Identifying Program Power Phase Behavior using Power Vectors

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

Abstract

Characterizing program behavior carries significant value in various avenues of computer engineering research from investigation of future architectures tailored better for emerging applications to OS based dynamic management techniques. Most modern applications exhibit distinctively 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 scales comparable to total program execution, 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. In our similarity analysis we use power vectors sampled at program runtime with our power estimation setup, which consist of power values for 22 processor sub-components. We define a power similarity metric as an intersection of both magnitude based and ratio-wise similarities and 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 such as thread scheduling, DVS and DFS , and to provide means to overcome prohibitively long simulations such as identifying representative execution points or reduced datasets.

Most programs show very variant behavior over their whole runtimes and the analysis of these behavior via simulation is usually too 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 demonstrated with several examples that different programs with similar average powers can show significantly different power variation (i.e. gcc, vpr, gzip), and same program with similar total power behavior, can have distinctively different power behavior –in terms of different power component ratios– in different execution phases (i.e. vpr). In this paper, we demonstrate an alternative phase analysis method which relates more directly to power. In our work, we use the *power vector* concept introduced in [5], 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, 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. Also, this in turn enables easy repeatability of the described similarity analysis. For example, 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, which would require times on the order of weeks with a modern computer.

The power phase analysis described in this paper can contribute to current research in various aspects. 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 power behavior characterization technique can be used for OS based dynamic management for thread scheduling, voltage or frequency scaling [4, 15]. Moreover, identified representative 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, 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 our discussion of presented work and our 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 [3, 10, 11, 12, 7, 2] and runtime [14] program profiling techniques to identify phase behavior. Varying goals of these work span diverse areas such as identifying representative simulation point samples, predicting phases, generating reduced datasets and managing multiconfigurable hardware with program signatures. Dhodapkar and Smith [3], 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 mulitconfigurable hardware. They propose a hardware-software implementation for dynamic configuration, where hardware phase tables collect working set signatures and low level software manages hardware configuration.

Sherwood et al. [10] propose *Basic Block Distribution Analysis* method, which uses basic block profiles of programs to identify phases and classifies this phase information into periodic and standalone behavior like initialization. A basic block is a portion of a program code that is entered at one point, executed in whole and has a single exit point. The work in [10] introduces basic block vectors, which represent the proportion of basic block executions within one sampling period. Then, it uses basic block vector differences with respect to a global vector, to identify phases. In order to determine the amount of resemblance between different windows of execution, collected over the program run, [11] defines the basic block similarity matrix, which consists of the manhattan distance between all pairs of basic block vectors. The similarity matrix presents both the duration of similarities and the similar repetitions. [11] then uses 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. These execution points are then used as representative simulations points, whose simulation results are weighted by a factor proportional to group size. Later, [12] extends 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.

Todi [14] uses Intel Itanium Processor performance counters to collect benchmark execution information, then applies principal component analysis to reduce variable dimensions and k-means partitioning algorithm to generate similarity groups, and finally selects representative execution points as closest neighbors to cluster centroids similar to [11].

Other related work include: [7], which provides reduced datasets for spec benchmarks based on functional simulation based profiling; [2], which identifies execution phases over long timescales based on metrics such as IPC (Instructions per clock), IPC variance and IPB similar to [12], and demonstrates workloads exhibit different execution phases along their lifetimes;[9], which uses a branch behavior buffer and detection counter to identify execution hotspots within workloads at runtime and describes program phases based on hotspot frequencies.

In comparison to the previous work, our research shares certain similarities. First of all, we use a similar similarity analysis technique as [11]. However, our similarity are based on estimated power vectors and we use a combination of normalized and non-normalized measures for power components. Second, similar to [14], the basis of our similarity analysis data rely 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 – which essentially stems from counter implementation differences in the P4 and Itanium architectures. We also collect counter data in lots of dimensions, but rather than rerunning the experiment at different times, we use counter rotations within a single experiment as the flexibility of P4 counter reading mechanism enables us to read 15 counters simultaneously, while Itanium architecture limits the auther to 3 counters per run. At the bottomline, we present a power oriented phase analysis methodology and since our technique is based on runtime power estimation rather than simulation, the generation of power similarities is at almost workload runtime speed. Therefore, the presented methodology is easily repeatable and extensible to new emerging workloads.

3 Methodology

In [5], we introduced a runtime methodology for component power estimation based on P4 performance counters [13] and we suggested power phase analysis as an alternative means for workload characterization, which utilized the power estimation methodology. In our following phase analysis, we make use of the same power estimation framework with the experimental setup shown in Figure 1. With this setup, 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.

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

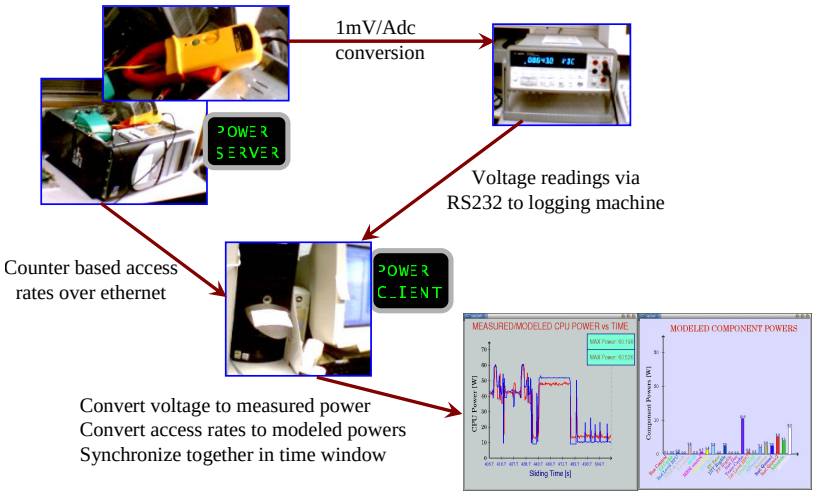


Figure 1. Power measurement and estimation setup

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, we use these generated component power estimates as 22-dimensional *power vectors* at each sampling point as our multivariate data for similarity analysis.

In reference to the broader aspect of our work, power phase analysis is enabled by the power estimation step, which provides us with the power vectors for a given benchmark, based on the utilized estimations. Power estimation step, in turn is enabled by performance monitoring and real power measurement steps as the latter two provide the power model with required information for estimation and verification.

Specifically during power phase analysis, we first collect power vectors, measured data and timing information for a given banchmark 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 the manhattan distance between the vector pairs. Based on a thresholding algorithm described in Section 6, we generate grouping matrices for a given similarity threshold. These demonstrate, for each execution point, what other execution points lie within its similarity threshold. Analyzing the produced grouping matrix, we partition the execution points into a smaller set of groups that contain vectors that satisfy specified amount of similarity. Afterwards, we identify representative vectors for each group, which represent the average group power behavior, and reconstruct power trace based on group distribution and representative vector informations and verify closeness of our approximation to original power behavior.

4 Power Behavior of Programs Constitute Phases

In [5], we showed that 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. This phase behavior is very common in applications can be observed in various other applications. 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 averager 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.

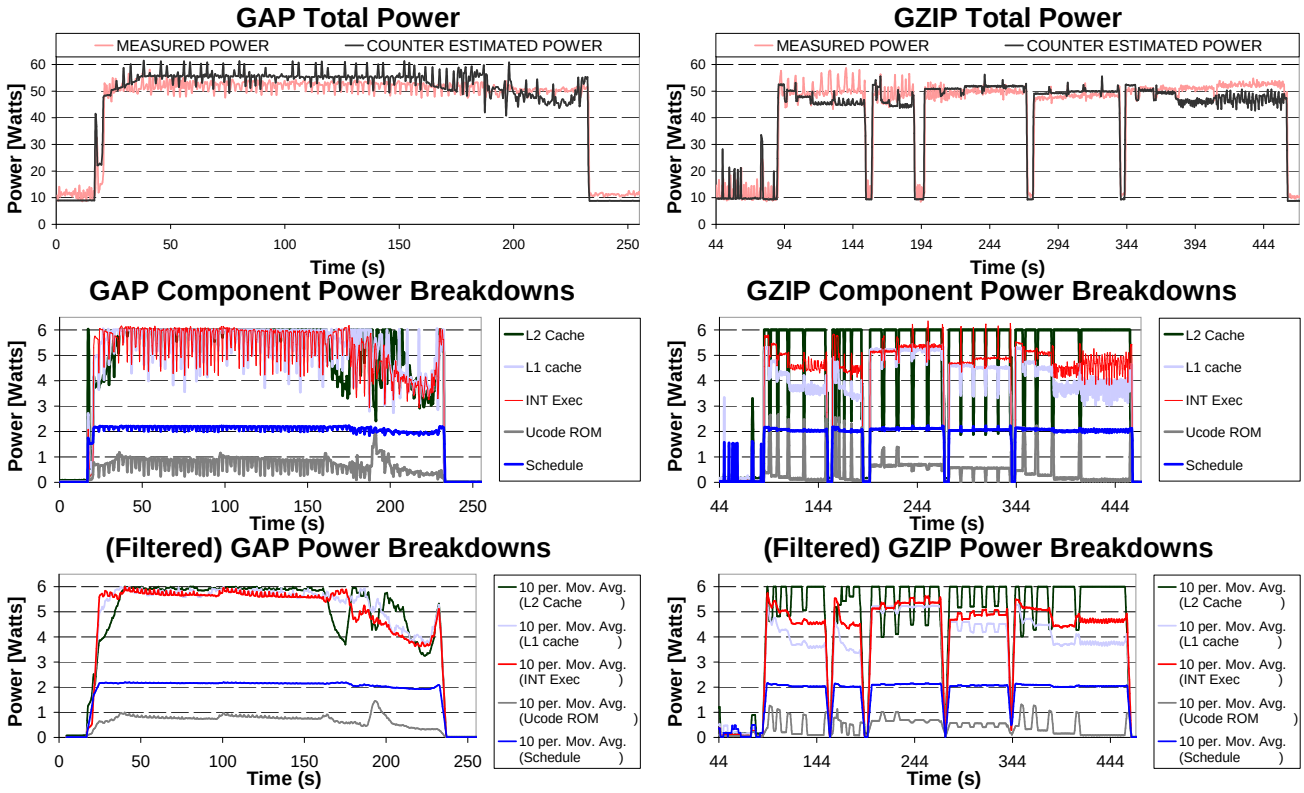


Figure 2. Program Power Phases for Gap and Gzip

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 obsrvations, 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 similarity analysis, we consider the power vectors as pointers to execution points in the positive quadrant of the power space defined by the 22 vector components. We define the amount of power behavior dissimilarity between any two pairs of execution points (or interchangibly power vectors) as the manhattan distance between the two vectors, which is defined as the absolute difference of vector elements summed over all vector components. We use a similarity matrix representation for our power vector sequences in order to quantify power behavior similarities within a benchmark [11]. The constructed upper diagonal power similarity matrices record the distance between all pairs of vectors such that, a matrix entry (r,c) shows the manhattan distance between the power vectors r and c. Only the upper diagonal needs to be constructed as distance from r to c is identical to distance from c to r. The diagonal of the matrix (r,r) corresponds to actual execution points and contains only zeros. Therefore, as each vector is perfectly similar to itself, perfect similarity corresponds to a zero in the matrix entry, while higher values represent higher dissimilarity. The execution time flow is along the matrix diagonal and for an execution point (r, c), points $(r1 < r, c)$ represent the similarity with respect to previous samples, while points $(r, c < c1)$ 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 end of timeline. In the similarity plots shown in Figures 3, 4, 5 and 6, the shading is scaled from white, for maximum dissimilarity, to black, for perfect similarity, where darker regions represent higher similarity between the corresponding component power vectors.

In [5], we had suggested the similarity matrix to be constructed from the original power vectors acquired from the runtime estimations. 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, finally arriving at our final similarity

metric.

5.1 Similarity Based on Total Power

In order to define 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 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.

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

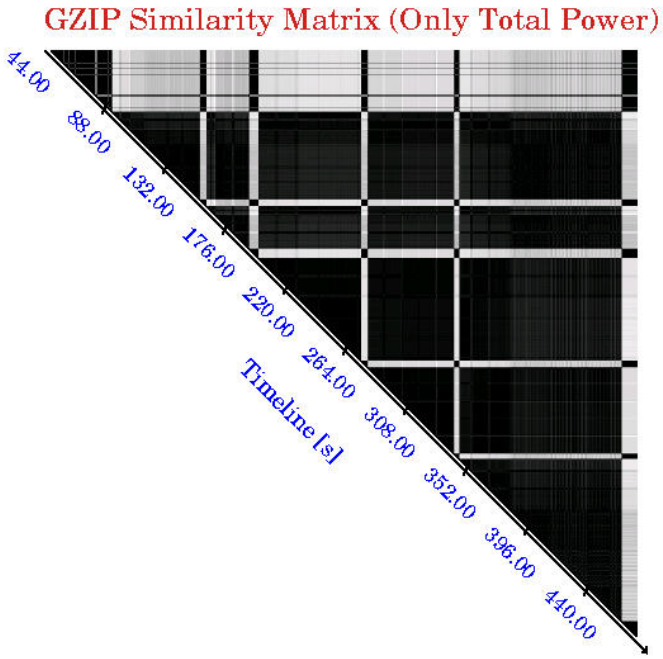


Figure 3. Similarity Matrix based on total power

The similarity matrix in Figure 3, hides significant amount of information that can be inferred from Figure 2 to identify different program power phases that represent different power behavior. 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. In order to identify these phases, we utilize power vector based similarity analysis as discussed in the next three sections.

5.2 Similarity Based on Original Power Vectors

Here, we describe our initially proposed similarity metric based on non-normalized power vectors. The similarity matrix is constructed from 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.

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

The generated matrix plot is shown in Figure 4, which identifies some of the phase information 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 increases within small time bursts. However, an inherent downside of this non-normalized approach is, vectors of smaller magnitude are bound to be considered similar even though they point to very different directions in power space as 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 as discussed in the next two subsections.

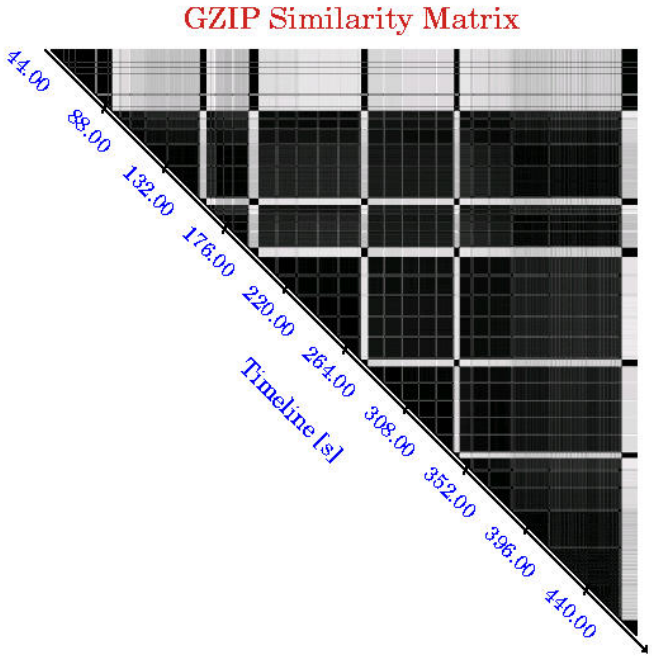


Figure 4. Similarity Matrix based on original power vectors.

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 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 generated matrix plot is shown in Figure 5, where the effects of normalization are readily observable. 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 Figure 4. For instance, 44-70s, 70-88s and 460-480s execution regions, which are identified as highly similar in figure 4 are distinctively discriminated in Figure 5. 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 can be observed at higher power regions such as 220-260s and 270-350s, which are considered as highly similar although the original similarity matrix inf Figure 4 shows a lower degree of similarity between the two regions. Finally, to avoid this pitfall, we present a combined approach as an intersection of the two similarity metrics in the next last subsection.

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. Hence, we do not normalize after the addition of two matrices in order to achieve a final similarity metric which emphasizes dissimilarities. In othere 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.

$$Final\ Similarity\ Matrix(r, c) = \min\left(\frac{OrigSimMatrix(r, c)}{\max_{r,c}(OrigSimMatrix(r, c)} + \frac{NormSimMatrix(r, c)}{\max_{r,c}(NormSimMatrix(r, c)}, 1\right) \tag{4}$$

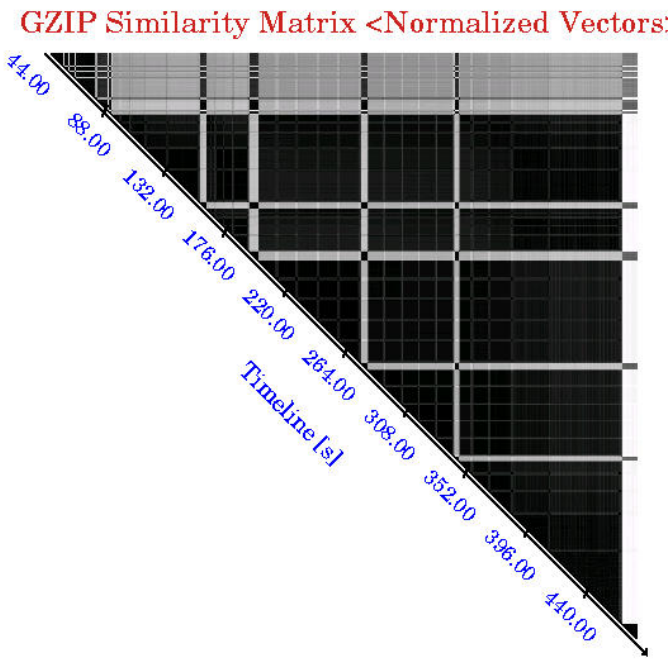


Figure 5. Similarity Matrix based on normalized power vectors.

The matrix plot representing this final similarity metric is shown in Figure 6. 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.

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 behavior. Moreover, this final similarity metric provides a more restrictive selection criterion, by eliminating both magnitude based and ratio based dissimilarities in power behavior. 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 SimilarityMatrix(r,c), we can understand the level of similarity between execution points r and c. Yet, to be able to use the similarity information, we need a more formal way of distinguishing this phase behavior, which will also be guided by the targeted application of this similarity behavior. One of our primary aims in power phase analysis is to achieve a small reduced workload size for a benchmark,

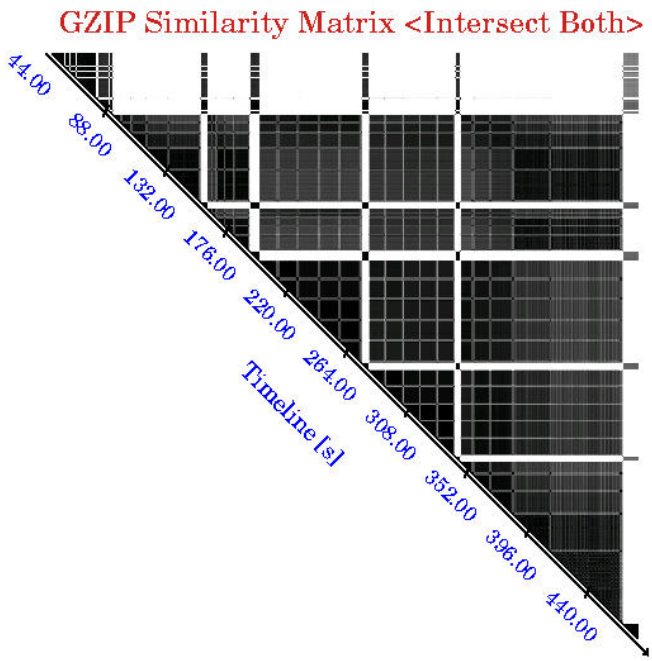


Figure 6. Similarity matrix as the intersection of both normalized and original similarity metrics.

which still captures most of its power behavior. As described in [14], there are two common ways of achieving this aim: reducing datasets [7], or time sampling of execution points [11]. Our methodology is best described as ”Representative Sampling Technique” [14], where we try to achieve a small set of execution points, which 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 power dissections and sequence of appearance in power trace. These signature vectors can be used in program identification and phase prediction.

In this section we introduce our *thresholding* algorithm in order to respond to the two aforementioned issues:

- Grouping execution points –power vectors– based on their similarity
- Representing power behavior with reasonable accuracy with a small number of ”signature vectors”

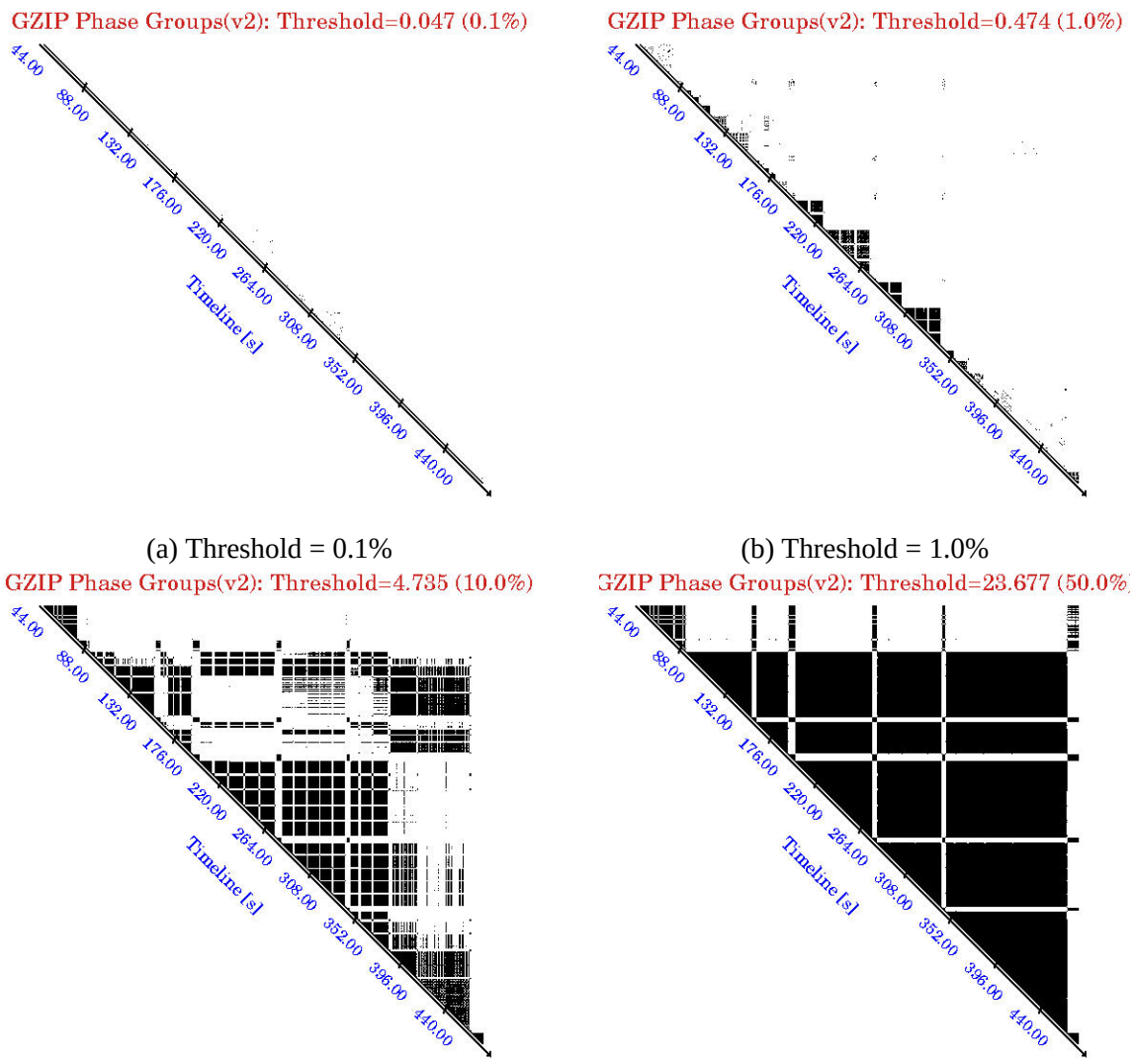
6.1 Thresholding Algorithm

Thresholding algorithm provides a simple means to group execution points, while 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 first execution point (0,0), it identifies the execution points in the forward execution path that lie within the threshold criterion. For example, for a threshold of 10%, a point is considered within threshold if the manhattan distance between the startpoint 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 similarity matrix, which demonstrates all the other similar points to each execution point, for a given threshold.

In Figure 7, we demonstrate the groupings for Gzip. Maximum difference for all pairs of vectors is 47.35 and for normalized vectors it is 1.69. Therefore, for a 1% threshold, the distance between original vectors should be less than 0.4735 and distance between normalized vectors should be less than 0.0169. The figures (a) and (d) show also the extreme cases, where for 0.1% threshold, almost all nodes are only similar to themselves (the second line parallel to timeline axis represents the execution points, which are points at locations (r, r)). For 50%, the only discrimination left is between the low power and high power values due to their large magnitude difference. For the values that lie within 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.

This first step of thresholding algorithm provides us with a more direct similarity information: For each execution point, it shows which other execution points are within the radius of the given threshold so that those points can be considered similar to the observed execution point. However, this doesn't yet divide the execution points into smaller sets of groups. To acquire this final identification of groups, we walk through the generated grouping matrix along the execution path and for an execution point (r, r) in the matrix, identify the points $(r, c > r)$ in the forward execution path that lie within thethreshold. Then, we tag the corresponding execution points (c, c) as the same group. Then, we find the next untagged execution point along execution 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 8, we first show in figure (a) the measured and modeled total power trace for Gzip. Then, in figures (b) and (c) we show the distribution of similarity groups generated by two different thresholds. Figure (b) shows the distribution of 254 groups along the same timeline for a tight threshold of 1%, while figure (c) shows the distribution of 33 groups for a more relaxed threshold of 10%. For the tighter threshold, the group assignments seem to have an almost monotonically increasing trend along the timeline, which means most of the points, which start a new group, can gather only their forward path near neighbors, while unable to include many execution points in further timescales. On the other hand, when we release the similarity threshold, several execution points begin to collapse into same similarity groups. In addition to the above two cases, we have generated group distributions for several other thresholds and 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, and as the tabulated data 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. Afterwards, the number of groups continues to decrease with a smaller pace, as the groupings start to spawn the standalone vectors at the



(a) Threshold = 0.1%

(b) Threshold = 1.0%

(c) Threshold = 10.0%

(d) Threshold = 50.0%

Figure 7. Grouping Matrices for Gzip with Various Threshold Values

group edges.

There are more established grouping algorithms such as k-means partitioning algorithm [8], which chooses k random center points for a set of vectors and iteratively updates center locations by assigning vectors to the groups defined by these centers based on minimum distance and then recomputing the group centroids. This algorithm is used in the SimPoint work of Sherwood et.al [11] and Todi's SPEClite work [14]. However, our thresholding algorithm serves for our power characterization purposes as the direct interpretation of manhattan distance for total power 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 for a given threshold and then the k-means

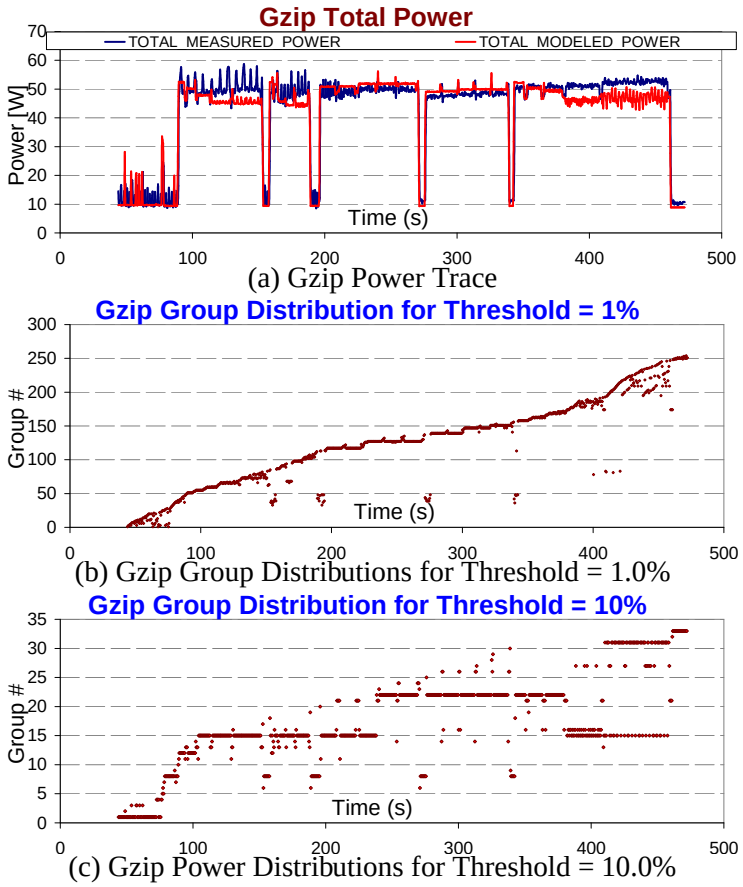


Figure 8. Grouping Distributions for Gzip for 1% and 10% Thresholds

algorithm performs the partitioning for the given number of groups might reveal better results for approximation.

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, as 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 9 and 10 we show the distribution of the 974 power vectors to the 33 groups for Gzip for a chosen 10% similarity threshold as well as the corresponding representative vectors for each group. Although these plots do not directly show the time relation of group vectors, this can be inferred by cross referencing Figures 9 and 10 with the group distributions in Figure 8.

In Figures 9 and 10, the first set of vectors show the vectors that correspond to the execution points that are

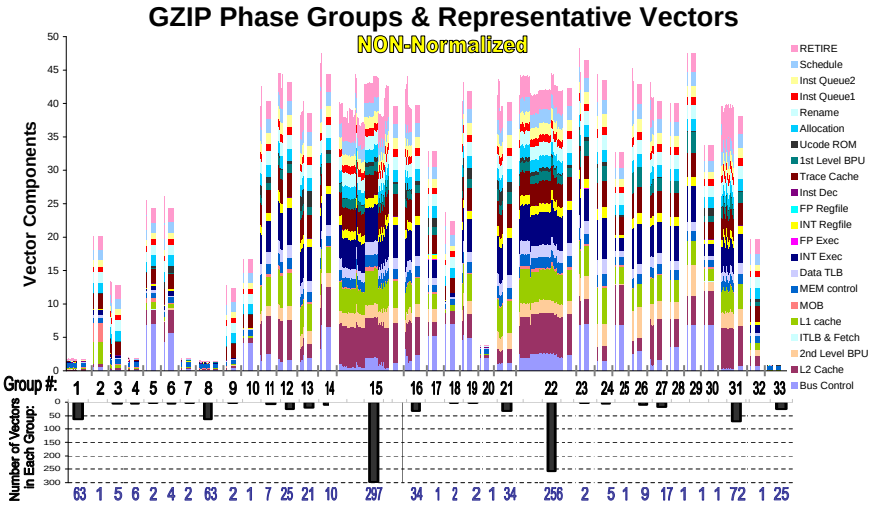


Figure 9. Gzip Power vectors Distributed into Groups and Representative Vectors. For each group number, the first bars are the actual vectors that fall into the that group. The last bar in each group shows the representative vector for that group. The histogram below the group numbers show the number of vectors per group.

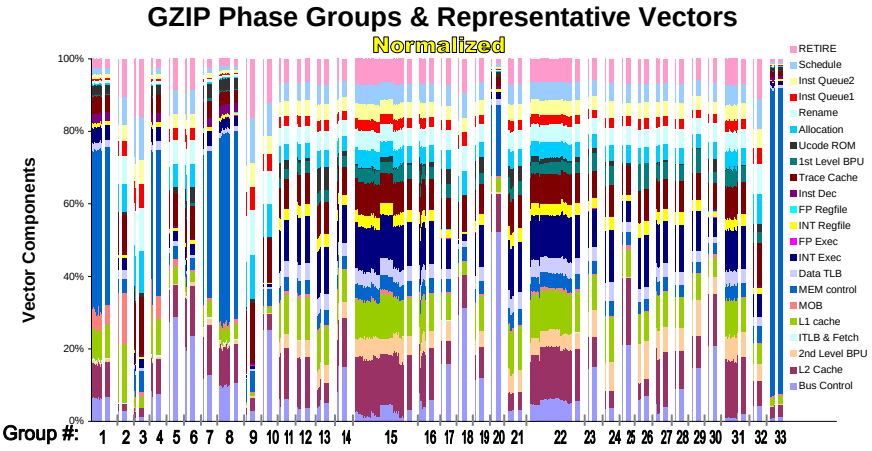


Figure 10. Normalized Gzip Power Vectors and Representative Vectors Distributed into Groups.

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 vs. Threshold for Gzip.

members of the shown group number in the x axis. The second bar shows the generated representative vector for that group as the arithmetic average of all vectors that belong to the same group. In Figure 9, we also show the number of vectors for each group with both the shown histogram, and the actual numbers below the histogram. The most immediate observation from the two figures is, there is a very uneven distribution of vectors into the 33 groups, where groups 15 and 22 actually represent more than 50% of the whole trace. Both normalized and non-normalized plots reveal, 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 given 10% threshold. Nevertheless, depending on the level of desired accuracy, a more greedy grouping mechanism can further focus on these fat groups and apply a second level thresholding with a tighter bound to identify these regions. Moreover, the non-normalized plots in Figure 9 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 such as groups 5 vs. 6 and 21 vs. 31. This last observation lets us assert, with a power vector based phase analysis, we can also discriminate phases into groups such as high L1 cache and low L2 power –i.e. group 21– or such as high L2 power with low bus power –i.e. group 31.

6.3 Selecting Execution Points

As we have discussed at the beginning of this section, our primary aim is to come up with a manageable set of execution points that capture most of the program power behavior. Unlike the representative vectors, the execution points should actually refer to an actual execution time so that those specified points can identify power simulation points similar to Calder et.al.ś SimPoints. The methodology both [11] and [14] use is, choosing the vector closest to the centroids of their generated clusters. This translates to our description as the execution points corresponding to the power vectors closest to the representative vectors for each group. However, as discussed in section 6.1, the main advantage of thresholding algorithm is that, the distance between the startpoint of a

group and all other members of the group is always bounded by the given threshold. Therefore, in our selection of the execution points, we choose the earliest occuring member of each group –the startpoint– as the selected execution point for that group. Thus, 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. Additionally, as [1] discusses, choosing representative simulation points earlier in the execution timeline reduces the time required to fast forward to the selected simulation points. As our implementation of the thresholding algorithm also takes this point into consideration, by walking along the forward execution path to discover the group startpoints, the selected execution points are always the 'early' simulation points in our experiments.

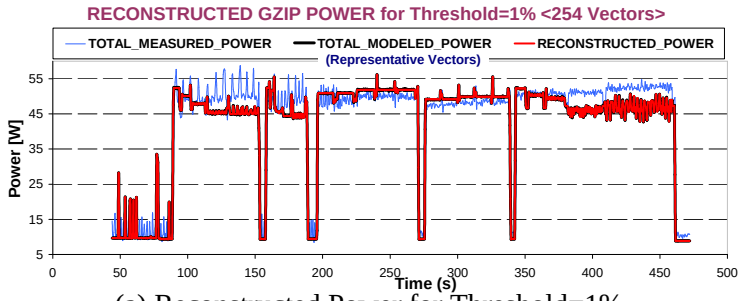
The power vectors for the selected execution points can be readily inferred from figures 9 and 10, as the first vector in each group's set of vectors. Further discussion of selected execution points and acquired results is included in sections 6.4 and 6.5, where we demonstrate the achieved power trace approximation and the range of approximation error.

6.4 Reconstructing Power Traces

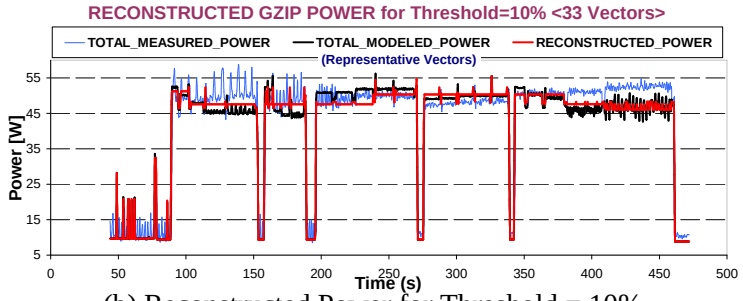
After having specified the representative vectors in section 6.2, for each execution point, we assign the representative vector for the corresponding group as that point's power vector and thus, reconstruct the whole power trace with only the representative vectors. Similarly, referring to the selected execution points in section 6.3, we identify the corresponding power vectors and construct the power trace based on selected execution points' vectors. These reconstructed power traces demonstrate the closeness of power behavior characterizations to the original power traces. In Figure 11, we show the reconstructed traces based on representative vectors for thresholds of 1% and 10% respectively in figures (a) and (b), and in Figure 12, we show the power traces based on selected execution points. For Figure 12, we only include the trace for 10% threshold as 1% is indistinguishable from Figure 11(a). All the three plots show actual measured as well as modeled power as the reconstructed power is actually a second approximation to the actual power through the performance counter based power estimation. Thus, we can also assess the accuracy of our characterization with respect to the real power behavior.

The reconstructed power trace for 1% threshold in Figure 11(a) shows almost a perfect matching to the original power behavior, with aproximately $1/4$ of the original power vectors. Figure 11(b) shows a distinguishable amount of mismatch between the traces, but it characterizes the whole power behavior with only 33 vectors, which are approximately 3.9% of the total vector samples. The power trace based on the vectors that correspond to the selected execution points in Figure 12 also shows a close but distinguishable approximation for the same 10% threshold. The difference in Figure 12 seems to be higher over the whole trace compared to Figure 11(b), which we discuss in more detail in section 6.5.

It is worth to note that, the comparisons in figures 11 and 12 only compare the total power behavior, while the



(a) Reconstructed Power for Threshold=1%



(b) Reconstructed Power for Threshold = 10%

Figure 11. Reconstructed Power Traces for Gzip based on Representative Vectors.

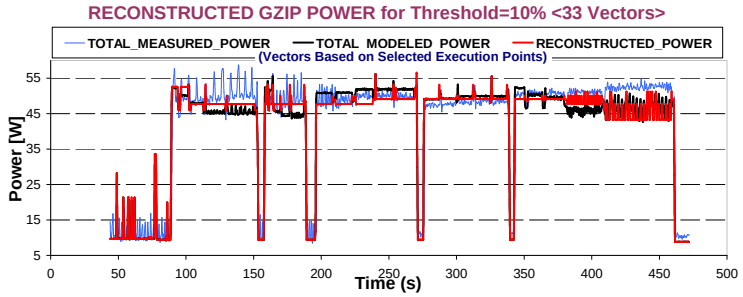
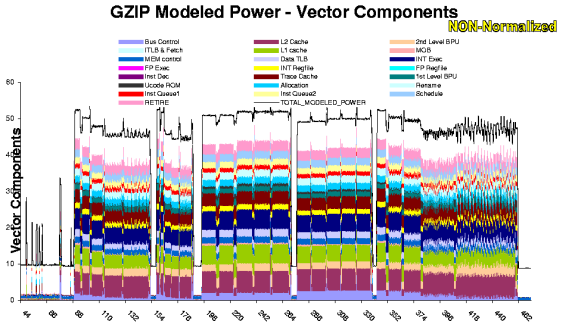
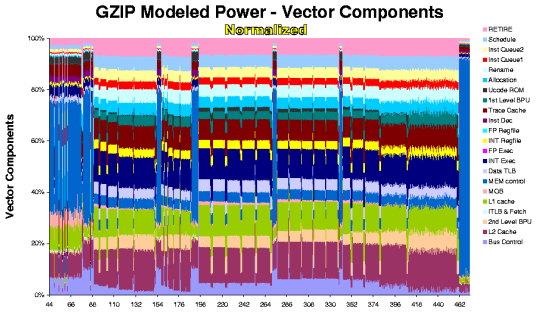


Figure 12. Reconstructed Power Trace for Gzip based on Selected Execution Point Vectors.,

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 13, 14 and 15. In Figure 13 we show the original counter estimated power vectors both with magnitudes and as normalized. In Figures 14 we show the resultant vector traces for representative vectors, and in Figure 15 we show the vectors based on the selected execution points, both normalized and non-normalized. In Figures 13, 14 and 15, we only show the vector traces for 10% threshold, as the 1% case is indistinguishable from the original vectors. In the non-normalized plots, we also show the total power trace, which is 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.

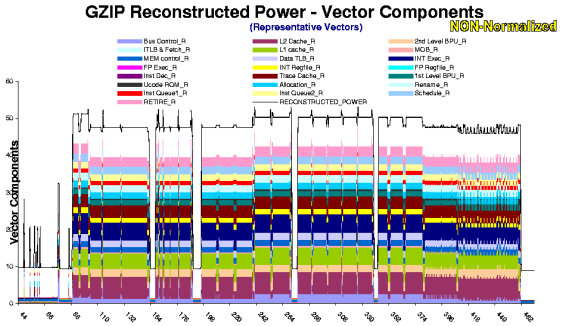


(a) Non-Normalized

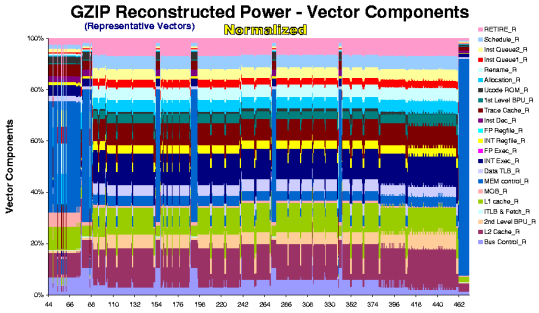


(b) Normalized

Figure 13. Gzip Original Power Vector Traces

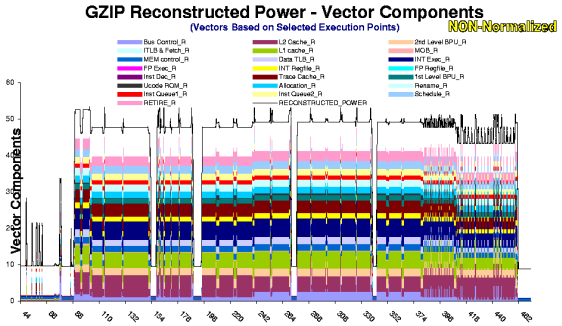


(a) Non-Normalized

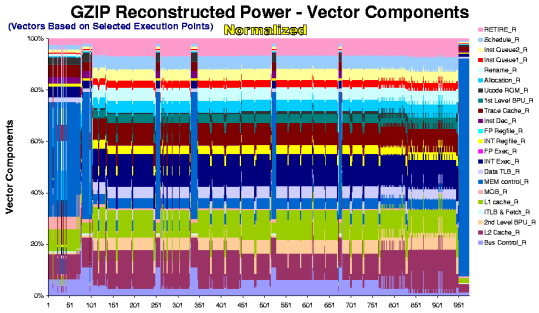


(b) Normalized

Figure 14. Gzip Power Vector Trace based on Representative Vectors



(a) Non-Normalized

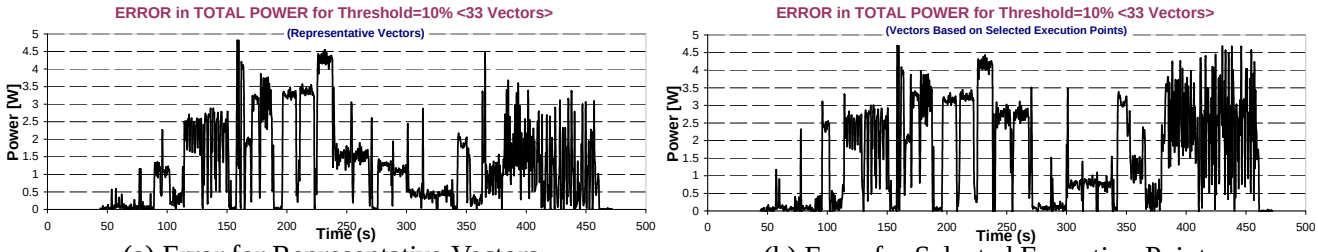


(b) Normalized

Figure 15. Gzip Power Vector Trace based on Selected Execution Points

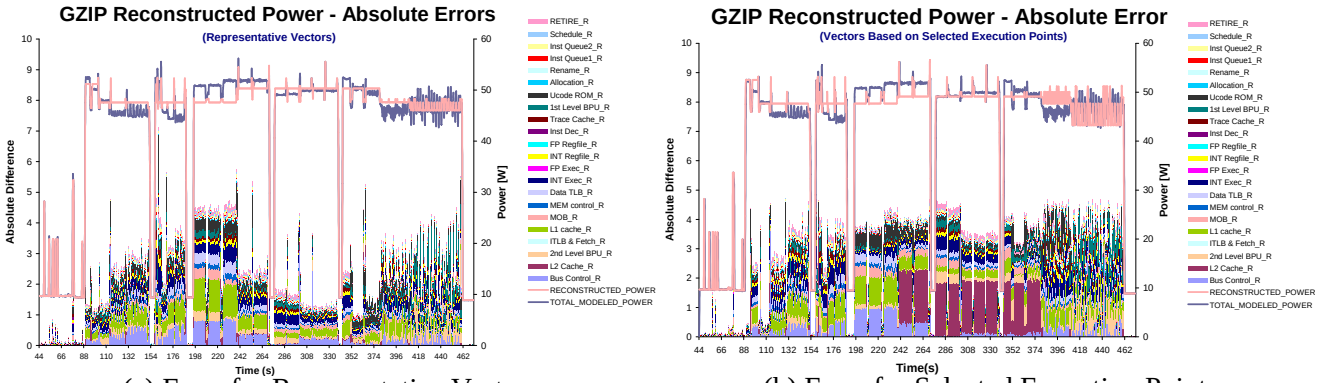
6.5 Error Analysis

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. We show the absolute error for total power in Figure 16 and we also show vector component based absolute errors in Figure 17.



(a) Error for Representative Vectors (b) Error for Selected Execution Points

Figure 16. The absolute error for total power characterizations



(a) Error for Representative Vectors (b) Error for Selected Execution Points

Figure 17. The component-wise absolute errors

The plots for representative vectors and vectors based on selected execution points differ in one major aspect. As mentioned in section 6.1, since we choose the startpoints of groups as the execution points, the sum of absolute errors for components is always within the specified threshold, while the errors for representative vectors are not necessarily bound with the same threshold. As a result, the errors for representative vectors occasionally shoot higher than the threshold –4.735 W for 10% threshold. However, as the representative vectors are at the center of each group, the cumulative error based on representative vectors is expected to be lower. These are reflected in the vector plots as a more evenly distributed error in execution point based plots, while representative vector plots show a lower average error over the whole timeline. For representative vectors, The RMS error is 2.31W and while maximum error is 7.10W. For execution points, the RMS error is 3.08W and the maximum error is 4.71W, which are in accordance with the discussed expectations. Hence, the maximum differences are higher than the range of total power errors in 16, as they are based on the manhattan distance between the power vectors. In other words,

while absolute errors based on total power show the absolute difference of the sum of vector components, the component based errors show the sum of absolute differences of vector components, thus preventing two counter power behaviors from cancelling each other.

7 Discussion and Future Work

Work reported in this paper is a preliminary decsription of our broader research related to power phase analysis and there are several issues that we plan to address in our current our future research. In this section, we discuss some of the issues that define our future work and also some shortcomings of our approach.

Although the variability in several dimensions of the power vectors is what enables the program power phase characterization, some of the dimensions show very similar variations such as the issue related components. 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 dimensionality of power vectors without loss of power behavior. [11] uses a random projection technique for the same purpose, while [14] uses the Principal Component Analysis (PCA)[6], which generates a new set of components as a linear combination of original components so that each component represents a different degree of variance in the applied dataset. In future research, we plan to use PCA, as it is a good way of removing the redundancies for our application, where some components tend to move together. Moreover, one of aims is to be able to introduce dimensions that are conceptually meaningful, while reducing dimensions, such as directions that represent high memory subsystem power, issue power, etc.

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 don't have access to a P4 power simulator. Therefore, one direction of our curent work involves relating the power phase behavior to program structure and identify execution points for a program that can be applied with a different architectural simulator.

Finally, as discussed in sections 6.1 and 6.2, there exist other possibilities for generating the phase groups, which can potentially perform better. A combination of thresholding and k-means algorithm may provide a better characterization, while still letting us identify execution points that satisfy the specified threshold or a two-pass thresholding algorithm can produce groups with significantly higher similarity with a slight increase in number of groups, such as separating the distinguishable regions in groups 15 and 22 of Figure 9.

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, 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 powertrace 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, while the maximum error for representative vectors can be significantly 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 and we believe, this power phase analysis technique can provide a significant insight to 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.