Identifying Program Power Phase Behavior Using Power Vectors

Canturk Isci and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{canturk,mrm}@ee.princeton.edu

Abstract

Characterizing program behavior is important for both hardware and software research. Most modern applications exhibit distinctly different behavior throughout their runtimes, which constitute several phases *of execution that share a greater amount of resemblance within themselves compared to other regions of execution. These execution phases can occur at very large scales, necessitating prohibitively long simulation times for characterization. Due to the implementation of extensive clock gating and additional power and thermal management techniques in modern processors, these program phases are also reflected in program power behavior, which can be used as an alternative means of program behavior characterization for power-oriented research.*

In this paper we present our methodology for identifying phases in program power behavior and determining execution points that correspond to these phases, as well as defining a small set of power signatures representative of overall program power behavior. We define a power similarity metric as an intersection of both magnitude based and ratio-wise similarities in the power dissipation of processor components. We then develop a thresholding algorithm in order to partition the power behavior into similarity groups. We illustrate our methodology with the gzip benchmark for its whole runtime and characterize gzip power behavior with both the selected execution points and defined signature vectors.

1 Introduction

Characterizing program phase behavior for current and emerging applications provides significant foundation to computer engineering research in several aspects and abstractions. Workload characterization can be used to develop more power-efficient, complexity-effective, high performance architectures, to provide feedback for multiconfigurable architectures for power/performance optimizations, to enable OS based dynamic management and to overcome prohibitively long simulations by identifying representative execution points or reduced datasets.

Most programs show very variable behavior over their whole runtimes and the analysis of these behavior via simulation is usually impractical due to extensive simulation times. However, most of these programs also exhibit some amount of repetitive behavior within different execution regions or at certain periods. Phase analysis is a formal method of identifying this repetitive behavior, which can be used to reduce the amount of redundant work in architectural research while preserving the workload characteristics, and to enable dynamic optimization techniques that benefit from this phase behavior.

In [5] we demonstrate with several examples that different programs with similar average powers can show significantly different power variation (i.e. gcc, vpr, gzip). Likewise, a single program with very stable total power can have distinctively different power behavior—in terms of different component power ratios—in different execution phases (i.e. vpr). In this paper, we demonstrate a phase analysis method which relates directly to power. In our work, we introduce the *power vector* concept, which represents the estimated power values for 22 processor components—such as trace cache, integer execution unit—at each sampled execution point, in a similar fashion as the basic block vector distribution analysis of [11]. We observe the variations in these power vectors in order to identify similar regions within a program, which define the several phases of execution a program goes through during its execution.

The most important aspect of our work is that it uses power signatures of programs and therefore is a way to analyze power phase behavior rather than relying on performance metrics or basic block information. The power vectors used in our analyses are acquired at runtime; therefore the similarity relations can be generated very quickly, without the need to perform lengthy simulations to collect the similarity data. To characterize the power behavior of a program for a different dataset or optimization level, we need to only rerun the program with the new configuration, rather than resimulating the whole execution.

The power phase analysis described in this paper contributes to current research in various ways. Representative power vectors, generated as one result of our similarity analysis, can be used as "program power signatures" in power oriented studies. As our analysis is based on a real system, it can directly be utilized in power aware research such as

[16] for runtime phase identification based on the signature vectors. With the ability to identify recurring phases over large scales of execution, our technique can be used for OS based dynamic management for thread scheduling, voltage or frequency scaling [4, 15]. Moreover, identified execution points for programs, as another outcome of the similarity analysis, can be used to define power simulation points similar to SimPoints of [11].

The remainder of this paper is structured as follows. Section 2 discusses the related work, Section 3 gives an overview of our power phase analysis methodology, Section 4 discusses the phase characteristics of program power behavior and Section 5 describes our power behavior similarity metric based on power vectors. Section 6 describes our phase identification technique and demonstrates our results for representative vectors and selected execution points, with a final error analysis. Section 7 provides discussion of presented work and future research related to workload characterization and Section 8 summarizes our conclusions.

2 Related Work

A number of previous works investigated various issues related to program phase behavior including simulation-based [2, 3, 7, 10, 11, 12] and runtime [9, 14] program profiling techniques to identify phase behavior. These works span diverse areas such as identifying representative simulation point samples, predicting phases, generating reduced datasets and managing configurable hardware with program signatures. Dhodapkar and Smith define working set signatures as a lossy compression of true working sets and use this working set information to detect phase changes and working set size, which are in turn used to find an optimal configuration for multiconfigurable hardware [3]. Sherwood et al. introduce *basic block vectors*, which represent the proportion of basic block executions within one sampling period. They use basic block vector differences with respect to a global vector to identify program phases [10]. In order to determine the amount of resemblance between different windows of execution, collected over the program run, they define the *basic block similarity matrix*, which consists of the manhattan distances between all pairs of basic block vectors. They use this similarity relation to cluster the sample points into a small set of groups, where each group is represented by a single execution point, chosen as the closest to group centroid [11]. Later, they extend the basic block vector idea to execution time phase tracking and phase predicting microarchitecture, by approximating basic block access information with instruction counts separated with branches [12]. In [14], Todi uses Intel Itanium processor performance counters to collect benchmark execution information. The author applies principal component analysis to reduce variable dimensions and k-means partitioning algorithm to generate similarity groups, and finally selects representative execution points similar to [11].

In comparison to the previous work, we use a similar similarity analysis technique as [11]. However, our analysis is based on estimated power vectors and we use a combination of normalized and non-normalized measures for phase identification. Similar to [14], the basis of our similarity analysis data relies on performance counters, but our approach provides power attributes to collected counter information. Thus as well as identifying phases, we can identify and focus on regions with high cache, execution power, etc. Moreover, the application of performance counter data collection bears significant differences. Rather than rerunning the experiment at different times, we use counter rotations within a single experiment as the flexibility of P4 counter reading mechanism [13] enables us to read 18 counters simultaneously. Overall, we present a power oriented phase analysis methodology and since our technique is based on runtime power estimation rather than simulation, power similarities are generated at almost real workload runtime speed. Therefore, the presented methodology is easily repeatable and extensible to new emerging workloads.

3 Methodology

In our power phase analysis, we make use of the runtime component power estimation framework that we introduced in [5]. With the experimental setup shown in Figure 1, we collect measured total processor power data and raw performance counter information, and use this counter information to estimate processor component powers and total processor power at runtime.

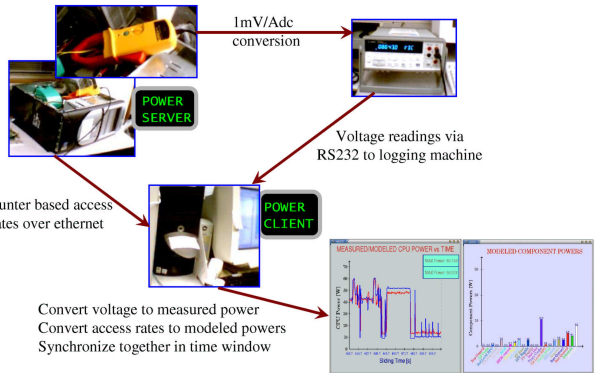


Figure 1. Power measurement and component power estimation setup.

In our setup, the current probe on P4 power lines measures the DC current through processor power lines and the digital multimeter at the other end sends this data to a logger machine over RS232. The tested machine runs a kernel module that collects counter information, and a server that sends the raw counter data to the logger machine over ethernet. The logger machine then processes the counter data to produce power estimates for 22 processor components at runtime, which, together with a constant idle power, add up to a total power estimate that can be verified against real measurement. In our work, these generated component

power estimates are considered as 22-dimensional *power vectors* at each sampling point that serve as our multidimensional data for similarity analysis.

During power phase analysis, we first collect power vectors, measured data and timing information for a given benchmark at runtime. We also generate a second set of data by normalizing the power vectors. Then, we use the original power vectors and normalized vectors together to generate a similarity matrix to quantify the amount of similarity between all pairwise combinations of execution points. Based on a thresholding algorithm described in Section 6, we generate *grouping matrices* for a given similarity threshold. These demonstrate, for each execution point, which other execution points lie within its similarity threshold. Analyzing the produced grouping matrix, we partition the execution points into a small set of groups, which contain vectors that satisfy a specified amount of similarity. Afterwards, we identify representative vectors for each group, which represent the average group power behavior. We use these representative vectors to reconstruct the power trace and verify how close our approximation is to original power behavior.

4 Power Behavior of Programs Constitute Phases

Most programs exhibit several distinct phases of execution even at the largest of scales like the whole execution timeframe. For example, benchmarks such as twolf can exhibit distinctively identifiable phases with respect to different datasets even though the—*measured*—total power reveals indistinguishable power behavior. On the other hand, with equake one sees that a benchmark can exhibit very different phases within a single dataset like initialization, computation and reporting [5]. In Figure 2 we show two benchmarks, SPEC2000 gap and gzip, where gap shows distinct phases for a single dataset and gzip shows periodic phases within a dataset as well as recurring phases across its 5 datasets. We also include plots for power breakdown traces filtered with a 10 point moving average so that we could filter down higher frequency phase components and look at distinct phases at the larger whole execution scale. Again, very distinct phases are identified at very large timescales, which are prohibitive to capture via simulations.

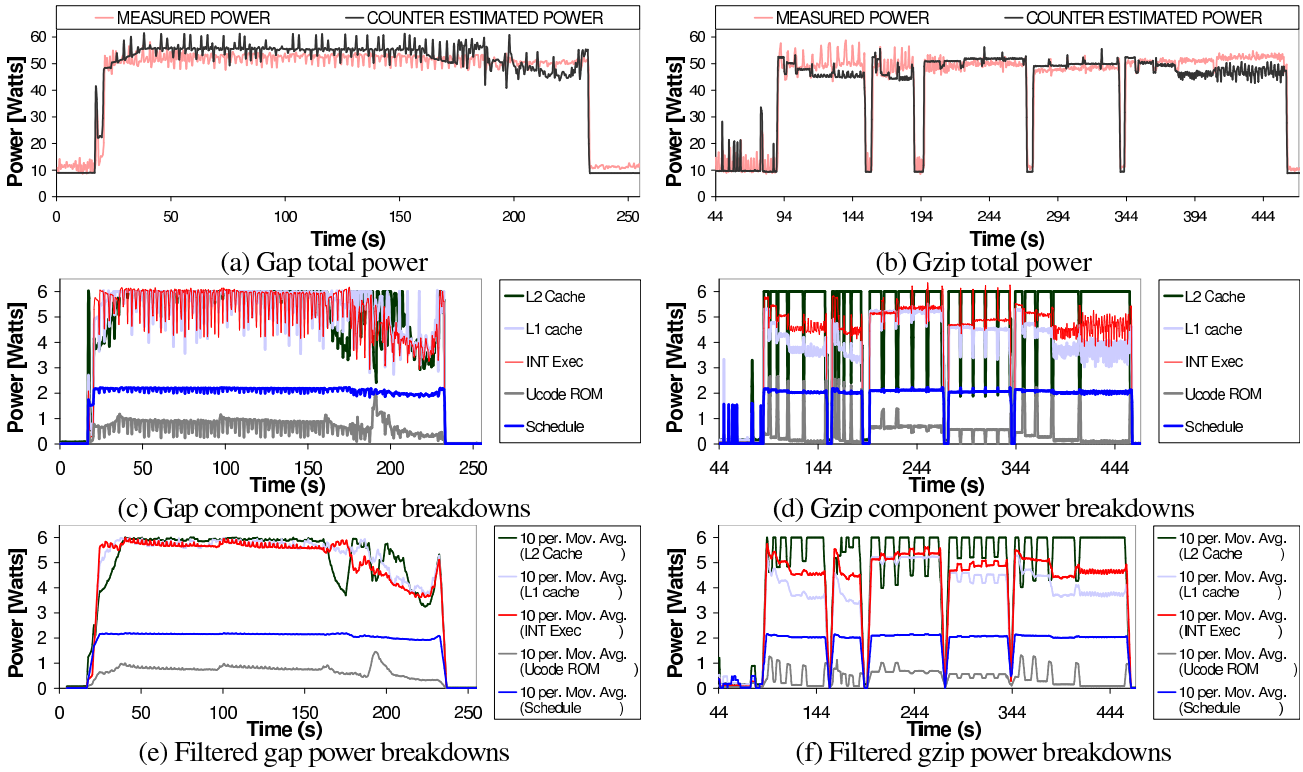
Power traces shown in Figure 2 reveal two important observations, which lay the grounds for our research. First, power behavior of programs are shown to exhibit phase behavior, similar to performance metrics such as IPC and miss rates. Additionally these phases may not be visible by sole total power observations, but can be hidden in the variations of power vectors. Secondly, the employed runtime technique enables observation of large scale phase behavior in the order of 10s of seconds. As discussed in [2], for most workloads, executing the first few billions of instructions, which correspond to a few seconds of actual execution, can produce a misleading view of program power behavior. Thus, these two observations set the ground rules of our power phase analysis research: to focus on complete power behavior of programs and to identify representative regions that can accurately and efficiently reconstruct program power behavior.

5 Using Power Vectors for Similarity

In our power phase analysis, we consider the generated power vectors as points in the positive quadrant of the power space spanned by the 22 vector dimensions. As each power vector corresponds to a specific execution time sample in the program trace, we evaluate power behavior similarity of execution regions by measuring the spatial closeness of the points specified by the corresponding power vectors. We use the manhattan distance between two vectors as our measure of closeness, which is defined as the absolute difference of vector elements summed over all vector components.

We record the manhattan distances for all vector pairs in an upper diagonal *similarity matrix* in the execution order such that, matrix entry (r,c) shows the manhattan distance between the power vectors corresponding to r^{th} and c^{th} execution time samples. Only the upper diagonal needs to be constructed as distance from the r^{th} vector to the c^{th} is identical to distance from the c^{th} vector to the r^{th}. The matrix entries are nonnegative real numbers. A 0 at entry (r,c) represents a perfect similarity between execution samples r and c, while higher values represent higher dissimilarity. The execution time flow is along the matrix diagonal and for an execution point r, entries in the upper column $(r_i < r, r)$ represent its similarity with respect to previous samples, while the entries in the right row $(r, c_i > r)$ represent similarity with respect to samples in the forward path. We demonstrate the generated power similarity matrices in terms of matrix plots that are aligned with the execution timeline along the diagonal, where the top left corner represents the start of the timeline and the lower right corner represents the end of the timeline. In the matrix plots shown in Figures 3, and 4, the matrix entries are presented as greyscale pixels, where the shading is scaled from white, for maximum dissimilarity, to black, for perfect similarity, by the entry values.

Initially, we had considered a similarity analysis based on original power vectors acquired from the runtime estimations [5]. Here, we suggest a more restrictive approach in order to also distinguish cases where vector component ratios are relatively different for vectors of smaller magnitudes and therefore also use normalized metrics in conjunction with the original vectors. Moreover, it is imperative to know that power vectors provide significant insight over the behavior that can be observed simply from total power dissipation. Therefore, we also provide a total power based similarity description in comparison to power vector based similarity analysis. In the following subsections we discuss these issues in a progressive manner, ultimately arriving at our final similarity metric.



(a) Gap total power (b) Gzip total power

(c) Gap component power breakdowns (d) Gzip component power breakdowns

(e) Filtered gap power breakdowns (f) Filtered gzip power breakdowns

Figure 2. Total and component-wise power traces for gap and gzip.

5.1 Similarity Based on Total Power

In order to observe the similarity based on total power, we constructed the similarity matrix in Figure 3 by considering total power as a single dimensional power vector. Therefore, the similarity matrix directly corresponds to the variation of absolute total power difference among execution points. Each matrix entry (r, c) is computed as shown in Equation 1, where $Total\ Power_{r,c}$ represent the total power samples at execution points r and c.

$$Total\ Similarity\ Matrix(r,c) = |Total\ Power_r - Total\ Power_c| \quad (1)$$

The similarity matrix in Figure 3, hides significant amount of information that can be inferred from Figure 2 to identify different program power phases. For instance, the region from from 200s to 380s is identified as almost completely similar except for the idle periods, while the power vector components show different power behavior for all the 3 datasets covered. To be able to extract this concealed phase information, we need to consider a finer grained analysis of power behavior based on variations of component-wise power distributions—power vectors, which we discuss in the following sections.

5.2 Similarity Based on Original Power Vectors

The most straightforward way of incorporating power vectors into the similarity analysis is defining similarity with respect to the distance between these vectors. The similarity matrix based on this approach is constructed from

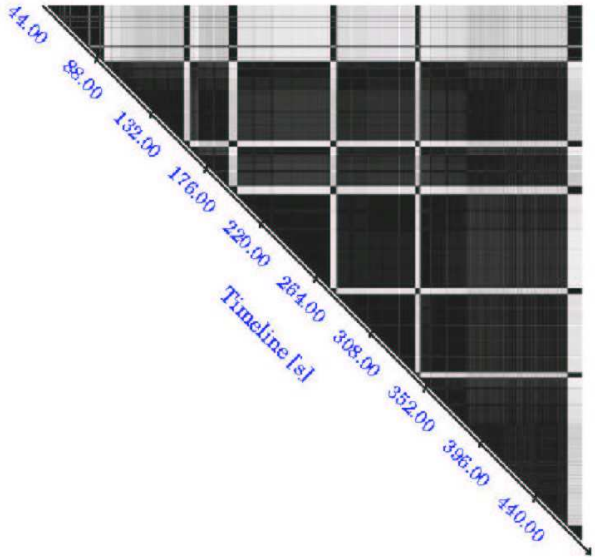


Figure 3. Total power similarity matrix.

manhattan distances of all the combination pairs of non-normalized power vectors. A single matrix entry (r, c) is computed as shown in Equation 2, where $PV_{r,c}$ represent the sample power vectors and $i \in \{1, 2, \ldots, 22\}$ correspond to vector component indices.

$$Original\ Similarity\ Matrix(r,c) = \sum_{i=1}^{22} |PV_r(i) - PV_c(i)| \quad (2)$$

The generated matrix identifies some of the phase in-

formation concealed by the total power metric, such as the obvious phase changes that occur within all 5 datasets, where memory-related power drops, while execution and issue power increase within small time bursts. There is, however, an inherent downside to this absolute approach. Namely, vectors of smaller magnitude are bound to be considered similar even though they may point very different directions in power space. This is simply because in low-power regions, the difference vector will also be of smaller magnitude compared to differences between higher power vectors. In order to overcome this pitfall, we also consider normalized metrics that help provide similarity measures independent of the range of total power.

5.3 Similarity Based on Normalized Power Vectors

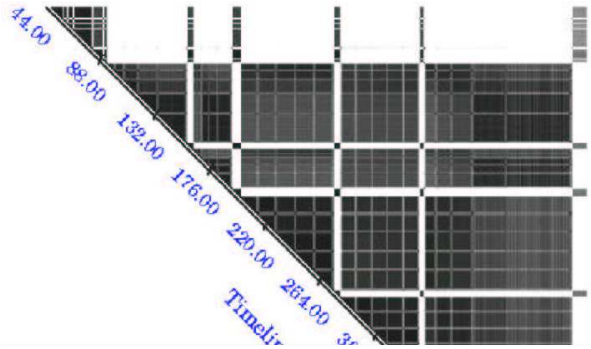
In order to single out the effects of normalization, here we consider a similarity metric based on only computed normalized power vectors. The similarity matrix is constructed from the manhattan distances of all the combination pairs of normalized power vectors. A single matrix entry (r, c) is computed as shown in Equation 3, where $NPV_{r,c}$ represent the sample normalized power vectors.

$$Normalized\ Similarity\ Matrix(r,c) = \sum_{i=1}^{22} |NPV_r(i) - NPV_c(i)| \tag{3}$$

The reason behind normalization is to emphasize the differences between the distribution of power into the vector components. In other words, the similarity metric demonstrated here is based on the relative ratios of component powers independent of vector magnitudes. Consequently, the similarity matrix reveals much better discrimination of low power vectors compared to original similarity matrix. However, one obvious shortcoming of the normalized vectors is their indifference with respect to magnitude as long as ratios prove to be similar. This unfair treatment is observed at some higher power regions, where the normalized similarity matrix exhibits significantly higher levels of similarity compared to the original similarity matrix. Finally, to avoid this pitfall, we present a combined approach as an intersection of the two similarity metrics in the next section.

5.4 Similarity Based on Both Normalized and Absolute Power Vectors

As discussed in Sections 5.2 and 5.3, both normalized and non-normalized techniques tend to disregard certain types of dissimilarities. Therefore, in order restrict ourselves to similarities that satisfy both cases, we developed an intersection of the above two matrices so that two vectors are considered similar only if they can be considered similar under both measures. We perform this by adding the two matrices after normalizing each to unity in order to weight both measures equally. We then limit the resultant matrix elements by 1 so that 1 is representative of maximum dissimilarity and 0 corresponds to perfect similarity. We perform a limiting operation, rather than normalization, after the addition of two matrices in order to achieve a final similarity metric which emphasizes dissimilarities. In other words, we want a similarity and a dissimilarity to result in dissimilarity. Consequently, the final similarity matrix is constructed from the two previous similarity matrices as shown in Equation 4, where FM, OM and NM represent final, original and normalized similarity matrices respectively.

$$FM(r,c) = \min\left(\frac{OM(r,c)}{\max_{r',c'}(OM(r',c'))} + \frac{NM(r,c)}{\max_{r',c'}(NM(r',c'))}, 1\right) \tag{4}$$

The matrix plot representing this final similarity metric is shown in Figure 4. This final plot identifies both ratio based and magnitude based dissimilarities relatively well. Moreover, the emphasis on dissimilar regions also provides much sharper distinction between the degrees of similarities. In comparison to Figure 3, the final similarity matrix plot reveals significantly higher information regarding program power phases, both at lower power and higher power execution regions.