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.

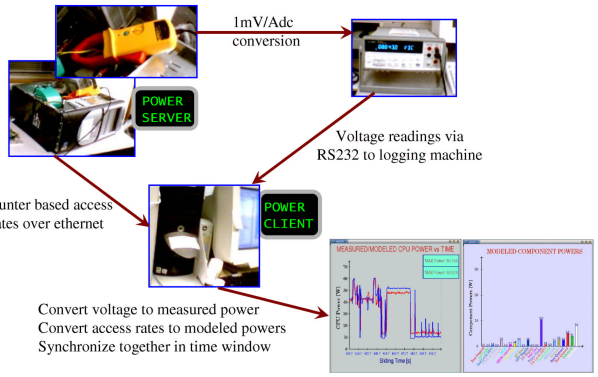


Figure 1. Power measurement and component power estimation setup.

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.

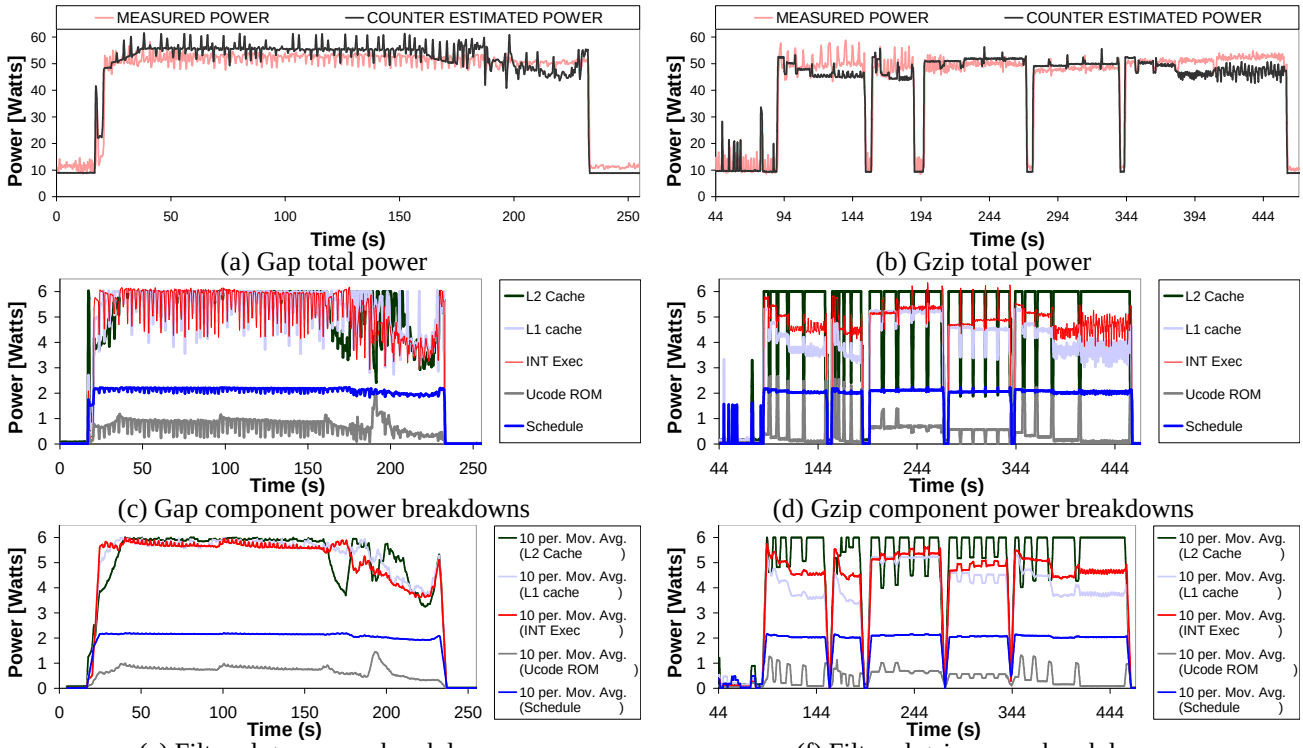
Power traces shown in Figure 2 reveal two important observations, which lay the grounds for our research. First, power behavior of programs are shown to exhibit phase behavior, similar to performance metrics such as IPC and miss rates. Additionally these phases may not be visible by sole total power 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

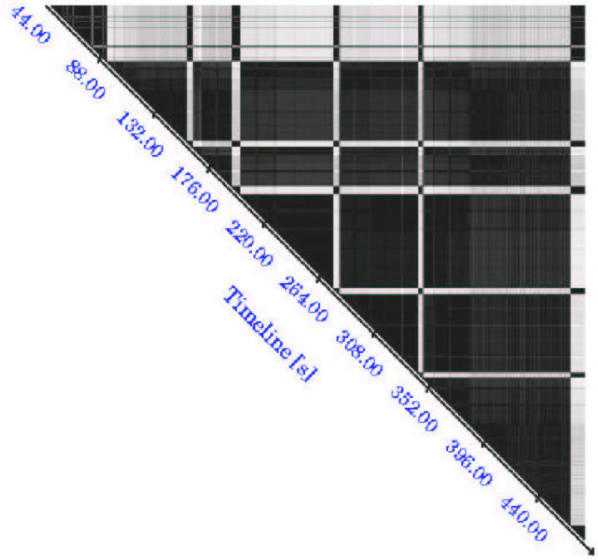


Figure 3. Total power similarity matrix.

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.

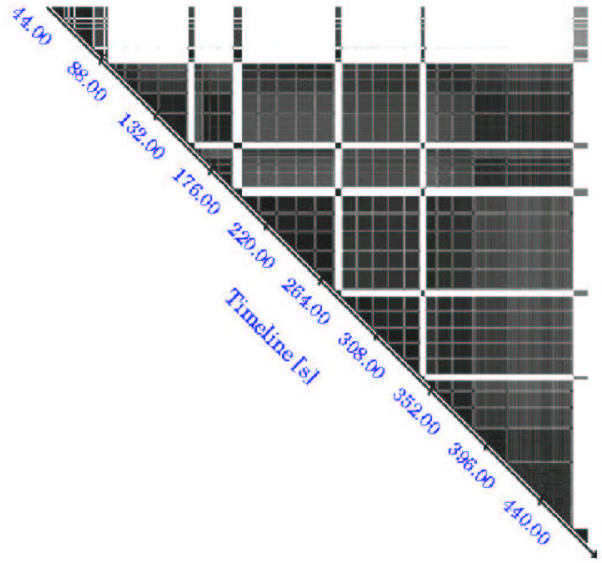


Figure 4. Similarity matrix based on both normalized and absolute similarity metrics.

With this final similarity metric, we demonstrate power vector based phase analysis provides certain amount of insight into workload power behavior, which cannot be directly extracted from total power. Moreover, it provides a more restrictive selection criterion. In the following research, we utilize this similarity metric to identify program phases and characterize program power behavior.

6 Similarity Groups Based on Thresholding

In Section 5 we have demonstrated how we can informally distinguish similar program phases from the similarity matrix. By assessing the degree of darkness of the similarity matrix entry with coordinates (r,c), we can understand

the level of similarity between sample execution points r and c in the timeline. Yet, to be able to use the similarity information, we also need more formal ways of distinguishing this phase behavior.

One primary aim in power phase analysis is to achieve a reduced workload size for a benchmark that still captures most of its power behavior. Our methodology is best described as "representative sampling technique" [14], where we identify a small set of execution points that are representative of the overall power traces of programs. Secondly, we also define a set of representative power vectors, which are not directly associated with execution points, but rather define a program "signature" based on their component powers and their order of appearance in the timeline. These signature vectors can be used in program identification and phase prediction.

For both of these problems, we need to get the sample set small enough to be manageable. Thus, in this section we introduce a *thresholding* algorithm with the following two goals:

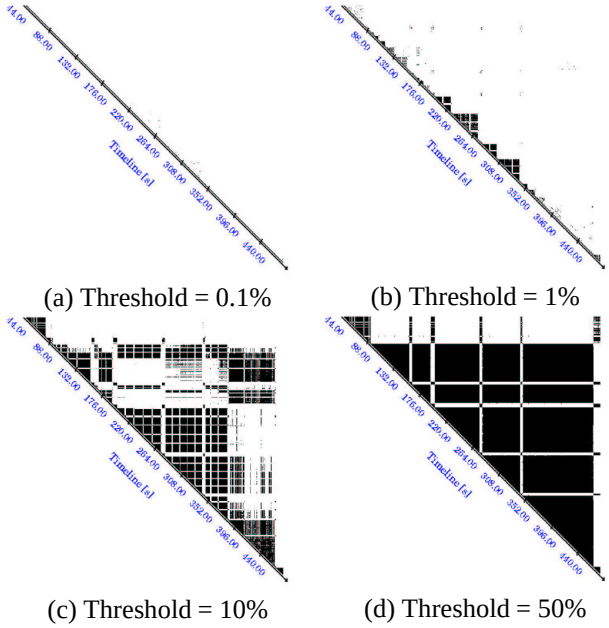
(i) Grouping execution points—power vectors—based on their similarity

(ii) Representing power behavior with reasonable accuracy with a small number of "signature vectors"

6.1 Thresholding Algorithm

Our thresholding algorithm provides a means to group execution points, guided by our aim to represent power behavior with generated signature vectors. First we specify a threshold as a percentage of maximum dissimilarity between all pairs. Then, starting from the first execution point (0,0), we move forward in time identifying the execution points that lie within the threshold criteria. The threshold criteria include both the absolute and normalized measures. For example, for a threshold of 10%, a point is considered within threshold if the manhattan distance between the start-point power vector and current power vector is less than 10% of maximum possible distance, and also if the distance between the normalized vectors at the two points lies within 10% of maximum possible distance between normalized vectors. The thresholding algorithm performs this similarity grouping for each execution point to generate a grouping matrix. Similar to the initial similarity matrix, the grouping matrix illustrates which other points are similar to each execution point, for a given threshold.

In Figure 5, we demonstrate the groupings for gzip. The maximum difference for all pairs of vectors is 47.35 (compared to a maximum power magnitude of 58.65W). For normalized vectors, the maximum difference is 1.69. Figure 5(a) shows an extreme case, where for the 0.1% threshold, almost all nodes are only similar to themselves. Figure 5(d) shows a very loose threshold of 50%. Here, the only discrimination left is between the low power and high power values due to their large magnitude difference. For the values between these extreme cases, even a 1% threshold shows some amount of captured similarity. When we increase the threshold to 10%, we already see that all execution points lie within at least one other point's 10% adjacency and a significant amount of similar groups can be identified for most of the points.



(a) Threshold = 0.1% (b) Threshold = 1%

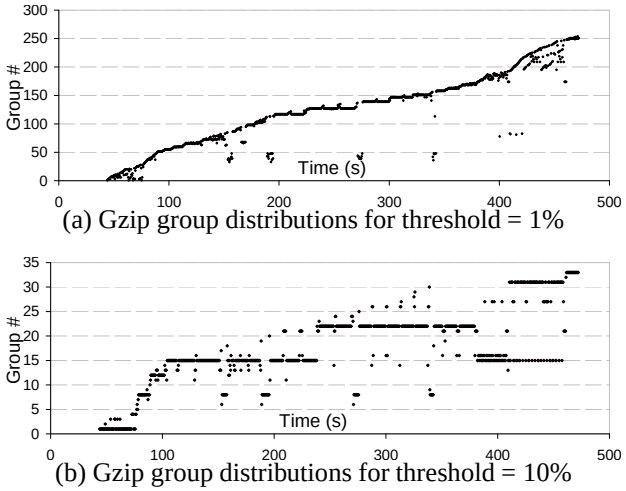
(c) Threshold = 10% (d) Threshold = 50%

Figure 5. Grouping matrices for gzip with respect to different threshold values.

For each execution point, the first thresholding step shows which other execution points are within the radius of the given threshold. However, this does not yet divide the execution points into smaller sets of groups. To identify these groups, we walk through the generated grouping matrix along the forward execution path. For each execution point r, in the matrix we identify the points $(r, c_i > r)$ in the forward execution path that lie within the threshold. Then, we tag the corresponding execution points c_i as the same group. Afterwards, we find the next untagged execution point in the timeline and perform the same tagging operation until we reach the end of execution. Thus, we prevent any tagged execution point from adding new elements to its belonging group.

In Figure 6, we show the distribution of similarity groups generated by two different thresholds. Figure 6(a) shows the distribution of 254 groups along the same timeline for a tight threshold of 1%, while Figure 6(b) shows the distribution of 33 groups for a more relaxed threshold of 10%. For the tighter threshold, the group assignments have an almost monotonically increasing trend along the timeline. This means that most of the points that start a new group can gather only near neighbors along their forward path, but are unable to include many execution points in further timescales. On the other hand, when we relax the similarity threshold, several execution points begin to collapse into same similarity groups.

In addition to the above two cases, we generated group



(a) Gzip group distributions for threshold = 1%

(b) Gzip group distributions for threshold = 10%

Figure 6. Group distributions for gzip.

distributions for several other thresholds. In Table 1 we show the number of generated groups for each of these applied thresholds. In our experiment, the total number of original execution points for gzip is 974. As the table shows, the number of groups decreases quickly to less than 7.2% of execution points within the first 5% threshold, due to the very regular and repetitive behavior of gzip.

Threshold	# Groups
0.1%	909
1%	254
3%	108
5%	70
7%	50
10%	33
20%	15
30%	9
50%	4
70%	3
100%	1

Table 1. Number of generated groups for different thresholds for gzip.

6.2 Generating Representative Vectors

In Section 6.1, the thresholding algorithm has provided our response to the first of the two raised questions: How to group the power vectors based on their similarity. For the second question—whether we could represent the power trace with a smaller set of signature vectors—we use the generated groupings as the startpoint and define a representative vector for each group. Consequently, the number of groups that depend on the set threshold is also the number of representative vectors for a given trace. For the representative vectors, we construct the vectors as the component-wise arithmetic average of all the vectors belonging to the corresponding group. In Figures 7 and 8 we show the distribution of the 974 power vectors into the 33 groups and the corresponding representative vectors for gzip for a chosen 10% similarity threshold.

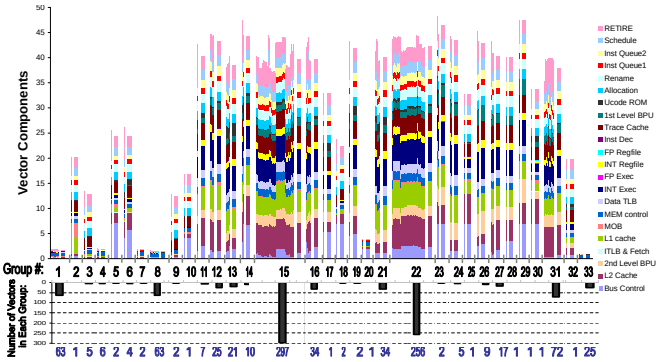


Figure 7. Gzip power vectors distributed into groups and corresponding representative vectors. The histogram below the groups shows the number of vectors per group.

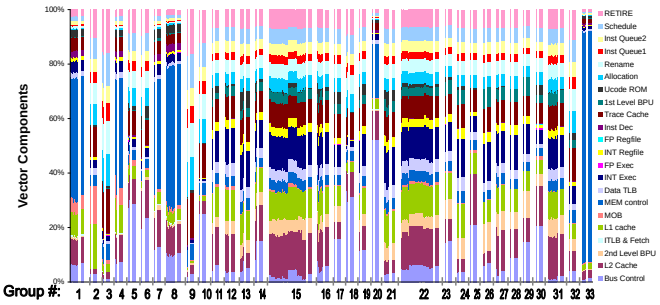


Figure 8. Normalized gzip power vectors and corresponding representative vectors distributed into groups.

In Figures 7 and 8, the first set of bars show the vectors that correspond to the execution points that are members of the shown group number in the x axis. The last bar per group shows the generated representative vector for that group as the arithmetic average of all vectors that belong to the same group. In Figure 7, we also show the number of vectors for each group with both the shown histogram, and the actual numbers below the histogram. There is a very uneven distribution of vectors into the 33 groups; groups 15 and 22 actually represent more than 50% of the whole trace. Both normalized and non-normalized plots reveal that there are a few regions within the 15th and 22nd groups, which present significantly higher similarity among themselves with respect to the rest of the members of the group. These regions are discriminated when we choose a tighter threshold, but they are grouped together for the applied 10% threshold. Depending on desired accuracy, a more greedy grouping mechanism can apply a second level thresholding with a tighter bound to identify these regions. Moreover, the non-normalized plots in Figure 7 show that some dimensions such as the trace cache and retirement logic move together, thus signifying a dependent power behavior, while

some other dimensions like the L1 cache, L2 cache and bus logic can show converse behavior. Thus, our power vector based phase analysis also lets us discriminate phases into groups such as high L1 cache and low L2 power (i.e. group 21) or high L2 power with low bus power (i.e. group 31).

6.3 Selecting Execution Points

Our primary aim in phase analysis is to come up with a manageable set of execution points that captures most of the program power behavior. The execution points should refer to actual execution times so that those specified points can identify power simulation points similar to SimPoints [1]. In our selection of the execution points, we choose the earliest occurring member of each group—the startpoint—as the selected execution point for that group. Thus—as the distance between the startpoint of a group and all other members of the group is always bounded by the given threshold, we can always formally specify an upper bound on the amount of difference between the originally estimated power and our power approximation based on the selected set of power vectors. Also, as [1] discusses, choosing representative simulation points earlier in the execution timeline reduces the time required to fast forward to the selected simulation points. Due to our selection scheme, the selected execution points are always the "early" simulation points in our experiments.

The power vectors for the selected execution points can be readily inferred from Figures 7 and 8, as the first vector in each group's set of vectors. Further discussion of selected execution points and acquired results is included in Sections 6.4 and 6.5, where we demonstrate the achieved power trace approximation and the range of approximation error.

6.4 Reconstructing Power Traces

After having specified the representative vectors in Section 6.2, for each execution point, we assign the representative vector for the corresponding group as that point's power vector and thus, reconstruct the whole power trace with only the representative vectors. Similarly, referring to the selected execution points in Section 6.3, we identify the corresponding power vectors and construct the power trace based on the selected execution point vectors.

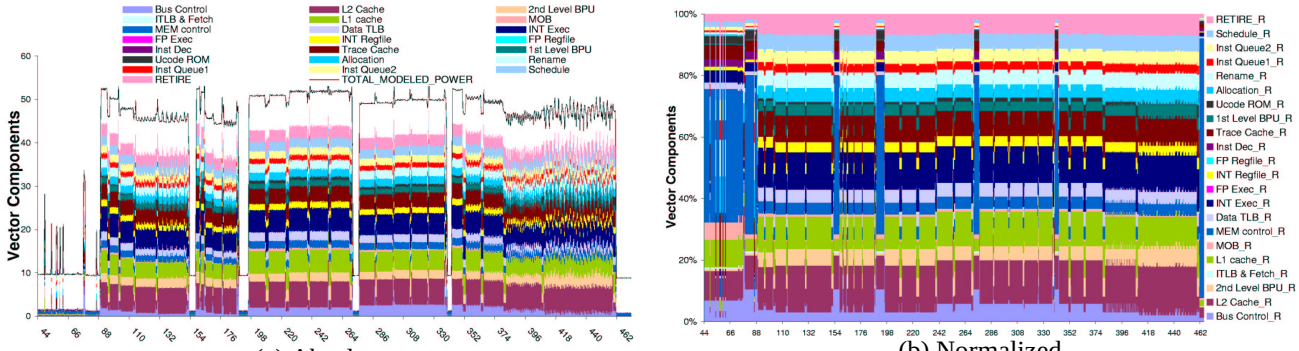
The reconstructed power traces for a 1% threshold show an almost perfect matching to the original power behavior for both representative vectors and selected execution points, using approximately $1/4$ (254) of the original power vectors. On the other hand, traces for a 10% threshold show distinguishable mismatches with respect to the original gzip total power, as can be observed in Figure 13. Nonetheless, both traces characterize the whole power behavior with only 33 vectors, which are approximately 3.9% of the total vector samples. In general, the differences in traces based on selected execution points are observed to be higher over the whole runtime compared to traces based on representative vectors, which we further discuss in Section 6.5.

It is worth noting that the above comparisons only consider the total power behavior, while the ultimate goal of similarity analysis is to be able to characterize power accurately across all dimensions. In order to show how the component powers are characterized, we show the power vector samples along the execution timeline in Figures 9, 10 and 11. In Figure 9 we show the original counter estimated power vectors both with magnitudes and as normalized. In Figures 10 and 11 we show the resultant vector traces for representative vectors, and the selected execution points, both absolute and normalized for a 10% threshold. In the magnitude plots, we also show the total power traces, which are the sum of all shown 22 vector components and a constant idle power of 8W that is not included in the vector plots. Both reconstructed traces demonstrate, they capture most of the large scale behavior, while they seem to filter out some power variations in smaller scales.

6.5 Error Analysis

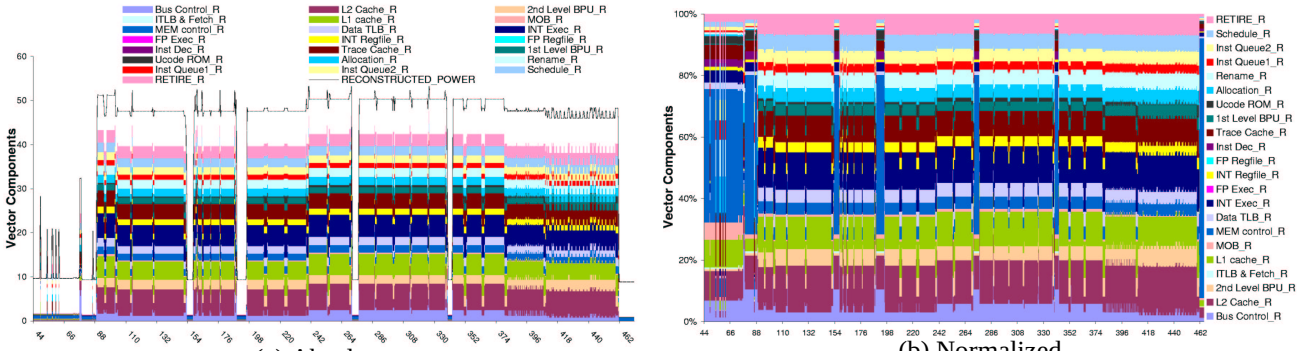
In previous sections, we have shown how the power behavior characterizations based on either representative vectors or selected execution points relate to the original power behavior with various descriptions. In this section, we quantify our approximation error with respect to the original counter-estimated powers.

Figure 12 shows the absolute error in reconstructed total power for both representative vectors and selected execution points. Figure 13 shows the component-wise absolute errors and how they accumulate at each time sample for both cases. Additionally, on the secondary y axes, it shows the reconstructed total power traces together with the original total power estimation for gzip. The component-wise errors for representative vectors and vectors based on selected execution points (Figure 13) differ in one major aspect. As mentioned in section 6.3, since we choose the startpoints of groups as the execution points, the sum of absolute errors for components is always within the specified threshold for selected execution points, while the errors for representative vectors are not necessarily bound with the same threshold. As a result, stacked component errors for representative vectors occasionally shoot higher than the threshold—4.735 W for the 10% threshold—in Figure 13(a). However, as the representative vectors are the centers of each group, they have a lower average error over the whole timeline. For representative vectors, the RMS error is 2.31W while maximum error is 7.10W. For execution points, the RMS error is 3.08W and the maximum error is 4.71W, in accordance with the discussed expectations. Hence, the component-wise errors in Figure 13 have a higher range than total power errors in Figure 12, as they are based on the manhattan distances between the power vectors. In other words, while absolute errors for total powers show the absolute difference of the sum of vector components, the component based errors show the sum of absolute differences of vector components. Thus, two converse power behaviors are prevented from canceling each other.



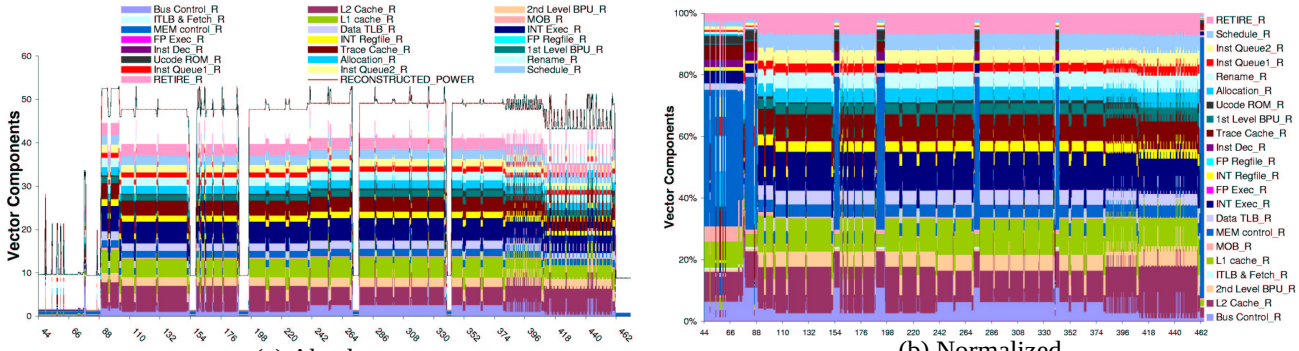
(a) Absolute (b) Normalized

Figure 9. Original gzip power vector traces.



(a) Absolute (b) Normalized

Figure 10. Gzip power vector traces based on representative vectors.



(a) Absolute (b) Normalized

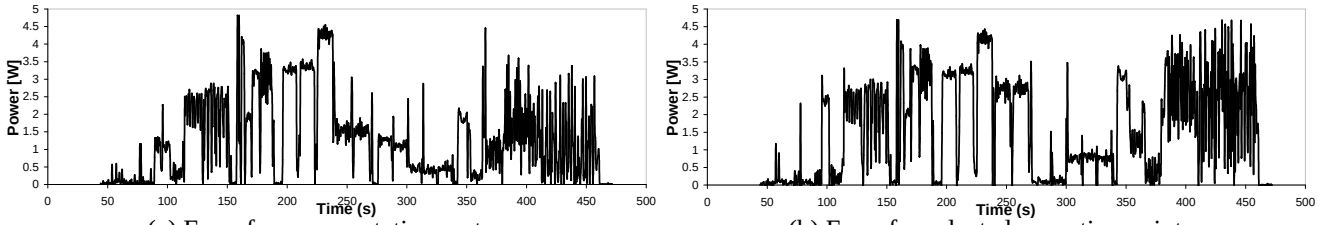
Figure 11. Gzip power vector traces based on selected execution points.

7 Discussion and Future Work

Work reported in this paper is a preliminary description of our broader research related to power phase analysis and there are several issues that we plan to address in future research. Here, we present some of these issues and discuss particular aspects of our work.

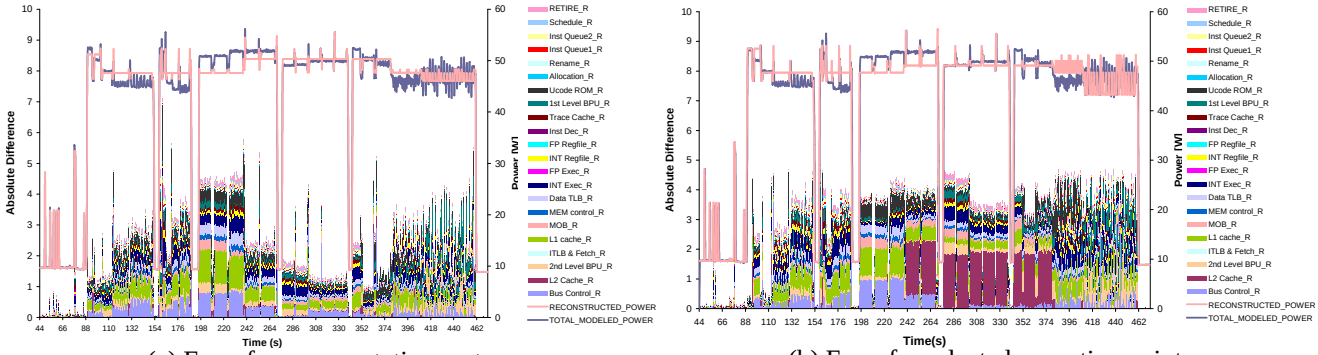
Although the variability in several dimensions of the power vectors is what enables the program power phase characterization, some of the dimensions such as the issue related components show very correlated variations. Moreover, although useful for processor component power estimations, some dimensions are actually driven by the same performance events, which do not directly contribute to phase identification. Therefore, one aspect of future work involves reducing the dimension of power vectors without loss of power behavior. In future research, we plan to apply principal component analysis (PCA)[6, 14] to generate a new set of components as a linear combination of the original set so that each component exhibits a different degree of variance. PCA is a good way of removing redundancies in our application, where some components tend to move together.

As described in Section 6, one of the primary aims of power phase analysis is to be also able to identify a small set of simulation points that characterize power behavior. However, although we can identify simulation points, we cannot verify our approach with power simulations as we do not have access to a sufficiently accurate P4 power simulator. Therefore, one direction of our current work involves relating the power phase behavior to program structure and



(a) Error for representative vectors (b) Error for selected execution points

Figure 12. Absolute error in total power estimates reconstructed from representative vectors and selected execution points.



(a) Error for representative vectors (b) Error for selected execution points

Figure 13. Component-wise absolute errors stacked along the timeline.

identify execution points for a program that can be applied in a different architectural simulator.

Finally, there exist other possibilities for generating the phase groups such as the k-means partitioning algorithm [8], which iteratively assigns a set of vectors to k centroids starting from random center points. However, our thresholding algorithm serves for our power characterization purposes as the direct interpretation of manhattan distance for total power. This provides confidence that the total power difference between the starting vector of a group and all other members of the group will be within the given threshold. Nonetheless, a combination of the two algorithms, where first the thresholding algorithm determines the number of of groups (k) for a given threshold and then the k-means algorithm performs the partitioning for the given number of groups might perform better. Moreover, as mentioned in Section 6.2, a two-pass thresholding algorithm can be used to further decompose large groups with intra-group phases. This method can produce groups with significantly higher similarity with a slight increase in the number of groups.

8 Conclusion

In this paper we presented a power phase analysis methodology for characterizing program power behavior based on *power vectors* sampled at program runtime with the performance counter based power estimation setup. We used our methodology to identify execution regions with similar power behavior for gzip and grouped these execution points using a restrictive similarity metric and a threshold based grouping algorithm. Furthermore, we identified execution points and representative power vectors for different specified thresholds based on the similarity groups generated by the thresholding algorithm and quantified the accuracy of our characterizations by comparing the original power trace to reconstructed power traces. Our investigation of different similarity metrics revealed that characterizing program power based on either the absolute differences of power vector components or the similarity of ratio distributions among components potentially identifies spurious similarities. Therefore, we defined a combined similarity metric, which identifies similarities common to both metrics and showed that it identifies only true similarities effectively. Moreover, we demonstrated that considering only total program power behavior conceals most of power phase information and can result in misleading conclusions. The experiments with different similarity thresholds revealed that the number of groups quickly decrease as thresholds increase within the 1-5% range, and reconstructed power traces produce an almost perfect match for thresholds around 1%, with only 1/4 of the original power vectors. The error analysis between the original power trace and reconstructed traces showed that the execution points always limit the error in characterization within a given threshold due to the generation of similarity groups and selection of execution points. The maximum error for representative vectors, however, can be higher than a given threshold. For whole program execution, selected execution points are shown to generate a more evenly distributed approximation error, but with a higher average error compared to representative vectors.

This research presents a different, power-oriented, program phase analysis technique that is based on runtime processor power estimation. The defined similarity metric characterizes program power behavior based on similarities in both total dissipated power and distribution of power to processor components. Unlike previously used performance metrics, the power vectors also provide a direct relation between the degree of similarity and the variation in total power, which enables us to limit total power variations within a threshold with the thresholding algorithm. The generated representative vectors can be used as *program power signatures* for program power characterization and the selected execution points represent a direct reference for power simulations. Moreover, as our power phase analysis is based on a real, available system, it can readily be used in several aspects of computer architecture research such as dynamic power and thermal management. In conclusion, this work offers a phase analysis methodology specifically targeted at characterizing power behavior. We believe this power phase analysis technique can provide significant insights for power aware and workload characterization research.

References

[1] B. Calder, T. Sherwood, E. Perelman, and G. Hamerly. SimPoint web page. http://www.cs.ucsd.edu/simpoint/.

[2] 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.

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

[4] M. Huang, J. Renau, and J. Torrellas. Profile-Based Energy Reduction in High-Performance Processors. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[5] 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.

[6] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley-Interscience, New York, 1991.

[7] A. KleinOsowski, J. Flynn, N. Meares, and D. J. Lilja. Adapting the SPEC2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization, International Conference on Computer Design*, Sept. 2000.

[8] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297, 1967.

[9] M. C. Merten, A. R. Trick, R. D. Barnes, E. M. Nystrom, C. N. George, J. C. Gyllenhaal, and W. mei W. Hwu. An architectural framework for runtime optimization. *IEEE Transactions on Computers*, 50(6):567–589, 2001.

[10] 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.

[11] 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/.

[12] 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.

[13] B. Sprunt. Pentium 4 performance-monitoring features. *IEEE Micro*, 22(4):72–82, Jul/Aug 2002.

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

[15] 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.

[16] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.