same cluster.

Error rates for estimation of total power dissipation are not shown here. Since our **BBV+IPC**

method only refines the clusters generated by SimPoint. The error rate should not be higher than the values in Figure 3.2.

Through refining the BBV-based phase classification using runtime event counts, we get a better classification method for program power behavior characterization.

Chapter 7

Infrequent Basic Block-based Program Phase Classification

Through simulation, we demonstrated that we can estimate the power consumption of a program using the power consumption of the selected representative intervals. When we want to validate the feasibility of SimPoint for power estimation on a real machine, we need to identify a representative interval during program execution to measure its power. In order to get the power curve that is as close to the real power behavior of the interval as possible, the instrumentation overhead should be very low so that it does not interfere the measured power behavior. Intervals with a fixed number of instructions do not work now, since dynamically counting the number of executed instructions brings high overhead.

Research in performance optimization often concentrates the optimization effort on frequently executed code. However, a large part of most programs is infrequently executed. The execution of an infrequently executed basic block often means a transition in program execution. It may be a transition from one phase to another, or a transition within the same phase, but from one group of instructions to another group. Using infrequently executed basic blocks to demarcate intervals may help us get better phase classification. What is more important is that through instrumenting these infrequently executed basic blocks, we can dynamically identify the beginning and the end of an interval during program execution with negligible overhead [26].

We measure the power consumption of both whole program execution and representative intervals using the measurement setup described in Chapter 5.

7.1 Which Basic Blocks are Infrequent?

Different program/input pairs execute different number of basic blocks. It is hard, if not impossible, to choose an absolute number as the best threshold for all programs to determine

infrequent basic blocks. After trying several methods of determining infrequent basic blocks, we use a percentage to find relatively infrequent basic blocks for each benchmark, instead of using an explicit frequency as the threshold. This percentage is the ratio of the total execution times of all infrequent basic blocks in that of all basic blocks. We sort the basic blocks based on their execution frequencies in decreasing order, then add up the numbers from the smallest one. When the sum is larger than the specified threshold, for example, 5% of the total executions times of all basic blocks, this procedure stops and the scanned basic blocks before the last one are selected as infrequent basic blocks for this program/input pair. Intuition tells us that when a low threshold is used, the selected infrequent basic blocks are distributed sparsely in program execution and the difference in size among the resulting intervals is larger than when a higher threshold is used.

Through simulating the MediaBench benchmarks, we also investigated the trade-off among infrequency threshold, program energy consumption estimation error, simulation workload and instrumentation overhead. Figure 7.1 shows the simulation result. The threshold on *x* axis is defined as in Section 7.2. The values are averaged over all of the benchmarks.

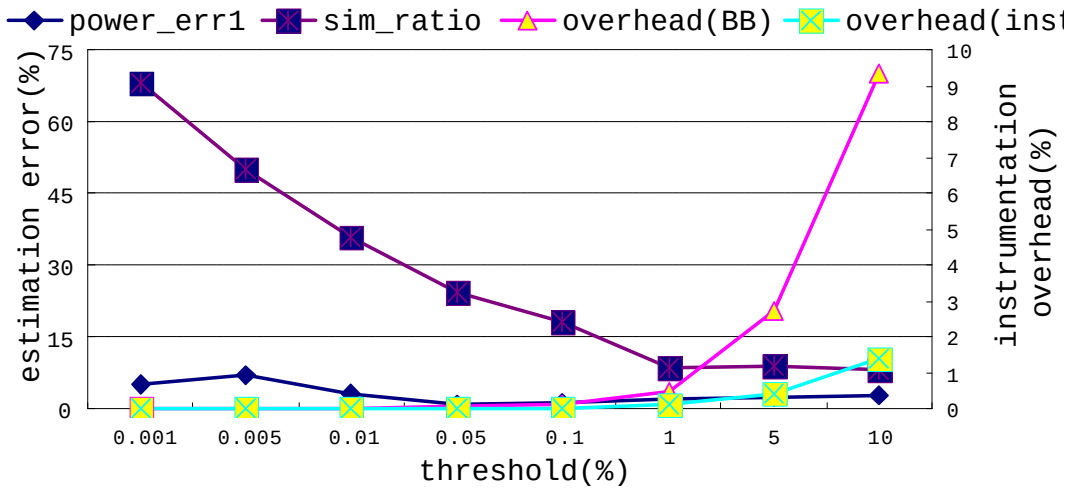


Figure 7.1: Trade-off between Accuracy and Simulation/Measurement Workload.

power_err is the error rate of using the energy consumption of representative intervals to estimate the energy consumption of the whole program. *sim_ratio* is the ratio the number of simulated instruction in the total number of instruction of the program execution. It means

the simulation workload. When the infrequency threshold is lower, the interval size is larger due to the sparse infrequent basic blocks, which results in a large number of instructions being simulated as instructions in the representative intervals. Estimation error rate is also higher because of the largely variant interval size. When the threshold is increased, there are more basic blocks that can be used to demarcate intervals and interval size is decreased. But instrumentation overhead is increased since more basic blocks should be instrumented in order to identify an interval during program execution. Figure 7.1 show that threshold 1% is the best one among the 8 ones we tried in experiments. This conclusion is consistent the one we draw in the experiments performed on a real system through physical measurement.

7.2 Basic Block Execution Frequency Profiling and Infrequent Basic Blocks Selection

Camino provides interfaces for basic block level instrumentation. At the entrance to each basic block, a call to a execution frequency counting library function is inserted. The distinct reference value of the basic block is passed to the function that increments the frequency of this basic block. Here we count the absolute execution frequency, that is, the number of times that a basic block is executed. Counts for the basic blocks are available after the instrumented program execution.

We try 3 different threshold values, 0.1%, 1%, and 5%, to investigate the trade-off between interval size variance and instrumentation overhead.

7.3 BBV Profiling and Program Execution Partition

As in SimPoint 2.0, we use BBV of both frequent and infrequent basic blocks as the fingerprint of an interval. Although infrequent basic blocks are used to demarcate intervals, partitioning program execution just based on the number of executed infrequent basic blocks may generate too small or too large intervals depending on the distribution of the infrequent basic blocks. Too small intervals often result in too many phases. Too large intervals are hard to measure with high precision using our measurement equipment. So we use a number of executed instructions to make the final interval lengths as uniform as possible.

Instrumentation for BBV profiling is similar to that for basic block execution frequency profiling, except that a different library function is called. All basic blocks are instrumented, so that we can get the complete fingerprint of the basic blocks in an interval. An interval size of 30 million instructions is used to avoid too large or too small intervals. The library function counts both the number of executed instructions for each basic block and the total number of executed instructions for the current interval. When an infrequent basic block is encountered, if the current total number of instructions is larger than or equal to 30 million, this basic block indicates the end of the current interval and it is the first basic block of the next interval. Figure 7.2 illustrates the interval partition using the combination of infrequent basic block and interval size. Here A, B, C, and D are basic blocks. C and D are infrequent and used to demarcate intervals. The lengths of the intervals can vary significantly and depend on the distribution of infrequently executed basic blocks across the program execution. Only the occurrences of C and D in shadow mark intervals. Other occurrences do not mark intervals because the interval size is smaller than 30 million when they are encountered. We get intervals of similar size by using this method. An execution frequency counter of C and D can be used to identify the exact execution of an interval. For example, the fourth interval starts when the counter is 5 and ends when the counter is 6.

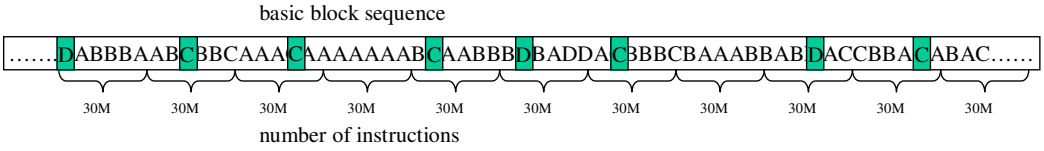


Figure 7.2: Interval partitioning using infrequent basic blocks and interval length.

7.4 A SimPoint-like Method for Phase Classification

Since the intervals are demarcated by infrequently executed basic blocks, they may have variable number of instructions. K-Means clustering is used for phase classification based on the BBVs collected in Section 7.3. As in SimPoint, the BBV of each interval is projected to a vector with much smaller dimension. Then k initial cluster centers are selected. The distance

between a vector and each center is calculated and each vector is classified into the cluster with the shortest distance. A cluster center is changed to the average of the current cluster members after each iteration. The iteration stops after the number of vectors in each cluster is stable. The simpoint of a phase is the one that is closest to the center of the cluster.

Weighting a simpoint with just the number of intervals in its phase cannot reflect the real proportion of this phase in whole program execution. We changed the weighting method such that each simpoint has two weights. The first weight is based on the percentage of the number of executed instructions of the corresponding phase in that of the whole program. Since we also profile the number of executed instructions for each interval in Section 7.3, it is easy to get this value and use it in the process of K-Means clustering. This weight is used to estimated the behavior of the whole program. The second weight is based on the number of intervals in the corresponding phase as in [3]. It tells us the number of occurrences of each simpoint in behavior estimation for the whole program execution.

The calculation of BIC (Bayesian Information Criterion) score is also changed to take variable interval lengths into account. We use the weights based on the number of executed instructions in each phase to calculate the log likelihood, such that phases with longer intervals have larger influence. It is similar to the calculation used in SimPoint 3.1 [37], but instead of using weight of each interval, we use the weight of each phase, which is simple since there are usually much fewer simpoints than intervals, and the weights based on variable interval lengths are already calculated.

Clustering is performed for different number of clusters and different cluster seeds. BIC scores from different clustering are compared and the one with the best trade-off between BIC score and number of phases is selected as the final clustering model. Intervals are clustered based on this model, and the simpoints and weights are calculated. The distinct reference values of the two infrequent basic blocks that demarcate each simpoint are recorded. These basic blocks are the final infrequent basic blocks that are instrumented for power measurement.

7.5 Low-overhead Instrumentation for Power Measurement

We use physical power measurement to verify that the selected simpoints are representative in energy consumption estimation. Instrumentation is needed to identify the data points for each simpoint in the final measurement result. We choose static instrumentation instead of using a dynamic instrumentation tool such as Pin [42] used in a previous work [31] because we want to instrument the program on basic block level, and at the same time lower the interference to the measured program as much as possible. We use *Camino* to instrument a program statically to generate special signals at the beginning and at the end of a simpoint, so that we can get a measurement result in high resolution and as close as possible to the real power behavior of each simpoint.

To identify a simpoint, we use the execution frequency of each infrequent basic block profiled in Section 7.3 and the final infrequent basic blocks recorded in Section 7.4. Our infrastructure supports two power measurement methods for any selected intervals.

One method is to measure the intervals selected by the phase classification, here the simpoints, in one program execution. By counting the execution times of the final infrequent basic blocks in all of the intervals, we get the number of execution times of the final infrequent basic blocks before each simpoint, and the number of execution times of these basic blocks in each simpoint. This information is put into a file for future reference by a library function to mark the beginning and the end of each simpoint. All of the final infrequent basic blocks are instrumented to call this library function, which counts up the execution times of these basic blocks and generates special signal to trigger the power measurement device when the counter reaches the recorded number of execution times before the beginning or to mark the end of a simpoint. To reduce comparison time, the simpoints are sorted in the order of their occurrence in program execution and the corresponding numbers are read into a linked list at the beginning of the program execution. A pointer to the node for the current simpoint moves one step after a simpoint is finished, such that we avoid searching for the fast-forwarding information. Off-line data analysis identifies each simpoint in the continuous measurement result based on the signals before and after the simpoint. The instrumentation overhead of this method is discussed in the next section.

The other method is to generate one executable for each simpoint for power measurement. Infrequent intervals that demarcate different simpoints are usually different, so this method has even lower instrumentation overhead than the first one. For a simpoint, only the final infrequent basic blocks that demarcate this simpoint are instrumented to call a library function, which increments a counter and generates special signals. The numbers of executed basic blocks for each simpoint are put into a separate file and are read into two variables at the beginning of program execution. This method separates the measurement of the simpoints into independent tasks. Users may choose to measure only the simpoints that represent long phases. It provides more detailed power behavior of the measured simpoints using our power measurement infrastructure, but the program is executed one time for each measurement, although the execution stops immediately after the measured simpoint.

7.6 Benchmarks

The benchmarks are from the members of SPEC CPU2000 INT that can be compiled by *Camino* successfully, shown in Table 7.1. *gzip, vpr, mcf, parser* and *twolf* are compiled with GCC 3.4.2. The other benchmarks are compiled with GCC 2.95.4 because the combination of *Camino* and GCC 3.4.2 fails in compiling these benchmarks correctly.

Table 7.1: SPEC CPU2000 INT benchmarks

164.gzip	Data compression using Lempel-Ziv coding (LZ77)
175.vpr	Integrated circuit placement and routing in FPGAs)
176.gcc	C compiler for Motorola 88100 based on gcc 2.7.2.2
181.mcf	Combinatorial optimization/Single-depot vehicle scheduling
197.parser	Syntactic parser that does grammar analysis for English text
253.perlbmk	Cut-down version of Perl v5.005_03
254.gap	Language and library designed for group-theoretic computation
255.vortex	Single-user object-oriented database transaction
256.bzip2	Block-sorting compression
300.twolf	Transistors placement and global connections

7.7 Instrumentation Overhead Evaluation

The original 10 SPEC CPU2000 integer benchmarks without any instrumentation are measured to obtain their CPU energy consumption. To show the low overhead of our instrumentation method, we also measure the CPU energy consumption with instrumentation on all final infrequent basic blocks obtained in Section 7.5. We control another LED instead of *numlock* in this instrumentation, so that there is no impact on the signal on the trigger channel, and the energy consumption is almost the same. Only the instrumentation overhead for the first method in Section 7.5 is measured here. The second method has even lower overhead since fewer basic blocks are instrumented. Figure 7.3 shows the overhead of the instrumentation using different thresholds. It is normalized to the measured energy consumption of the uninstrumented benchmarks. A positive value means the measured energy consumption for this configuration is larger than that of the uninstrumented one. A negative value means the opposite. The measured energy consumption for any threshold is almost the same as that of the uninstrumented ones. For some benchmarks, for example, *perlbmk* and *bzip2*, the energy consumption of the instrumented program is even lower than the uninstrumented program. One possible reason is that inserting instructions somewhere might accidentally improve the performance or power consumption, possibly due to a reduction in conflict misses in the cache because of different code placement. We notice that the four values are almost the same for *mcf*. The reason is that the all the frequently executed basic blocks are in 4 of the 30 identified phases when SimPoint is used. The final instrumented basic blocks for the large phases are mostly infrequent. SimPoint has a very high overhead for most benchmarks because the basic blocks demarcating the intervals are executed frequently. Figure 7.4 shows the same trend in measured execution time when different thresholds are used.

7.8 Measuring Energy Consumption of Simpoints

Energy consumption of each simpoint is measured using the trigger mode of the oscilloscope. Measuring all simpoints in one program execution in auto mode takes shorter time, but the resolution is much lower than the other method because the communication latency between the oscilloscope and the data acquisition machine put an upper bound on the resolution we can use,

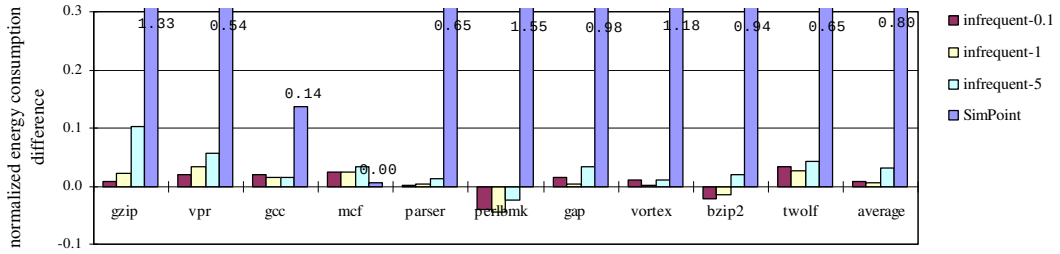


Figure 7.3: Normalized overhead in energy consumption of instrumented benchmarks using different thresholds.

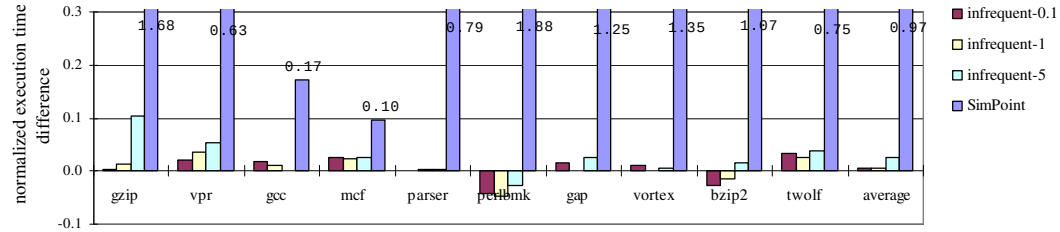


Figure 7.4: Normalized overhead in execution time of instrumented benchmarks using different thresholds.

otherwise, some data points will be lost. Measuring the simpoints one by one removes this limitation, so we can get very high resolution. Program execution and data acquisition are on the same machine. Reading data from the oscilloscope is always performed after the measurement of a simpoints is done. There is still no interference to the measured program execution. We use the second instrumentation method in Section 7.5, and implement an automatic measurement and data acquisition process to do measurement of any number of simpoints as a single task.

7.9 Error Rates in Whole Program Energy Consumption Estimation

We tried three different thresholds to find infrequent basic blocks, 0.1%, 1%,and 5%. In each case, the total absolute execution frequency of the selected infrequent basic blocks are less than 0.1%, 1%, or 5% of total execution frequency of all basic blocks. The basic blocks instrumented for power measurement are a subset of these. Actually, at most two of them are instrumented for physical measurement of each simpoint. A maximum number of clusters, 30, is used to find the best clustering as in SimPoint [3]. We also show the experimental results

of SimPoint with a fixed interval length of 10 million instructions. We do not claim that our phase classification method is more accurate than SimPoint. Rather, we show that we can also find the representative slices for program execution using infrequent basic blocks to demarcate intervals, and this method enables power physical measurement of simpoints with very low instrumentation overhead and provides a way to get fine-grained time-dependent power behavior through measurement.

Using the power measurement infrastructure described in Section 5.3, we measured the CPU power curves for the uninstrumented benchmarks, the ones with all final basic blocks instrumented, the simpoints of the two instrumentation methods mentioned in Section 7.5. Energy consumption is calculated as

$$E = U \times \sum (I \times t)$$

where E is energy consumption, U is the voltage of CPU, I is the measured current on the CPU power cable, t is the time resolution of the power data points. The sum is over all of the data points for one benchmark or simpoint.

Due to the variable interval lengths, we estimate the total energy consumption using the weight based on our modified weighting scheme in Section 7.4. *energy/instruction* is calculated for each simpoint, the products of this value and the weight are added up, and the estimated energy consumption is the product of this weighted *energy/instruction* and the total number of instructions. Energy estimation error rate is calculated as

$$error = \frac{|energy\_estimated - energy\_measured|}{energy\_measured}$$

Time estimation is similar to energy estimation.

Figure 7.5 shows the error rates of the infrequent basic block-based phase classification method with different thresholds and SimPoint with fixed interval size of 10M instructions. Error rate is based on the comparison between estimated energy and measured energy of the uninstrumented benchmarks. The columns show us the trade-off between interval size variance and instrumentation overhead. Low threshold results in variable length intervals clustered into the same cluster. But the interference to the program execution is also low since only a few

basic blocks are instrumented. The opposite is true when a high threshold of 5% is used. The interval size is more stable in the same phase, but the instrumentation overhead is high. When threshold 1% is used, we get the lowest error rate among the three. SimPoint has high error rates due to the high frequency of the instrumented basic blocks. In our experiment result, the number of intervals increases with the increase in the threshold used to find infrequent basic blocks. Graphs are not shown here due to space limitation.

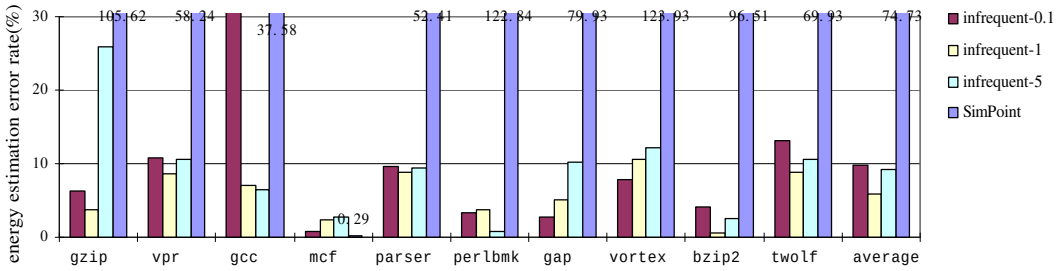


Figure 7.5: Error rates of energy consumption estimation when different thresholds are used, based on comparison between estimated and measured energy of uninstrumented benchmarks.

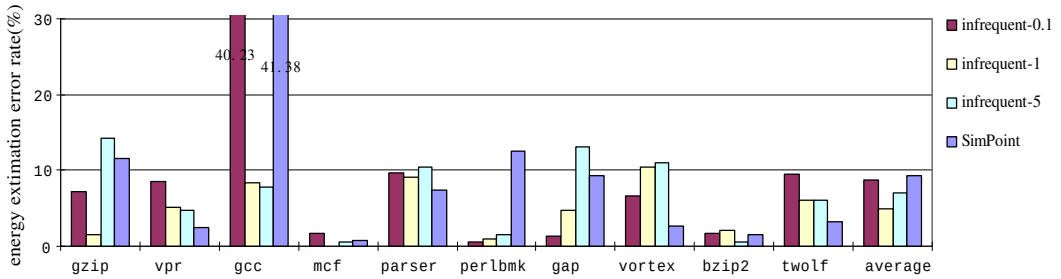


Figure 7.6: Error rates of energy consumption estimation when different thresholds are used, based on comparison between estimated and measured energy of instrumented benchmarks shown in Figure 7.3.

To verify that the low error rates in Figure 7.5 are not obtained by accident and the selected simpoints are really representative of the program execution, in Figure 7.6, we show the error rates when the estimated energy consumption is calculated from the measured simpoints with all final infrequent basic blocks instrumented and compare this estimation to the measured energy consumption of the whole benchmark with all final infrequent basic blocks instrumented. Here SimPoint has very low average error rate since instrumentation overhead does not affect

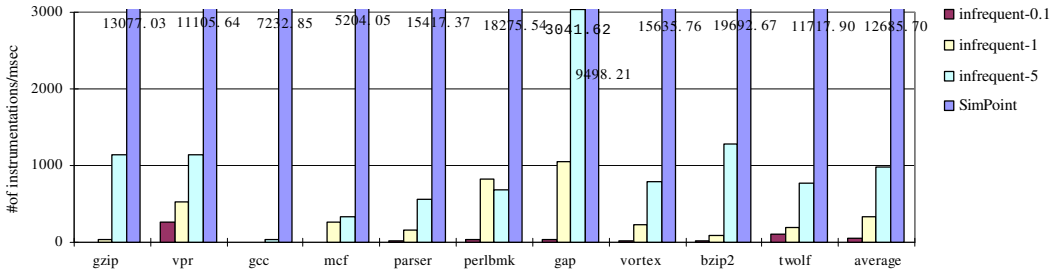


Figure 7.7: The number of instrumentation per millisecond during the program execution in simpoints.

the estimation accuracy now. It is less than 6% after *gcc* is removed from the benchmark group. This error rate might be higher than the error rate evaluated through simulation because there is an operation to set the voltage to a low value at the end of each simpoint, but there is no such operation at the end of other intervals. This causes higher estimated energy consumption, but does not affect the precision of the measured power behavior of simpoints. Our new phase classification has lower error rates for some benchmarks. One possible reason is that the program behavior of these benchmarks is hard to characterized by simpoints of the same size. The low error rate in Figure 7.6 shows that the selected simpoints are truly representative of the program execution and our low overhead instrumentation method enables us to get the program time-dependent power behavior that is very close to the real power behavior.

Lau *et. al* proposed variable length intervals and hierarchical phase behavior [37]. We did not validate the profiling and clustering in the new SimPoint version in energy estimation consumption because time-dependent power behavior observation is our objective and thus small and similar interval sizes, detailed BBVs, and low overhead are necessary.

Figure 7.7 shows the frequency of instrumented basic blocks during the program execution in simpoints. Here we can see that instrumentation overhead increases with the increased threshold. This is consistent with our explanation of the trade-off between interval size variance and instrumentation overhead. SimPoint has the highest value because of the high frequency of the basic blocks that demarcate the simpoints.

Using infrequently executed BBs to demarcate intervals results in negligible instrumentation overhead and enables accurate and efficient program power behavior characterization on

real systems.

Chapter 8

An Infrastructure for Efficient Power Behavior Characterization

This chapter describes the current state of our infrastructure. Infrequent basic blocks are used to demarcate intervals. Phase classification is based on the combination of control-flow information and runtime event count. To evaluate the phase classification accuracy in terms of time dependent power behavior, we perform FFT on the measured power curves and use the transformation results to evaluate the similarity between two intervals.

8.1 A New Phase Classification Method

The flowchart in Figure 8.1 illustrates the components in our current infrastructure. The two-stage phase classification is similar to the one described in Chapter 6.

8.1.1 Using EV as Fingerprint

In Chapter 3 and Chapter 7, we show that using BBV as interval fingerprint results in low error rate in energy consumption estimation through both simulation and physical measurement. Compared to BBVs, EVs give us more information about the control behavior of the program at run-time. BBVs contain information about what parts of a program were executed, but EVs tell us what decisions were made in arriving at these parts of the program. This extra information allows a classification of phases that more accurately reflects program behavior. For the same BBV, it is possible that there are several EVs depending on the dynamic paths taken during program execution. An example is shown in Figure 8.2.

In our current infrastructure, we use the EV of conditional edges as the fingerprint of an interval. This vector is the absolute count for each control-flow edge traversed during the execution of an interval [33].

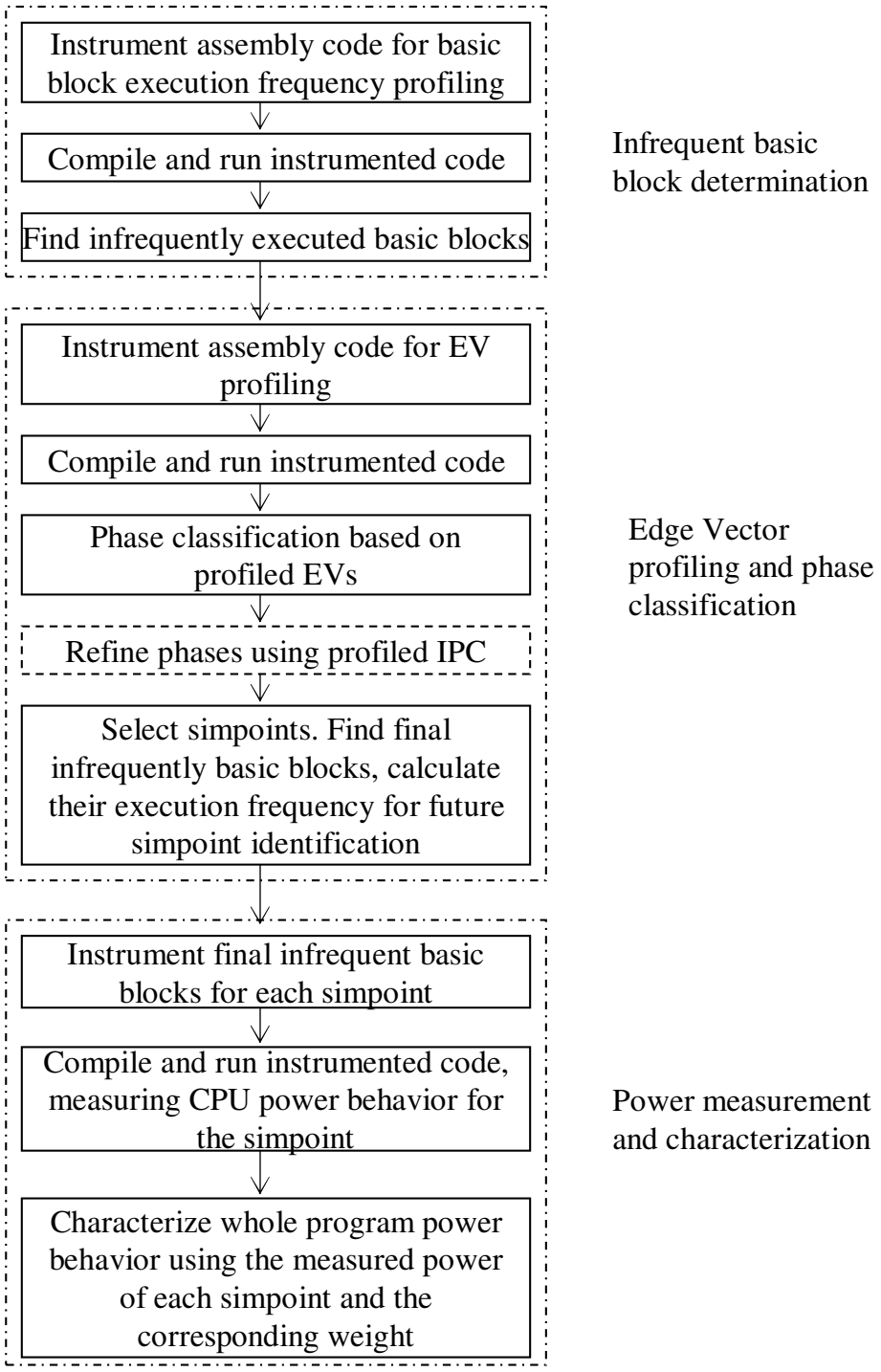


Figure 8.1: Infrequent basic block-based phase classification and power measurement of simpoints.

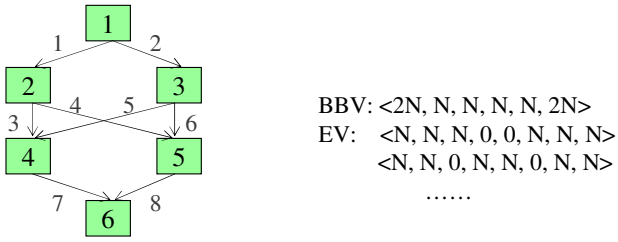


Figure 8.2: Several EVs are possible for the same BBV.

8.1.2 EV profi ling

Before we profile the EV for each interval, infrequent basic blocks are selected using a relative threshold. Three thresholds, 0.05%, 0.1%, and 1%, are tried. We do not try 5% because 1% is proved to be the best of the threshold tried in the experiments where 0.1%, 1%, and 5% are tried. For lower instrumentation overhead as well as low error rate, we try the thresholds that is no higher than 1% here. For each threashold, we perform infrequent basic block selection, EV profiling, phase classification, power measurement of representative intervals and error rate calculation, as described in Chapter 3.

Instrumentation for EV profiling is similar to that for basic block execution frequency profiling shown in Section 7.3. A pre-defined interval size of 30 million instructions is used to avoid too large or too small intervals. All basic blocks are instrumented so that we can get the complete fingerprint of an interval. The analysis routine, a library function, remembers the last executed basic block and knows the taken edge based on the last and the current executed basic blocks. It counts each control flow edge originating in a basic block that ends in a conditional branch. It counts the total number of executed instructions for the current interval as well. When an infrequent basic block is encountered, if the count is larger than or equal to 30 million, this basic block indicates the end of the current interval and it is the first basic block of the next interval.

Note that, because we only have coarse control over where the demarcating infrequent basic blocks will occur, the actual interval might be somewhat longer than 30 million instructions; thus, the intervals are variable-length. We use the adjusted BIC score calculation to take variable interval lengths into account.

8.1.3 Refining Phase Classification Using IPC

In our current infrastructure, we use a two-stage phase classification method, as shown in Chapter 6, which combines EV and IPC to get the representative intervals for power behavior characterization. By this method, intervals with similar EVs but different IPCs are classified into different phases. The difference level between IPCs is adjustable to balance the accuracy and time cost.

8.1.4 Linux Device Driver for Event Counter Profiling

Through using the device driver mentioned in Section 4.2.4, profiling IPC is easy to do in our infrastructure. The device driver is initialized to read the number of execution cycles for each interval After the program execution is partitioned into intervals, all of the infrequent basic blocks that demarcate the resulting intervals are instrumented to collect the number of clock cycles taken by each interval. By running the instrumented program once, we can get the IPC values of all intervals by dividing the number of instructions by the number of cycles. We already have the number of instructions executed from the edge vector profiling. This technique very slightly underestimates IPC because of system activity that is not profiled, but we believe this has no impact on the accuracy of the classification since IPC tends to vary significantly between phases. Since we identify intervals based on infrequent basic block counts, the overhead is low and has a negligible impact on the accuracy of the profiling result.

8.1.5 Combining EV Clustering with IPC Clustering

For a program execution, we first perform the phase classification in Section 7.4 to group intervals with similar EVs together. Then we do another phase classification based on the profiled IPC values. The two-stage phase classification and refining control in Chapter 6 are used to get phases in terms of both energy consumption and time-dependent power behavior.

8.2 Validation on Real System

We validate our infrastructure through physical power measurement of the CPU of a Pentium 4 machine, using the same experimental setup and benchmarks as described in Chapter 5 and

Chapter 7.

8.2.1 Comparing Error Rates in Energy Consumption Estimation

The first step to verify that this infrastructure is useful in power behavior characterization is to calculate the error rate when the measurement result of the selected simpoints is used to estimate the power consumption of the whole program. Although we use EVs as the fingerprint of an interval in our infrastructure, we also measured the CPU power of the simpoints using BBVs for comparison.

The energy consumption of each simpoint is measured using the trigger mode of the oscilloscope. We generate an executable for each simpoint and measure the simpoints one by one so we can get very high resolution as well as the lowest possible instrumentation overhead. Program execution and data acquisition are on the same machine. Reading data from the oscilloscope is scheduled after the measurement of a simpoint is done. Data acquisition does not interfere with the running program. We implement an automatic measurement and data acquisition process to measure any number of simpoints as a single task.

8.2.2 Power Behavior Similarity Evaluation

Even though we can get low error rates in estimating whole program energy consumption, energy consumption is the average behavior of an interval. Intervals that are classified into the same phase may have different time-dependent power behavior. If intervals in the same phase have largely different power behavior, we cannot characterize the time-dependent power behavior of the whole program execution using the measurement result of the simpoints.

Comparing in the Frequency Domain

Energy consumption of an interval does not reflect time-dependent power behavior. The two power curves in Figure 1.1, section 1.1.1, have the same energy consumption and the same average power, but they are significantly different in time-dependent power behavior.

Our power measurements come in the form of discrete samples in the time domain. Power behavior is characterized by periodic activity, so a comparison in the frequency domain is more

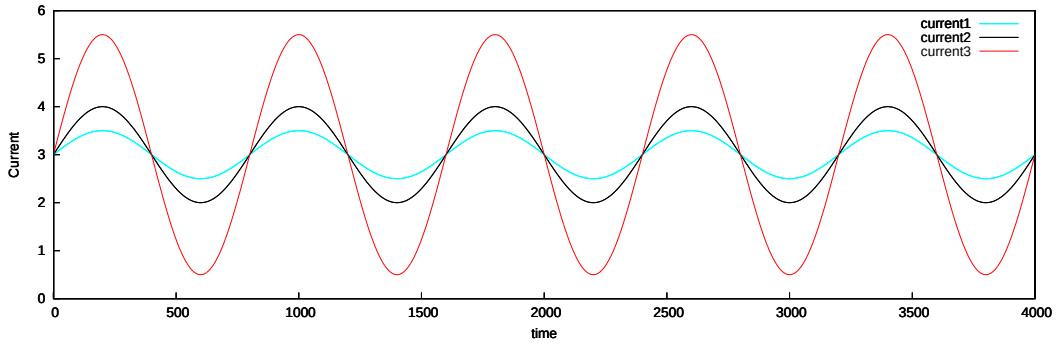
appropriate for determining whether two intervals are similar. Fast Fourier Transform (FFT)is a computationally fast way to calculate the frequency, amplitude and phase of each sine wave component of a signal. Thus, we compare the power behavior similarity of two intervals by comparing their discrete Fourier transforms computed using FFT. After the FFT calculation of a power curve, each frequency is represented by a complex number. In power curve similarity comparison, the phase offset of the same frequency should not affect the similarity of two curves. For instance, two power curves might be slightly out of phase with one another, but have exactly the same impact on the system because they exhibit the same periodic behavior. So when we compare two power curves, we calculate the absolute value of the complex number for each frequency, the distance between two corresponding absolute values, and the Root Mean Square (RMS) of the distances for all frequencies. The equation is given in a following section.

Figure 8.3 shows the FFT distance between the *sine* curves with different values in amplitude, frequency and phase offset, calculated using our method mentioned above. We generate 4096 samples for each curve. Ideally, there is only one frequency in the FFT output of each *sine* curve. But we get multiple frequencies due to the discrete data samples. This is the reason why the calculated distance values are not 0's in Figure 8.3 (c). The three curves in Figure 8.3 (a) have the same frequency and phase offset, but different amplitude, which determines the similarity of two curves. Figure 8.3 (b) shows the effect of frequency in our similarity calculation. The small(compared to the values in (a) and (b)) distance between the curves in Figure 8.3 (c) demonstrate that the effect of phase offset is eliminated.

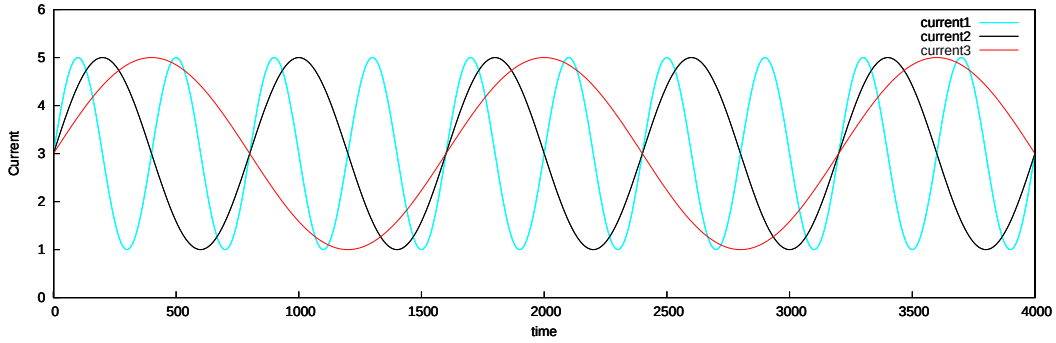
All of the curves shown in Figure 8.3 have the same average power, such that they have the same energy consumption. But they have significantly time-dependent power behavior. We can not tell this difference based on total energy consumption. FFT distance can be used to evaluate time-dependent power behavior similarity.

A More Robust Sampling Approach for Verification

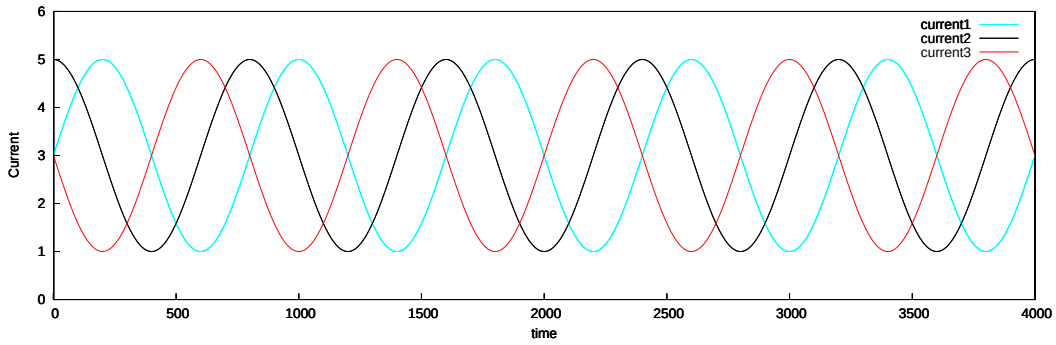
Measuring every interval in a long-running program is infeasible because of time and space constraints (indeed, this fact motivates our research). Thus, we use a more robust sampling methodology to verify that power behavior is consistent within a phase. We choose 20 intervals at random for each phase of each program to compare the FFT results of their curves. If



(a) same frequency and phase offset, different amplitude. dist(1,2)=22.4, dist(1,3)=89.8, dist(2,3)=67.3



(b) same amplitude and phase offset, different frequency. dist(1,2)=119.6, dist(1,3)=115.6, dist(2,3)=120.8



(c) same amplitude and frequency, different phase offset. dist(1,2)=6.5, dist(1,3)=8.1, dist(2,3)=3.0

Figure 8.3: Power curve distances calculated using our similarity calculation method

the number of intervals in some phase is less than 20, all of the intervals are selected. The selected intervals for each phase are selected from a uniformly random distribution among all the intervals in the phase.

Instrumenting for Verification

Infrequent basic blocks demarcating the intervals from the same phase are instrumented to measure each interval in the same way we measure a simpoint. Each selected interval is measured separately. Then the FFT is performed on the measured power curve of each interval. The Root Mean Square (RMS) error of the FFT results is used to evaluate the variation of the power behavior of the intervals in this phase. For each phase, we calculate the arithmetic average over the frequencies in the FFT result of all measured intervals as the expected FFT of the phase. The distance between an interval i and the expected FFT is:

$$D_i = \sqrt{\frac{\sum_{j=1}^{N} (\sqrt{{c_j}^2 + {d_j}^2} - \sqrt{{a_j}^2 + {b_j}^2})^2}{N}}$$

c_j and d_j are the real and imaginary part of the jth frequency of interval i, respectively. a_j and b_j are the real and imaginary part of the jth frequency of the expected FFT respectively. N is the number of frequencies in the output of Fast Fourier Transform. Then the FFT RMS of a phase is calculated as:

$$FFT_{RMS} = \sqrt{\frac{\sum_{i=1}^{M} {D_i}^2}{M}}$$

M is the number of measured intervals in the phase. The lower FFT_{RMS} is, the high the similarity among the time-dependent power behavior of the intervals in the phase.

The FFT_{RMS} for each phase is then weighted by the weight of the corresponding phase to get the RMS for the whole benchmark. We evaluated the weighted FFT_{RMS} for all of the 10 benchmarks in two cases: when phase classification is based on EV only, and when IPC is used to refine phase classification.

8.2.3 Interval Length Variance

Using infrequent basic blocks to partition program execution into intervals results in variable interval length. We use a pre-specified interval size to avoid intervals that are too small. Intervals of large size are still possible due to the distribution of the infrequent basic blocks during program execution. We analyze the resulting size for each interval of each benchmark to show the distribution of the interval sizes.

We evaluate the interval length variance of a benchmark as the weighted RMS of the interval lengths in each phase. If this value is high, intervals that are of largely different number of instructions are classified into the same phase, the simpoint for the phase can not be representative of the phase in terms of power behavior.

8.3 Experimental Results and Evaluation

Using the power measurement infrastructure for Pentium 4 described in Chapter 5, we measured the CPU power curves for the instrumented benchmarks, the ones with all final infrequent basic blocks instrumented, the simpoints, and the selected intervals from each phase.

8.3.1 Instrumentation Overhead

Figure 8.4 shows the overhead of the instrumentation using different thresholds. It is normalized to the measured energy consumption of the uninstrumented benchmarks. A positive value means the measured energy consumption for this configuration is larger than that of the uninstrumented one. A negative value means the opposite. For some benchmarks, for example, *perlbmk* and *gap*, the energy consumption of the instrumented program is slightly lower than the uninstrumented program. One possible reason is that inserting instructions somewhere might accidentally improve the performance or power consumption, possibly due to a reduction in conflict misses in the cache because of different code placement. Overhead in execution time when different thresholds are used follow the same trend. Instrumentation overhead for power measurement of a single simpoint is even lower because only one or two of the final infrequent basic blocks are instrumented.

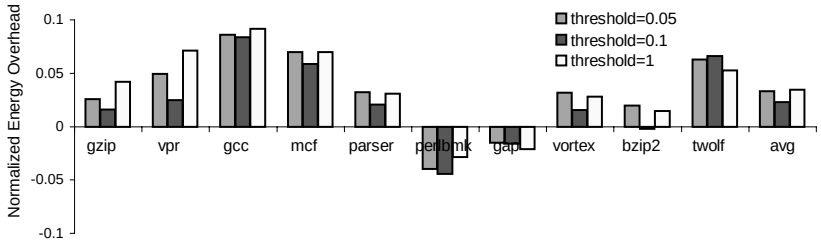


Figure 8.4: Normalized instrumentation overhead in energy consumption. The difference between the energy consumption of the instrumented and uninstrumented benchmark divided by the energy consumption of the latter.

8.3.2 Total Energy Consumption Estimation

We investigate both BBV and EV as the fingerprint of intervals in phase classification. A maximum number of clusters, 30, is used to find the best clustering in both cases. Simpoints are measured and the whole program energy consumption is estimated as

$$E_{est} = \sum_{i=1}^{k} E_i \times W_i$$

E_i is the measured energy consumption of the ith simpoint, W_i is its weight, and k is the number of phases. Although intervals have variable sizes, we estimate the total energy consumption using the weight based on the number of intervals in each phase.

For BBV-based phase classification, we use three percentage values 0.1%, 1%, and 5% to get the threshold for infrequent basic blocks. The measured energy consumption of simpoints are used to estimate the whole program energy consumption. The error rate is the lowest when threshold is 1% due to the trade-off between uniform interval size and instrumentation overhead. Then we use 1%, 0.1% and 0.05% as threshold in EV-based phase classification. The calculation of energy consumption of a measured benchmark or simpoint, and the energy estimation error rate are calculated as described in Section 7.9:

$$E = U \times \sum (I \times t)$$

$$error = \frac{|energy\_estimated - energy\_measured|}{energy\_measured}$$

Execution time estimation is similar to energy estimation.

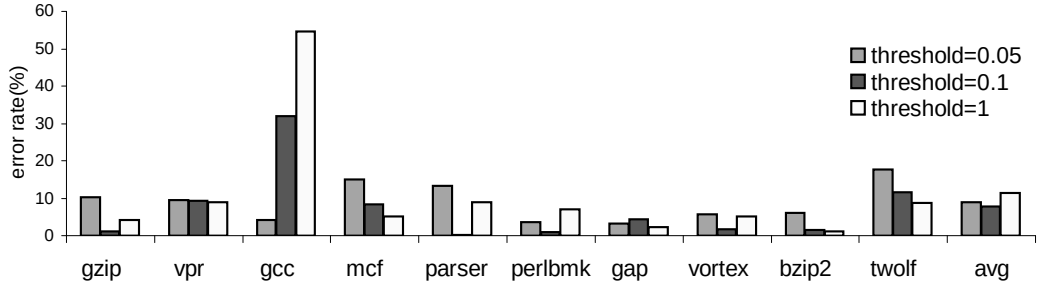
The error rate in total energy consumption estimation for each threshold is shown in Figure 8.5 (a).

For comparison, we perform the same operations for the 3 thresholds of BBV using the same pre-defined interval size, 30 million. The calculated error rates is shown in Figure 8.5 (b).

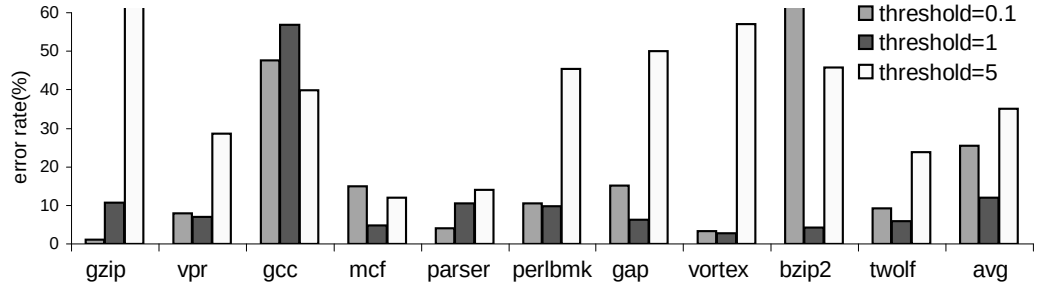
Figure 8.5 (c) shows the error rates of the infrequent basic block-based phase classification method using different program execution fingerprints. The error reported is that of the estimate using the threshold that delivered the minimum overall error for each method: 1% for BBVs, and 0.1% for EVs. The figure shows that EV performs better than BBV for almost all of the benchmarks. EV improves the estimation accuracy on average by 35%. One possible reason for the higher error rate of EV for some benchmarks is that we only record conditional edges taken during program execution. Some benchmarks have many unconditional edges, such as *jmp*, so it is possible that some information is lost in EV, although we significantly reduce the edge vector size. For example, method *sort_basket* of *mcf* is called 14683023 times and many of its edges are non-conditional edges. We can improve the phase classification accuracy through recording execution frequency of all edges, at the cost of larger edge vectors and slower phase classification. All of the following analysis and evaluation are for the experimental results of EV-based phase classification if there is no specification.

8.3.3 Time-dependent Power Behavior Similarity

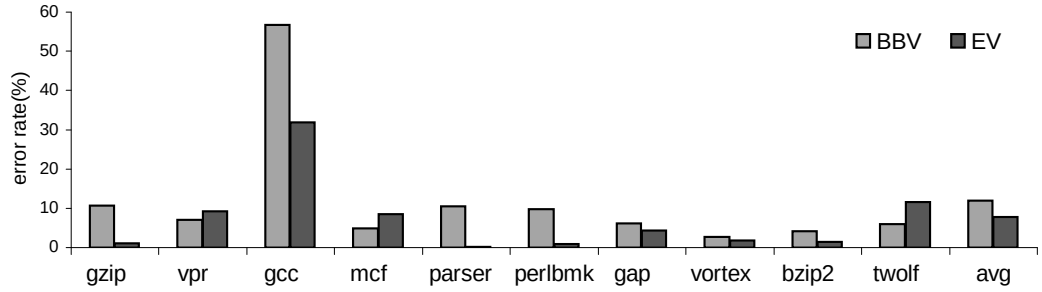
As mentioned in Section 8.2.2, we use the distance between the FFT results of their power curves to evaluate the similarity of two intervals in terms of power behavior. We use 4096 points in the Fast Fourier Transform. The maximum number of data points for a curve is 10,000 when the oscilloscope is in trigger mode. If the measured data points for the curve of an intervals is less than 4096, the curve is repeated to reach the number of frequencies. Figure 8.6 (a) shows the measured CPU current curves of two intervals from the same identified phase, while (b) shows that of two intervals from two different phases. Distance between the FFT



(a) Error rate in total energy consumption estimation when EV is used as interval fingerprint.



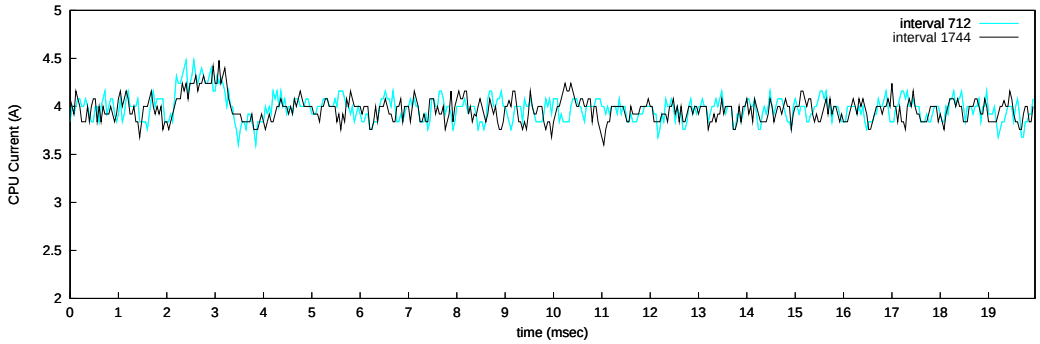
(b) Error rate in total energy consumption estimation when BBV is used as interval fingerprint.



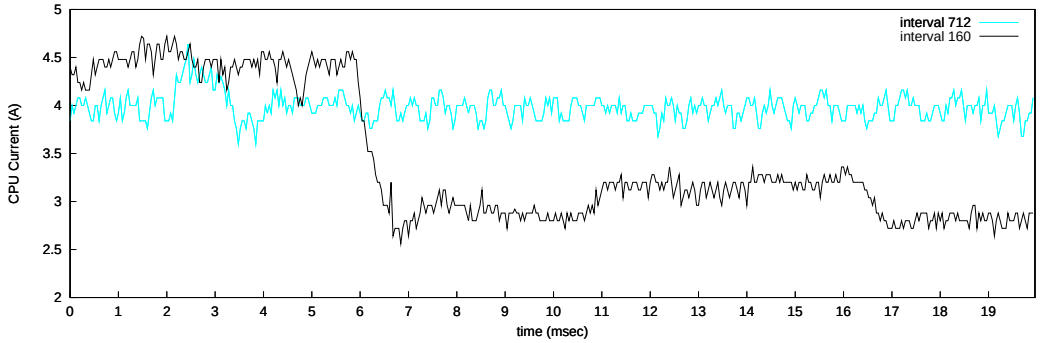
(c) Comparison between EV and BBV as interval fingerprint.

Figure 8.5: Error rates in total energy consumption estimation, EV vs. BBV

values is included to show the relation between time-dependent power behavior similarity and FFT distance. In Figure 8.6 (a), the upper curve uses the left *y* axis, while the other one use the right *y* axis, to avoid overlapping curves. The second column of each group in Figure 8.7 is the weighted FFT_{RMS} for each benchmark when EV is used for phase classification.



(a) Power curves of intervals from the same phase(distance=5.4).



(b) Power curves of intervals from different phases(distance=55.1).

Figure 8.6: Similarity between measured CPU current of intervals.

We measure the IPC using performance counters for each interval and do phase classification based on IPC to refine the EV-based phase classification. The third column in each group in Figure 8.7 is the weighted FFT_{RMS} for each benchmark when EV+IPC is used for phase classification. The similarity among the intervals is improved by 22% over using BBVs. Compared to the FFT distance between an interval and another interval from a different phase, the distance inside a phase is much smaller. This shows that the combination of EV and IPC enables us to classify intervals into phases in which the intervals have similar power behavior. Thus the power behavior of the whole program can be characterized by the measured behavior of the simpoints.

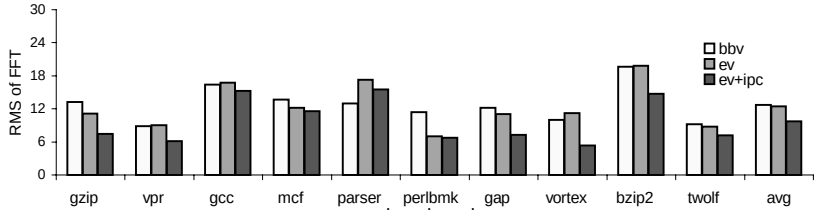


Figure 8.7: Root Mean Squared error of the FFT calculated based on RMS of FFT and the weight of each phase.

8.3.4 Interval Length Variance

Figure 8.8 shows the variance of interval length when different thresholds are used. Higher threshold results in more uniform interval size, but lower threshold has lighter instrumentation overhead. When threshold is 1%, this trade-off reaches the best value.

Figure 8.9 shows the weighted average of interval length variance of each phase for each benchmark when BBV and EV is used in phase classification respectively. A smaller number means the intervals of the same phase have very close interval size. Again it shows that EV is better for our infrastructure because, on average, it causes much lower interval length variance than BBV no matter which threshold is used. Again One possible reason for the higher RMS of EV for some benchmarks is that we only record conditional edges taken during program execution, which results in information loss. Although the possible reason is the same as in Section 8.3.2, the higher error rate or RMS happens to different benchmarks in these two set of experiments. The reason is that total power consumption is an average metric, if the energy consumption of the selected representative interval is close to the average energy consumption of all of the intervals in the same phase, the error rate should be low. While RMS of interval length is used to evaluate the similarity among intervals in the same phase, low error rate in total energy consumption does not mean this RMS value is small. This also applies to time-dependent power behavior and is also one of the motivation to use FFT to evaluate the time-dependent power behavior similarity among intervals in the same phase.

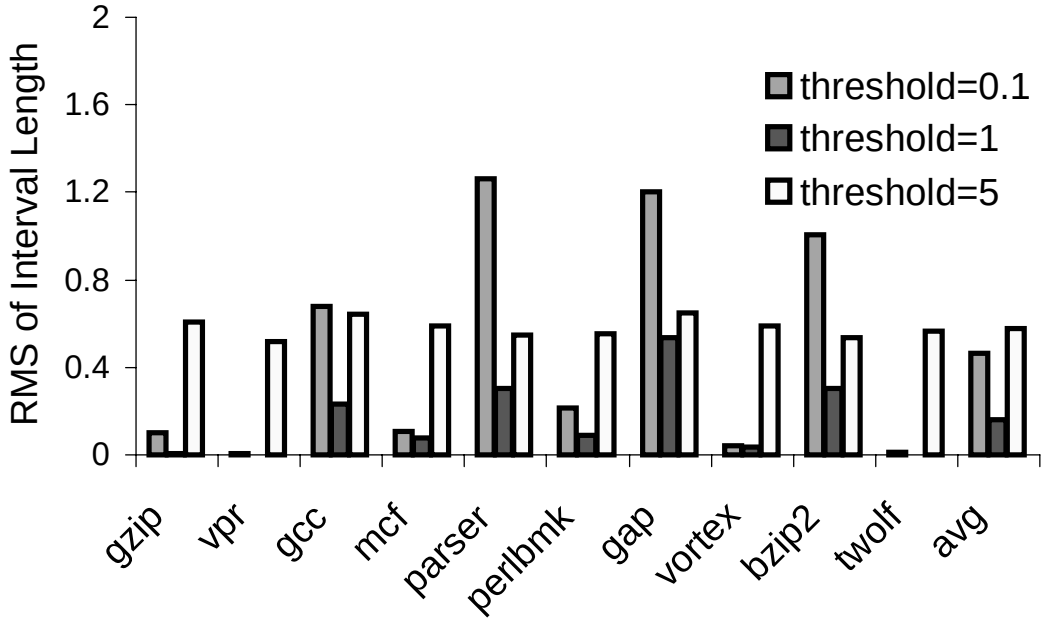


Figure 8.8: RMS error of the interval length of the whole benchmark.

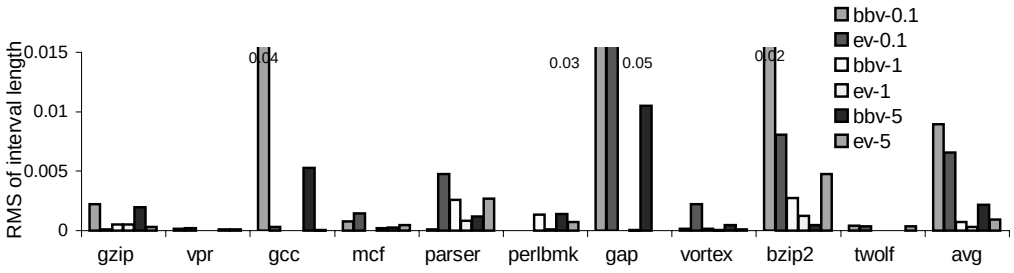


Figure 8.9: Weighted average of the RMS error of interval length in the same phase.

Chapter 9

Applications

9.1 Peak Power Optimization

In Section 1.1.1, Figure 1.1.1 shows the power behavior of two copies of a benchmark running on a hyper-threading machine. The benchmark is described in Figure 9.1. It finds the primes in the range of 1 to 20000, and then sleeps for 3 msec. This process is repeated 6 times. The power curve of first two iterations is not as apparent as the last four iterations due to the power behavior of the beginning of program execution. The CPU is very busy during the calculation to find primes and almost idle between the calculations.

In order to lower the CPU peak power, it should be avoided that two simultaneous programs are in their high-power region at the same time. These high-power regions can be treated as critical sections in Operating Systems. Some synchronization mechanism is required at the entry and exit of these regions. We use a semaphore to keep two simultaneous threads from running in high-power regions at the same time. At the entry of a high-power region, a **P** operation is performed to check if this region should be executed. At the exit, a **V** operation is performed to free the semaphore for future use. Thus at any time only one program is running in high-power region and peak power is controlled. The simplified implementation of the semaphore is shown in Figure 9.1.

Figure 1.1.1(c) shows that peak power is significantly reduced.

Our infrastructure can get detailed power behavior of the representative intervals with high resolution. Based on the power behavior similarity among intervals of the same phase that is already proved in Section 8.3, it can be used to find out high-power regions of programs and instrument the programs to be synchronized for lower peak power.

```
#define high_power 0       /* semaphore array index*/
int sem_init(void)
{
     /* create new semaphore set of 1 semaphore */
     int semid = semget (IPC_PRIVATE, 1, IPC_CREAT | 0600);
     /* initialize the semaphore to 1 */
     semctl (semid, high_power, SETVAL, 1);
     return semid;
}
/* perform a P or wait operation on a semaphorex */
void P(int semid, int index)
{
   struct sembuf sema_op[1];
   sema_op[0].sem_num = index;
   sema_op[0].sem_op   = -1;
   sema_op[0].sem_flg = 0;
   semop (semid, sema_op, 1);
}
/* perform a V or signal operation on semaphore */
void V(int semid, int index)
{
   struct sembuf sema_op[1];
   sema_op[0].sem_num = index;
   sema_op[0].sem_op   = 1;
   sema_op[0].sem_flg = 0;
   semop (semid, sema_op, 1);
}
int main (int argc, char** argv)
{   int semid;            /* identifier for a semaphore set */
     int i, max_x, x, max_y, y, count = 0, fd;
     max_x = atoi(argv[1]);
     semid = sem_init();
     for(i=0; i<6; i++){
        P(semid, high_power);
        for (x=2; x<=max_x+1; x++){
           /*determine if x is a prime*/
           y = 2;    max_y = sqrt(x);
           while (y <= max_y){
              if ((x % y) == 0){
            count++; break;
              }else
                 y = y + 1;
           }
        }
        V(semid, high_power);
        usleep(3000);
     }
     printf("found %d primes\n", count);
     semctl (semid, 0, IPC_RMID);    /*remove semaphore*/
     return;
}
```

Figure 9.1: Implementation and use of semaphore in peak power optimization.

9.2 DVFS Metric and Threshold Selection

DVFS is an important technique for runtime power optimization. DVFS is supported on various levels, as described in Section 2.5. A typical DVFS method periodically profiles some runtime events to predict the CPU computation ability requirement of the system in the next time interval and make CPU frequency scaling decisions, that is, what is the CPU power state for the next time interval? This requirement is usually determined by a metric and a pre-defined threshold. For example, if IPC is the metric, the CPU computation ability requirement is predicted to be high if the predicted IPC is higher than 1.0. If high CPU computation ability is not needed, the CPU frequency can be lowered to save energy consumption without significant performance loss.

Using the right runtime event and its corresponding threshold is critical for the efficiency of a DVFS policy. Our infrastructure identifies the representative intervals in terms of power behavior and measures the objective power consumption of any program region. Through profiling runtime events of these selected intervals and evaluating their power behavior in different power states, we can find out the right metric and its threshold to use in DVFS.

On an Intel Conroe E6600, we run the benchmarks listed in Table 7.1 under different CPU frequencies and measure the power consumption of each representative interval found by our current phase classification method. All benchmarks are compiled by GCC 4.1.1 and the OS is Fedora 6. E6600 has four power states, the frequency is 2.4GHz, 2.1GHz, 1.8GHz, and 1.6GHz, respectively. We profile some event counters, including *uop* per cycle (UPC) and memory access per *uop* (MPU), when CPU is running at 2.4GHz. Since E6600 is a dual core machine but we only investigate single program execution, the tested benchmark is always running on one of the two cores through using CPU affinity. The unused core is always set to the 1.6GHz to ensure that the frequency of the core in use is effective.

9.2.1 Selecting DVFS Metric

Power consumption of the same interval when different CPU frequencies are used is compared to estimate if we can reduce energy consumption without significant performance loss when CPU frequency is scaled to a lower value. To take performance into consideration, we use

power-delay product (PDP), instead of power, as a metric in evaluating frequency/voltage scaling benefit. If an interval has lower PDF when CPU runs at a lower frequency, it benefits from frequency scaling down during program execution.

DVFS metric selection includes the following steps:

- Run phase classification to get representative intervals for each benchmark.
- Profile runtime events for the selected intervals, including number of cycles, number of retired micro-instructions and number of memory accesses.
- Measure the power consumption and execution time of each selected interval under different frequencies.
- Calculate the PDP for each measured interval. All PDP values are normalized by the PDP values under the highest frequency, 2.4GHz. If the normalized PDP is less than 1, the interval benefits from running in a lower frequency.
- Calculate UPC and MPU for each selected interval based on event profiling results.
- Find the right metric and the corresponding threshold through analyzing the metric values, here UPC or MPU, for the intervals that benefit from a lower CPU frequency and the other intervals, as shown in the following graph.

Figure 9.2 shows the experimental result. The representative intervals of 10 SPEC2000 integer benchmarks are partitioned into 2 groups. The triangles are intervals that have lower PDP and the circles are those that have higher PDP. The result shows that UPC is not a good metric for DVFS by itself, since it is hard to delimit the UPC of intervals based on whether they benefit from lower CPU frequency or not. Although the triangles cover almost all possible values of MPU, it is easy to find a MPU threshold, say 0.01 in this figure. When the MPU of an interval is higher than 0.01, it benefits from lower CPU frequency with very high probability. An even more conservative DVFS policy can use the combination of the two metrics. For example, it lowers the CPU frequency for intervals with high MPU and low UPC. An analytical procedure to find the right metric and threshold is in the future work.

Since the selected intervals are representative in terms of power behavior, the metrics and thresholds found by this method can be used for whole program power optimization. The

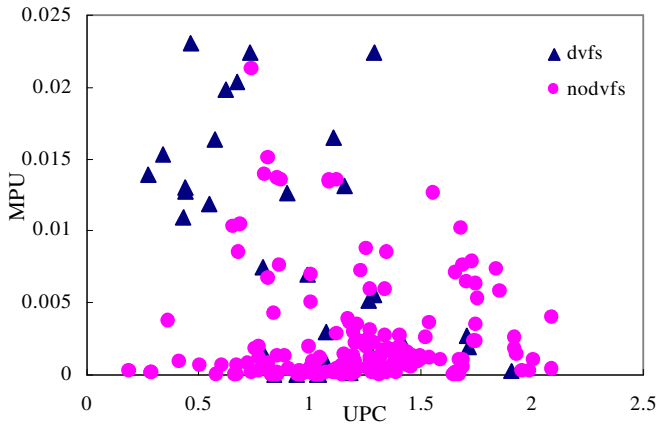


Figure 9.2: MPU and UPC distribution for all representative intervals selected by our phase classification.

exploration space of a threshold is usually big. With our infrastructure, it is easy to compare the effect of different metrics and to find a good threshold for a metric.

9.2.2 Applying Selected Metric and Threshold in DVFS

To show that we can find the right metric and the corresponding threshold for dynamic voltage/frequency scaling, we examine the metrics shown in Figure 9.2. As discussed in Section 9.2.1, MPU is a better metric than UPC, and 0.01 is a good threshold for MPU. Since most representative intervals do not benefit from a lower CPU frequency when MPU is lower than 0.06, we increase CPU frequency to the highest level when MPU is lower than 0.06.

E6600 has a model-specific register (MSR) that controls the frequency and voltage of the processor. We change CPU frequency through writing a specific value to this MSR using our device driver. Due to the high CPU computation requirement of the benchmarks used in our experiments, only two CPU frequencies, 2.4GHz and 2.1GHz are used. Increasing CPU frequency means changing it from 2.1GHz to 2.4GHz, or keeping it unchanged if the frequency is 2.4GHz. Decreasing CPU frequency means changing it from 2.4GHz to 2.1GHz, or keeping it unchanged if the frequency is 2.1GHz.

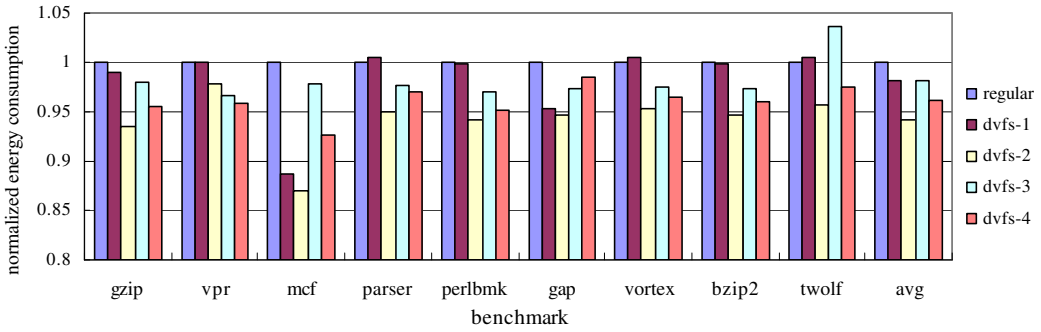
Table 9.1 shows the metrics and threshold used in our validation experiments. The thresholds are derived from Figure 9.2. A simple time-interval-based DVFS method is used. It checks

Table 9.1: Validation experiments for DVFS metric and threshold.

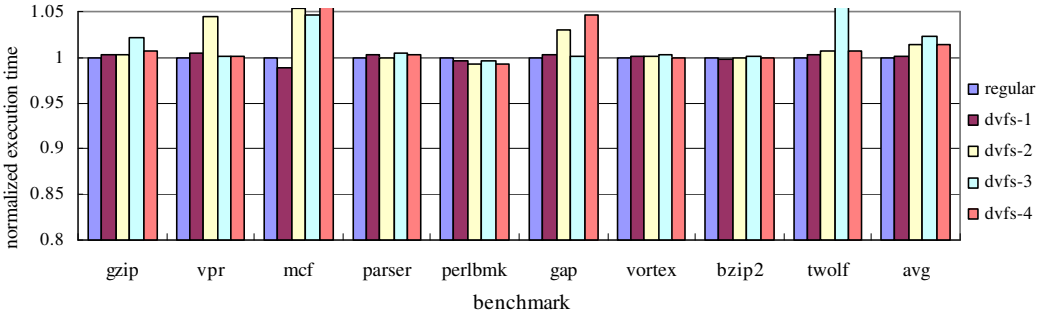
Method	Description
dvfs-1	If MPU $\geq$ 0.012, decrease frequency; if MPU $<$ 0.006, increase frequency; for other MPU values, keep the current frequency
dvfs-2	If MPU $\geq$ 0.01, decrease frequency; if MPU $<$ 0.006, increase frequency; for other MPU values, keep the current frequency
dvfs-3	If UPC $\geq$ 0.8, decrease frequency; otherwise, increase frequency
dvfs-4	If MPU $\geq$ 0.1, decrease frequency; if MPU $<$ 0.006 and the number of executed *uops* is smaller than the last time interval, increase frequency; for other MPU values, if the number of executed *uops* is smaller than the last time interval, increase frequency; otherwise, decrease frequency

the metric of the current interval and decides the CPU frequency for the next time interval.

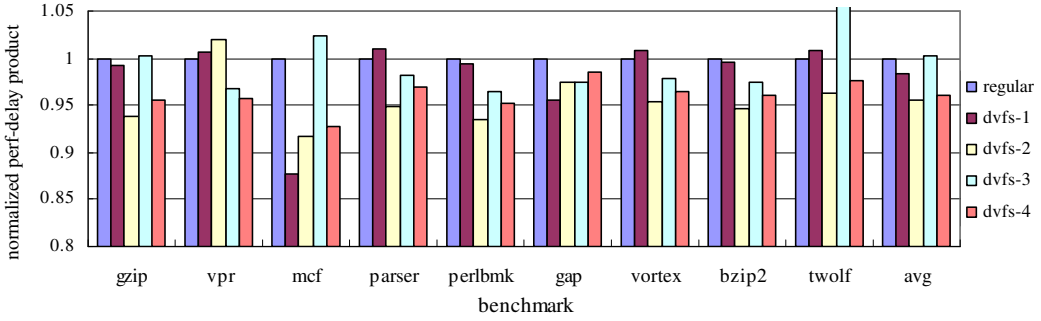
Figure 9.3 shows the experimental results. The *regular* column in each figure is the measurement result without DVFS. The other four are corresponding to the four methods described in Table 9.1, respectively. All measurement results are normalized by the result without DVFS. Figure 9.3 (a) shows that using 0.01 as MPU threshold brings us more energy saving than 0.012, but performance loss is higher because lower CPU frequency is used for more intervals, as shown in Figure 9.3 (b). Since we evaluate whether an interval benefits from lowering CPU frequency based on the measured PDF, the better MPU threshold inferred from Figure 9.2 has a lower PDP. A UPC threshold 0.8 is used because it results in a better classification of the representative intervals in terms of DVFS benefit. For *twolf*, this DVFS method results in higher energy consumption with high performance loss. The possible reason is that the profiled UPC of twolf is consistently low, but there are few memory access. We also examined another threshold, 0.5. The experimental result is similar to that of threshold 0.8. Using UPC as a metric results in more intervals suffering from false positive DVFS decision. That is, an interval with low UPC is treated as one that benefits from lower CPU frequency, but the truth is the opposite. *dvfs-4* is more complicated than the other three. An additional event, number of retired *uops*, is used to avoid high performance loss. Since the CPU utilization if already high for these benchmarks, this method results in higher performance loss and less energy saving than *dvfs-2* due to more computation at the end of each time interval.



(a) Normalized energy consumption.



(b) Normalized execution time.



(c) Normalized power-delay product.

Figure 9.3: Experimental results of the DVFS methods in Table 9.1.

The experimental results show that our infrastructure can be used in finding good DVFS metrics and thresholds. It can help researchers in observing the relation between power optimization and various runtime events. Furthermore, the detailed power behavior obtained for each representative interval shows the impact of any power optimization method on program power behavior.

9.3 Program Power Behavior Understanding

Characterizing the time-dependent power behavior of whole program execution is one objective of our infrastructure. Measured power curves can give us a better understanding of dynamic program power behavior if we can find the source code corresponding to a specific power curve. Semantic relation between measurement result and source code can also help in understanding the power behavior of program structures, such as procedures and loops. Interval partitioning method based on fixed interval length or infrequent basic blocks makes it very hard to get direct relation between the measured power curve of a representative interval and the source code executed in this interval. We implemented a two-level profiling in the Camino compiler to profile procedures and loops in addition to EV and event counter profiling, which is illustrated in Section 4.2.5.

Although our procedure and loop profiling is performed when the EVs are profiled, which means only the control-flow information is collected, we want to put a limit on the memory space used by the profiling. Due to the high frequency of some short procedures and loops, recording the information for each invocation of such a procedure or loop is time- and memory-consuming. Furthermore, procedures and loops of size that is much smaller than interval size are hard to be analyzed based on the measurement result. Smaller interval size can be used to get the dynamic power behavior of more procedures and loops. We set two profiling thresholds. One is for the execution frequency of a procedure or loop, and the other is for procedure or loop size. When the invocation frequency of a procedure or loop reaches the first threshold, its average number of instructions is compared to the second threshold. If higher, this procedure or loop continues to be profiled, otherwise, it is no longer profiled. The size threshold is relative to the interval size, such that the power behavior of a profiled procedure or loop is easy to

characterize.

Using the combination of the Interval Vector (IV) of a procedure/loop, the phase classification result, and the measured power behavior of the representative interval, we can get the power behavior of this procedure/loop. Definition of IV is in Section 4.2.5. Figure 9.4 shows the profiled IVs of a loop of *181.mcf*. Each number is the phase number for the corresponding interval. The difference among the IVs indicates that different iterations of the same loop have different power behavior or even they execute different source code. We can evaluate if a procedure/loop has consistent or variant power behavior based on its IVs.

```
{13:1  14:7  18:14  19:7  24:3  25:10}
{5:19  13:1  14:6  18:1  24:5  25:6}
{5:17  14:2  18:2  19:1  25:10}
{5:19  18:8  24:4  25:2}
{5:19  16:1  18:5  24:6}
{5:22  6:1  12:2  15:1  18:5}
{5:23  6:1  16:1  18:2}
{5:27  6:1  16:1}
{5:27  6:1  16:1}
{5:27  6:1  16:1}
```

Figure 9.4: Interval Vector of a loop in method *price_out_impl* of *mcf*.

As described in Section 4.2.5, it is easy to find the source code corresponding to this loop based on identification of the loop and the recorded *procname_no* information. Sometimes GCC performs reordering optimization, so it is necessary to check the first BB of the loop in the source code to ensure the sequence number is the same in both the assembly code and the source code in C. The source code for this profiled loop is the second *for* loop in method *price_out_impl*. Figure 9.5 shows that control flow graph of this loop, which is generated by our Camino compiler. Since the identification of an edge is related to the hash value of the BB that this edge originates from, it is easy to figure out the paths taken in different stages of program execution based on EVs. The profiled EVs for each IV show that in each invocation of the loop, which may have many iterations, the path taken in the first several iterations is different from the one taken in the later iterations. Arcs are inserted into a graph and replaced later, shown as the blue path (12-16-17-8) and the red path (12-13-14-15), respectively. This is the possible reason that there are different EVs in an IV although the same loop is profiled.

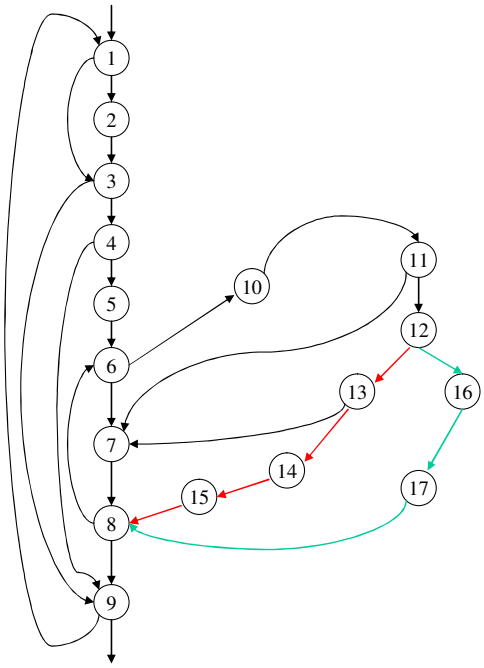
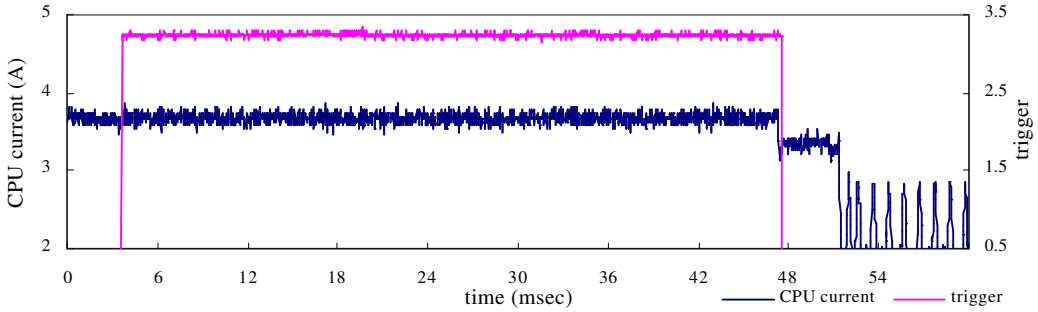
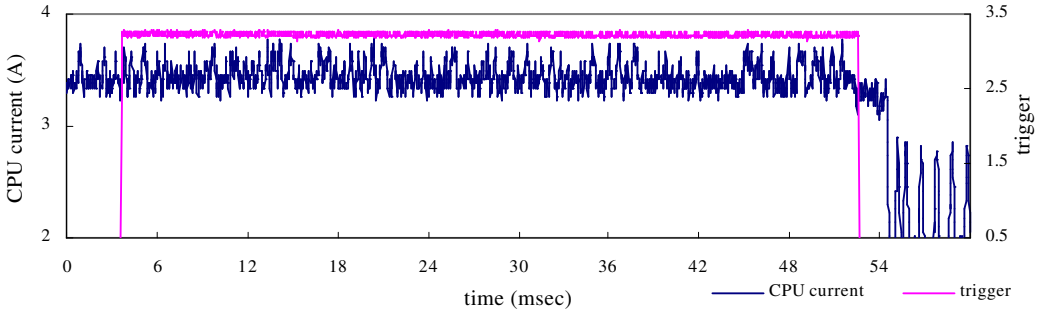


Figure 9.5: CFG of a loop in method *price_out_impl* of *mcf*.

Variance in power behavior is even higher due to runtime events. In this example, intervals in phase 14 have the highest MPU and those in phase 5 have the lowest MPU, as shown in Figure 9.6. This fact shows that sometimes loop can not be used as an optimization unit for DVFS. Some DVFS policies profile the CPU utilization of a loop in there first several iterations and then perform DVFS on it. For the loop shown here, if a policy determines that this loop has many memory accesses and the CPU frequency should be scaled down, it will cause performance loss since many iterations have very low MPU as we measured; if a policy profiles many iterations of the loop and concludes that CPU frequency should not be scaled down for this loop, it will lose the opportunity to save energy during the iterations with high MPU. Figure 9.7 shows the measured power curve of the two intervals in Figure 9.6 when a lower CPU frequency, 2.1GHz, is used.

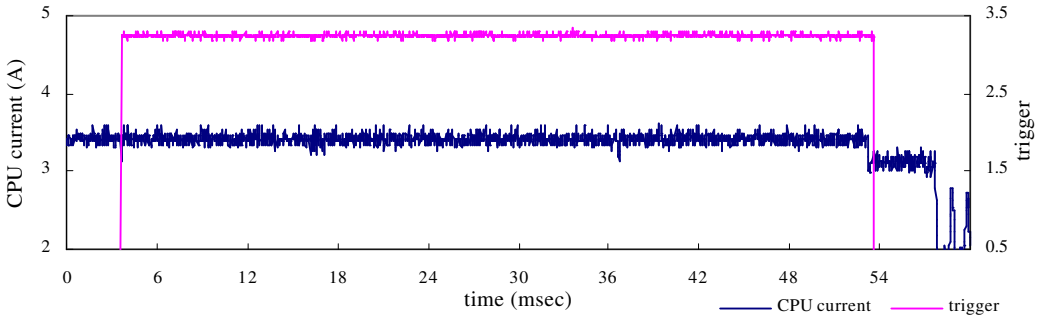


(a) representative power curve of phase 5.

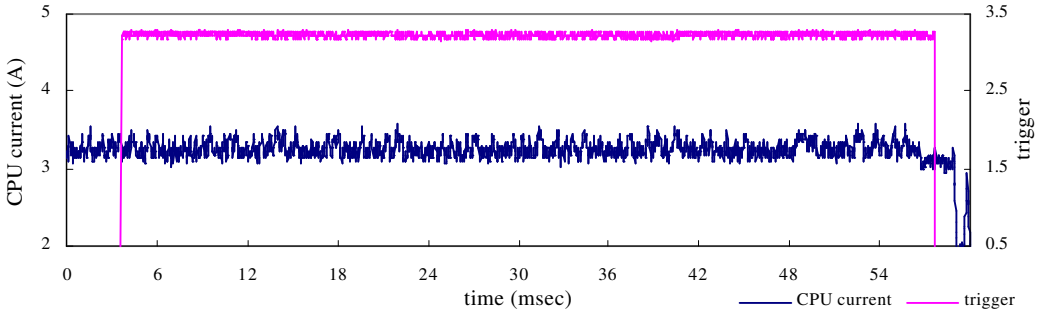


(b) representative power curve of phase 14.

Figure 9.6: Power behavior of different phases of *mcf*, CPU frequency = 2.4GHz.



(a) representative power curve of phase 5.



(b) representative power curve of phase 14.

Figure 9.7: Power behavior of different phases of *mcf*, CPU frequency = 2.1GHz.

Chapter 10

Conclusion

Detailed time-dependent power behavior is useful in both program power behavior analysis and power optimization evaluation. However, existing simulation and physical power measurement methods are inefficient because of high cost in time and space, imprecision, or lack of semantic meanings in measured power curve.

This thesis describes our infrastructure for detailed time-dependent program power behavior characterization. It enables instrumentation on various levels of program assembly code, and provides routines for control-flow and runtime event profiling. Our new program power phase classification method partitions program execution into intervals based on infrequently executed basic blocks, uses the combination of Edge Vector and runtime events as the fingerprint of each interval, classifies intervals into phases, and selects a representative interval for each phase. Our power measurement method used in this thesis overcomes hardware limitation and provides precise and objective power behavior for any measured program region. Validation experiments on real systems show that using infrequent basic blocks to demarcate intervals results in negligible instrumentation overhead in identifying intervals during program execution, and our new phase classification method can find the representative intervals in terms of time-dependent power behavior. Our two-level profiling enables the semantic relation between the measured time-dependent power behavior and the corresponding source code.

This infrastructure can be used in not only program power behavior characterization, but also power/performance optimization opportunity observation and optimization evaluation. It can be used to find the good metric and the corresponding threshold for DVFS. Its low instrumentation overhead and low error phase classification make it a useful tool for program power/performance behavior analysis and optimization research on real systems.

This thesis work investigated static program instrumentation, control-flow and runtime

event profiling, program power phase behavior, phase classification and program power behavior characterization.

10.1 Static Program Instrumentation Tool

Assembly-level instrumentation supported by our infrastructure is useful in program profiling and power measurement. It is easy to identify control-flow structures, such as basic blocks and procedures, in assembly code. There are many static and dynamic instrumentation tools. Dynamic instrumentation can instrument dynamically generated code, but its instrumentation overhead is too high for our purpose of identifying an interval during program execution and measuring its power behavior. Static instrumentation has advantages in our infrastructure, since only the selected infrequent basic blocks are instrumented for interval power measurement, and it is needed to profile procedures and loops for program detailed power behavior analysis and understanding. It also enables the mapping between the measurement result and the corresponding source code.

Our Camino compiler is used as the instrumentation tool in our infrastructure. Currently Camino supports basic block count and edge count profiling, inter-procedural path profiling, procedure and loop profiling, and EV/BBV profiling for SimPoint-like phase classification. Implementation of new profiling or instrumentation is very easy because of the clear internal representation of Camino. Camino is currently used as a test bed for low-level performance, power or energy optimizations. We successfully use Camino in charactering program power behavior with low error rates using a SimPoint-like phase classification method.

10.2 Accurate Program Power Behavior Phase Classification

There is phase behavior in physical power measurement of program execution. Although control-flow-based classification achieves low error rate in estimating some overall metrics, such as IPC, branch misprediction, and energy consumption, it is not a good classification in terms of time-dependent power behavior. Different executions of the same source code may generate different power curves due to runtime events. Similarly, runtime-event-based classification can not efficiently characterize program time-dependent power behavior, since two

intervals with the same runtime event reading may execute totally different source code and their power curves are largely different.

Compared to BBVs, EVs give us more information of program execution since they track the taken edges. Intervals with the same BBV may have different EVs. Through experiments on real systems, we show that EV is better than BBV as interval fingerprint used in phase classification. Combination of EV and IPC can be used as interval fingerprint in time-dependent power behavior classification. A robust validation shows that our new two-stage phase classification method can find representative intervals in terms of power behavior. Power behavior similarity is evaluated as the distance between the FFT results of interval power curves. Phase classification accuracy is evaluated as the RMS of FFT results of intervals that are classified into the same phase.

10.3 Infrequent Basic Block-based Interval Partitioning

In order to characterize the power behavior of whole program execution, we measure the time-dependent power behavior of the representative intervals selected by our phase classification method. Different from simulation, physical power measurement is sensitive to instrumentation overhead. Objective measurement requires dynamic identification of the beginning and end of an interval during program execution. We use infrequent basic blocks to demarcate program execution into intervals, and use a new similarity evaluation method to adapt to the variable length intervals. By instrumenting only the basic blocks that are necessary for the identification of an interval, we achieve negligible instrumentation overhead and get the objective power curve of any measured interval.

10.4 Dynamic Voltage/Frequency Scaling

Voltage/frequency scaling is supported by more and more computing systems. There are many policies of determining when to scale up/down CPU voltage/frequency, different in scaling unit and metric. Using our infrastructure, we compared two metrics and found that MPU is a better metric than UPC for DVFS decision making. Although high MPU results in low UPC, low UPC is not always because of memory latency. Another possible reason is data dependency.

If memory access rate is low, slowing down CPU frequency does not hide any memory access delay, instead, it harms performance with out energy saving.

It is possible that time-interval-based DVFS is better than procedure/loop-based DVFS. Our experimental results show that there are phases in program time-dependent power behavior and the same procedure/loop has largely different power behavior in different invocations, exposing different DVFS opportunities as shown in Section 9.3.

References

[1] http://www.cs.wisc.edu/~mscalar/simplescalar.html.

[2] http://www.eecs.umich.edu/~panalyzer/.

[3] http://www.cs.ucsd.edu/~calder/simpoint/index.htm.

[4] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*.

[5] Anantha P. Chandrakasan Amit Sinha. Jouletrack - a web based tool for software energy profiling the Seville statement on violence. *Design Automation Conference*, 2001.

[6] Bryan Black and John Paul Shen. Calibration of microprocessor performance models. *IEEE Computer, p59-65*, 1998.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. *In 1st International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

[8] Bruno De Bus, Dominique Chanet, Bjorn De Sutter, Ludo Van Put, and Koen De Bosschere. The design and implementation of fit: a flexible instrumentation toolkit. *Proceedings of the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering(PASTE)*, pages 29–34, 2004.

[9] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*, October 1994.

[10] Margaret Martonosi Canturk Isci. Runtime power monitoring in high-end processors: Methodology and empirical data. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture, p93*, 2003.

[11] Margaret Martonosi Canturk Isci. Runtime power monitoring in high-end processors: Methodology and empirical data. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, page 93, 2003.

[12] J. W. Chen, M. Dubois, and P. Stenstrm. Integrating complete-system and user-level performance/power simulators: The simwattch approach. *In Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2003.

[13] Eric Chi, A. Michael Salem, and R. Iris Bahar. Combining software and hardware monitoring for improved power and performance tuning. *Proceedings of the Seventh Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'03)*, 2003.

[14] Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. Reducing state loss for effective trace sampling of superscalar processors, October 1996.

[15] Margaret Martonosi David Brooks, Vivek Tiwari. Wattch: a framework for architectural-level power analysis and optimizations. *Proceedings of the 27th annual international symposium on Computer architecture, p83-94*, 2000.

[16] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, page 220, 2003.

[17] Jason Flinn and M. Satyanarayanan. Powerscope: A tool for profiling the energy usage of mobile applicationthe. *Second IEEE Workshop on Mobile Computing Systems and Applications*, 1999.

[18] Dirk Grunwald, Philip Levis, Keith I. Farkas, Charles B. Morrey III, and Michael Neufeld. Policies for dynamic clock scheduling. *In Fourth Symposium on Operating System Design and. Implementation(OSDI 2000)*, pages 73–86, October 2000.

[19] Vivek Haldar, Christian W. Probst, Vasanth Venkatachalam, and Michael Franz. Virtual machine driven dynamic voltage scaling. *Technical Report, University of California, Irvine*, 2003.

[20] Kim Hazelwood and David Brooks. Eliminating voltage emergencies via microarchitectural voltage control feedback and dynamic optimization. *International Symposium on Low-Power Electronics and Design*, August 2004.

[21] Alvin R. Lebeck Amin Vahdat Heng Zeng, Carla S. Ellis. Ecosystem: Managing energy as a first class operating system resource. *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, 2002.

[22] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI'03)*, pages 38–48, 2003.

[23] Chunling Hu, Daniel A. Jiménez, and Ulrich Kremer. Toward an evaluation infrastructure for power and energy optimizations. *19th International Parallel and Distributed Processing Symposium (IPDPS 2005, Workshop 11), CD-ROM / Abstracts Proceedings*, April 2005.

[24] Chunling Hu and Jack Liu. Vm + dbs: Dynamic power consumption and performance optimization. *Proceedings of the International Conference on Thermal Issues in Emerging Technologies Theory and Application (ThETA)*, January 2007.

[25] Chunling Hu, John McCabe, Daniel A. Jiménez, and Ulrich Kremer. The camino compiler infrastructure. *SIGARCH Comput. Archit. News*, 33(5):3–8, 2005.

[26] Chunling Hu, John McCabe, Daniel A. Jiménez, and Ulrich Kremer. Infrequent basic block-based program phase classification and power behavior characterization. *Proceedings of The 10th IEEE Annual Workshop on Interaction between Compilers and Computer ArchitecturesThe Camino Compiler Infrastructure*, 2006.

[27] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide*.

[28] Intel. *StrongARM SA-110/21285 Evaluation Board*.

[29] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, 2006.

[30] Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-6)*, 2003.

[31] Canturk Isci and Margaret Martonosi. Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. *In 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, Febrary 2006.

[32] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. *Proceedings of the conference on Design, Automation and Test in Europe (DATE'01)*, pages 190–196, 2001.

[33] Chunling Hu Daniel A. Jiménez and Ulrich Kremer. Efficient power behavior characterization. *2007 International Conference on High Performance Embedded Architectures and Compilers*, January 2007.

[34] Daniel A. Jiménez. Code placement for improving dynamic branch prediction accuracy. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI'05)*, pages 107–116, June 2005.

[35] David A. Patterson John L. Hennessy. *Morgan Kaufmann*, 2002.

[36] AJ KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the spec 2000 benchmark suite for simulation-based computer architecture research. *Workload Characterization of Emerging Computer Applications*, pages 83–100, 2001.

[37] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. *In the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, pages 135–146, 2005.

[38] J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. *In the Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'04)*, pages 57–67, 2004.

[39] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. *International Symposium on Microarchitecture*, pages 330–335, 1997.

[40] Tao Li and Lizy Kurian John. Run-time modeling and estimation of operating system power consumption. *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 31(1):160–171, June 2003.

[41] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, 2005.

[42] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'05)*, pages 190–200, 2005.

[43] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 14–27, 2003.

[44] Srilatha Manne, Artur Klauser, and Dirk Grunwald. Pipeline gating: speculation control for energy reduction. *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA'98)*, pages 132–141, 1998.

[45] D. Marculescu. On the use of microarchitecture-driven dynamic voltage scaling. *In Workshop on Complexity-Effective Design*, June 2000.

[46] M. I. Irwin H. S. Kim W. Ye N. Vijaykrishnan, M. Kandemir. Energy-driven integrated hardware-software optimizations using simplepower. *Proceedings of the 27th annual international symposium on Computer architecture, p95-106*, 2000.

[47] Priya Nagpurkar and Chandra Krintz. Phase-based visualization and analysis of java programs. *Science of Computer Programming*, 59(1-2):64–81, 2006.

[48] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. *In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, page 244, September 2003.

[49] Kambiz Rahimi. Minimizing peak power in synchronous logic circuits. *GLSVLSI '07: Proceedings of the 17th great lakes symposium on Great lakes symposium on VLSI*, pages 247–252, 2007.

[50] Stephen W. Keckler Rajagopalan Desikan, Doug Burger. Measuring experimental error in microprocessor simulation. *Proceedings of the 28th annual international symposium on Computer architecture,p266-277*, 2001.

[51] R. Rao and S. Vrudhula. Battery optimization vs energy optimization: which to choose and when? *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 439–445, 2005.

[52] Vijay Janapa Reddi, Alex Settle, and Daniel A. Connors. Pin: A binary instrumentation tool for computer architecture research and education. *Proceedings of the Workshop on Computer Architecture Education*, June 2004.

[53] K. Chen S. Huang, K. Cheng and T. Lee. A novel methodology for transistor-level power estimation. *ISLPED'96: Proceedings of the 1996 international symposium on Low power electronics and design*, pages 67–72, 1996.

[54] Greg Semeraro, David H. Albonesi, Steven G. Dropsho, Grigorios Magklis, Sandhya Dwarkadas, and Michael L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367, 2002.

[55] Xipeng Shen, Yutao Zhong, and Chen Ding. Locality phase prediction. *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, pages 165–176, 2004.

[56] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, pages 3–14, 2001.

[57] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[58] Dirk Grunwald Soraya Ghiasi. A comparison of two architectural power models. *Proceedings of the First International Workshop on Power-Aware Computer Systems, P137-152*, 2000.

[59] Brinkley Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, 2002.

[60] Ram Srinivasan, Jeanine Cook, and Shaun Cooper. Fast, accurate microarchitecture simulation using statistical phase detection. *Proceedings of The 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'05)*, 2005.

[61] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205, November 1994.

[62] C-L. Su, C-Y. Tsui, and A.M. Despain. Low power architecture and compilation techniques for high-performance processors. *IEEE COMPCON*, pages 489–498, February 1994.

[63] Mary Jane Irwin N. Vijaykrishnan et al. Sudhanva Gurumurthi, Anand Sivasubramaniam. Using complete machine simulation for software power estimation: The softwatt approach. *International Symposium on High Performance Computer Architecture(HPCA)*, 2001.

[64] M.C. Toburen, T. Conte, and M. Reilly. Instruction scheduling for low power dissipation in high performance microprocessors. *Power Driven Microarchitecture Workshop*, June 1998.

[65] Carla Schlatter Ellis Todd L. Cignetti, Kirill Komarov. Energy estimation tools for the palm. *International Workshop on Modeling Analysis and Simulation of Wireless and Mobile Systems*, 2000.

[66] Madhavi Valluri and Lizy John. Is compiling for performance == compiling for power? *The 5th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-5)*, January 2001.

[67] Qiang Wu, Margaret Martonosi, Douglas W. Clark, V. J. Reddi, Dan Connors, Youfeng Wu, Jin Lee, and David Brooks. A dynamic compilation framework for controlling microprocessor energy and performance. *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 271–282, 2005.

[68] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling, June 2003.

[69] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: Opportunities and limits. *In the Proceedings of Programming Language Design and Implementation (PLDI 2003)*, June 2003.

[70] H-S. Yun and J. Kim. Power-aware modulo scheduling for high-performance VLIW. *International Symposium on Low Power Electronics and Design (ISLPED'01)*, August 2001.

Vita

Chunling Hu

Education

2008	Ph.D. Computer Science, Rutgers University
2007	M.S. Computer Science, Rutgers University
2001	M.S. Computer Science, Beijing Univ of Posts&Telecom
1998	B.E. Computer Communications, Beijing Univ of Posts&Telecom

Research Experience

2004-2007	Research Assistant, Department of Computer Science, Rutgers Univ
2006	Technical Graduate Intern, Intel Corporation

Teaching Experience

2004	Computer Architecture (undergraduate level)
2003	Computer Architecture (graduate level)
2001-2002	Internet Technology (undergraduate level)

Publications

- Chunling Hu, Daniel A. Jiménez, Ulrich Kremer. Combining Edge Vector and Event Counter for Time-dependent Power Behavior Characterization. *Invited paper to LNCS Transactions on High-Performance Embedded Architectures and Compilers*(to appear).

- Chunling Hu, Daniel A. Jiménez, Ulrich Kremer. An Evaluation Infrastructure for Power and Energy Optimizations. *International Journal of Embedded Systems (IJES)* (to appear).

- Chunling Hu, John McCabe, Daniel A. Jiménez and Ulrich Kremer. The Camino Compiler Infrastructure, *ACM SIGARCH Computer Architecture News*, Vol. 33, Issue 5, Dec. 2005.

- Chunling Hu, Jack Liu. JVM+DBS: Dynamic Power Consumption and Performance Optimization. *International Conference on Thermal Issues in Emerging Technologies, Theory and Application-ThETA*, Jan. 2007.

- Chunling Hu, Daniel A. Jiménez, Ulrich Kremer. Efficient Program Power Behavior Characterization. *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC2007)*, Jan. 2007.

- Chunling Hu, John McCabe, Daniel A. Jiménez and Ulrich Kremer. Infrequent Basic Block-based Program Phase Classification and Power Behavior Characterization. *Proceedings of The10th IEEE Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT-10)*, Feb. 2006.

- Chunling Hu, John McCabe, Daniel A. Jiménez and Ulrich Kremer. The Camino Compiler Infrastructure, *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications (WBIA), held in conjunction with PACT2005*, Sept. 2005.

- Chunling Hu, Daniel A. Jiménez, Ulrich Kremer. Toward an Evaluation Infrastructure for Power and Energy Optimizations. *The First Workshop on High-Performance, Power-Aware Computing (HP-PAC)*, Apr. 2005.