Phase Characterization for Power: Evaluating Control-Flow-Based and Event-Counter-Based Techniques

Blind Review

Abstract

Computer systems increasingly rely on dynamic, phase-based system management techniques, in which system hardware and software parameters may be altered or tuned at run-time for different program phases. Prior research has considered a range of possible phase analysis techniques, but has focused almost exclusively on performance-oriented phases; the notion of power-oriented phases has not been explored. Moreover, the bulk of phase-analysis studies have focused on simulation evaluation; there is need for real-system experiments that provide direct comparison of different practical techniques (such as control flow sampling, event counters, and power measurements) for gauging phase behavior.

In this paper, we propose and evaluate a live, real-system measurement framework for collecting and analyzing power phases in running applications. Our experimental framework simultaneously collects control flow, performance counter and live power measurement information. Using this framework, we directly compare between code-oriented techniques (such as "basic block vectors") and performance counter techniques for characterizing power phases. Across a collection of both SPEC2000 benchmarks as well as mainstream desktop applications, our results indicate that both techniques are promising, but that performance counters consistently provide better representation of power behavior. For many of the experimented cases, basic block vectors demonstrate a strong relationship between the execution path and power consumption. However, there are instances where power behavior cannot be captured from control flow, for example due to differences in memory hierarchy performance. We demonstrate these both with microbenchmarks and examples from real applications. Overall, counter-based techniques offer average classification errors of 1.9% for SPEC and 7.1% for other benchmarks, while basic block vectors achieve 2.9% average errors for SPEC and 11.7% for other benchmarks respectively.

1 Introduction

In recent years, phase behavior of applications has drawn a growing research interest for two main reasons. First, the increasing complexity and power demand of processor architectures mandate workload dependent dynamic management techniques. These techniques extensively benefit from tracking application phases to optimize power/performance trade-offs and to identify critical execution regions for management actions [1, 3, 9]. Second, in parallel with increasing processor complexities, architectural simulation studies develop a growing need to research long execution timescales to capture the increasingly variable behavior of applications. These studies benefit from phase characterizations that summarize application behavior with representative execution regions, alleviating the prohibitively high computational costs of large-scale simulations [28, 32].

Various prior studies demonstrated that phase behavior can be observed via different features of applications. Most of these approaches fall into two main categories: In the first category application phases are determined from the control flow of the applications or the program counter (PC) signatures of the executed instructions [9, 20, 16, 28, 31, 32, 33, 23]. In the second category, phases are determined based on the performance characteristics of the applications [3, 7, 11, 18, 35, 36].

Although there have been some previous efforts to compare or evaluate phase characterization techniques [2, 8, 22], they do not perform a direct comparison of the two main approaches. Moreover, there is generally a missing link between phase characterizations and their ability to represent power behavior, especially with real-system experiments. Such power characterization is very important for real-systems, as a primary goal of phase characterization is dynamic power management

of running systems.

Following from these motivations, in this work, we compare phase characterizations based on PC signatures and performance behavior of applications. Our study primarily evaluates these techniques for accurate power behavior characterization on a real-system. We compare these with respect to the actual, measured runtime power dissipation behavior of applications. Specifically, we look at phase analysis based on basic block vector (BBV) features of an application [32] to determine regions of similar power behavior. We compare this to phases determined by a particular set of performance monitoring counter (PMC) events that are chosen to reflect power dissipation [19]. We test the power characterization accuracy of these methods on 21 benchmarks from SPEC2000 suite and 9 other benchmarks derived from commonly used desktop and multimedia applications. We show that, in general, tracking performance metrics performs better than tracking control flow in identifying power phase behavior of applications. Additionally, we present specific examples from microbenchmarks and real applications demonstrating cases where power phase behavior cannot be deduced from code signatures.

There are three primary contributions of this work. First, we have designed an accurate, real-system method for synchronizing BBV signatures, performance events, and power measurements on running machines. This method allows us to study large-scale application behavior on running systems rather than being limited to simulation approaches. Second, utilizing this experimental framework, we evaluate how how BBV and PMC based approaches perform from a real power characterization point of view. Compared to an uninformed phase characterization, both phase based techniques achieve significantly higher accuracies in identifying power phases, leading to 2-6x less errors for benchmarks with significant power variations. Last, we compare control flow (BBV) and performance (PMC) based approaches against each other for their power phase classification abilities. Overall tracking performance behavior leads to 30-40% less errors than tracking control flow in representing real power phase behavior.

The rest of the paper is organized as follows. Section 2 discusses the reasons why control flow and performance phases can significantly differ. Section 3 describes our experimentation platform. Section 4 explains the collection of BBV and PMC information with our experimental setup. Section 5 describes our phase classification methods. Section 6 describes our quantitative evaluation and presents the power phase characterization results. Section 7 provides detailed observations from performed experiments. Section 8 provides a final discussion of BBV and PMC based approaches and their applications. Section 9 summarizes related work and Section 10 offers our conclusions.

2 What Control Flow Information Does Not Show

Before delving into the details of our experimentation and phase characterization methodology, here we discuss the reasons why control flow and power/performance behavior of an application may disagree. We then show the extent of disagreement for one case, with a synthetic benchmark example.

There are multiple aspects of application behavior that can cause the control flow and performance based approaches to reach different phase characterization conclusions. *Dynamic change in data locality* during an application's execution can cause the power behavior to significantly change. While this change can be easily recovered from memory related performance metrics, code signatures cannot reflect this as execution footprints are not altered. *Effectively same execution* represents the converse of the above effect. In various applications, multiple procedures or code segments perform similar processes, leading to practically identical power behavior. These are considered as fairly different phases in terms of control flow, which may result in many different phase clusters that do not reflect actual changes in program power. Typical examples for these are scientific or other iterative processing applications performing different tasks on an input with similar power/performance implications. *Operand dependent behavior* may result in similar effects as the first case, where power

and latency of a unit depends on the input operands, despite the same control flow. Typical cases for these are overflow handling and scaling of execution based on the input operand values or widths [4].

Below, we demonstrate a complete case study to show the differences that can arise between control flow and and performance based phase tracking for power. This presents one aspect of the sources of disagreement, varying data locality. We do not provide examples to the other two cases here for space limitations. However, we revisit these after presenting our power phase characterization study, with observations from real experimented applications.

2.1 Dcache Microbenchmark

We design a simple synthetic example benchmark, `dcache`, to demonstrate the effect of data locality on power and performance. While increasing the address range for data accesses impacts power and performance drastically with reduced cache affinity, this change of behavior goes unnoticed by the control flow observations.

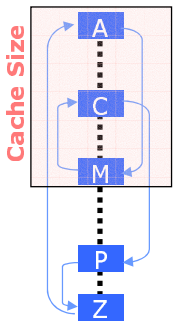
We implement this in the `dcache` microbenchmark with a random list traversal over a single dimensional vector. This vector is constructed such that, each vector element contains the address of the next element to be accessed. Each next access is determined randomly from the pool of yet "untouched" elements, thus providing uniform probabilities that each access can be to any location in the vector address space. The last accessed element is linked back to the first vector element, forming a complete cycle. The main microbenchmark loop then continuously travels through these links for a large number of iterations to avoid cold start effects.

In Figure 1, we show the relevant C and assembly code snippets for the main microbenchmark loop for random list traversal. Inside the C code, we also layout the generic vector traversal path constructed at initialization. The length of this vector determines whether the elements reside in L1 cache, L2 cache or memory. For example, for our experimentation platform, a single threaded Pentium 4 processor, L1 and L2 caches are 8KB and 256KB. Therefore, integer vector sizes less than 2K and 64K will be mostly resident in L1 and L2 caches respectively.

In both C and assembly codes, the parts in italics represent the main microbenchmark loop. `cmpl` instruction at 0*x*804874*C*

```
/* FOR VARIOUS ARRAY SIZES: */

  /* Initialize The Array Data: */
            :
```




```
            :
  /* Main MicroBenchmark Loop */
  array_index = 0;
  for (ind=0; ind<100000000; ind++)
  {
    array_index = data[array_index];
    count = count + 1;
  }
  /* MicroBenchmark END*/
```

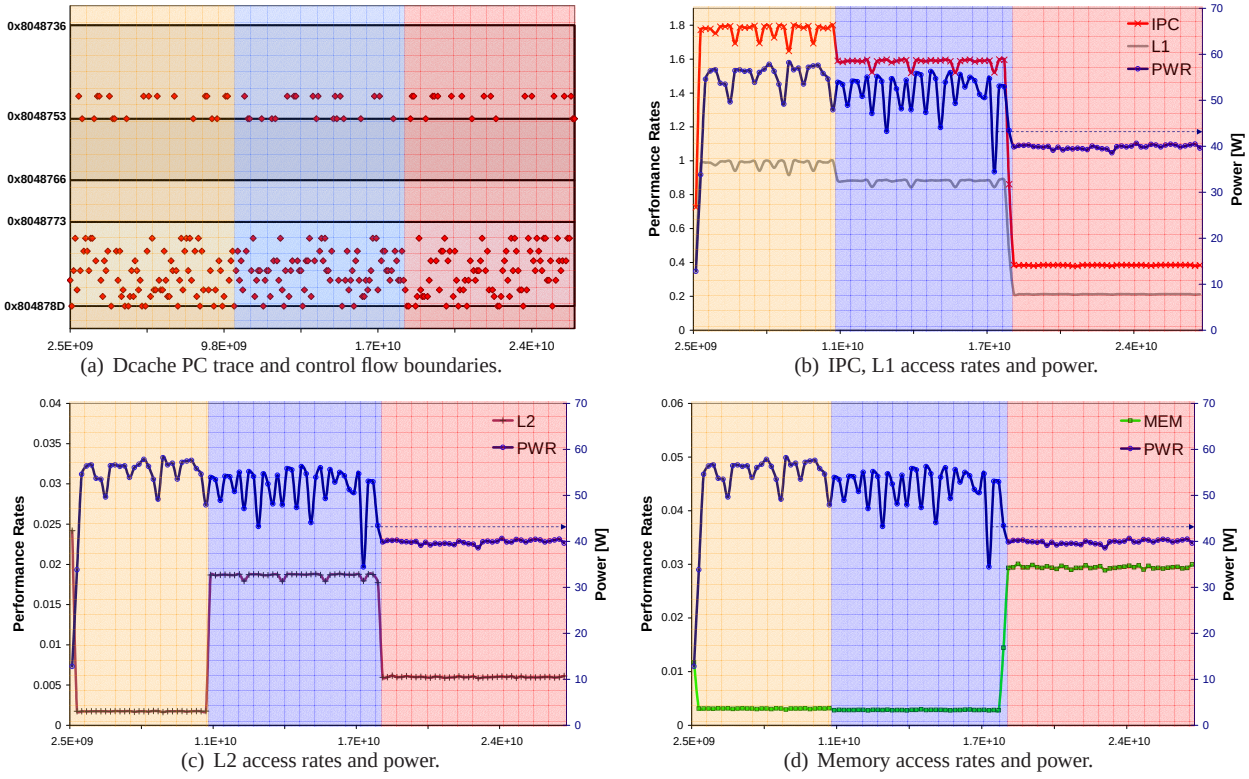
```
8048736:      call   80483b8 <_init+0x38>
804873b:      add    $0x10,%esp
804873e:      movl   $0x0,0xffffff88(%ebp)
8048745:      movl   $0x0,0xffffff8c(%ebp)
804874c:      cmpl   $0x5f5e0ff,0xffffff8c(%ebp)
8048753:      jle    8048778 <main+0x258>
8048755:      sub    $0x4,%esp
8048758:      pushl  0xffffff9c(%ebp)
804875b:      push   $0x8048922
8048760:      pushl  0x8049a98
8048766:      call   80483b8 <_init+0x38>
804876b:      add    $0x10,%esp
804876e:      lea    0xfffffa0(%ebp),%eax
8048771:      incl   (%eax)
8048773:      jmp    8048588 <main+0x68>
8048778:      imul   $0x4,0xffffff88(%ebp),%edx
804877c:      mov    0xfffffb4(%ebp),%eax
804877f:      mov    (%eax,%edx,1),%eax
8048782:      mov    %eax,0xffffff88(%ebp)
8048785:      lea    0xffffff9c(%ebp),%eax
8048788:      incl   (%eax)
804878a:      incl   0xffffff8c(%ebp)
804878d:      jmp    804874c <main+0x22c>
804878f:      nop
8048790:      mov    $0x0,%eax
8048795:      lea    0xfffffff4(%ebp),%esp
8048798:      pop    %ebx
8048799:      pop    %esi
804879a:      pop    %edi
804879b:      pop    %ebp
804879c:      ret
```

Figure 1. Dcache microbenchmark

is the loop exit condition check and `mov` instruction at $0x804877F$ is our "indeterministic load" to retrieve next vector index. In the assembly code, bold lines show the control flow boundaries, where the execution path may divert.

2.2 Showing Effect of Data Locality on Control Flow, Performance and Power Characteristics with Dcache

We show the control flow, performance and power behavior of `dcache` benchmark for three distinct vector configurations: *L1 intensive*, *L2 intensive* and *memory intensive*. In L1 intensive case, data vector practically resides in L1, leading to very high L1 cache hit rates for the vector element accesses. In L2 intensive case, vector accesses incur around 90% L1 misses, but almost perfect L2 hits. In memory intensive operation, many of the accesses also initiate a memory transaction. We acquire control flow information by sampling PC every 1 million instructions (with a random jitter of 100 instructions to eliminate biased sampling). We collect performance metrics—L1, L2 and memory access rates, and instructions per cycle (IPC)—by sampling PMCs. We collect power information from real measurements via a current probe.



(a) Dcache PC trace and control flow boundaries.

(b) IPC, L1 access rates and power.

(c) L2 access rates and power.

(d) Memory access rates and power.

Figure 2. Control flow, performance and power behavior of `dcache` microbenchmark during three modes of operation: L1 intensive, L2 intensive and memory intensive execution. In each plot, x axis is executed instructions.

In Figure 2, we show the resultant behavior in terms of control flow (a) and power/performance (b-d) for the three configurations. In each plot, the three shaded regions correspond to the three different configurations. First region corresponds to L1 intensive execution, 2nd to L2 intensive execution and 3rd region to memory intensive execution. Figure 2.(a), the solid horizontal lines represent the control flow boundaries as shown in the assembly plot of 1. The PC scatterplots show, which sequential execution parts the sampled instruction addresses fall into.

It is clearly seen from Figures 2.b-d that, the three configurations lead to distinctly different execution phases in terms of both power and performance. All performance metrics show very different behavior in all three phases. Power behavior change is subtle, but observable between L1 and L2 intensive modes. On the other hand, it is distinctly different between L2

and memory intensive execution. While performance metrics easily identify these three phases, there is no observable pattern in control flow behavior, as major executed code lies in the same control flow boundaries.

3 Software and Hardware Measurement Platform

To collect synchronous PC, PMC and power information during an application's execution, we use dynamic instrumentation via Pin [25]. Pin provides several flexible methods to dynamically instrument the binary at different granularities. This first step, *instrumentation*, simply decides where in the native code the additional procedures to analyze the application behavior should be inserted. Afterwards, whenever one of these instrumentation checkpoints are reached, Pin gains the control of the application and injects corresponding analysis routines. During execution, each time the instrumented locations are visited, their injected analysis routines also execute, providing the dynamic application information. This second phase of operation is called *analysis*. Although conceptually *instrumentation* and *analysis* are two exclusive processes. Pin operates similarly as a just-in-time (JIT) compiler. Instrumentation for a code trace happens immediately before it is executed for the first time. Therefore, *instrumentation* and *analysis* are usually temporally intermixed, with instrumentation-dominant execution at the beginning of an application and analysis-dominant execution towards the end, as very few new traces are encountered. Pin utilizes a single executable, *Pintool*, to perform instrumentation and analysis on an application. Each Pintool contains separate routines for instrumentation and analyses to perform these processes.

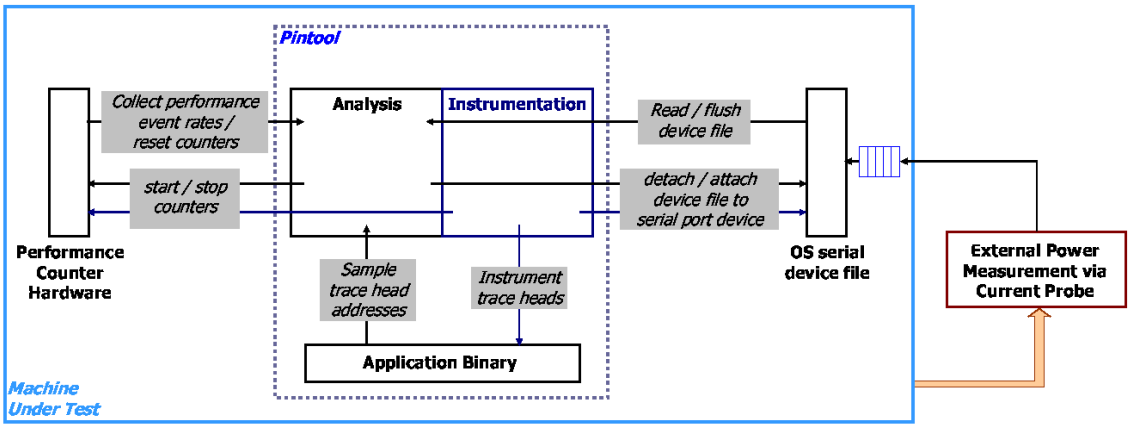
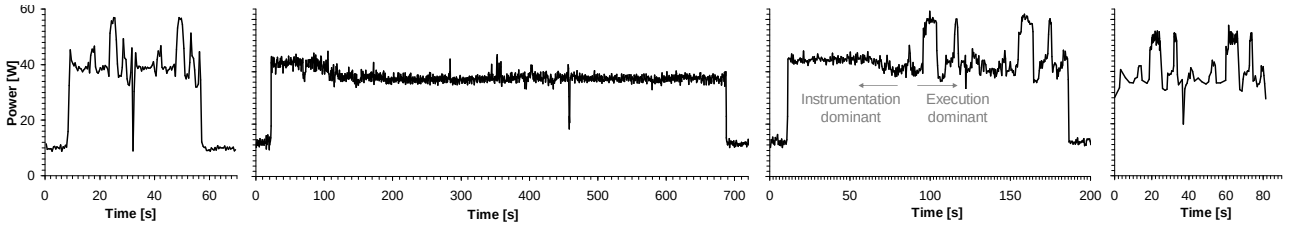


Figure 3. Experimental setup for power phase analysis with Pin.

Figure 3 presents an overview of our experimental setup for power phase analysis with Pin. In our Pintool, we use trace level instrumentation to keep track of executed code traces. Our analysis routine consists of three levels of hierarchy. First level simply provides an account of executed instructions. Second level samples one PC address approximately every 1 million instructions. Highest level analysis is evoked every 100 million instructions. This routine generates one BBV from the 100 PC samples, reads performance statistics from PMCs and logs the measured power history from the serial device file. These three sources of data collection are shown with the three incoming arrows to the analysis routine of our Pintool.

It is important to isolate application behavior from Pin operation. Pin provides application exclusive control flow information, however, performance monitoring and power measurements are independent of Pin operation. Therefore, we provide handles to our Pin routines to disable the logging of data for power and performance at routine entries; and to reenable data logging at routine exits. As we mentioned, instrumentation and analysis are not mutually exclusive temporally. Therefore, we use these handles during both instrumentation and analysis, as shown with the arrows in Figure 3. Nonetheless, there exist additional handles exclusive to the analysis tool such as reseting the PMCs and flushing device file at the end of a complete



(a) Native execution power behavior without instrumentation

(b) Flattened power behavior with Pin basic block instrumentation

(c) Improved external power behavior with Pin trace instrumentation and conditional inlining

(d) Power behavior assigned to application execution by Pintool

Figure 4. Effect of Pin instrumentation on application power behavior for SPEC `gcc` benchmark with `166` dataset. (a) shows the native power behavior without instrumentation. (b) shows Pin instrumentation overwhelming `gcc` power behavior with naive BBL instrumentation. (c) shows the retained power behavior with trace instrumentation and conditional inlining with external power measurement (including both Pin and `gcc`). (d) shows power behavior assigned to application as isolated by our Pintool power analysis routine (`gcc` only, by selectively disabling device logging during Pin execution).

sampling period. We provide further details to each of the data collection mechanisms in the following sections, where we also provide information on the fidelity of our experiments.

All our experiments described in this paper were performed on a 1.4GHz Pentium 4 processor with Linux operating system, kernel 2.4.7-10. The experiments are carried out with the SPECCPU 2000 benchmarks using reference datasets and other benchmarks derived from well-known suites and desktop applications. All benchmarks are compiled with gcc and g77 compilers with base compiler flags.

3.1 Instrumentation Details

In our experiments, initially we used basic block level (BBL) instrumentation to collect PC information. In this case, first analysis routine—called at every basic block entry—kept track of executed instructions. At every 1 million instructions, we called the 2nd analysis routine to log PC address. However, as on average every 5-7 instructions make a BBL, doing instrumentation at this granularity caused the instrumentation tool to overwhelm benchmark power behavior. Although it performs a minor operation, the 1st routine has been the bottleneck due to its call frequency. As this research relies on collecting real measured power behavior, the most important specific consideration is to be able to preserve the power variations induced by benchmark behavior. Therefore, we applied following optimizations to instrumentation to retain variation in power behavior:

- *Trace Instrumentation:* This is a one level coarser instrumentation than BBL granularity provided by Pin. A trace is a single entry, multiple exit (unlike single exit BBLs) code region. As we keep track of each trace head (i.e. each change in control flow), this method still provides required control flow information, while reducing the amount of instrumentation to $1/2 - 1/3$.

- *Conditional Inlining:* One obvious feature of our analysis hierarchy is that, first level analysis is performed very frequently, to determine the condition for applying the second level analysis. Here, the condition is, whether 1 million instructions are executed. Therefore, the task of first level analysis is to simply count instructions up to 1 million, and call the second level whenever the condition is met, at a much lower frequency. Starting with toolkit 1795, Pin API provides routines for inlining this first level condition checking. This is seen to drastically improve performance as long as the "then" routine has significantly lower frequency. Therefore, in our implementation, we make use of this feature. This helps reduce the execution times by 5x and also helps retain the original power behavior of applications.

In Figure 4, we show the effects of instrumentation on power behavior for `gcc` benchmark with `166` dataset. Figure

4.(a) shows the actual measured power behavior of `gcc` without any instrumentation. Figure 4.(b) shows the result of our initial experiments. Here, power behavior of `gcc` is overwhelmed by the frequent calls to the first level analysis routine. Consequently, the observed power behavior is rather flat, not reflecting the variations in actual benchmark execution. In Figure 4.(c), we show the improvement achieved with trace instrumentation and conditional inlining. This figure also shows the two phases of Pin operation clearly, where the initial stages of benchmark is instrumentation dominant and as lots of new traces are encountered and later it becomes execution and analysis dominant as mostly the previously entered traces re-execute. The power behavior reported in this case is from "external" measurements. We mean by this, an external power monitor is used to collect power information during the execution of the application within Pin. Therefore, what we see is the aggregate power behavior of both Pin and `gcc`. As we have briefly discussed, we use Pin to exclusively collect isolated benchmark power behavior, by excluding power logging during Pin instrumentation and analysis routines. We show in Figure 4.(d) the actual observed power behavior for `gcc` by our Pintool. Here, we can demonstrate most of the instrumentation (and analysis) effects are filtered out from the power behavior seen from within Pin with our power monitoring method. Note that this trace cannot be identical to that of Figure 4.(a). Although what we measure is mostly benchmark execution, there are inevitable differences due to inlined first level analysis routine, and imperfect synchronization of power and performance measurements. Nonetheless, it can be seen that the original temporal power behavior of the application, as well as the magnitudes of application power variations are both preserved, which are the most important prerequisites of our research.

3.2 External Power Measurement and Pin Interface

We provide real power behavior feedback to our power phase characterizations via external, live power measurements. We perform power measurements by measuring the current flow into the processor with a current probe. This measurement information is then fed back to the measurement system over the serial port interface.

This power measurement is performed continuously at runtime, reflecting the actual power consumption as experienced by the processor. Therefore, this power behavior reflects the aggregate effects of all the Pin tasks running on the system at a given time period. As the purpose of our power characterization research is to efficiently identify phases reflecting real power behavior of an application, it is crucial to be able to disect the application power behavior from Pin analysis and instrumentation. To perform this, we use certain controls from the instrumentation and analysis routines of our Pintool eliminating their contribution to collected power information.

In our experimentation system, measured power information comes to the serial interface as 15 bytes of ASCII followed by a newline. By default, this data passes through the serial buffer, and is silently logged into a 4KB serial device file—if the serial port is programmatically opened. The tail of this file extends as new data is received from the serial port and current file position advances as data is read by the user level program. In order to log only application execution specific power data in this file, we use separate controls inside the Pintool routines that can detach/attach serial device driver from the device file via `termios` flags. This approach allows us to preserve previous power history, while preventing further logging while inside an instrumentation/analysis routine. At the exit of a routine, the driver is enabled to log further power data for continued application execution. At the end of a 100 million instruction sampling period, the highest level analysis function halts logging and reads the logged power history for the past sampling period. This history is then averaged and is assigned as the observed power for the past sampling quantum. Afterwards, the buffer is flushed and reenabled for logging at the start of next sampling interval.

With this approach, we provide a valid matching between application execution flow, performance statistics and application specific power behavior. Inevitably, there exist sources of error due to measurements, transient operations that perform the control functions for selective logging and asynchronous operation of different data sources. However, in the experi-

mented cases, our selective power collection process produces power information with good fidelity, as compared to native execution behaviors of applications. Measured power behavior in all cases are similar, both temporally and in terms of delta variations.

4 Generating BBV and Performance Information from Pin/Hardware Structure

4.1 Program Counter Sampling and BBV Generation

To track control flow based application phases, we use the basic block vector (BBV) approach [32]. BBVs summarize application execution by tracking both which basic blocks of the application are touched and how many times each basic block is visited during a sampling interval. BBVs are shown to represent application execution behavior by providing both working set information and execution frequencies for different basic blocks [8]. BBVs are constructed from execution flow by mapping executed PC addresses to the basic blocks of an application binary. Originally, each component of a BBV is a specific basic block, and the magnitude of the component represents how often the corresponding basic block has been executed for a past sampling period. For practical purposes, BBVs are generally mapped into smaller dimensional vectors via random projection/hashing or eliminating least significant dimensions [2, 22, 32, 33].

In our implementation, we use Pin to sample the PC addresses at trace heads. Using trace instrumentation provides certain advantages to BBV generation. As each trace head is also a basic block start address, each sampled PC actually corresponds to a specific basic block. This eliminates any need for prior profiling of the binary to identify basic block boundaries and searching through these boundaries at runtime PC sampling to map the PCs to basic blocks. Consequently, different sampled PCs represent different elements of the BBV and number of samples for a specific PC represents the execution frequency of the corresponding block. For sampling periods, we use previously published granularities [2]. We sample one PC every 1 million instructions and construct a BBV at every 100million instructions. Thus, each BBV has an L1 norm of 100. We perform static instrumentation of applications with gcc to determine the dimensions of basic block profiles. Even after eliminating untouched basic blocks and libraries, applications exhibit large BBV dimensions ranging from 33000 (`gcc`) to 100 (`swim`). These lead to highly sparse and impractical to implement BBVs. Consequently, we also apply dimension reduction. For the reduced dimensions, we choose 32 buckets, based on previous work [33]. We use a variation of Jenkins' 32 bit integer hash function [21] to reduce the large and variable BBV dimensions into common 32 dimensional vectors.

As has been discussed in previous studies [22], sampling always incurs some amount of information loss. However, for any practical implementation of runtime control flow tracking, sampling is inevitable. As we have shown, our choice of sampling provides acceptable intrusion to program power behavior. In addition, our observations show, our sampled PC information still leads to similar similarity information for large scale control flow behavior. We compare full-blown BBVs, constructed from complete PC information, to our sampled BBVs with similarity matrices [32]. Both methods reflect the major phase content in terms of execution flow similarity and lead to similar phases for small numbers of target phase clusters.

4.2 Monitoring Using Performance Counters

Our performance oriented methods read performance counters at runtime via handles in our Pintool. In order to track power phases, we use a set of 15 performance counters that are good proxies for power estimation. The counters include CPU instruction counts, L1 and L2 access rates, and bus utilizations for memory behavior. The method is similar to prior research [19], but streamlined to avoid counter rotations. The final set of 15 PMC events can be monitored simultaneously without conflicts. Therefore, no PMC configuration is required except at the initial Pintool startup.

We developed several handles to control PMC monitoring from within our Pintool. At Pintool initialization, we use

PrepCntrs call to configure the 15 events to be monitored. This is the most heavyweight call, and is called only once, before application execution commences. We provide *StartCntrs* and *StopCntrs* calls to selectively start/halt performance monitoring at instrumentation and analysis routine exits/entries. These are used to avoid polluting the PMC information with Pin execution. Although we provide the start/stop handles to all routines, after our initial experiments, we do not invoke them for instrumentation and 2nd level analysis routines, as their costs are seen to be comparable. Note that, this trade-off only effects PMC information, without any effect on BBV generation and power measurements. Our experiments show that, PMC information still performs superior in phase characterization and most of the large scale phase behavior is preserved. After every 100 million instruction execution, the highest level analysis routine calls *ReadCntrs* to collect the past performance statistics for the current sampling period. These 15 event rates are used to construct a 15 dimensional *PMC vector* which is used the gauge the similarity of execution samples in a similar manner as BBVs. After the collection of PMC information, the analysis routine resets the counters with *ResetCntrs* and initiates monitoring for the next sampling interval.

5 Phase Classification

We cluster gathered BBV and PMC vector samples into phases with multiple clustering algorithms. First, we develop a fast, but less accurate method based on the descriptions of previous work [18]. This method is more suitable for runtime analysis as it assigns samples to phases as they are observed. We call this method *First Pivot Clustering*. To corroborate the observed characterization results are not due to the choice of clustering, we also experiment with a very computationally expensive method, *Agglomerative Clustering*. We use two variations of this method: *complete linkage* and *average linkage*. Patil et al. [28] show in their representative phase generations, SPEC INT and FP lead to on average 4 and 5 phases respectively. Therefore, in this study, to provide consistent results and error metrics across all applications, we target towards 5 final phases for all benchmarks. Afterwards, we show that observed results are consistent as the target number of phases changes.

5.1 First Pivot Clustering

First Pivot Clustering uses *pivot* samples to represent different phases. In the original description of this method, a new gathered sample is compared to all previous pivots, i.e. starters of different phases. If the current sample is within a specified threshold distance of a pivot, it is assigned to that phase. If it is not within the similarity distance of any of the pivots, it starts a new phase and is added to the list of pivots as the representative sample for the new phase. By this way, The original description can assign samples to phases at runtime. This approach provides an upper bound to the distance within each phase, but it does not guarantee a fixed number of phases.

We change this to an iterative process, where the threshold is changed dynamically based on both the acquired and target number of phases. With this modification, we classify both BBVs and PMC vectors into 5 final phases after a few iterations.

5.2 Agglomerative Clustering

Agglomerative clustering is a tedious bottom-up approach to clustering samples into phases. In this approach, clustering algorithm starts with an initial clustering solution of N clusters, where N is the number of samples. At each iteration, the algorithm compares all pairwise combinations of the current set of clusters and finds the best candidate pair of clusters to combine into a single cluster. The pairs are compared based on a *linkage* criterion, which determines the best candidates. This iterative process continues until a final target number of clusters are reached or a distance threshold among clusters is exceeded. For agglomerative clustering, we experiment with two types of linkages, complete and average linkage. We describe these below.

5.2.1 Average Linkage

Average linkage compares the average distance between all sample pairs belonging to two different clusters. For two clusters with i and j samples respectively, it computes the distance between all the $i \cdot j$ pairs and finds the average distance between the clusters. Performing this operation for all cluster combinations, it chooses to combine two clusters with the minimum average distance. This leads to clusters with similar ranges in all dimensions, but can result in significantly different ranges for different clusters.

5.2.2 Complete Linkage

Complete linkage does similar comparisons as average linkage. However, it compares the maximum pairwise sample-distance among clusters. It combines the clusters with the least maximum distance among all their pairs. Consequently, the final set of clusters have similar ranges among most of their samples, although the range across each dimension can be different.

In all our analyses we use L1—manhattan—distance, as our measure of distance between two samples. For BBV based clustering, we compute the L1 distance between the two corresponding 32 dimensional BBVs. For PMC based clustering, we use the two 15 dimensional PMC vectors to gauge the similarity between points. We apply above three clustering methods and evaluate clustering criteria based on these distances.

6 Power Phase Characterization: Evaluation of Techniques and Results

We apply our described power phase classification methods to several benchmarks. Using both control flow and performance features, we cluster each benchmark into 5 phases with multiple clustering methods. Here, we discuss first how we evaluate the fidelity of these phases in terms of power behavior characterization. Afterwards, we provide the complete set of results based on these evaluations. With the demonstrated results, we show how code signatures and PMC phases perform in identifying power behavior characteristics with respect to a "gold standard" phase classification as our lower bound and an "uninformed" classification as the upper bound. We also present a direct comparison between BBV phases and PMC phases for power characterization.

6.1 Evaluating the Error of Power Phase Characterization

We evaluate the quality of generated phase clusters by comparing the measured power at each sample to the aggregate power for the whole cluster the sample belongs to. For a benchmark with N samples, each sample i $(i = 1,\ldots,N)$ is an element of one of the final phase sets P_j $(j = 1,\ldots,5)$. Each sample has a corresponding set of data $[bbv_i, pmc_i, pwr_i]$, where bbv_i and pmc_i are the corresponding BBV and PMC vectors used during phase clusterings, and pwr_i is the measured power value during sample i's execution. For each phase P_j, we compute a "representative power", R_j, as the arithmetic average of the power values for the total N_j samples belonging to that phase. Then, for each sample i, we compute the squared difference between the sample's actual power value pwr_i and the representative power R_j for its owner phase P_j. We denote R_j values corresponding to each sample i with Rj_i. Afterwards, we compute the rooted average of these squared differences over all samples for our final RMS error figure E_{RMS}. We summarize this error computation in Equation 1.

$$R_j = \frac{\sum_{i \in P_j} pwr_i}{N_j} \qquad (j = 1,\ldots,5)$$

$$E_{RMS} = \sqrt{\frac{\sum_{i=1}^{N} (pwr_i - Rj_i)^2}{N}} \tag{1}$$

This error value represents the quality of power phase characterization for a given phase classification method on the evaluated benchmark. The methods are the combinations of tracked feature (BBVs or PMCs) and clustering algorithm (first pivot, agglomerative with average or complete linkage). We use this error measure to gauge the effectiveness of BBV and PMC based features in representing power phase behavior of applications in our experiments with various benchmarks.

6.2 Error Boundaries

To gauge the ability of the phase classification techniques in discerning application power behavior, we also provide the error boundaries that can be achieved with perfect knowledge of power information—lower bound—as well as without any knowledge of application behavior—upper bound.

To compute lower error bounds, we look directly at the measured power, which is the independent target experiment parameter in all other analyses. We apply all three clustering algorithms to each benchmark's power information and for each case choose the smallest error value achieved. We refer to this "gold standard" measure as *baseline error* in our results.

For the upper error bounds, we design a separate clustering method, which assigns each sample to any of the final target phases randomly, without using any application behavior information. We refer to the results of this "uninformed" phase characterization as *random error*. We show the results achieved with these approaches to for each benchmark. These demonstrate opportunities for improvement that remain and how much improvement each tested phase analysis feature brings to power characterization.

6.3 Experimented Benchmarks

For our power phase analysis experiments, we obtain control flow, performance and power characteristics for several benchmarks on our test machine. We look at 11 SPEC INT benchmarks—all except `perlbmk` due to compilation problems—and 10 SPEC FP benchmarks—excluded are F90 benchmarks. We experiment with all reference datasets for the 21 SPEC benchmarks leading to a total of 37 different experiments.

In addition to SPEC, we also use 9 other benchmarks from previous studies and derived from well-known applications. These benchmarks are `ghostscript`, `dvips`, `gimp`, `lame`, `cjpeg`, `djpeg`, `mesh`, `stream` and `mdbnch`. For some cases, we alter the dataset or iterations for the benchmarks to achieve longer execution times. We describe these benchmarks and any modifications here.

In the first category, `ghostscript` and `dvips` are conversion utilities commonly used in document creation. Their behavior can depend on the nature and layout of the input document. Next, `gimp`, `lame`, `cjpeg` and `djpeg` and media processing tools, used to convert among formats or manipulate media files. Last, `mesh`, `stream` and `mdbnch` are iterative applications with multiple sequential functions similar to many scientific computation tools.

For `ghostscript` and `dvips` we use a large document of 190 pages, with different size images in the middle of document. `ghostscript` converts a postscript input to pdf, and `dvips` converts dvi input to postscript.

`Gimp` is an image manipulation tool [13]. We use `gimp` in batch mode to perform several image processing operations such as blurring, filtering and applying digital effects. Depending on the computation and memory intensity of the applied functions, they can lead to different power behavior. We use `lame` MP3 encoder [34] to encode a wave file under varying quality settings. Both power levels and the total execution increase with the quality settings. `Cjpeg` and `djpeg` are image compression and decompression programs from MediaBench [24]. We use `cjpeg` to encode a very large (110 MB) ppm image file into jpeg and `djpeg` to decode the jpeg file into ppm. Their power behavior also changes during execution and with input data.

`Mesh` is a well-known program used in dynamic program optimization studies [10, 30]. It performs various computations

over the input mesh edges and faces, with sequentially executed repetitive functions. Our mesh input consists of 10K nodes and 60K edges, leading to very quick iterations. To emphasize the execution of separate functions, we alter the original `mesh` code to repeat each function 100-200 times. `Mdbnch` is a relatively older, scalar molecular dynamics benchmark [12]. It performs seven different molecular dynamics tasks with different sizes or complexities. To extend its execution, we increase the number of time steps for each task by 4-50x. Both `mesh` and `mdbnch` have similar iterative properties of scientific computation. Although they iterate within different control paths, each task usually has similar computation properties—except for changes in memory intensity. These lead to fairly flat behavior with small data footprints. `Stream` is actually a synthetic benchmark, commonly used to measure sustainable memory bandwidth [27]. It iterates over four small tasks doing different computations. Similar to the above two applications, `stream` also exhibits a stable power behavior during normal operation. However, it has a loop carried positive feedback that eventually overflows the inputs for its tasks, resulting in a drastic change in power behavior. For our `stream` experiments, we use an iteration count of 275 and data size of 2million entry arrays.

6.4 Power Phase Characterization Results

We show the overall results for our experiments in Figures 5-7. Three figures show phase characterization errors for the three clustering algorithms. In each figure, we show the upper—*random*—and lower—*baseline*—error bounds for each application and the achieved error with BBV and PMC based approaches. We also show the average accuracies for SPEC INT, SPEC FP and other experimented benchmarks.

First, obtained characterization results are consistent, independent of the applied clustering algorithm. In general Figure 5 shows relatively higher errors due to the cheaper clustering method. However, the general accuracy relation between BBVs and PMCs are preserved.

Comparing among the three sets of applications, SPEC FP applications lead to relatively low errors even with random phase clustering for some cases. This is due to the generic flat power behavior of these benchmarks (`applu`, `art`, `sixtrack`, `wupwise`). In some other cases, benchmarks go through specific initialization (i.e. `equake`) or periodic (i.e. `ammp`) phases with significant changes in all control flow, performance and power features. In these cases, both BBVs and PMCs achieve very good power characterizations approaching baseline errors.

SPEC INT shows significantly higher errors for all approaches due to higher variations in behavior. In many of the shown cases, BBVs and PMCs are seen to have significant improvement over random clustering. This shows the benefits of phase tracking for power behavior characterization.

Most of the other experimented benchmarks show significantly higher error ranges due to their high power variability based on input data characteristics and functional behavior. In these cases, applying phase analysis, especially with PMCs, proves to be very useful in identifying similar power behavior.

Overall, for the three benchmark sets, BBV phases achieve errors on average 52% of random clustering errors, for benchmarks with non-flat power behavior. PMC phases lead to 34% of random errors. For PMC based approach, power characterization accuracies vary between 2-6x improvements over random clusterings for these benchmarks. Performing same comparisons with respect to baseline errors show, BBVs on average achieve 2.9x higher errors compared to baseline, while PMCs errors 1.8x of baseline figures. These comparisons show, BBV and PMC phase analyses have significant benefit in characterizing power behavior. However, there still exist opportunities to improve power phase behavior characterization of applications.

As above measures also indicate, in almost all experimented cases, PMC based phase analysis performs better than BBV based approach for representing power behavior. Direct comparison shows, PMCs lead to 2.2% and 1.4% errors for SPEC

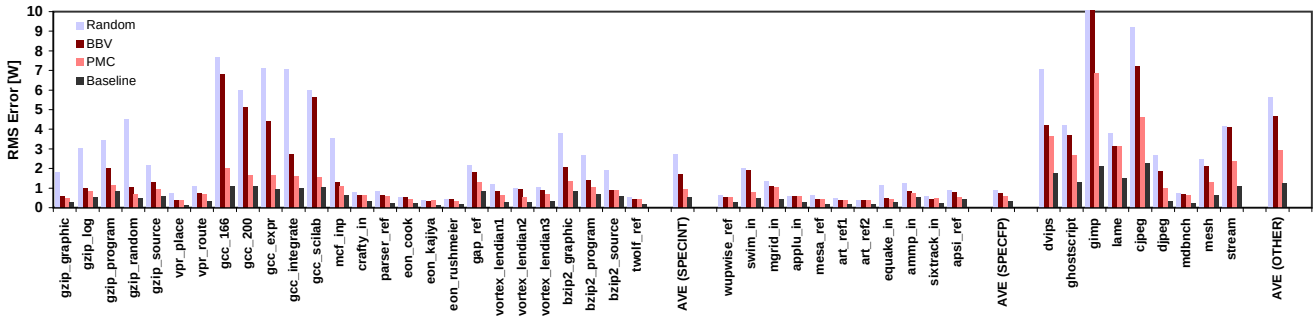


Figure 5. Power characterization errors (absolute) for BBV and PMC phases with first pivot clustering.

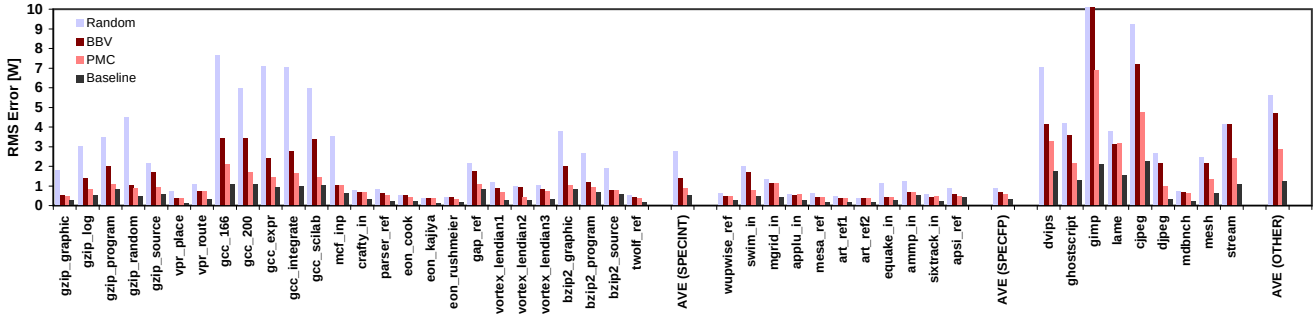


Figure 6. Power characterization errors (absolute) for BBV and PMC phases with agglomerative clustering-average linkage.

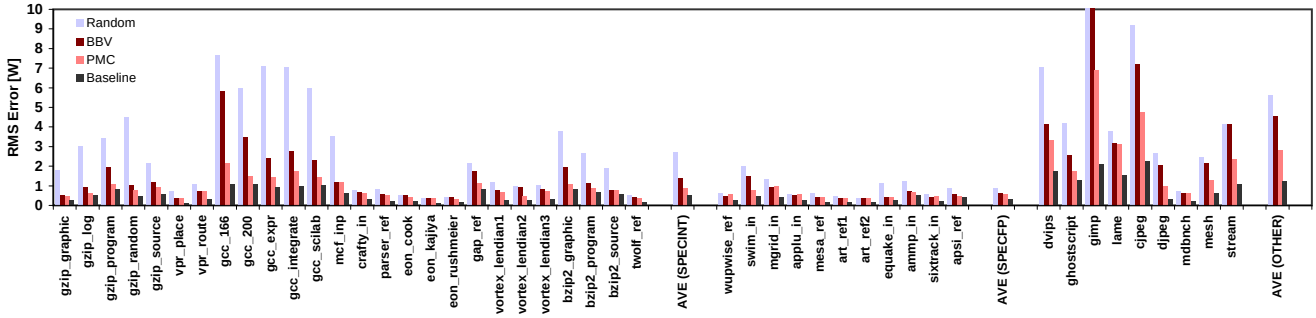


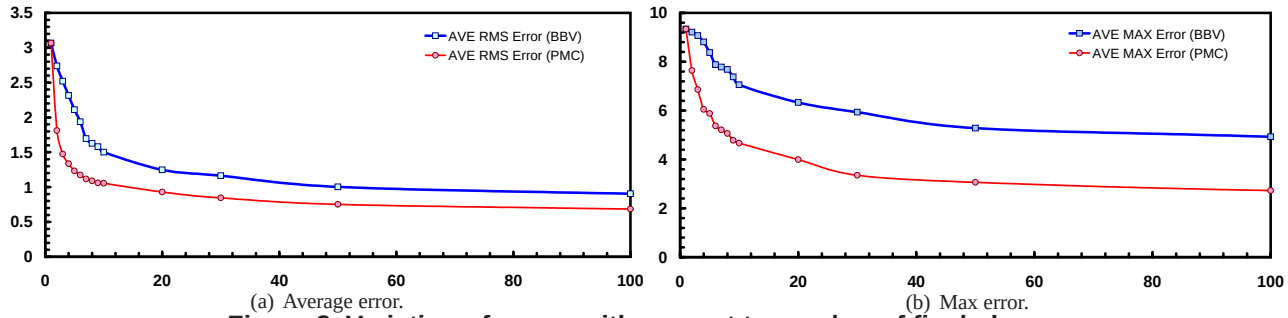
Figure 7. Power characterization errors (absolute) for BBV and PMC phases with agglomerative clustering-complete linkage.

INT and FP, while BBVs achieve 3.4% and 1.5% errors. For the other experimented benchmarks, PMCs and BBVs have 7.1% and 14.7% average errors respectively. For most of the benchmarks PMCs achieve 30-40% less errors than BBVs with an average of 33%. This direct comparison between BBVs and PMCs show, although both techniques provide useful features to identify power phase behavior, in general PMCs features are better candidates for identifying power phases.

6.5 Sensitivity to Different Target Number of Phases

We have presented our complete analysis for a fixed target number of 5 phases for consistency. However, we have also experimented with various number of target phases to verify the reliability of our results. We show these in Figure 8.

Here, we show the effect of target phases with agglomerative clustering/complete linkage. For all the benchmarks, we perform clusterings for final phase numbers varying from 1 to 5000. We show the achieved errors as both RMS and maxi-



(a) Average error. (b) Max error.

Figure 8. Variation of errors with respect to number of final phases.

mum observed values. For each benchmark, we compute the RMS and maximum error figure for each target phase count. Afterwards, we average these values over all benchmarks to reach a single error figure for each target phase count.

Intuitively, for a single final phase, both BBVs and PMCs will reach the same error, equivalent to the power standard deviation of the whole benchmark samples. Afterwards, as the number of phases increase, errors for both methods will decrease with different slopes. As phase numbers grow towards infinity, both error curves will converge to 0, i.e. where each phase is a singleton sample.

In Figure 8, we show the behavior up to 100 phases for demonstration purposes. As phase counts grow beyond 100, both curves reach 0. For all practical purposes, PMC based phases perform consistently better, independent of the number of final phase clusters.

7 Observations from Experimented Applications

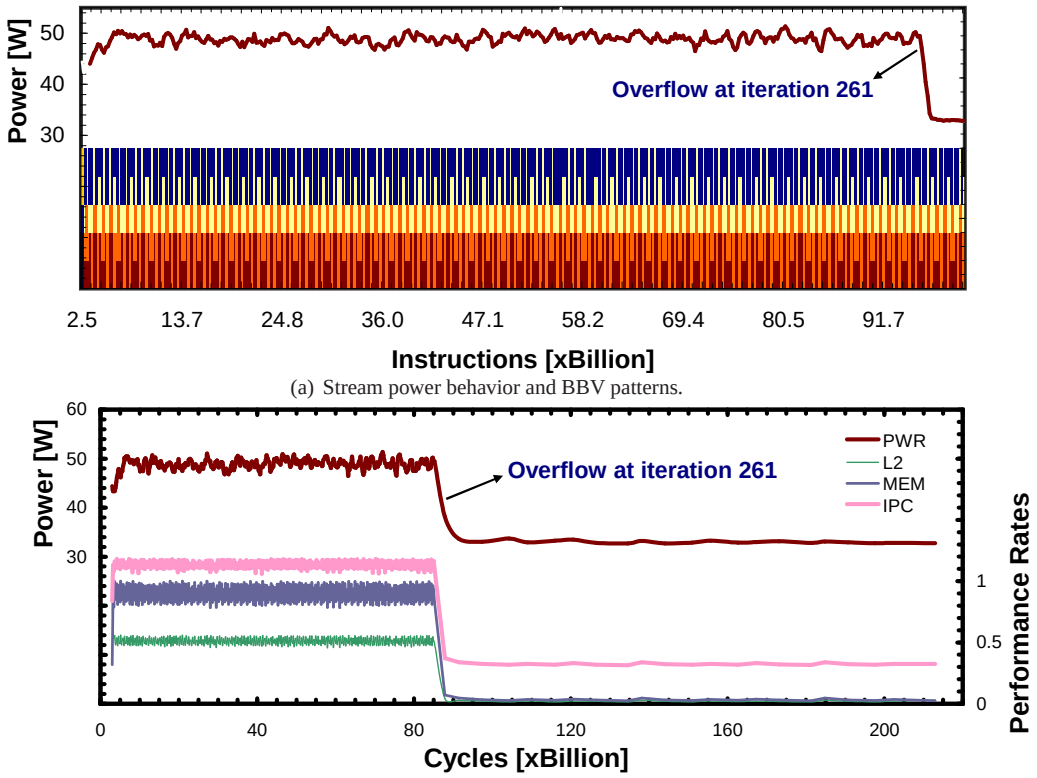
Initially we discussed some of the possible reasons that can cause control flow information and performance statistics to arrive at different conclusions about application power behavior. We showed how control flow information and power/performance characteristics of an application differ under varying data locality with the `dcache` microbenchmark.

Here, we show our observations from actual applications that we experimented on. We demonstrate the effects of other sources of disagreement, *operand dependent behavior* and *effectively same execution*.

7.1 Operand Dependent Behavior

We show an interesting example to operand dependent behavior with the `stream` benchmark. `Stream` performs four repetitive operations with simple vector kernels. It operates on three vectors, `a`, `b` and `c`. The four operations are *copy* ($c[j] = a[j]$), *scale* ($b[j] = scalar * c[j]$), *add* ($c[j] = a[j] + b[j]$) and *triad* ($a[j] = b[j] + scalar * c[j]$). It targets at measuring sustainable memory bandwidth with vectors larger than cache sizes and by avoiding data reuse. Here, we use this application to show an interesting operand dependent behavior and its implications on power. There exists a positive feedback between each iteration of the four described operations. This causes the the FP operations to overflow at iteration 261, where first vector `a` overflows at *triad*. This is then propagated to vectors `b` and `c` in the next iteration. This overflow causes the three FP kernels to experience a slowdown larger than 10x, while the *copy* operation is not significantly effected. Consequently, power dissipation experiences a drastic phase change, while execution path is still conserved.

In Figure 9, we show the resulting behavior in terms of power, BBV signatures and PMC signatures. Figure 9.(a) shows, the power (top) and BBV signatures (bottom) with respect to executed instructions. We show BBV signatures as stacked vector sample bars, where magnitude of each vector component adds on top of the stack. Here, we see the repetitive BBV vector patterns throughout the execution, corresponding to the 4 different operations repeated 275 times. As the control flow is repetitive, the sudden power drop goes undetected with BBVs. In Figure 9.(b), we show the same execution with few of the PMC vector samples. Here, we show the execution with respect to cycles, to emphasize the actual effect of overflow



(a) Stream power behavior and BBV patterns.

(b) Stream power and PMC patterns

Figure 9. Power phase change at overflow condition for `stream` benchmark. (a) shows BBV signatures, unable to detect the phase change, (b) shows PMCs detecting the change. (b) is drawn with respect to elapsed cycles to show the actual time behavior.

on elapsed time in different power phases. While, the lower power phase occupies less than 6% of executed instructions, the time spent in this phase is more than half of total execution. Tracking PMCs easily identifies this power phase change resulting from operand dependent behavior of `stream`.

7.2 Impact of Effectively Same Execution

Phase characterizations of applications have two related outcomes. First, phase characterizations provide feedback for identifying phase changes in program behavior. Second, they classify applications into similar regions of execution. These two aspects have an inverse relation, which can be considered in terms of *similarity* and *granularity* [15]. Dictating more restrictive similarity features within each phase results in higher number of phases with smaller granularity. These may, then, lead to numerous false alarms for spurious phase transitions, as many of the small variations in tracked features do not reflect in application (power) behavior. Thus, a desired property for phase characterization is to lead to high granularity phases that capture major application behavior; balancing similarity and granularity.

Effectively same execution represents a characteristic behavior when PMC and BBV approaches perform differently in achieving this balance. In many occurrences, applications walk through different code paths, while performing similar computational tasks. These lead to different code signatures, indicating different phases, while actual power phase behavior is similar.

We demonstrate the impact of this effect with the `mesh` benchmark. During its execution, `mesh` first reads an input mesh configuration and performs various tasks on the input mesh. Most of these tasks have computationally similar properties, leading to effectively same execution behavior—while in different execution address spaces. In Figure 10, we show part of

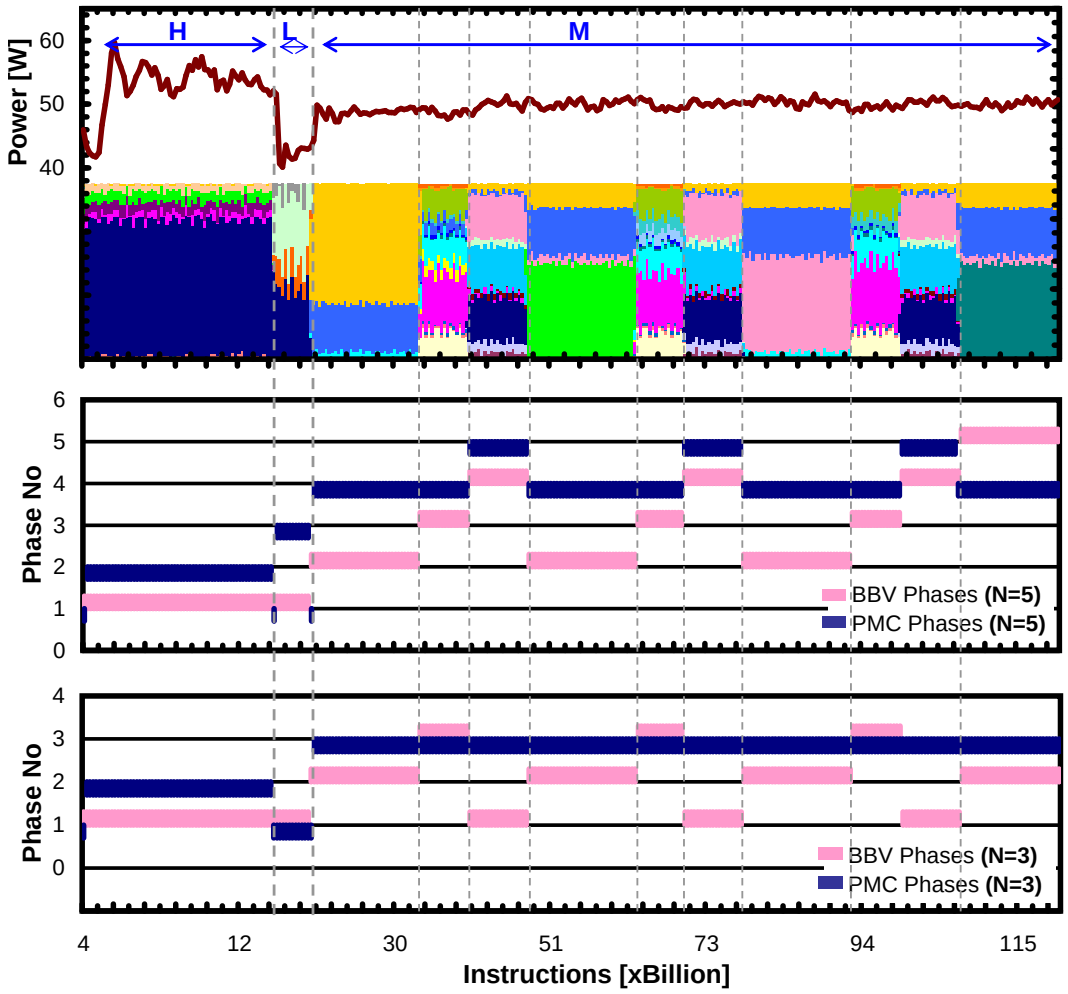


Figure 10. `Mesh` **power and BBV signatures (top) and generated PMC and BBV phases with target cluster numbers of 5 (middle) and 3 (bottom). Multiple control flow phases with effectively same power characteristics disguise actual power phases in BBV based classification. Actual power phases are labeled as** H**,** L **and** M**, for high, low and medium power dissipation regions.**

the execution characteristics for `mesh`. In the figure, we first show the measured power behavior. We can easily separate `mesh` execution into three power phases by observing the power trace. We label these "actual" power phases as H, L and M on the power trace. Representing phases with high, low and medium power consumption. Underneath the power trace, we show the corresponding BBV vector patterns for each sample. Again, we present the 32 dimensional BBVs as stacked bars, where each vector component adds up to the stack based on its magnitude. Several distinct control flow phases are observable from the BBV patterns. We separate each of these regions with vertical dotted lines. These correlate well with `mesh` tasks. First high power phase corresponds to the sorting task after reading nodes and initialization. This task sorts nodes based on their types. It operates mainly in L1 cache and performs several arithmetics. The following low power phase, results from *SetBoundaryData* task which sets the values for boundary nodes. This task mostly accesses L2, and has low overlapping computation, which leads to less power. After this task, `mesh` repetitively operates on three computation tasks, namely, *ComputeForces()*, *ComputeVelocityChange()* and *SmoothenVelocity()*. These constitute the medium power phase of `mesh`. All these tasks also make significant L2 accesses. However, their overlapping higher FP computations lead to relatively higher power.

In the lower two plots of Figure 10, we show the phase classifications performed by BBVs and PMCs. We apply agglomerative clustering with complete linkage and use target phase numbers, N, of 5—as our general choice—and 3 for a more restrictive case. In these plots, y axis shows different phases ranging from 1 to 5 for the first case and 1 to 3 for the second. For each sample, we add a tick mark above the horizontal line corresponding to its phase assigned by BBV classification. We also add a tick mark below the horizontal line that corresponds to each sample's PMC phase. These marks then form the bands of phases seen in these plots. For example, for the case with 5 phases, low power phase of `mesh` is classified into phase "1" by BBVs and phase "3" by PMCs.

These plots show the significant impact of effectively same execution in phase classification. For $N = 5$, PMCs correctly identify the three actual power phases. BBVs on the other hand, collapse the high and low power phases into a single phase, leading to a false characterization. This is because, BBVs identify several different large-scale control flow phases. Clustering starts to overlap these based on their L1 distances, and these result in combining the high and low phases of power. The three repetitive control flow phases with effectively same power behavior are seen as the more different phases by BBVs, and are assigned to different clusters. These indicate several false alarms to spurious phase changes. For $N = 3$, BBV phases still show more sensitivity to the three repetitive tasks of medium power phase and assign them to three different phases. In this case, all high, low and parts of medium power phases are assigned to same phase ("1") by BBVs. In comparison, PMCs show very good fidelity. They successfully identify three power regions and assign them to different phases.

This example demonstrates the clear impact of effectively same execution on control flow based power phase characterization. It is important to note that, this effect has implications for not only phase characterizations, but also runtime phase detection. Various control flow phases, with similar power behavior can cause a detection framework to produce several false alarms for phase transitions. These in turn lead to worsened receiver operating characteristics and pollution of actual phase behavior.

In general, there exist other cases where differences between PMC and BBV approaches arise including some SPEC benchmarks such as `mcf`. We do not present these here for brevity. Nonetheless, overall both BBV and PMC phases provide a good account of application power phase behavior; in many cases showing good correlation between power and both control flow and performance measures. PMCs usually show a better binding to power behavior due to both their proximity to the actual flow of power in the processor, as well as these discussed sources of disagreement between power and code signatures.

8 Summary and Recommendations

Here, we first make a final comparison of BBVs and PMCs for power phase characterization. We discuss different pros and cons of the two approaches. Afterwards, we recommend a combination of methods and discuss their applicability to different dynamic management techniques.

BBVs are widely studied and are shown to have several benefits for summarizing application performance or tracking application phases. Most important advantage of BBVs is the repeatability of the observed phase behavior. Tracked code signatures do not change due to system effects or with the application of dynamic management actions that affect system power and performance.

The biggest disadvantage of BBVs lies in runtime applicability. It is impractical to collect full blown BBV information during application runtime. Sampling methods, as applied in this study, provide acceptable resolution, but BBV generation still requires mapping PC samples into control flow blocks. These require additional profiling or instrumentation of applications. Another related issue is the high dimensionality of BBVs that requires processing for dimension reduction. In addition to these, false alarms due to changes only in control flow are an important consideration for a runtime detection system. Fi-

nally, the indifference of BBVs to varying data locality can be a significant impediment also in power phase characterizations for certain real applications [2].

The important advantages of PMCs are, their straightforward runtime applicability and their proximity to processor power consumption. PMCs are easily accessible at runtime with lightweight interfaces, which makes them good candidates for dynamic applications on real-systems. Several PMCs show good correlations with processor power behavior, therefore they don't suffer significantly from false alarms. Also as simultaneously monitored PMCs are on the order of 10, they require no data processing for dimension reduction during phase characterizations.

The most important consideration with PMCs is repeatability. As PMC data comes from several event counts over the processor, the values are not identical among repetitions of phases. A phase detection method that utilizes PMCs requires to consider event count ranges or has to track deltas together with events to detect phases or phase changes. Our previous studies show, quantization can be unreliable, while tracking deltas produce higher fidelity. PMC based approach also requires range considerations. Different dimensions of PMC vectors are not of similar strength. For example, memory access counts and instructions issued have different orders of magnitude. Therefore, scaling of vector components or normalizations may be necessary to emphasize the impact of certain events.

Our quantitative results showed that PMCs have relatively higher fidelity in characterizing power phase behavior. However, we believe a better solution can be achieved by combining the strengths of BBVs with PMC approach. For a general power phase characterization study, we suggest a hierarchical approach between BBVs and PMCs. We consider using PMC based phase tracking as the global mechanism to identify phase changes and using BBVs to track the repetitive execution progress. In terms of decision hierarchy, PMCs can provide confidence to phase changes detected from control flow and provide the final decision whether this is an actual or spurious phase change. On the other hand, BBVs can enhance the repeatability of observed PMC phases, by informing PMC method when a repetitive control flow is detected.

We consider application of such control flow feedback to PMC based phase detection in our current research for runtime dynamic management on real-systems. We envision several applications to such power phase characterizations that can be used in both architectural studies and real-systems. Temperature aware scheduling[3] can benefit from detecting repetitive power phases to select among tasks with different power/temperature behavior to reduce performance degradation due to idling or throttling. Multicore power balancing and activity migration [14, 29] rely on application behavior to distribute or transfer activity among different components. Power phases can provide both history and phase change information to decision policies of these techniques. These phases can also be used for dynamic voltage scaling [6, 26] to evaluate costs and benefits at runtime based on diversity and duration of different power phases.

9 Related Work

Several previous studies investigate phase behavior of applications for adaptation and characterization purposes. Most of these research studies focus on either control flow or performance characteristics of applications. Iyer and Marculescu [20], Dhodapkar and Smith [9], Sherwood et al. [32, 33], Huang et al. [17] and Lau et al. [23] analyze control flow behavior of applications via different features such as subroutines, working sets and basic block profiles. These studies use simulation based methods to identify application phases for summarizing performance and architectural studies. Patil et al. [28] also look at control flow phases with real-system experiments. They use similar dynamic instrumentation to identify BBV phases of applications. Their work uses basic block profiles of applications to find representative execution points, while we look at power characterizations with BBV and PMC phases.

Cook et al. [7] show the repetitive performance phase characteristics of different applications using simulations. Todi

[35], Weissel and Bellosa [36] and Duesterwald et al. [11] utilize performance counters to identify performance based phases. They use performance statistics to guide dynamic optimizations and metric predictions. These works do not consider power behavior of applications. Isci and Martonosi [18] employ runtime power measurements and power estimation with performance counters to identify phases of applications. Chang et al. [5] apply process power profiling to determine software power breakdowns. While these studies also look at power behavior, they do not investigate control flow approaches. Hu et al. [16] describe a compile time methodology to find basic block phases at runtime for power studies. This study looks at control flow information from a compiler perspective, while we investigate runtime power phase behaviors of both control flow and performance statistics.

There are also previous studies that compare or evaluate phase characterization techniques. Dhodapkar and Smith [8], perform a comparison between different control flow techniques, working set signatures and BBVs. Annavaram et al. [2] sample executed program counters as a proxy to control flow and show the correlations between code signatures and application performance. They show that, control flow does not always correlate well with application performance. Lau et al. [22] also look at control flow and performance of applications to show a strong correlation can be established by linking program counter to procedures and loops of applications via profiling. In comparison, our work looks at the direct comparison of two phase characterization features, BBVs and PMCs with runtime measurement feedback for real power evaluation on a real-system.

10 Conclusion

Phase analysis is increasingly important for computer systems first because simulation-based techniques rely on phase-directed sampling to reduce simulation time, and second, because real-life adaptive hardware and software mechanisms rely on dynamic phase-directed readjustments.

With power being such a pressing constraint in current processors, it becomes important to understand not just the phases of performance metrics, but also of their related-but-distinct power counterparts. Observing power phase behavior on real systems is particularly important because the real-system phases show the impact of a comprehensive range of systems effects typically excluded from simulations.

This work has explored methods for real-system power phase generation. Drawing on prior work, we have developed an experimental framework for comparing both control-flow-based and performance-monitoring-based phase techniques, and for comparing against live power measurements. Our results show that both control-flow and performance statistics provide useful hints to power phase behavior. In general, performance-based phase tracking leads to approximately 33% less power characterization errors than code signatures.

In some cases where power behavior depends on aspects other than control flow (e.g. data locality, operand values, or other characteristics), phases based on control flow can “miss” some transitions. In other cases, control flow phase classification can result in “extra” phases, where applications perform different tasks with effectively the same execution characteristics. These effects lead to both false alarms for power phase changes and incorrect power phase classifications.

Overall, the results presented here show a roadmap to effective power phase analysis in real systems. Control-flow techniques offer a good base, but may well be best applied as hybrid techniques together with performance counters that can more closely track the details of program behavior, needed for detection of power phases with high fidelity.

References

[1] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis, M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, and S. Schuster. Dynamically tuning processor resources with adaptive processing. *IEEE Computer*, 36(12):43–51, 2003.

[2] M. Annavaram, R. Rakvic, M. Polito, J.-Y. Bouguet, R. Hankins, and B. Davies. The fuzzy correlation between code and performance predictability. In *Proceedings of the 37th annual International Symp. on Microarchitecture*, 2004.

[3] F. Bellosa, A. Weissel, M. Waitz, and S. Kellner. Event-driven energy accounting for dynamic thermal management. In *Proceedings of the Workshop on Compilers and Operating Systems for Low Power (COLP'03), New Orleans*, Sept. 2003.

[4] D. Brooks and M. Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan. 1999.

[5] F. Chang, K. Farkas, and P. Ranganathan. Energy driven statistical profiling: Detecting software hotspots. In *Proceedings of the Proceedings of the Workshop on Computer Systems*, 2002.

[6] K. Choi, R. Soma, and M. Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED)*, Aug. 2004.

[7] J. Cook, R. L. Oliver, and E. E. Johnson. Examining performance differences in workload execution phases. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[8] A. Dhodapkar and J. Smith. Comparing program phase detection techniques. In 36th International Symp. on Microarchitecture, 2003.

[9] A. Dhodapkar and J. Smith. Managing multi-configurable hardware via dynamic working set analysis. In 29th Annual International Symposium on Computer Architecture, 2002.

[10] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.

[11] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *IEEE PACT*, pages 220–231, 2003.

[12] F. Ercolessi. MDBNCH - A molecular dynamics benchmark. International School for Advanced Studies in Trieste. www.fisica.uniud.it/ ercolessi/mdbnch.html.

[13] GIMP. GNU Image Manipulation Program. http://www.gimp.org/.

[14] S. Heo, K. Barr, and K. Asanovic. Reducing power density through activity migration. In *Proceedings of International Symposium on Low Power Electronics and Design (ISLPED), Seoul, Korea*, Aug. 2003.

[15] M. J. Hind, V. T. Rajan, and P. F. Sweeney. Phase Shift Detection: A Problem Classification. IBM Researh Report RC-22887, IBM T. J. Watson, Aug. 2003.

[16] C. Hu, D. Jimenez, and U. Kremer. Toward an evaluation infrastructure for power and energy optimizations. In *Workshop on High-Performance, Power-Aware Computing*, 2005.

[17] M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the International Symp. on Computer Architecture*, 2003.

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

[19] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. In *Proceedings of the 36th International Symp. on Microarchitecture*, Dec. 2003.

[20] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of Design Automation and Test in Europe, DATE*, Mar. 2001.

[21] R. Jenkins. Hash functions. *Dr. Dobb's Journal*, 9709, Sept. 1997.

[22] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Mar. 2005.

[23] J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *11th International Symposium on High Performance Computer Architecture*, 2005.

[24] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual International Symposium on Microarchitecture*, 1997.

[25] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation (PLDI)*, June 2005.

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

[27] J. McCalphin. STREAM: Sustainable Memory Bandwidth in Current High Performance Computers. Technical report, University of Virginia.

[28] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium programs with dynamic instrumentation. In *Proceedings of the 37th annual International Symp. on Microarchitecture*, 2004.

[29] M. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: Leveraging smt and cmp to manage power density through the operating system. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI*, 2004.

[30] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Eleventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, Oct. 2004.

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

[32] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior, 2002. In Tenth International Conference on Architectural Support for Programming Languages and Operating Systems, October 2002. http://www.cs.ucsd.edu/users/calder/simpoint/.

[33] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA-30)*, June 2003.

[34] Sourceforge.net. The LAME Project. http://www.mp3dev.org/.

[35] R. Todi. Speclite: using representative samples to reduce spec cpu2000 workload. In *Proceedings of the IEEE International Workshop on Workload Characterization (WWC-4)*, 2001.

[36] A. Weissel and F. Bellosa. Process cruise control: Event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002), Grenoble, France,*, Aug. 2002.