Phase Complexity Surfaces: Characterizing Time-Varying Program Behavior

Frederik Vandeputte Lieven Eeckhout

ELIS Department, Ghent University
Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium
Email: {fgvdeput,leeckhou}@elis.UGent.be

Abstract

It is well known that a program execution exhibits time-varying behavior, i.e., a program typically goes through a number of phases during its execution with each phase exhibiting relatively homogeneous behavior within a phase and distinct behavior across phases. In fact, several recent research studies have been exploiting this time-varying behavior for various purposes.

This paper proposes phase complexity surfaces to characterize a computer program's phase behavior across various time scales in an intuitive manner. The phase complexity surfaces incorporate metrics that characterize phase behavior in terms of the number of phases, its predictability, the degree of variability within and across phases, and the phase behavior's dependence on the time scale granularity.

1 Introduction

Understanding program behavior is at the foundation of computer system design and optimization. Deep insight into inherent program properties drive software and hardware research and development. A program property that has gained increased interest over the past few years, is time-varying program behavior. Time-varying program behavior refers to the observation that a computer program typically goes through a number of phases at run-time with relatively stable behavior within a phase and distinct behavior across phases. Various research studies have been done towards exploiting program phase behavior, for example for simulation acceleration [9,26], hardware adaptation for energy consumption reduction [1,6,7,27], program profiling and optimization [11,21], etc.

This paper concerns characterizing a program's phase behavior. To identify phases, we divide a program execution into non-overlapping intervals. An *interval* is a contiguous sequence of instructions from a program's dynamic instruction stream. A *phase* is a set of intervals within a program's execution that exhibit similar behavior irrespective of temporal adjacency, i.e., a program execution may go through the same phase multiple times.

Basically, there are four properties that characterize a program's phase behavior.

– The first property is the *time scale* at which time-varying program behavior is being observed. Some programs exhibit phase behavior at a small time granularity while other programs only exhibit phase behavior at a coarse granularity; and yet other programs may exhibit phase behavior at various time scales, and the phase behavior

may be hierarchical, i.e., a phase at one time scale may consist of multiple phases at a finer time scale.
- The second property is the *number of phases* a program goes through at run-time. Some programs repeatedly stay in the same phase, for example when executing the same piece of code over and over again; other programs may go through many distinct phases.
- The third property concerns the *variability* within phases versus the variability across phases. The premise of phase behavior is that there is less variability within a phase than across phases, i.e., the variability in behavior for intervals belonging to a given phase is fairly small compared to intervals belonging to different phases.
- The fourth and final property relates to the *predictability* of the program phase behavior. For some programs, its time-varying behavior is very regular and by consequence very predictable. For other programs on the other hand, time-varying behavior is rather complex, irregular and hard to predict.

Obviously, all four properties are related to each other. More in particular, the time scale determines the number of phases to be found with a given degree of homogeneity within each phase; the phases found, in their turn, affect the predictability of the phase behavior. By consequence, getting a good understanding of a program's phase behavior requires all four properties be characterized simultaneously.

This paper presents *phase complexity surfaces* as a way to characterize program phase behavior. The important benefit over prior work in characterizing program phase behavior is that phase complexity surfaces capture *all* of the four properties mentioned above in a unified and intuitive way while enabling the reasoning in terms of these four properties individually.

As a subsequent step, we use these phase complexity surfaces to characterize and classify programs in terms of their phase behavior. Within SPEC CPU2000 we identify a number of prominent groups of programs with similar phase behavior. Researchers can use this classification to select benchmarks for their studies in exploiting program phase behavior.

2 Related Work

There exists a large body of related work on program phase behavior. In this section, we only discuss the issues covered in prior work that relate most closely to this paper.

Granularity. The granularity at which time-varying behavior is studied and exploited varies widely. Some researchers look for program phase behavior at the 100K instruction interval size [1,6,7]; others look for program phase behavior at the 1M or 10M instruction interval granilarity [23]; and yet others identify phase behavior at yet a larger granularity of 100M or even 1B instructions [22,26]. The granularity chosen obviously depends on the purpose of the phase-level optimization. The advantage of a small time scale is that the optimization can potentially achieve better performance because the optimization can be applied more aggressively. A larger time scale on the other hand has the advantage that the overhead of exploiting the phase behavior can be amortized more easily.

Some researchers study phase behavior at different time scales simultaneously. Wavelets for example provide a natural way of characterizing phase behavior at various time scales [4,13,24], and Lau et al. [17] identify a hierarchy of phase behavior.

Fixed-length versus variable-length phases. Various researchers aim at detecting phase behavior by looking into fixed-length instruction intervals [1,6,7]. The potential problem with the fixed-length interval approach though is that in some cases it may be hard to identify phase behavior because of the effect of dissonance between the fixed-length interval and the natural period of the phase behavior. In case the length of the fixed-length interval is slightly smaller or bigger than the period of the phase behavior, the observation made will be out of sync with the natural phase behavior. To address this issue, some researchers advocate identifying phases using variable-length intervals. Lau et al. [17] use pattern matching to find variable-length intervals, and in their follow-on work [16] they identify program phases by looking into a program's control flow structure consisting of loops, and methods calls and returns. Huang et al. [12] detect (variable-length) phases at method entry and exit points by tracking method calls via a call stack.

Microarchitecture-dependent versus microarchitecture-independent characterization. Identifying phases can be done in a number of ways. Some identify program phase behavior by inspecting microarchitecture-dependent program behavior, i.e., they infer phase behavior from inspecting time-varying microarchitecture performance numbers. For example, Balasubramonian et al. [1] collect CPI and cache miss rates. Duesterwald et al. [8] collect IPC numbers, cache miss rates and branch misprediction rates. Isci and Martonosi [14] infer phase behavior from power vectors. A concern with microarchitecture-dependent based phase detection is that once phase behavior is being exploited, it may affect the microarchitecture-dependent metrics being measured; this potentially leads to the problem where it is unclear whether the observed time-varying behavior is a result of natural program behavior or is a consequence of exploiting the observed phase behavior.

An alternative approach is to measure microarchitecture-independent metrics to infer phase behavior from. Dhodapkar and Smith [7,6] for example keep track of a program's working set; when the working set changes, they infer that the program transitions to another phase. Sherwood et al. [26] use Basic Block Vectors (BBVs) to keep track of the basic blocks executed — BBVs are shown to correlate well with performance in [18]. Other microarchitecture-independent metrics are for example memory addresses [13] and data reuse distances [24], a program's control flow structure such as loops and methods [11,12,16], a collection of program characteristics such as instruction mix, ILP, memory access patterns, etc. [9,19].

Phase classification. Different researchers have come up with different approaches to partitioning instruction intervals into phases. Some use threshold clustering [6,7,27]; others use machine learning techniques such as k-means clustering [26], pattern matching [17,24]; yet others use frequency analysis through wavelets [4,5,13,24].

Phase prediction. An important aspect to exploiting phase behavior is to be able to predict and anticipate future phase behavior. Sherwood et al. [27] proposed last phase, RLE and Markov phase predictors. In their follow-on work [20], they added confidence counters to the phase predictors. Vandeputte et al. [28] proposed conditional update which only updates the phase predictor at the lowest confidence level.

Relation to this paper. In this paper, we characterize program phase behavior at different time scale granularities. To this end, we consider fixed-length intervals, use BBVs to identify phase behavior, use threshold clustering for phase classification, and use a theoretical predictor to study phase predictability. We will go in more detail about our phase characterization approach in the next section.

The important difference between this paper compared to prior work is that the explicit goal of this paper is to *characterize* the complexity of a program's phase behavior in an intuitively understandable way. Most of this prior work on the other hand concerned *exploiting* program phase behavior. The work mostly closely related to this paper probably is the work done by Cho and Li [4,5]. They use wavelets to characterize the complexity of a program's phase behavior by looking at different time scales. This complexity measure intermingles the four phase behavior properties mentioned in the introduction; phase complexity surfaces on the other hand provide a more intuitive view on a program's phase behavior by factoring out all four properties.

3 Phase Complexity Surfaces

As mentioned in the introduction, there are four properties that characterize the program's overall phase behavior: (i) the time scale, (ii) the number of phases, (iii) the within and across phase variability, and (iv) phase sequence and transition predictability. The phase behavior characterization surfaces proposed in this paper capture all four properties in a unified way. There are three forms of surfaces: the phase count surface, the phase predictability surface and the phase complexity surface. This section discusses all three surfaces which give an overall view of the complexity of a program's time-varying behavior. Before doing so, we first need to define a Basic Block Vector (BBV) and discuss how to classify instruction intervals into phases using BBVs.

3.1 Basic Block Vector (BBV)

In this paper, we use the Basic Block Vector (BBV) proposed by Sherwood et al. [25] to capture a program's time-varying behavior. A basic block is a linear sequence of instructions with one entry and one exit point. A Basic Block Vector (BBV) is a one-dimensional array with one element per static basic block in the program binary. Each BBV element captures how many times its corresponding basic block has been executed. This is done on an interval basis, i.e., we compute one BBV per interval. Each BBV element is also multiplied with the number of instructions in the corresponding basic block. This gives a higher weight to basic blocks containing more instructions. A BBV thus provides a picture of what portions of code are executed and also how frequently those portions of code are executed.

We use a BBV to identify a program's time-varying behavior because it is a microarchitecture-independent metric and by consequence gives an accurate picture of a program's time-varying behavior across microarchitectures. Previous work by Lau et al. [18] has shown that there exists a strong correlation between the code being executed — this is what a BBV captures — and actual performance. The intuition is that if two instruction intervals execute roughly the same code, and if the frequency of the portions of code executed is roughly the same, these two intervals should exhibit roughly the same performance.

3.2 Phase classification

Once we have a BBV per instruction interval, we now need to classify intervals into phases. As suggested above, and intuitively speaking, this is done by comparing BBVs to find similarities. Intervals with similar BBVs are considered belonging to the same program phase.

Classifying instruction intervals into phases can be done in a number of ways. We view it as a clustering problem. There exist a number of clustering algorithms, such as linkage clustering, k-means clustering, threshold clustering, and many others. In this paper, we use threshold clustering because it provides a natural way of bounding the variability within a phase. As will become clear later, the advantage of using threshold clustering is that, by construction, it builds phases for which the variability (in terms of BBV behavior) is limited to a threshold θ. Classifying intervals into phases using threshold clustering works in an iterative way. It selects an instruction interval as a cluster center and then computes the distance with all the other instruction intervals. If the distance measure is smaller than a given threshold θ, the instruction interval is considered to be part of the same cluster/phase. Out of all remaining instruction intervals (not part of previously formed clusters), another interval is selected as a cluster center and the above process is repeated. This iterative process continues until all instruction intervals are assigned to a cluster/phase.

In our clustering approach we scan all instruction intervals once from the beginning until the end of the dynamic instruction stream. This means that the clustering algorithm has a complexity of $O(kN)$ with N the number of instruction intervals and k clusters ($k << N$), which is much more efficient than the iterative approach as described above which has an $O(N^2)$ computational complexity.

We use the Manhattan distance as our distance metric between two BBVs:

$$d = \sum_{i=1}^{D} \|A_i - B_i\|,$$

with A and B being two BBVs and A_i being the i-th element of BBV A; the dimensionality of the BBV, or the number of basic blocks in the program binary, equals D. The advantage of the Manhattan distance over the Euclidean distance is that it weighs differences more heavily. Assuming that the BBVs are normalized — the sum over all BBV elements equals one — the Manhattan distance varies between 0 (both BBVs are identical) and 2 (maximum possible difference between two BBVs). The θ threshold is

expressed as a percentage of the maximum possible Manhattan distance between two instruction intervals.

After having applied threshold clustering, there are typically a number of clusters that represent only a small fraction of the total program execution, i.e., clusters with a small number of cluster members. We group all the smallest clusters to form a single cluster, the so called transition phase [20]. The transition phase accounts for no more than 5% of the total program execution.

3.3 Phase count surfaces

Having discussed how to measure behavioral similarity across instruction intervals using BBVs and how to group similar instruction intervals into phases through threshold clustering, we can now describe what a phase count surface looks like. A *phase count surface* shows the number of program phases as a function of intra-phase variability across different time scales, i.e., each point on a phase count surface shows the number of program phases at a given time scale at a given intra-phase variability threshold. The time scale is represented as the instruction interval length, and the per-phase variability is represented by θ used to drive the threshold clustering.

3.4 Phase Predictability Surfaces

As a result of the threshold clustering step discussed in the previous section, we can now assign phase IDs to all the instruction intervals. In other words, the dynamic instruction stream can be represented as a sequence of phase IDs with one phase ID per instruction interval in the dynamic instruction stream. We are now concerned with the regularity or predictability of the phase ID sequence. This is what a phase predictability surface characterizes.

Prediction by Partial Matching. We use the Prediction by Partial Matching (PPM) technique proposed by Chen et al. [3] to characterize phase predictability. The reason for choosing the PPM predictor is that it is a universal compression/prediction technique which presents a theoretical basis for phase prediction, and is not tied to a particular implementation.

A PPM predictor is built on the notion of a Markov predictor. A Markov predictor of order *k* predicts the next phase ID based upon *k* preceding phase IDs. Each entry in the Markov predictor records the number of phase IDs for the given history. To predict the next phase ID, the Markov predictor outputs the most likely phase ID for the given *k*-length history. An *m*-order PPM predictor consists of (*m+1*) Markov predictors of orders *0* up to *m*. The PPM predictor uses the *m*-bit history to index the *m*th order Markov predictor. If the search succeeds, i.e., the history of phase IDs occurred previously, the PPM predictor outputs the prediction by the *m*th order Markov predictor. If the search does not succeed, the PPM predictor uses the *(m-1)*-bit history to index the *(m-1)*th order Markov predictor. In case the search misses again, the PPM predictor indexes the *(m-2)*th order Markov predictor, etc. Updating the PPM predictor is done by updating the Markov predictor that makes the prediction and all its higher order Markov predictors. In our setup, we consider a 32-order PPM phase predictor.

Predictability surfaces. A *phase predictability surface* shows the relationship between phase predictability and intra-phase variability across different time scales. Each point on a phase predictability surface shows the phase predictability as a function of time scale (quantified by the instruction interval granularity) and intra-phase variability (quantified by the θ parameter used during threshold clustering). Phase predictability itself is measured through the PPM predictor, i.e., for a given θ threshold and a given time scale, we report the prediction accuracy by the PPM predictor to predict phase IDs.

3.5 Phase Complexity Surfaces

Having discussed both the phase count surface as well as the phase predictability surface, we can now combine both surfaces to form a so called *phase complexity surface*. A phase complexity surface shows phase count versus phase predictability across different time scales. A phase complexity surface is easily derived from the phase count and predictability surfaces by factoring out the θ threshold. In other words, each point on the phase complexity surface corresponds to a particular θ threshold which determines phase count and predictability at a given time scale. The motivation for the phase complexity surface is to represent an easy-to-grasp intuitive view on a program's phase behavior through a single graph.

3.6 Discussion

Time complexity. The time complexity for computing phase complexity surfaces is linear as all of the four steps have a linear-time complexity. The first step computes the BBVs at the smallest interval granularity of interest. This requires a functional simulation or instrumentation run of the complete benchmark execution; the overhead is limited though. The second step computes BBVs at larger interval granularities by aggregating the BBVs from the previous step. This step is linear in the number of smallest-granularity intervals. The third step applies threshold clustering at all interval granularities. As mentioned in the paper, the basic approach to threshold clustering is an iterative process, our approach though makes a linear scan over the BBVs. Once the phase IDs are determined through the clustering step, the fourth step then determines the phase predictability by predicting next phase IDs — again, this is linear-time complexity.

Applications. The phase complexity surfaces provide a number of potential applications. One is to select representative benchmarks for performance analysis based on their inherent program phase behavior. A set of benchmarks that represent diverse phase behaviors can capture a representative picture of the benchmark suite's phase behavior; this will be illustrated further in section 6. Second, phase complexity surfaces are also useful in determining an appropriate interval size for optimization. For example, reducing energy consumption can be done by downscaling hardware resources on a per-phase basis [1,6,7,27]. An important criterion for good energy saving and limited performance penalty, is to limit the number of phases (in order to limit the training time at run time of finding a good per-phase hardware setting) and to achieve good phase predictability (in order to limit the number of phase mispredictions which may be costly in terms of missed energy saving opportunities and/or performance penalty).

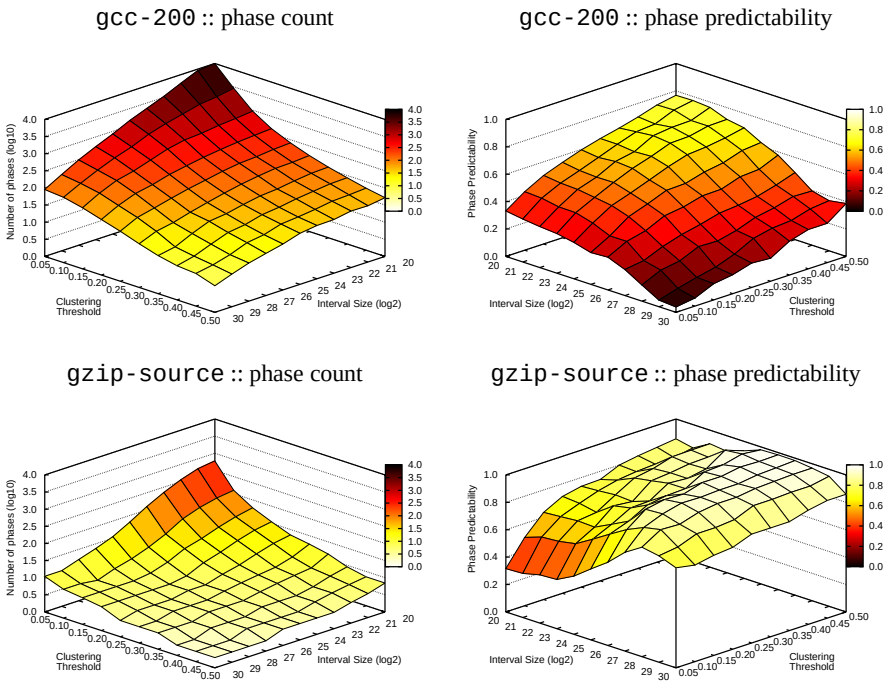

Fig. 1. Phase count surfaces (left column) and phase predictability surfaces (right column) for `gcc-200` (top) and `gzip-source` (bottom).

4 Experimental Setup

We use all the SPEC CPU2000 integer and floating-point benchmarks, use reference inputs for all benchmarks and run all benchmarks to completion. We use the SimpleScalar Tool Set [2] for collecting BBVs on an interval basis.

5 Program Phase Characterization

Due to space constraints, it is impossible to present phase complexity curves for all benchmarks. Instead we present and discuss typical example phase complexity surfaces that we observed during our study. Example surfaces are shown in Figures 1 and 2: Figure 1 shows phase count and predictability surfaces for `gcc-200` and `gzip-source`, and Figure 2 shows surfaces for `bzip2-graphic` and `fma3d`. As mentioned before, a phase count surface shows the (logarithm of the) number of phases on the Z-axis versus the clustering threshold (which is a measure for intra-phase variability) and the interval size (which is a measure of time granularity) on the X and Y axes; the phase predictability surface shows phase predictability on the Z-axis versus clustering threshold and interval size. The θ clustering threshold is varied from 0.05 up to 0.5 in 0.05

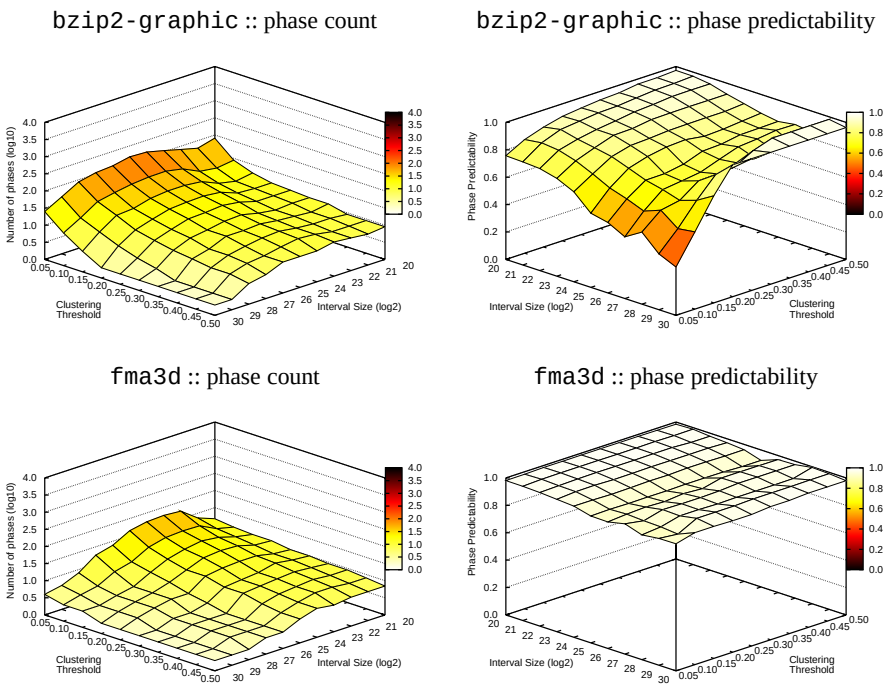

Fig. 2. Phase count surfaces (left column) and phase predictability surfaces (right column) for `bzip2-graphic` (top) and `fma3d` (bottom).

increments — the smaller the threshold, the smaller the intra-phase variability; interval size is varied from 1M up to 1G — note the labels are shown as log_2 of the interval size.

There are basically two types of phase count surfaces. The first type shows a decreasing number of program phases at larger time granularities. This is illustrated in Figure 1. The second type shows an increasing number of program phases at larger time granularities and a decreasing number of program phases at a yet larger time granularity, see Figure 2.

The first type of phase count surface can be explained by the observation that phase behavior at a small time granularity gets averaged out at a larger time granularity. As a result, more and more portions of the program execution start looking similar which is reflected in a decreasing number of program phases. The second type of phase count surface appears for programs with obvious phase behavior, however, this obvious phase behavior seems to be difficult to capture over a range of time scales. This can occur in case the period of the inherent phase behavior is not a multiple of a given time granularity. For the purpose of illustration, consider the following example of a phase ID sequence: ‘AAABBAAABBAAABB...’ with ‘A’ and ‘B’ being phase IDs. The number of phases at time granularity 1 equals 2, namely ‘A’ and ‘B’. At the time granularity of 2, there are 3 phases observed, namely ‘AA’, ‘AB’ (or ‘BA’) and ‘BB’. At the time granularity of 4, there are only 2 phases observed: ‘AAAB’ and ‘AABB’. In some sense

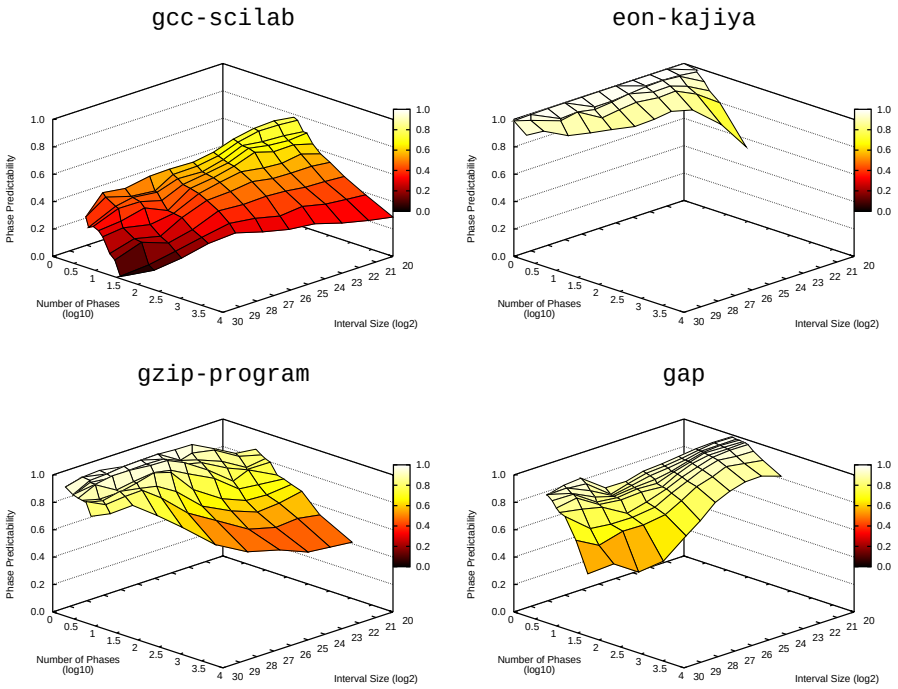

Fig. 3. Phase complexity surfaces for `gcc-scilab` (top left), `eon-kajiya` (top right), `gzip-program` (bottom left) and `gap` (bottom right).

this could be viewed of as a consequence of our choice for fixed-length intervals in our phase-level characterization, however, we observe the large number of phases across a range of time granularities. This seems to suggest that this phase behavior has a fairly long period, and that variable-length intervals (which are tied to some notion of time granularity as well) may not completely solve the problem.

It is also interesting to observe that for both types of phase count surfaces, phase predictability can be high or low. For example, the predictability is low for `gcc-200`, `gzip-source` and `bzip2-graphic` and is very high for `fma3d`. For some benchmarks, phase predictability correlates inversely with the number of phases, see for example `gzip-source`: the higher the number of phases, the lower their predictability. For other benchmarks on the other hand, the opposite seems to be true: phase predictability decreases with a decreasing number of phases, see for example `gcc-200`.

Figure 3 shows the phase complexity surfaces for `gcc-scilab`, `eon-kajiya`, `gzip-program` and `gap` which combine the phase count and predictability surfaces. These examples clearly show two extreme phase behaviors. The phase behavior for `eon-kajiya` is much less complex than for `gcc-scilab`: `eon-kajiya` has fewer program phases and shows very good phase predictability; `gcc-scilab` on the other hand, exhibits a large number of phases and in addition, phase predictability is very poor.

6 Classifying Benchmarks

Having characterized all the benchmarks in terms of their phase behavior using phase complexity surfaces, we can now categorize benchmarks according to their phase behavior. To this end we employ the methodology proposed by Eeckhout et al. [10] to find similarities across benchmarks.

6.1 Methodology

As input to this methodology we provide a number of characteristics per benchmark: we provide phase predictability and (the logarithm of) the number of phases at three threshold values ($\theta = 5\%$, $\theta = 10\%$ and $\theta = 25\%$) at four time scales (1M, 8M, 64M and 512M) — there are 24 characteristics in total. Intuitively speaking, we sample the phase complexity surface. This yields a data matrix with the rows being the benchmark-input pairs and the columns being the 24 phase characteristics.

This data matrix serves as input to Principal Components Analysis (PCA) [15] — the goal of PCA is (i) to remove correlation from the data set and (ii) to reduce the dimensionality. PCA computes new dimensions, called *principal components*, which are *linear combinations* of the original phase characteristics. In other words, PCA tranforms the $p = 24$ phase characteristics $X_1, X_2, \ldots, X_p$ into p principal components $Z_1, Z_2, \ldots, Z_p$ with $Z_i = \sum_{j=1}^{p} a_{ij} X_j$. This transformation has the properties (i) $Var[Z_1] \geq Var[Z_2] \geq \ldots \geq Var[Z_p]$ — this means Z_1 contains the most information and Z_p the least; and (ii) $Cov[Z_i, Z_j] = 0, \forall i \neq j$ — this means there is no information overlap between the principal components. Some principal components have a higher variance than others. By removing the principal components with the lowest variance from the analysis, we reduce the dimensionality of the data set while controlling the amount of information that is thrown away. On our data set we retain 3 principal components that collectively explain 87.4% of the total variance in the original data set. Note that prior to PCA we normalize the data matrix (the columns have a zero mean and variance of one) to put all characteristics on a common scale; also after PCA, we normalize the principal components to give equal weight to the underlying mechanisms extracted by PCA.

We now have a reduced data matrix, i.e., we are left with three principal component values for all benchmark-input pairs. This reduced data set now serves as input to cluster analysis which groups benchmark-input pairs that exhibit similar phase behavior. We use linkage clustering here because it allows to visualize the clustering through a dendrogram. Linkage clustering starts with a matrix of distances between the benchmarks. As a starting point for the algorithm, each benchmark is considered as a group. In each iteration of the algorithm, groups that are closest to each other are merged and groups are gradually merged until we are left with a single group. This can be represented in a so called *dendrogram*, which graphically represents the linkage distance for each group merge at each iteration of the algorithm. Having obtained a dendrogram, it is up to the user to decide how many clusters to take. This decision can be made based on the linkage distance. Indeed, small linkage distances imply strong clustering while large linkage distances imply weak clustering. There exist several methods for calculating the distance between clusters. In this paper we use the weighted pair-group

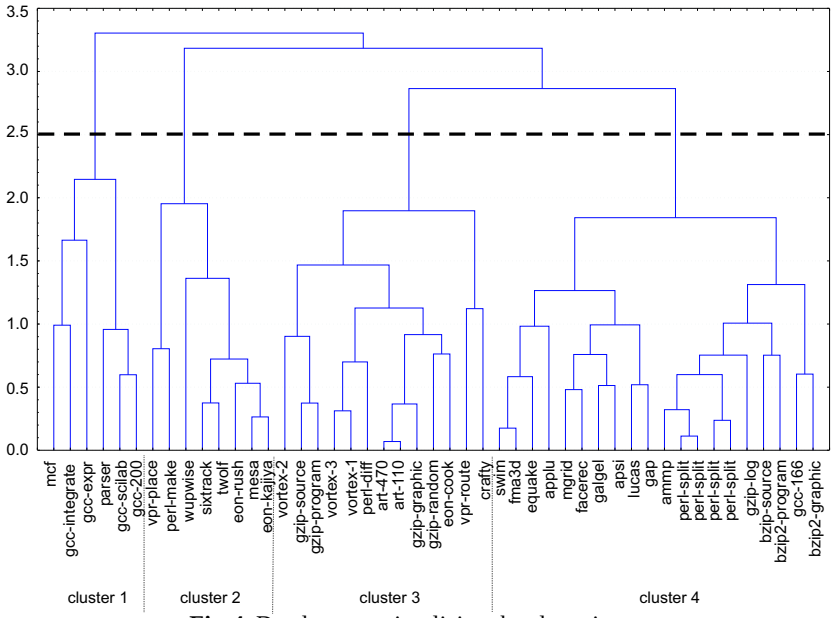

Fig. 4. Dendrogram visualizing the clustering.

ID	benchmarks
1	gcc-200, **gcc-scilab**, gcc-expr, gcc-integrate, parser, mcf
2	**eon-kajiya**, eon-rush, mesa, twolf, sixtrack, wupwise, perl-make, vpr-place
3	crafty, vpr-route, eon-cook, **gzip-program**, gzip-source, gzip-graphic, gzip-random,art, perl-diff, vortex
4	bzip2, gcc-166, gzip-log, perl-split, ammp, **gap**, lucas, apsi, galgel, facerec, mgrid, applu, equake, fma3d, swim

Table 1. Classifying benchmarks in terms of their phase behavior; cluster representatives are shown in bold.

average method which computes the distance between two clusters as the weighted average distance between all pairs of program-input points in the two different clusters. The weighting of the average is done by considering the cluster size, i.e., the number of program-input points in the cluster.

6.2 Results

Figure 4 shows the dendrogram obtained from clustering the benchmarks based on their phase behavior. Classifying the benchmarks using this dendrogram with a critical threshold of 2.5, results in four major clusters representing the most diverse phase behaviors across the SPEC CPU2000 benchmarks, see also Table 1. Note that in case a more fine-grained distinction needs to be made among the benchmarks in terms of their phase behavior, the critical threshold should be made smaller; this will result in more

fine-grained types of phase behavior. We observe the following key phase characteristics in each of the four major clusters:

- cluster 1 :: very poor phase predictability and a very large number of phases
- cluster 2 :: very small number of phases and very good phase predictability;
- cluster 3 :: a relatively poor predictability and a high number of phases at small time granularities, in combination with relatively better predictability and relatively fewer phases at large time granularities;
- cluster 4 :: a moderate number of phases across all time granularities, and mostly good to excellent predictability.

In summary, cluster 2 exhibits the simplest phase behavior. Clusters 3 and 4 show moderately complex phase behaviors, with cluster 3 showing poorer phase predictability at small time granularities. Cluster 1 represents the most complex phase behaviors observed across the SPEC CPU2000 benchmark suite. Referring back to Figure 3, the phase complexity surfaces shown represent an example benchmark from each of these groups: `eon-kajiya` as an example for the simple phase behavior in cluster 2; `gzip-program` and `gap` as examples for the moderately complex phase behaviors in clusters 3 and 4, respectively; and `gcc-scilab` as an example for the very complex phase behavior in cluster 1.

Researchers exploiting program phase behavior can use this classification to select benchmarks for their experiments. A performance analyst should pick benchmarks from all groups in order to have a representative set of program phase behaviors.

7 Conclusion

Program phase behavior is a well-known program characteristic that is subject to many optimizations both in software and hardware. In order to get a good understanding in a program's phase behavior, it is important to have a way of characterizing a program's time-varying behavior. This paper proposed phase complexity surfaces which characterize a program's phase behavior in terms of its four key properties: time scale, number of phases, phase predictability and intra- versus inter-phase predictability. Phase complexity surfaces provide a good intuitive and unified view of a program's phase behavior. These complexity surfaces can be used to classify benchmarks in terms of their inherent phase behavior.

Acknowledgements

We would like to thank the reviewers for their valuable comments. Frederik Vandeputte and Lieven Eeckhout are supported by the Fund for Scientific Research in Flanders (Belgium) (FWO-Vlaanderen). Additional support is provided by the FWO project G.0160.02, and the HiPEAC European Network-of-Excellence.

References

1. R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, Dec. 2000.
2. D. C. Burger and T. M. Austin. The SimpleScalar Tool Set. Computer Architecture News, 1997. See also `http://www.simplescalar.com` for more information.
3. I. K. Chen, J. T. Coffey, and T. N. Mudge. Analysis of branch prediction via data compression. In *ASPLOS*, pages 128–137, Oct. 1996.
4. C.-B. Cho and T. Li. Complexity-based program phase analysis and classification. In *PACT*, pages 105–113, Sept. 2006.
5. C.-B. Cho and T. Li. Using wavelet domain workload execution characteristics to improve accuracy, scalability and robustness in program phase analysis. In *ISPASS*, Mar. 2007.
6. A. Dhodapkar and J. E. Smith. Dynamic microarchitecture adaptation via co-designed virtual machines. In *International Solid State Circuits Conference*, Feb. 2002.
7. A. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, May 2002.
8. E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *PACT*, Oct. 2003.
9. L. Eeckhout, J. Sampson, and B. Calder. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC*, pages 2–12, Oct. 2005.
10. L. Eeckhout, H. Vandierendonck, and K. De Bosschere. Workload design: Selecting representative program-input pairs. In *PACT*, pages 83–94, Sept. 2002.
11. A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In *OOPSLA*, pages 270–287, Oct. 2004.
12. M. Huang, J. Renau, and J. Torrellas. Positional adaptation of processors: Application to energy reduction. In *ISCA*, June 2003.
13. T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *PACT*, pages 95–104, Sept. 2006.
14. C. Isci and M. Martonosi. Identifying program power phase behavior using power vectors. In *WWC*, Sept. 2003.
15. R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, fifth edition, 2002.
16. J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *CGO*, pages 135–146, Mar. 2006.
17. J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *ISPASS*, pages 135–146, Mar. 2005.
18. J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *ISPASS*, Mar. 2005.
19. J. Lau, S. Schoenmackers, and B. Calder. Structures for phase classification. In *ISPASS*, pages 57–67, Mar. 2004.
20. J. Lau, S. Schoenmackers, and B. Calder. Transition phase classification and prediction. In *HPCA*, pages 278–289, Feb. 2005.
21. P. Nagpurkar, C. Krintz, and T. Sherwood. Phase-aware remote profiling. In *CGO*, pages 191–202, Mar. 2005.
22. H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of larhe Intel Itanium programs with dynamic instrumentation. In *MICRO*, pages 81–93, Dec. 2004.

23. E. Perelman, G. Hamerly, and B. Calder. Picking statistically valid and early simulation points. In *PACT*, pages 244–256, Sept. 2003.
24. X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, Oct. 2004.
25. T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT*, pages 3–14, Sept. 2001.
26. T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *ASPLOS*, pages 45–57, Oct. 2002.
27. T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, pages 336–347, June 2003.
28. F. Vandeputte, L. Eeckhout, and K. De Bosschere. A detailed study on phase predictors. In *Euro-Par*, pages 571–581, Aug. 2005.