Program Phase Classification for Power Behavior Characterization

Chunling Hu John McCabe Daniel A. Jiménez Ulrich Kremer

Department of Computer Science
Rutgers University, Piscataway, NJ 08854
{chunling, jomccabe, djimenez, uli}@cs.rutgers.edu

Abstract

Power and energy optimizations include both reducing total energy consumption and improving time-dependent power behavior. Fine-grained program power behavior is useful in evaluating power optimizations and observing power optimization opportunities. In this paper, we present a physical measurement-based infrastructure for program time-dependent power behavior characterization and optimization evaluation. This infrastructure does basic block profiling, a SimPoint-like phase classification, and precise physical measurement for selected representative program execution slices. Our phase classification method uses infrequently executed basic blocks to demarcate intervals and uses a user-specified interval size to control the length of the resulting intervals. This partitioning method incurs low instrumentation overhead for dynamic identification of simpoints during program execution. This physical measurement infrastructure enables precise power measurement for any program execution. This infrastructure is built on our Camino *compiler, which supports static instrumentation on various levels. This infrastructure can be used to characterize detailed program power behavior, as well as evaluate compiler and architecture level power and energy optimizations.*

We validate the feasibility of this phase classification method in power behavior characterization through the physical measurement of 10 SPEC CPU2000 integer benchmarks on a Pentium 4 machine. Experimental results show that our method can also find representative slices for power behavior estimation, and the physical measurement with negligible instrumentation overhead enables us to estimate detailed time-dependent program power behavior.

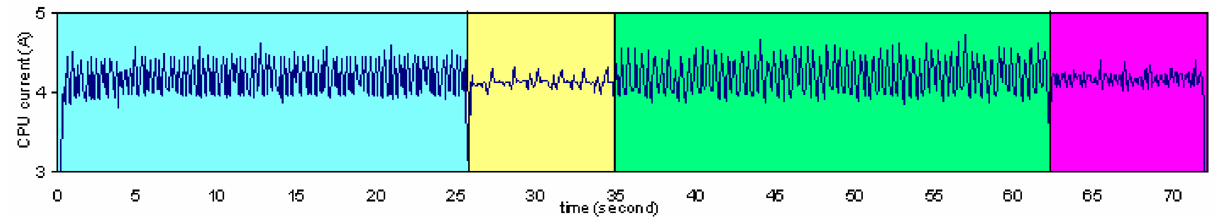
1. Introduction

Research in power and energy optimizations focuses not only on reducing overall program power 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 has very high cost in terms of time and space. For instance, simulating a benchmark with 300 billion or more instructions with SimpleScalar [1] takes approximately 1 month of CPU time at a simulation rate of 400 million instructions per hour [5]. Including power simulation will increase the simulation time even more. While physical measurement is much faster, fine-grained power measurement requires proper measurement equipment and a large amount of space to store measurement results.

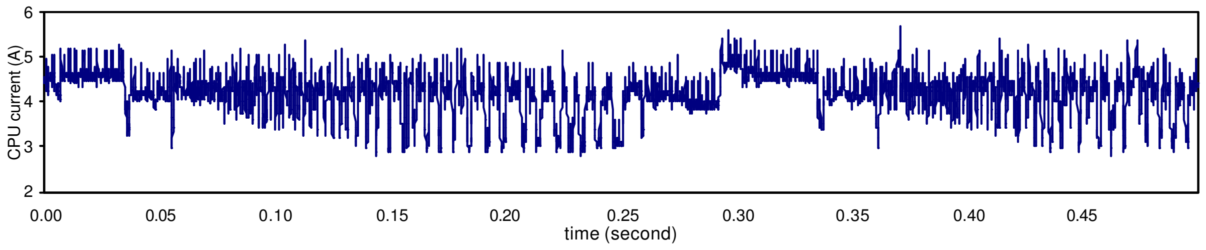
1.1. Characterizing Phases with Representative Slices

Program phase behavior shows that many program execution slices have similar behavior in several metrics, such as instructions-per-cycle (IPC), cache miss rate, and branch misprediction rate. Phase classification makes it easier to capture the fine-grained program behavior. Representative slices from each phase instead of the whole program execution are measured and analyzed, and then the whole program behavior can be characterized based on the analysis result. If we use this whole program behavior characterization method in power behavior analysis, we can obtain fine-grained power behavior with significant savings in both time and storage space. Figure 1 shows the measured CPU current of *bzip2* of SPEC2000. Figure 1(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(b) is the measured power behavior of a small part of 0.5 second in the first phase with a resolution that is 100 times higher than the one used for Figure 1(a). It shows that we can get finer phase classification if we use smaller intervals. There is a repeated power behavior period of 300 milliseconds. Figure 1(c) shows the detailed power behavior of a piece of 0.05 second, from 0.1 second to 0.15 second in Figure 1(b). It presents repeated power behavior periods of less than 5 milliseconds, indicating possible finer phase classification than Figure 1(b). With our infrastructure, we can get even finer power behavior.

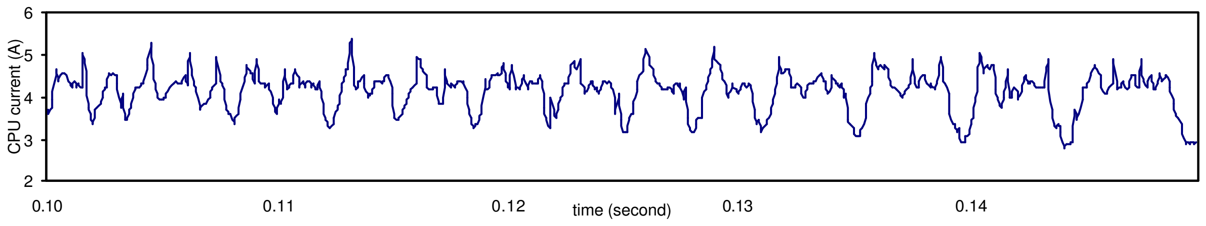
[0]Proceedings of MASPLAS'06 Mid-Atlantic Student Workshop on Programming Languages and Systems Rutgers University, April 29, 2006



(a) Very coarse granularity



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



(c) Detailed CPU power behavior of a small slice from 0.10 second to 0.15 second in (b)

Figure 1: Measured power behavior of bzip2 with different granularity.

1.2. Using Infrequent Basic Blocks to Reduce Instrumentation 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.

1.3. An Infrastructure for Characterizing Time-Dependent Power Behavior

In this paper, we present our infrastructure for program time-dependent power behavior characterization and optimization evaluation. This infrastructure is built on the *Camino* compiler of our lab [8]. *Camino* statically instruments the assembly code of a program for profiling and physical measurement. A SimPoint-like [19] idea is used for phase classification. SimPoint identifies a few intervals, or *simpoints*, of program execution that characterize the behavior of the entire program execution. It is often used to speed up simulation by simply simulating the simpoints and estimating, for instance, IPC, by taking a weighted average of the IPCs of each simpoint. Instead of using a fixed number of instructions as interval length, we use infrequently executed basic blocks to demarcate intervals. This results in variable interval length, but much lower instrumentation overhead for physical power measurement of a representative interval. We also use Basic Block Vector (BBV) as the fingerprint of each interval of the program execution, but the finally selected simpoints are weighted based on the number of instructions in each phase, instead of number of intervals. We do not claim that our phase classification method is better than SimPoint. Rather, we show that this method can also find the representative slices for program execution, and enables us to do power measurement for simpoints with very low instrumentation overhead. Unlike simulation, physical measurement is sensitive to the overhead for identification of simpoints during program execution. So this low instrumentation overhead is very important. This infrastructure can be used to evaluate optimizations for energy consumption or time-dependent power behavior, for example, the impact on power behavior of pipeline gating [16] or dynamic voltage/frequency scaling [6].

We evaluate our methodology by measuring 10 SPEC CPU2000 integer benchmarks on a Pentium 4 machine, and we present the error rates in whole program energy consumption estimation based on the measurement result of the selected simpoints. We also show the low instrumentation overhead because of the use of infrequently executed basic blocks for interval demarcation.

This paper has the following primary contributions: 1)

We present a new phase classification method based on infrequently executed basic blocks, which has SimPoint's ability to find representative intervals, but causes negligible instrumentation overhead in power measurement of simpoints. 2) We provide a compiler infrastructure that enables users to do static instrumentation for profiling on various levels, from procedure to instruction, as well as instrumentation for precise fine-grained power measurement. 3) We present a power measurement infrastructure to implement precise measurement of whole program execution and any selected program execution slice in combination with our *Camino* compiler infrastructure. We demonstrate the feasibility of our new phase classification method in efficient power behavior characterization through power measurement on a real system.

2. Related Work

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 [18, 19, 17, 11, 13], 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 [3, 4, 20, 9], such as IPC, power, cache misses rate and branch misprediction, to identify phases.

Lau *et al.* compare different architecture-independent structures used for phase classification [14], 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. Basic block vectors perform almost the best among the structures in terms of CPI coefficient of variation and calculated CPI error. Our work in this paper identifies phases based on the BBV of each interval, which is easy to obtain using *Camino*. The low coefficient of variation is important in power behavior characterization from the measured result of selected intervals.

SimPoint [18, 19] partitions a program execution into intervals with the same number of instructions and identifies the phases based on the BBV of each interval. One interval, called a *simpoint*, is selected as the representative of its phase. These simpoints are simulated or executed to estimate the behavior of the whole program execution. Sherwood *et al.* proposed an Off-line Phase Clustering Analysis and used it to find simpoints [19]. They applied SimPoint to some SPEC benchmarks to find their simpoints and estimated their program IPC, cache miss rate, branch misprediction rate. The error rates are very low and the simulation time saving is significant.

Shen *et al.* proposed a data locality phase identification method for run-time data locality phase prediction [17]. 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.* [13] shows a hierarchy of phase behavior in programs and the feasibility of variable length intervals in program phase classification. They break up variable length intervals based on procedure call and loop boundaries. 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 measurement of simpoints.

Isci *et al.* [2] proposed a coordinated measurement approach to monitor runtime power behavior of a real architecture. 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 [7]. Isci *et al.* [10] 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. Our work is different from [10] 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 paper 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.

3. Phase Classification Based on Infrequently Executed Basic Blocks

Phase classification and power measurement of representative intervals for a benchmark is implemented as an automatic process. The threshold for infrequent basic blocks, the minimum number of instructions in each interval, and the number of phases are the input to this process. The flowchart in Figure 2 illustrates its steps. The implementation of each step will be presented in the following sections.

3.1. Instrumentation Infrastructure for Profiling and Measurement

All of the instrumentation for various profiling and physical measurement of selected program execution slices in this paper are implemented using *Camino*, a compiler infrastructure [8] under development in our lab. It is a GCC post-processor written in C++. The goal of *Camino* is to serve as a testbed for various low-level optimizations. It is currently used to study performance and power and energy optimizations.

Camino reads the assembly code generated by GCC and the options selected by the user. It parses the assembly code into three basic abstractions: procedures, basic blocks, and lines(similar to instructions). It analyzes the control-flow of the input assembly code and constructs control-flow graph(CFG). Each basic block has a distinct reference value. *Camino* performs various transformations on the input assembly code according to user options, including branch alignment, static instrumentation for profiling, and pattern history table partitioning [12]. The output of *Camino* is the transformed assembly code that is compiled into an executable by GCC to run later.

The distinct reference value is used as the identification of a basic block. Instrumentation using *Camino* is simple. Only two routines are required: an instrumentation routine and an analysis routine. The former inserts a call to the analysis routine at proper positions in each basic block. The latter is 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 performed by an external device.

3.2. 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.

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. Instead of using an explicit frequency as the threshold, we use a percentage to find relatively infrequent basic blocks for each benchmark. 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. We try 3 different threshold values, 0.1%, 1%, and 5%, to investigate the trade-off between interval size variance and instrumentation overhead.

3.3. 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 10 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 10 million, this basic block indicates the end of the current interval and it is the first basic block of the next interval. Figure 3 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 10 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 8.

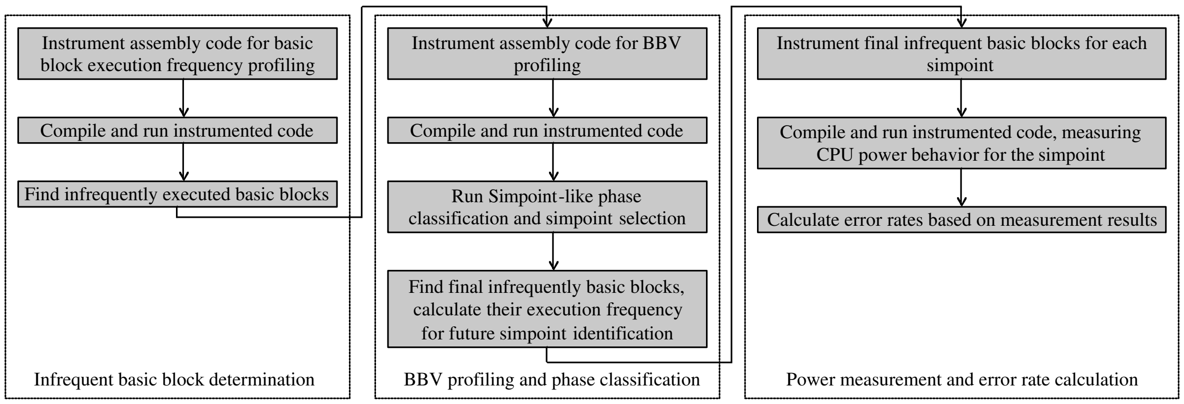

Instrument assembly code for basic block execution frequency profiling
Compile and run instrumented code
Find infrequently executed basic blocks
Infrequent basic block determination
Instrument assembly code for BBV profiling
Compile and run instrumented code
Run Simpoint-like phase classification and simpoint selection
Find final infrequently basic blocks, calculate their execution frequency for future simpoint identification
BBV profiling and phase classification
Instrument final infrequent basic blocks for each simpoint
Compile and run instrumented code, measuring CPU power behavior for the simpoint
Calculate error rates based on measurement results
Power measurement and error rate calculation

Figure 2: Infrequent basic block-based phase classification and power measurement of simpoints.

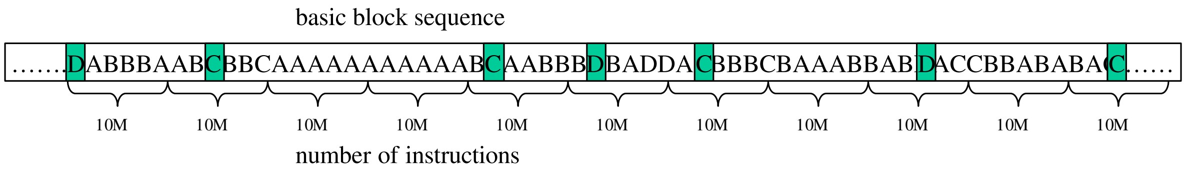


Figure 3: Interval partitioning using infrequent basic blocks and interval length.

3.4. 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 3.3. 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 3.3, 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 [19]. 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 [13], 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.

3.5. 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 [15] used in [10] 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 3.3 and the final infrequent basic blocks recorded in Section 3.4. Our infrastructure supports two power measurement methods for any selected intervals.

One method is to measure the intervals, 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.

4. CPU Power Measurement

All of the profiled and measured benchmarks in this paper run on a Pentium 4 machine running Linux 2.6.9, GCC 3.4.2 and GCC 2.95.4. Benchmarks are from the members of SPEC CPU2000 INT that can be compiled by *Camino* successfully, shown in Table 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. 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 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 4 is our experimental setup, data acquisition machine is not shown in the picture.

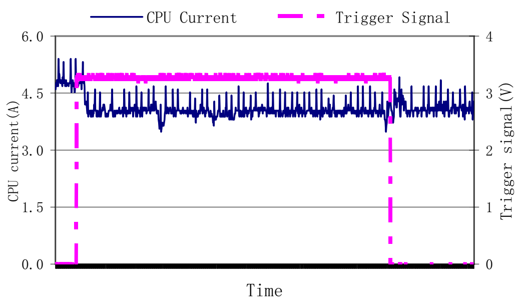


Figure 5: The window of the oscilloscope after the execution of a simpoint. The dotted line is the trigger signal. The power curve for the measured simpoint is surrounded by the trigger signal.

4.1. 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 4. The voltage difference between when the light is on and off is more than 3.0V, which is big enough to trigger the oscilloscope.

The voltage on the trigger channel is set to high to trigger the oscilloscope at the beginning of the program slice to measure. This voltage is consistently high until when it is set to low at the end of this slice. Figure 5 shows the measurement result using trigger signals. It is easy to identify the power behavior of the measured slice.

4.2. 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

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

Table 1: SPEC CPU2000 INT benchmarks

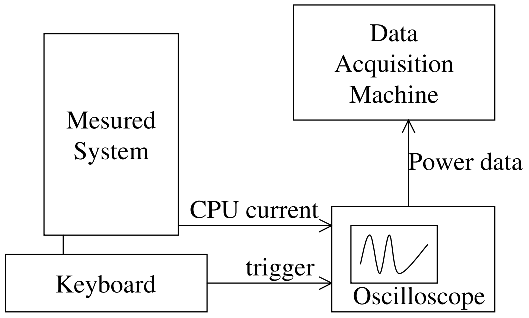


Figure 4: The physical measurement infrastructure used in the experiments.

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.

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 3.5. 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 3.5 is measured here. The second method has even lower overhead since fewer basic blocks are instrumented. Figure 6 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 shows the same trend in measured execution time when different thresholds are used.

4.3. 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, 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 in-

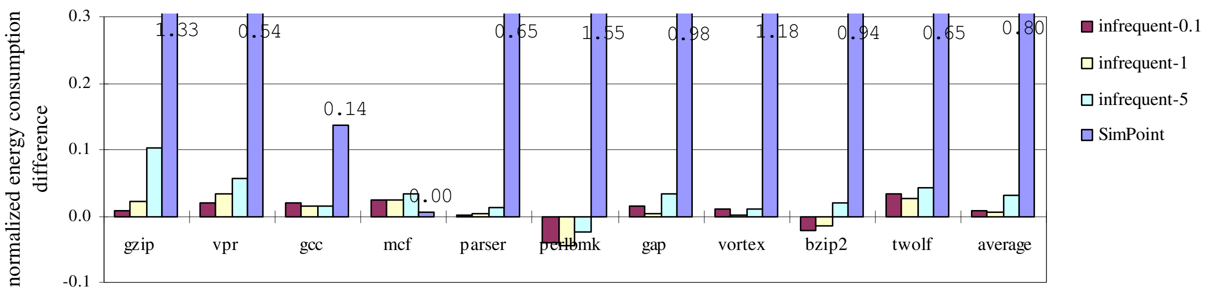


Figure 6: Normalized overhead in energy consumption of instrumented benchmarks using different thresholds.

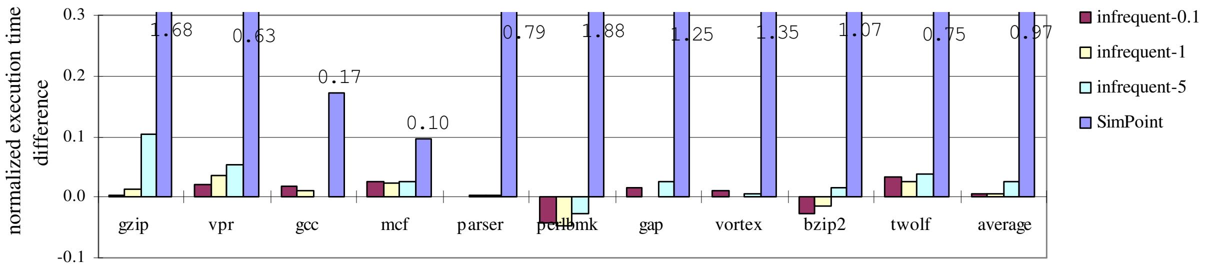


Figure 7: Normalized overhead in execution time of instrumented benchmarks using different thresholds.

strumentation method in Section 3.5, and implement an automatic measurement and data acquisition process to do measurement of any number of simpoints as a single task.

5. Experimental Results

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 [19]. 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 4, 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 3.5. Energy consumption is calculated as

$$E = U * \sum It \tag{1}$$

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 3.4. *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

$$error = \frac{|energy\_estimated - energy\_measured|}{energy\_measured} \tag{2}$$

Time estimation is similar to energy estimation.

Figure 8 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.

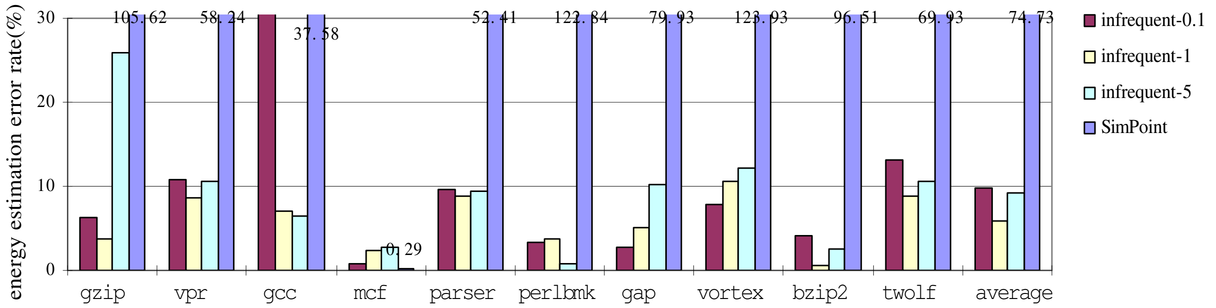


Figure 8: Error rates of energy consumption estimation when different thresholds are used, based on comparison between estimated and measured energy of uninstrumented benchmarks.

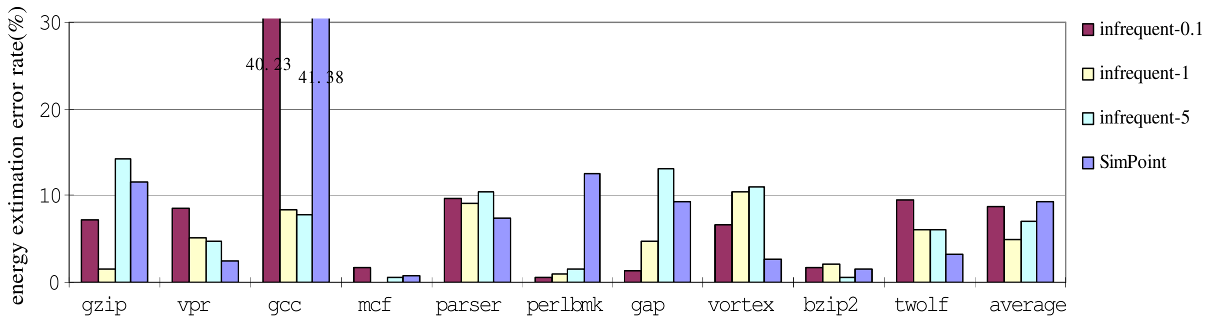


Figure 9: Error rates of energy consumption estimation when different thresholds are used, based on comparison between estimated and measured energy of instrumented benchmarks shown in Figure 6.

To verify that the low error rates in Figure 8 are not obtained by accident and the selected simpoints are really representative of the program execution, in Figure 9, 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 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 9 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 in [13]. 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 10 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.

6. Conclusion

This paper introduced our infrastructure for efficient program power behavior characterization and evaluation, including the *Camino* compiler for profiling and instrumentation, a new phase classification method, and the physical measurement setup for precise power measurement. By demarcating intervals using infrequently executed basic block, we get intervals with variable lengths and negligible instrumentation overhead for physical measurement of simpoints. Through experiments on a real system, we demonstrated that our new phase classification method can also find representative intervals for energy consumption estimation as SimPoint. The ability of instrumenting programs on various levels, identifying phases, and obtaining detailed power behavior of program execution slices makes this infrastructure useful in power behavior characterization and optimization evaluation.

References

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

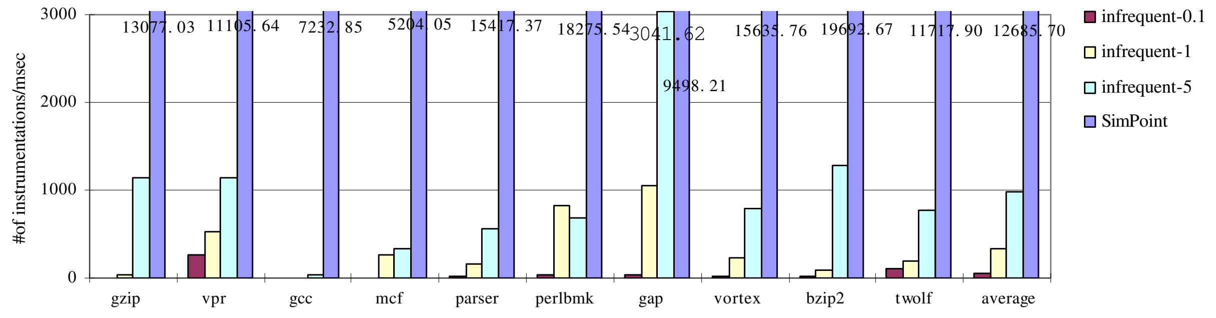


Figure 10: The number of instrumentation per millisecond during the program execution in simpoints.

[2] Canturk Isci, Margaret Martonosi. 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.

[3] E. Chi, A. M. Salem, and R. I. 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.

[4] E. Duesterwald, C. Cascaval, and S. 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.

[5] Erez Perelman, Greg Hamerly, and Brad Calder. Picking statistically valid and early simulation points. *International Conference on Parallel Architectures and Compilation Techniques (PACT'03)*, 2003.

[6] C.-H. Hsu and U. 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)*.

[7] C. Hu, D. A. Jiménez, and U. Kremer. Toward an evaluation infrastructure for power and energy optimizations. In *19th International Parallel and Distributed Processing Symposium (IPDPS'05, Workshop 11), CD-ROM / Abstracts Proceedings*, April 2005.

[8] C. Hu, J. McCabe, D. A. Jiménez, and U. Kremer. The camino compiler infrastructure. *Proceedings of the 2005 Workshop on Binary Instrumentation and Applications (WBIA)*, 2005.

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

[10] C. Isci and M. 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. 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.

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

[17] X. Shen, Y. Zhong, and C. 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.

[18] T. Sherwood, E. Perelman, and B. 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.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, October 2002.

[20] R. Srinivasan, J. Cook, and S. 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.