# Power-Sleuth: A Tool for Investigating your Program's Power Behavior

Vasileios Spiliopoulos, Andreas Sembrant, Stefanos Kaxiras
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{vasileios.spiliopoulos, andreas.sembrant, stefanos.kaxiras}@it.uu.se

***Abstract*—Modern processors support aggressive power saving techniques to reduce energy consumption. However, traditional profiling techniques have mainly focused on performance, which does not accurately reflect the power behavior of applications. For example, the longest running function is not always the most energy-hungry function. Thus software developers cannot always take full advantage of these power-saving features.**

**We present Power-Sleuth, a power/performance estimation tool which is able to provide a full description of an application's behavior for any frequency from a single profiling run. The tool combines three techniques: a power and a performance estimation model with a program phase detection technique to deliver accurate, per-phase, per-frequency analysis.**

**Our evaluation (against real power measurements) shows that we can accurately predict power and performance across different frequencies with average errors of $3.5\%$ and $3.9\%$ respectively.**

## I. Introduction

In the past decades, design of computer systems has focused on delivering the highest possible performance. Optimizing for speed has been the main goal in all levels of building a system, from architecture-level decisions (e.g., complex cache hierarchies, out-of-order execution) and physical-layer design (faster transistors) to program development. Regarding the latter, profiling software [2] has proven to be a powerful tool in the hands of developers to optimize their code for speed. In the last few years, however, it is power consumption that is turning into the most critical constraint in system design. Although hardware design has taken large steps towards minimizing power consumption through advanced power saving techniques (e.g., clock gating, power gating, voltage-frequency scaling), less effort has been spent on developing power-aware software. One of the reasons for this is the lack of advanced profiling tools for providing software developers with power-related information required to improve energy-efficiency of their code. This paper introduces Power-Sleuth, a tool for investigating your program's power behavior.

Power-Sleuth is unique in that it brings together three techniques: efficient run-time phase detection and identification (Section II-A), performance estimation based on analytical Dynamic Voltage and Frequency Scaling (DVFS) models (Sections II-B, IV), and power estimation based on novel correlation models (Section V). All three components are important for a complete understanding of the power behavior of a program.

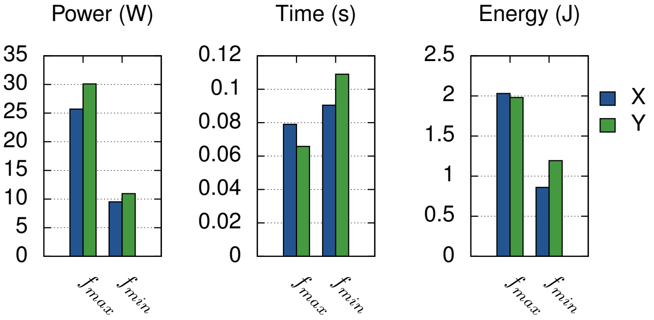


Figure 1. The power, time and energy consumption of two phases, $X$ and $Y$, at different frequencies, $f_{min}$ and $f_{max}$, in gcc/166. The figure illustrates the importance of considering all frequencies. For example, phases $X$ and $Y$ have similar energy consumptions at $f_{max}$, but have distinctly different energy consumptions at $f_{min}$.

Figure 1 motivates why both power and timing information are required for characterizing energy behavior. The figure illustrates two of the phases of gcc/166 (SPEC2006 [11]), detected with a phase detection tool, and the corresponding core power, execution time and energy consumption. Traditional profiling tools have focused on analyzing an application with respect to time. When it comes to energy, however, it is not always the case that phases with the longest execution time are the most energy-hungry phases. In our example, phase $X$ executes for longer time in maximum frequency compared to phase $Y$, but it consumes less power, so the total energy of the two phases is roughly the same. This means that both time and power are required to classify phases regarding energy.

Modern processors support multiple clock-frequency steps. As we show in Figure 1, ignoring this functionality can provide misleading information about the program behavior. Phase $X$ is the most important of the two regarding execution time under maximum frequency, however phase $Y$ runs for longer time at minimum frequency. This means that different phases are affected in a different way by frequency scaling, thus determining where a program spends most of its execution time is frequency dependent. Moreover, since time and power do not change uniformly with frequency, it is not valid to claim that phases $X$ and $Y$ are similar in terms of energy; although energy consumptions are roughly the same at maximum frequency, phase $Y$ consumes about $39\%$ more energy when the two phases are executed at minimum frequency.

In addition, analyzing a program in phases, as opposed to, say, execution intervals, is indispensable in two ways. First, phases allow us to relate the performance and power analysis back to the source code. A program phase typically comprises a small set of function calls, thus when we talk about phase $A$, we can actually reason about the power/performance behavior of specific parts of the program. Second, if we want to understand the power/performance behavior of a program in sufficient detail so as to optimize it, by necessity we need to collect profiling information at a finer granularity than the whole program. Breaking up the execution of a program in intervals, and profiling each interval individually, allows us to do just that. However, adding phase detection on top of the intervals, brings significant leverage in how we can collect the profiling information. Since, with phase detection, each interval is assigned to a specific phase, we know that it has similar behavior to other intervals of the same phase. We can thus guide our profiling to track many more events than what the hardware allows us to sample at any time. The end result is that phase detection allows us to profile a single run of the application without restricting our ability to gather the necessary profiling information.

In brief, Power-Sleuth works as follows: it runs an application once, collects the data required, and then from this data it can provide power and performance information in any frequency of interest.

The main contributions of this paper are:

- We present a novel power correlation model that is independent of frequency.
- We combine three main components (power and performance estimation models and a phase detection and classification technique) to deliver per-phase and per-frequency power/performance analysis.
- We evaluate our approach against real, fine-grained power measurements.
- We demonstrate how accurate power/performance prediction can be utilized for understanding and improving the power efficiency of applications.

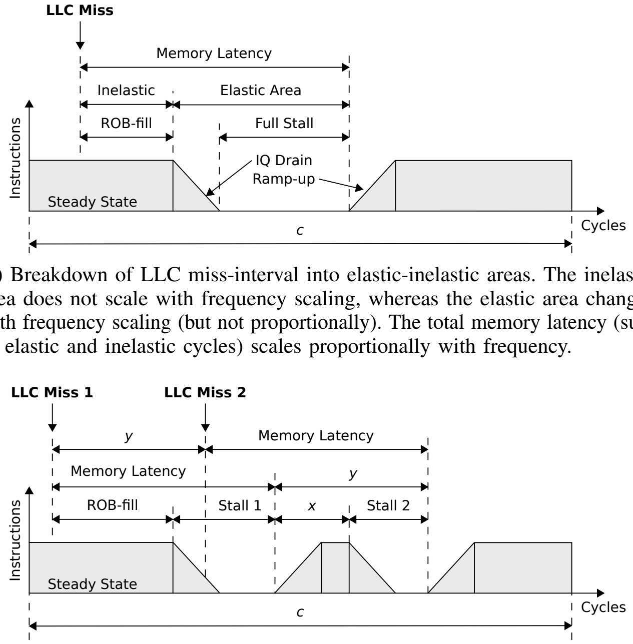
The evaluation of our approach shows that we can predict power and performance with average errors of 3.5% and 3.9% respectively.

## II. Background

### A. Detecting Program Phases

Power-Sleuth analyzes programs at the level of program-phases. We use the ScarPhase [25] library to detect and classify phases. ScarPhase is an execution history based, low overhead (2%), online phase detection library. Since it is based on the application's execution history, it detects hardware independent phases [27, 23]. Such phases can be readily missed by performance-counter based phase detection.

To detect phases, ScarPhase monitors executed code, based on the observation that changes in executed code reflect changes in many different metrics [26, 27, 4, 28, 21]. To accomplish this, execution is divided into non-overlapping intervals. During each interval, hardware performance counters are used to sample conditional branches using Intel Precise Event Based Sampling [22, 13]. The address of each branch is hashed into a vector of counters called a conditional branch vector (CBRV), similar to a basic block vector (BBV) [26] but with only conditional branches. Each entry in the vector shows how many times its corresponding conditional branches were sampled during the interval.



(a) Breakdown of LLC miss-interval into elastic-inelastic areas. The inelastic area does not scale with frequency scaling, whereas the elastic area changes with frequency scaling (but not proportionally). The total memory latency (sum of elastic and inelastic cycles) scales proportionally with frequency.

(b) Case of overlapping LLC misses handled by stall-based model. Due to the forwarding of the second miss to the head of the Reorder Buffer, an additional stall interval is introduced.

Figure 2. Interval-based DVFS model

The vectors are then used to determine phases by clustering them together using an online clustering algorithm, such as leader-follower [6]. Intervals with similar vectors are then grouped into the same cluster and considered to belong to the same phase.

### B. Analytical DVFS Models

In our previous work [20] we described two analytical interval-based models, named *stall-based* and *miss-based* models, to predict the impact of frequency scaling in an application's execution time. Both models are derived from Karkhanis and Smith [18, 8] interval-based performance model. The two models extend the basic performance model by identifying: 1) the critical events that are affected by frequency scaling and 2) how these events are affected by frequency. In particular, the models suggest that execution time measured in cycles remains unaffected by frequency unless an off-chip request occurs. In this case, slowing down the processor results in a reduction of the latency of the main memory (measured in cycles). Two more groups, working independently, came up with models similar to ours [7, 24].

Table I
PERFORMANCE COUNTERS

| **Phase Detection** | | |
|---|---|---|
| *EVENT NAME* | *EVENT CODE* | |
| INST_RETIRED.ANY | FIXED_CTR | |
| BR_INST_RETIRED | 0x01C4 | |
| **Power Estimation** | | |
| *EVENT NAME* | *EVENT CODE* | *PARAM* |
| UOPS_EXECUTED.PORT_234_CORE | 0x80B1 | 0.75 |
| L2_RQSTS.MISS | 0xAA24 | -4.51 |
| L2_RQSTS.REFERENCES.ANY | 0xFF24 | 3.08 |
| RESOURCE_STALLS.ANY | 0x01A2 | -1.38 |
| FP_COMP_OPS_EXE.SSE_FP | 0X0410 | 0.94 |
| BR_MISP_EXEC.ANY | 0x7F89 | 0.35 |
| POWER MODEL CONSTANT | – | 2.11 |
| **Performance Estimation** | | |
| *EVENT NAME* | *EVENT CODE* | |
| CPU_CLK_UNHALTED | FIXED_CTR | |
| UOPS_EXECUTED.CORE_STALL_CYCLES | 0x3FB1 | |
| LLC_MISSES | 0x412E | |

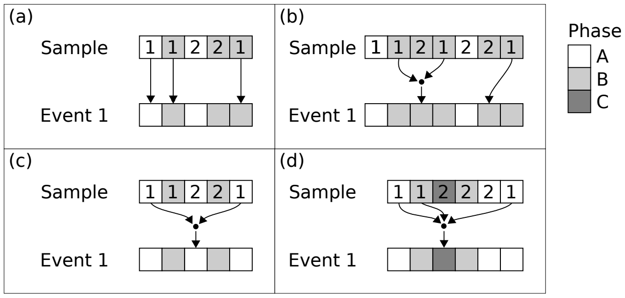


Figure 3. Multiplexing/Interpolation methodology for event 1. (a) shows the case that an event is sampled in an interval. (b) shows how an event is approximated when it is not sampled in the same interval, but in intervals of the same instance of the phase. In (c), the event is not monitored in the current instance, so it is approximated using information from past and future instances. Finally, in (d) the event is never monitored for this phase and is approximated using information of the whole program execution. Note that for cases b, c and d, for the intervals that event 1 is actually sampled, the same approach as in case (a) is used.

**Stall-based model.** The basic interval-based performance model breaks execution of program into intervals. During the *steady state intervals*, the processor executes instructions at a constant rate, limited by the processor's width and the program's Instruction Level Parallelism. Steady state intervals are punctuated by *miss events* (e.g., cache misses, branch mispredictions), which introduce stall cycles. Figure 2a shows how the basic interval-based performance model represents a miss-interval due to a Last Level Cache miss. When an LLC miss occurs, the processor continues to issue instructions for a few cycles until the *Reorder Buffer* (ROB) fills up. The processor then keeps executing instructions until the *Instruction Queue* (IQ) drains out of instructions independent to the pending miss. When the miss is serviced, new instructions enter the instruction window and the issue rate ramps up until it reaches the steady state. Off-chip requests are crucial events for core frequency scaling, thus it is important to understand how memory latency changes with frequency. Since $mem\_lat\_in\_core\_cycles = mem\_lat\_in\_nsec \times core\_freq$ and memory latency measured in nsec is not affected by core frequency scaling, memory latency measured in cycles scales proportionally with frequency. The stall-based model assumes that stall cycles due to off-chip requests scale proportionally with frequency. This is of course an approximation, since it disregards the ROB-fill area:

$$\begin{aligned} stall\_cycles &= mem\_lat - ROB\_fill \\ &\approx mem\_lat \end{aligned} \tag{1}$$

Figure 2b shows the case of overlapping LLC misses. In addition to the first stall interval, a second stall interval appears due to the forwarding of the second miss to the head of the ROB. In this case, the stall-based model can still be applied, since total stall cycles are approximately equal to memory latency:

$$\begin{aligned} stall_1 + stall_2 &= y + mem\_lat - ROB\_fill - x \\ &\approx mem\_lat \end{aligned} \tag{2}$$

In any case, stall cycles are approximately equal to memory latency, thus the total number of stall cycles (for a given part of the program) will scale proportionally to frequency. On the other hand, non-stall cycles remain intact when frequency is scaled. If $c$ is the total execution cycles for a given amount of instructions and $st$ is the total stall cycles under frequency $f_0$, then for frequency $f_1$ (with a scaling factor $k = f_0/f_1$) stall cycles, execution cycles and execution time can be approximated as

$$\begin{aligned} st_{new} &= \frac{st}{k} \\ c_{new} &= c - st + \frac{st}{k} \\ t_{new} &= \frac{c_{new}}{f_1} = \frac{c_{new} \times k}{f_0} = \frac{(c - st) \times k + st}{f_0} \end{aligned} \tag{3}$$

**Miss-based model.** The miss-based model on the other hand improves prediction accuracy by taking into account the existence of $ROB - fill$ area. As shown in Figure 2a, the whole miss interval equals memory latency, and thus it is the whole miss interval that scales *proportionally* with frequency. In the case of overlapping misses (Figure 2b), if $LLC\ Miss$ 2 occurs $y$ cycles after $LLC\ Miss$ 1, it will also be serviced $y$ cycles after $LLC\ Miss$ 1, regardless the frequency, since $y$ belongs to the inelastic area of the first miss interval. Thus, the stalls generated by $LLC\ Miss$ 2 do not scale with frequency, meaning that *only the miss interval of the first miss in a cluster of overlapping misses scales with frequency*. By counting the number of these misses, our model predicts execution time over different frequencies with great accuracy (1% error on average). More details about the miss-based model can be found in [20].

## III. POWER-SLEUTH OVERVIEW

Power-Sleuth is a power/performance estimation tool. To use Power-Sleuth, the user needs to run an application only once. During this run, the tool collects all the data required, and then it estimates the energy consumption of each program phase. Moreover, without any additional runs of the application, Power-Sleuth is able to predict how power and performance of each phase will be affected if it is executed under any other frequency.

Power-Sleuth combines three basic components: an analytical DVFS performance model, an effective capacitance correlation model and a phase detection technique. Execution of the program is divided into intervals of 100M committed instructions each. Using ScarPhase, each interval is characterized by a phase ID. Performance and power models rely on information gathered by hardware performance counters: with our current setup, 11 different performance counter events have to be monitored per interval. Execution cycles and instructions retired can be monitored using two of the fixed counters, but the remaining 9 events have to be monitored using programmable performance counters. However, only 4 counters are available in our processor. One way of gathering all the required information is running the application multiple times and monitoring different events in every run. This would imply performing 3 complete runs of the same application before applying the power/performance models to give a picture of the application's behavior. Power-Sleuth manages to collect all the required information from a **single** run of the application by using a phase-guided multiplexing/interpolation technique.

In the remainder of the paper we use the following terminology. We refer to a window of 100M retired instructions as an *interval*. Each interval belongs to a *phase* and is assigned a *phase ID* by the phase detection component of Power-Sleuth. Finally, we refer to consecutive intervals of the same phase as an *instance* of this phase.

### A. Phase-Guided Counter Multiplexing

Since we need to concurrently monitor more events than the hardware supports, we have to multiplex multiple events in the available set of counters. Employing a simple time-based multiplexing methodology requires the program to have rather uniform behavior, otherwise approximating events that are not monitored with previous values of these events results in high errors. Instead of a simple time-based approach, we use a *phase-guided* time-based multiplexing methodology, shown in Figure 3. In this example we assume that there is only one available counter and two different events to measure, named *event 1* and *event 2*. The same approach can be extended for different number of events to monitor. The performance counter is programmed to measure events in a per-phase, round-robin fashion. This means that Power-Sleuth remembers the type of event monitored in the last interval of each phase and programs the performance counter to measure the next event when an interval with the same phase ID is about to execute again. Since the counter has to be programmed at runtime, Power-Sleuth needs to predict the phase ID of the next interval. We use the same prediction method as in [25]. Figures 3a-d show how events 1 and 2 are sampled for different patterns of phases $A$, $B$ and $C$.

### B. Phase-Guided Interpolation

As a consequence of multiplexing counters, not all of the events are sampled in every interval. To solve this issue we develop a phase-aware interpolation method. There is a total of 4 different cases for an event during an interval. The event is:

1) monitored in this interval,
2) not monitored in this interval but monitored in other intervals of the current instance of the phase,
3) not monitored in the current instance of the phase, but monitored in other (future or past) instances of the same phase,
4) never monitored for this phase.

Figures 3a-d show the priority for approximating event 1 in various intervals, with (a) being the highest priority and (d) the lowest. Of course, the same idea applies for approximating event 2. Figure 3a shows case 1, when event 1 was actually sampled in some interval. Then the value sampled is used for this interval. In Figure 3b consecutive intervals of phase $B$ are executed, and thus sampling is interleaved between events 1 and 2. For the intervals that event 2 was sampled, event 1 is approximated as the average of the values of event 1 sampled in the current instance of this phase. Figure 3c depicts an example of case 3: there is an instance of phase $A$ that event 1 is never sampled (third interval of this example). In this case, the average of the values of event 1 sampled in all past and future intervals of the same phase are used as an approximation. Finally, as shown in Figure 3d, there is an extreme case that event 1 is never sampled for some phase (phase $C$ in the example). In this case, the average of values of event 1 sampled in the whole execution of the program, regardless the phase, is used to approximate event 1.

The approach described above lets us have one value for each event of interest (either sampled or approximated) in every interval. This information is then fed to our models to predict power and execution time of each interval not only for the frequency the application was profiled in, but for any frequency the user wants, even if this frequency is not included in the processor's possible V-f configurations. Intervals of the same instance of a phase are grouped together to get a more smooth view of the average behavior per instance of the phases, and finally different instances of the phases are accumulated to get a per-phase performance/energy breakdown for the whole program. Note however that since we use a multiplexing and interpolating method, Power-Sleuth can still provide per interval information.

### C. Experimental Setup

We run our experiments in an Intel Core i7 920 machine (Nehalem micro-architecture). The processor is a quad-core machine with 9 frequency steps (2.66GHz to 1.6GHz), 4 programmable performance counters and 3 more counters monitoring fixed events. We run benchmarks from SPEC2006 suite in a 2.6.38-12 Linux kernel. Power gating (turning off idle cores to save leakage power) is deactivated so that we can

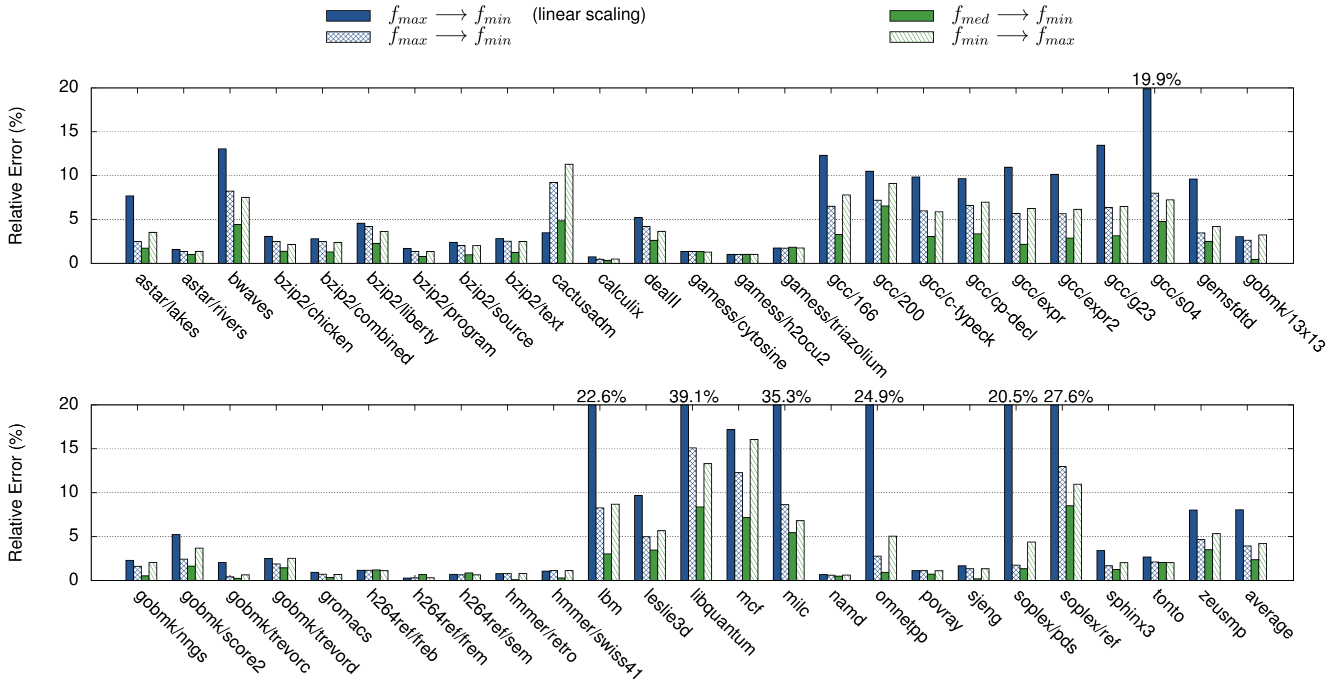


Figure 4. Relative error in predicting execution time. The figure shows (from left to right) prediction: a) using data collected in maximum frequency and predicting for minimum frequency assuming execution time scales linearly with frequency, b) from maximum to minimum frequency using the stall-based model, c) from median to minimum frequency using the stall-based model and d) from minimum to maximum frequency using the stall-based model.

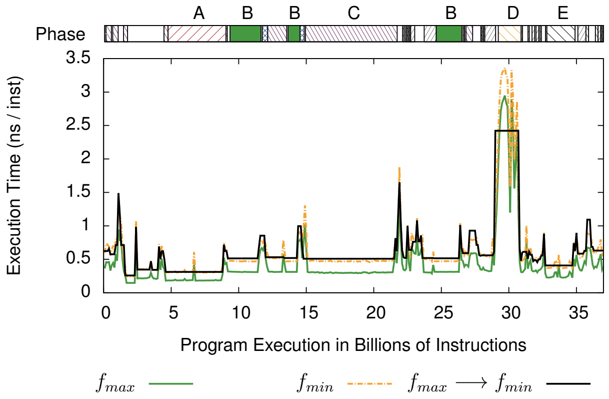


Figure 5. nsec per instruction for the first half of gcc/166. The second half is identical and thus omitted. The top of the figure shows the phases detected. $f_{max}$ and $f_{min}$ show the execution times for maximum and minimum frequency and $f_{max} \longrightarrow f_{min}$ is the predicted execution time for minimum frequency when profiling is performed in maximum frequency.

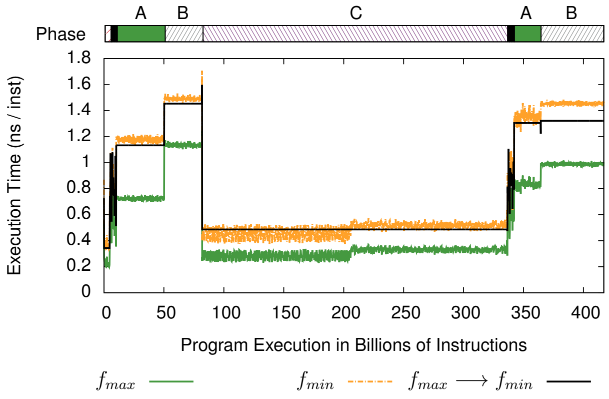


Figure 6. nsec per instruction for astar/lakes. The top of the figure shows the phases detected. $f_{max}$ and $f_{min}$ show the execution times for maximum and minimum frequency and $f_{max} \longrightarrow f_{min}$ is the predicted execution time for minimum frequency when profiling is performed in maximum frequency.

measure idle power in all frequencies. We collect real power samples using current sensors and a 16-channel A/D device. These samples are used as reference for our power estimation.

## IV. Performance Estimation

In [29] we employed the models described in Section II-B to develop frequency scaling governors. These governors adapt to program behavior and pick the frequency that minimizes various energy efficiency metrics, such as *Energy Delay Product* (EDP) with/without performance constraints. In this work we use the same models, but from a different perspective: the goal now is to provide a per-phase estimation of how execution time is affected by frequency scaling. Moreover, performance prediction is crucial for our novel, cross-frequency power estimation method described in Section V.

After running an application once and collecting appropriate statistics, performance of each interval/phase can be estimated using the models described in Section II-B. The miss-based model, though more accurate, cannot be applied in our processor since there is no performance-counter event monitoring the number of clusters of misses. Neither the stall-based model can be applied as it is; there is no counter for measuring stalls due to off-chip requests. These stalls, however, can be approximated by the minimum between all the pipeline execution stalls and the worst case stalls due to off-chip misses. The latter ones are simply the number of last level cache misses multiplied by the memory latency. More details about applying the stall-based model in a Nehalem processor can be found in [29].

Figure 4 illustrates the error of our prediction. The error shown per benchmark is not just the error in predicting the whole execution time. Consecutive intervals of the same phase are grouped together and the execution time of this instance of the phase is predicted. The prediction is compared to the actual execution time of the instance and the relative error is calculated. The figure shows the average of all of these errors along the execution of the program, weighted with the number of intervals of each phase. This way possible over/underestimations of different phases do not cancel each other and the error presented corresponds to the accuracy of predicting execution times of separate phases across different frequencies.

The leftmost bar shows the accuracy of a simple linear scaling model which assumes that execution time scales proportionally with frequency. Though this estimation is accurate for CPU bound programs (such as calculix and h264ref), the error is huge for the memory bound applications, reaching up to $39.1\%$ for libquantum. This proves the necessity of using a model that takes into account the asynchronous nature of off-chip accesses. The next bar shows the prediction error when a profile run is performed in maximum frequency (2.66GHz) and execution time is predicted for minimum frequency (1.6GHz). As shown in the figure, most of the benchmarks suffer low prediction errors, within $5\%$, whereas the worst case prediction is $15.1\%$ (libquantum). Bare in mind that libquantum is the most memory bound application, so disregarding the ROB-fill area results in higher prediction error. The average error over all benchmarks is about $3.9\%$. The next bar shows how prediction accuracy can be improved by performing the profiling run in the median frequency between max and min (2.13GHz). Profiling time is now longer, but the closer the profiling frequency is to the target frequency, the better the prediction accuracy, thus worst case and average errors are reduced down to $8.4\%$ and $2.4\%$ respectively. Finally, the rightmost bar shows that prediction accuracy is not affected when Power-Sleuth collects profiling data in low frequency and predicts performance for a high frequency, since average error is $4.2\%$.

Power-Sleuth predicts execution time individually for each interval/phase. The metric we use is nsec per instruction: we divide execution time of each interval with the instructions executed in that interval (100M). Figures 5, 6 show the execution of the phases of gcc/166 and astar/lakes respectively. We run Power-Sleuth in maximum frequency and we measure execution time ($f_{max}$). At the same time, we predict execution time for minimum frequency ($f_{max} \longrightarrow f_{min}$). Finally, we run Power-Sleuth once more to actually measure execution time in minimum frequency and evaluate the accuracy of our prediction. At the top of each figure, we show the phases detected by Power-Sleuth. The figures show that Power-Sleuth accurately predicts the increase in execution time due to frequency scaling. The relative increase of execution time between maximum and minimum frequency is a metric of how memory bound an application is. The closer the ratio to 1, the more memory bound the phase is. For the predicted execution time, we average intervals of the same instance of a phase together, to get a more smooth view of the phase behavior. Thus, we can see the average performance of phase $D$ in gcc, filtering out the peaks that appear in some intervals. However, since we perform per interval prediction, the user can skip this step.

## V. Power Estimation

We model total power consumption as the sum of dynamic power consumption (dissipated by the switching activity of transistors) and static power (dissipated by leakage currents)[19]:

$$P = P_{static} + afCV^2 = P_{static} + fC_{eff}V^2 \qquad (4)$$

where $P_{static}$ is the static power, $f$ is the operating frequency, $V$ is the supply voltage, $C$ is the processor capacitance, $a$ is the activity factor and $C_{eff} = a \times C$ is the *effective capacitance*.

Several researchers have correlated power with performance counter events [3, 10, 14, 17]. Their work is limited by voltage and frequency scaling in the sense that different models have to be formed to predict power in different frequencies. Moreover, these models, unless coupled with a performance prediction model, are not able to predict power when target frequency is not the same as the profiling frequency. Modern processors' power measuring capabilities [12] suffer from similar limitations: power monitoring can only provide power for current V-f configuration and for the whole package. Thus, it is impossible to estimate power per core in multiprocess workloads, as well as for different frequencies. To address these limitations, we develop a model that correlates core's effective capacitance with performance counters. As explained later in this section, the model is frequency independent and thus fulfills the requirements of Power-Sleuth.

### A. Methodology

What is unique in our approach is that rather than correlating power with events, we develop a model that investigates directly the source of power consumption: charging and discharging of processor node-capacitances. Component activity makes capacitors switch state, and this switching activity results in power consumption which depends on the processor frequency and voltage supply. Component activity (i.e. effective capacitance) does not depend on voltage and frequency; it is only related to architecture level details and application behavior.

We develop a correlation model for estimating effective capacitance by training the model in maximum frequency, and then using the same parameters we can estimate power in any Voltage-frequency setting with Equation (4). Thus, the problem of estimating power is reduced to estimating effective capacitance. More importantly, we do not need to collect the data fed to the model from a profiling run in the same frequency as the target frequency: we monitor performance counter events

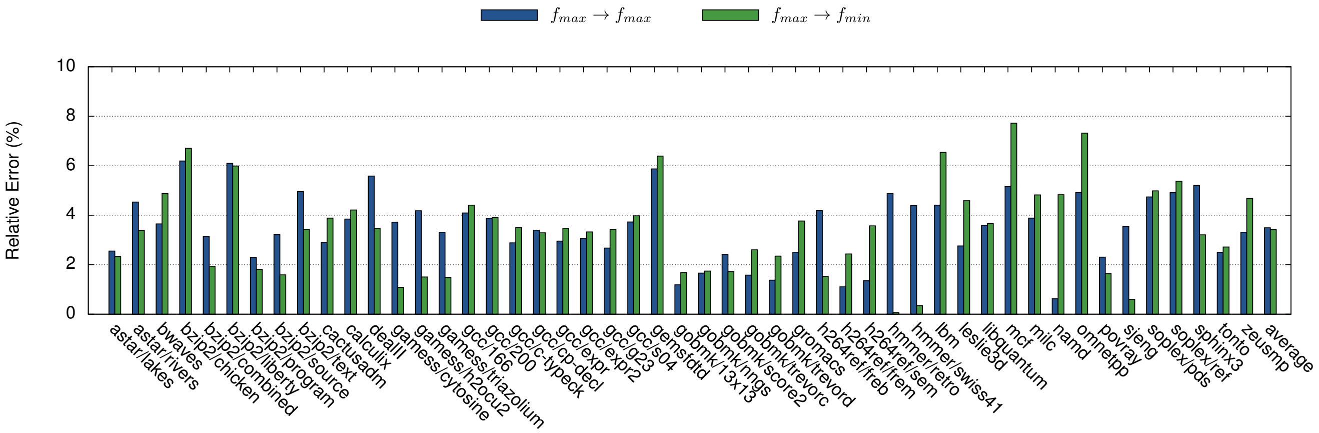


Figure 7. Relative error in predicting power consumption. We run the application in maximum frequency and predict power for maximum (left bar) and minimum (right bar) frequency.

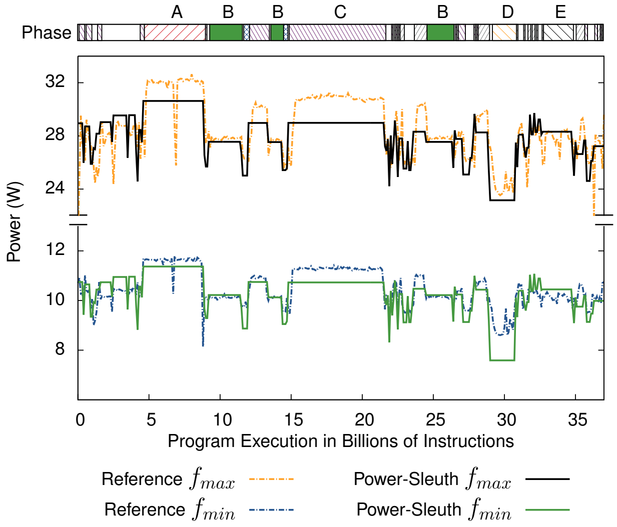


Figure 8. Power consumption (measured and predicted) for the first half of gcc/166 at maximum (upper part) and minimum (bottom part) frequency. The second half is identical and thus omitted. Both predictions are based on the data collected from a profiling run in maximum frequency.

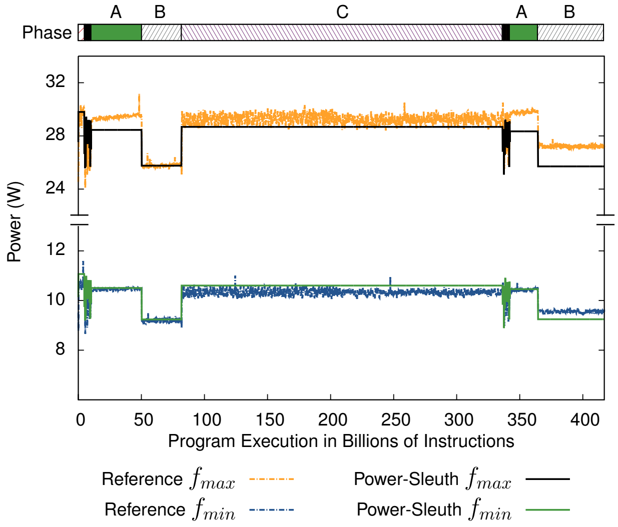


Figure 9. Power consumption (measured and predicted) for astar/lakes at maximum (upper part) and minimum (bottom part) frequency. Both predictions are based on the data collected from a profiling run in maximum frequency.

in maximum frequency (minimizing profiling time) and then we estimate event rates by using the analytical DVFS performance model presented in Section II-B. This is possible because most events of interest (like micro-ops executed, cache misses etc.) are not affected by frequency scaling and the event rates (on a per cycle basis) are simply the event counts divided by the prediction of execution cycles under the target frequency.

We assume that effective capacitance is a linear function of various event rates that best describe the power behavior of the processor. We tried different events in order to track the activity of different processor components: execution units, cache hierarchy, branch predictor, and utilization of various other resources, such as load buffers and reservation station. We ended up with the events shown in Table I, which is a good compromise between good accuracy and low number of events monitored. One of the events with high correlation is the RESOURCE_STALLS.ANY event, which is a cycle count event and thus can be affected by frequency scaling: if resource stalls overlap with memory stalls they scale with frequency, otherwise they do not. Since it is impossible to monitor a union of the two events (cycles that are both resource and execution stalls), we use the heuristic that if resource stalls are more than execution stalls, a part of them overlaps with execution stalls and thus scales with frequency in the same way as execution stalls. This approximation, though based on a -educated- guess, provides good results: it improves the error compared to the case that no special care is taken for the scaling of resource stalls. To predict effective capacitance, we use the following equation:

$$C_{pred} = \sum_{k=0}^{5} \frac{param_k \times event_k}{cycles} + param_6 \qquad (5)$$

where $event_k, k = 0, ..., 5$ are shown in Table I. To train our model for the specific hardware, we use real power measurements. We run a set of benchmarks in maximum frequency, measure processor total power consumption, subtract static power (measured for all frequencies when the processor is idle) and finally divide with $f \times V^2$ . This way we compute the average effective capacitance $C_i$ for benchmark $i$. We train our model by minimizing $\sum\limits_{i \in specs} \left(C_i - C_{pred_i}\right)^2$, or

$$\sum_{i \in specs} \left(C_i - \sum_{k=0}^{5} \frac{param_k \times event_{k,i}}{cycles_i} - param_6\right)^2 \quad (6)$$

After obtaining the parameter values $param_k, k = 0, 1, ..., 6$ that minimize expression (6) (shown in Table I), we can use them for effective capacitance estimation of any application.

### B. Evaluation

We run all the applications in maximum frequency and let Power-Sleuth collect the profiling data. Similarly to performance estimation, we estimate power consumption for each interval separately and then we average intervals of the same instance of the phase. Unlike performance prediction, we have to predict power consumption even for the profiling frequency, since no real power measurements are used by the end tool. The left bar in Figure 7 shows the error when profiling frequency is maximum and we predict for the same frequency. The worst case error is about 6%, while the average error is below 4%. The right bar of the same figure shows the error when from the same data (collected in the profiling run under maximum frequency) we predict power consumption under minimum frequency. Remember that to do so, we first need to use the performance model to get the event rates under minimum frequency, and then we can use the power correlation model to estimate power. The figure shows that we can accurately predict power even when we profile the application under a frequency different than the target frequency.

Finally, Figures 8 and 9 show power predicted over time for gcc/166 and astar/lakes in maximum and minimum frequency. Each application was profiled in maximum frequency, and the data was used to predict for both frequencies. The figures show that Power-Sleuth successfully tracks power variation over time, with a worst case prediction of about 10% (phase $D$ for gcc/166 in minimum frequency).

### C. Power Measurement Infrastructure

In Sections V-A, V-B we used real power measurements to train our model and to evaluate our method. To achieve the high level of accuracy and resolution we need for Power-Sleuth, we created the infrastructure depicted in Figure 10. We use current sensors [5] to measure the current through each voltage rail supplying the motherboard: the main ATX connector 3.3V, 5V and 12V, as well as the separate 12V rail supplying the processor. Though only the latter one is needed for this work, our future work includes extending Power-Sleuth for estimating uncore (L3 cache and memory controller in Intel architectures) and memory power, thus our measuring setup tracks them as well. The current sensors are 3-pin components. Pins 1 and 2 are connected across a sense resistor $R_s$. The current flows through the resistor and, depending on this current, an output voltage is produced in pin 3. To measure currents for the whole system, we design a PCB which is installed between the power supply and the motherboard, with a current sensor inserted between the two ends of the cables of interest. The outputs of the sensors are connected to a 16-channel Analog-to-Digital data acquisition device capable of sampling 200K samples per second. We use a sampling rate of 1KHz per channel, which is enough for measuring power at a phase granularity and evaluating the accuracy of phase-power prediction.

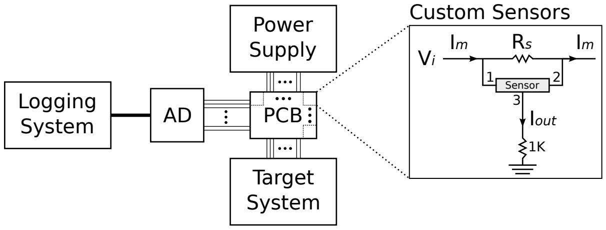


Figure 10. Our power measurement setup. A PCB board with an array of current sensors is inserted between the power supply and the motherboard of the target system. The outputs of the sensors are tracked by an ADC, which sends the results through a USB connection to a separate logging PC.

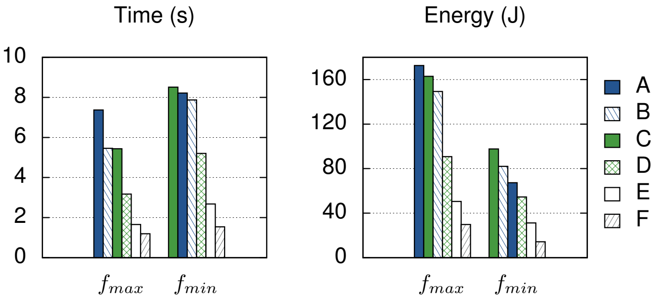


Figure 11. Per-phase execution time and energy for gcc/166. The 6 most important phases are shown. The figure shows that ranking of phases is different for execution time and energy, and also depends on frequency.

## VI. Using Power-Sleuth

Having evaluated the accuracy of Power-Sleuth in the previous sections, in this section we demonstrate how a user can employ the tool to characterize and possibly improve the energy efficiency of an application.

### A. Phase-Energy Characterization

Power-Sleuth can give the user a breakdown of execution time and energy of program phases, for any possible frequency. Figure 11 shows the 6 most important phases of gcc/166. As the figure shows, ordering the phases by execution time depends on frequency: phase $A$ is the longest running in maximum frequency, but in minimum frequency it is phase $C$ that runs for the longest time. By comparing execution times of the phases for maximum and minimum frequency, it is obvious that phase $A$ is memory bound (execution time is not affected much by

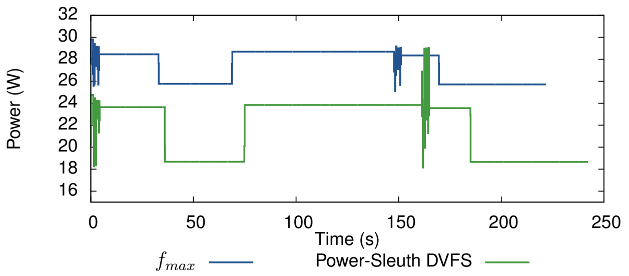


Figure 12. Power over time predicted by Power-Sleuth for astar/lakes. The two curves show power for maximum frequency, as well as power when a DVFS schedule that aims to minimize EDP within 10% of performance penalty is applied. The schedule was calculated by Power-Sleuth, and the prediction for this schedule is performed without an extra run of the application.

frequency scaling), whereas phase $C$ is CPU bound (scaling frequency down results in significant performance overhead). Using this information, the user can select to run phase $A$ in low frequency, by injecting a DVFS command in the source code, at the beginning of phase $A$. The energy consumed in this case will be reduced from 170J to 70J, as shown in the right part of Figure 11. Alternatively, since memory is the bottleneck in phase $A$, one can try to optimize the memory behavior of this phase using appropriate tools [1]. By doing so, power will increase (since the processor will execute instructions at a higher rate), but execution time will decrease, so energy could either increase or decrease: in this case, the user can evaluate both versions of the program using Power-Sleuth and pick the most efficient code.

Phases $B$ and $C$, on the other hand, are significantly affected by frequency scaling and thus are CPU-bound phases. Frequency scaling should be avoided in such phases to keep performance at a high level. High energy consumption in these phases comes from high processor utilization, and thus this piece of code is already efficient enough. However, the user can still apply various optimizations, such as improving hit rate in the lower levels of the cache hierarchy or reordering instructions to increase Instruction Level Parallelism, and see how these optimizations impact energy.

### B. Optimal DVFS Schedule

After the profiling run, Power-Sleuth has all the required data to predict performance and energy under any frequency. It is thus capable of providing an optimal DVFS configuration for each phase, according to user specifications. The metric of interest for optimization can be any metric involving performance or power, such as minimum EDP (Energy Delay Product) or ED$^2$P, or even metrics under constraints (e.g., limited performance penalty, etc.). Figure 12 shows astar/lakes power consumption over time predicted by Power-Sleuth for maximum frequency, as well as for the recommended DVFS schedule. In this example, minimizing EDP within a performance overhead of 10% for each phase is our optimization criterion. The schedule then can be applied in practice with DVFS commands at the beginning of each phase.

## VII. Related Work

In this section we discuss work related to power profiling and estimation.

**Power measurement.** Ge et al. [9] developed a tool, called PowerPack, to profile power dissipation of a computer system (processor, memory, disks etc.). The authors use extra hardware to measure power consumption, and they modify the target application by inserting library calls to communicate with the profiler so as to monitor specific code regions. Instead, we only use real power measurements once, to get the characteristics of the processor, and then we estimate power of applications at a program-phase granularity (10s of msec). Moreover, we do not modify the target application, since mapping of power estimation with code is achieved using the phase-detection component.

**Power estimation.** Joseph and Martonosi [17] correlate power with hardware performance counters to estimate power consumption. They evaluate their method in both a simulator and a real processor. The authors use circuit-level power information and approximate component activities using heuristics (when this information is not readily available from the performance counters) to get a per-component breakdown of power consumption. Conrteras and Martonosi [3] use a total of 7 performance counter events to estimate core and main memory power consumption in Intel XScale processor. The power model formed can be parameterized in the sense that different regression parameters are used for the different processor Voltage-frequency configurations. More recently, Goel et al. [10] followed a similar approach to get power estimation for multithreaded and multiprocess workloads and employ it for implementing a power-capping scheduler.

Unlike the above mentioned approaches, we correlate *effective capacitance* (the processor's capacitance coupled with node activity factors) [29] with processor performance counters events and then estimate power using Equation (4). This approach allows us to have a unified model across all different $V$-$f$ combinations.

**Phase detection.** Isci and Martonosi [16] compared control-flow-based (e.g., ScarPhase [25], SimPoint [27]) phase detection with power-event-counter-based [14, 15] phase detection. They used 15 power related performance counters to monitor the execution and to detect phases. They found that event-counter-based phase detection produced slightly more accurate results (i.e., more homogeneous power behavior within phases). However, the goal of their work was to use phase detection for runtime optimizations (i.e., apply new hardware settings at phase changes). In this work, we focus on profiling, and we want to map the power profile back to the code, which control-flow-based phase detection does by default.

## VIII. Conclusions and Future Work

In this paper we introduced Power-Sleuth, a tool that estimates performance and power consumption of an application in different frequencies. The tool is capable of characterizing the behavior of an application in any frequency, from a set of data collected in a single frequency. To achieve this, we utilize an analytical DVFS performance model, a novel power-estimation model and a phase detection and classification technique. We show that we can predict power and performance with high accuracy, not only for the whole execution of an application, but also for each program phase individually. Finally, we show use-cases of how the information provided by Power-Sleuth can be used to improve the power-efficiency of an application.

In our future work we plan to extend our methodology to account for memory and uncore power, as well as port and evaluate Power-Sleuth in more platforms.

## References

[1] Roguewave threadspotter. URL `http://www.roguewave.com/products/threadspotter.aspx`.

[2] Intel vtune. URL `http://www.intel.com/software/products/vtune/`.

[3] G. Conrteras and M. Martonosi. Power prediction for intel xscale processors using performance monitoring unit events. In *Int. Symposium on Low Power Electronics and Design*, 2005.

[4] A. S. Dhodapkar and J. E. Smith. Comparing program phase detection techniques. In *Int. Symposium on Microarchitecture*, 2003.

[5] *ZXCT1009: High-side Current Monitor*. Diodes Incorporated, document number: ds33441 rev. 12 - 2 edition, April 2011.

[6] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*, chapter 10.11. On-line Clustering, pages 559–565. Wiley-Interscience, 2 edition, 2001. ISBN 0-471-05669-3.

[7] S. Eyerman and L. Eeckhout. A counter architecture for online dvfs profitability estimation. *Computers, IEEE Transactions on*, 2010.

[8] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 2009.

[9] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron. Powerpack: Energy profiling and analysis of high-performance systems and applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5): 658–671, may 2010.

[10] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *Int. Green Computing Conference*, 2010.

[11] J. L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[12] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, volume 3a: system programming guide edition, September 2010. 14.7 Platform Specific Power Management Support.

[13] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, volume 3b: system programming guide edition, September 2010. 30.4.4 Precise Event Based Sampling (PEBS).

[14] C. Isci and M. Martonosi. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Int. Symposium on Microarchitecture*, 2003.

[15] C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *Int. Workshop on Workload Characterization*, 2003.

[16] C. Isci and M. Martonosi. Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *Int. Symposium on High-Performance Computer Architecture*, 2006.

[17] R. Joseph and M. Martonosi. Run-time power estimation in high performance microprocessors. In *Int. Symposium on Low Power Electronics and Design*, 2001.

[18] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st annual international symposium on Computer architecture*, 2004.

[19] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008. ISBN 0-471-05669-3.

[20] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval based models for run-time dvfs orchestration in superscalar processors. In *Int. Conf. on Computing Frontiers*, 2010.

[21] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Int. Symposium on Performance Analysis of Systems and Software*, 2004.

[22] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. Technical Report Version 1.0, Intel Corporation, 2009.

[23] N. Peleg and B. Mendelson. Detecting change in program behavior for adaptive optimization. In *Int. Conf. on Parallel Architecture and Compilation Techniques*, 2007.

[24] B. Rountree. Theory and practice of dynamic voltage/frequency scaling in the high-performance computing environment. *Ph.D. dissertation, University of Arizona*, 2010.

[25] A. Sembrant, D. Eklov, and E. Hagersten. Efficient software-based online phase classification. In *Int. Symposium on Workload Characterization*, 2011.

[26] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Int. Conf. on Parallel Architecture and Compilation Techniques*, 2001.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[28] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Int. Symposium on Computer Architecture*, 2003.

[29] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Int. Green Computing Conference*, 2011.