# Characterizing Phase Behavior Through Time-Varying Microarchitecture Independent Characteristics Clustering

**Rafael Mendonça Soares, Rodolfo Azevedo**

[1]Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Av. Albert Einstein, 1251 – Cidade Universitária – Campinas – SP – Brazil

`rafael.soares@students.ic.unicamp.br`
`rodolfo@ic.unicamp.br`

***Abstract.*** *Programs often exhibit repeating behaviors, which are known as program phases. The automatic discovery of such structured behavior has benefited many applications. However, many existing phase signatures lack the ability to reason about what are the key factors of each phase. Also, programs exhibit phase behavior at many different granularities, and some exhibit hierarchical phase behavior. Many techniques focus on a single granularity, which can cause an out of sync classification with the actual phase behavior. We solve these problems by adopting a recently proposed method of subsequence clustering of multivariate time series. Using this method, the phases started to have a much more interpretable signature (MRF). We graphically showed that the method partitions the execution into a temporally consistent way. We showed the effectiveness of MRF's signature by using a centrality measure to identify the most important characteristics within a program phase. Finally, we present a case study to show the relationship between the MRF signature and source code.*

## 1. Introduction

The execution of a program can often be broken down into a sequence of phases, each having a unique behavior, that can reoccur multiple times throughout execution. The automatic discovery of such structured behavior benefits many applications. For instance, the knowledge of the variable resource requirements throughout execution has been exploited for reconfigurable architectures [Shen et al. 2004, Isci and Martonosi 2006] and simulation acceleration [Sherwood et al. 2002].

A large body of phase analysis methods focuses on giving the location of each program phase (i.e., the start of unique behaviors). However, there still exists a lack of making the explanation of the classification understandable. In other words, to have a representation of a phase that explains why such a slice of the program is indeed a phase — a unique behavior (state) of the program. This representation should answer the question: what are the key factors and relationships that characterize each phase?

This representation can benefit other tasks in which the location of each program phase alone is not enough information. Knowing "the why" (e.g., phase's requirements) can better help, for instance, hardware resource adaptability, the study of application balance in benchmarks, and design space exploration. Another advantage is that we can learn previously unknown properties of a workload runtime behavior. Finally, by knowing only the resulting segmentation, it is often hard to check if the metric contributions to the resulting classification are aligned with the architectural metric of interest.

Although the strong correlation between executed code and performance [Sherwood et al. 2002] has been largely exploited for phase classification, program code structures lack interpretability. Code independent signatures have also been used for phase classification. However, they typically rely on distance-based methods, which are not very suitable for discovering interpretable structures. To reduce the signature dimensionality, some works employ PCA [Eeckhout et al. 2005], which complicates the understandability of the lower-dimensional workload space, and genetic algorithm [Hoste and Eeckhout 2007b], which may lead to a suboptimal set and do not explain their interactions.

Another drawback of conventional phase analysis is that they typically divide a program's execution into fixed-length intervals and group similar intervals together with clustering techniques. Such an approach can result in a phase classification that is out of sync with actual periodic behavior of the program, as they do not account for the change in periodicity of program behavior throughout the execution [Lau et al. 2005].

Some solutions to this phase behavior dissonance problem focus on finding phase transitions that match the procedure call and loop structure of programs [Lau et al. 2006]. However, phase classification using program code structures lacks interpretability. Other approaches have been proven useful to a single metric such as data reuse distance (locality phases) [Shen et al. 2004], however, they explore a priori knowledge of memory access patterns of the applications.

The goal of having interpretable phases and a segmentation sync with the natural phase behavior requires simultaneous segmentation and clustering of the program's execution. We tackle these problems by using a recently proposed method of subsequence clustering of multivariate time series known as TICC [Hallac et al. 2017]. The metrics we use are a large set of microarchitecture-independent metrics (e.g., instruction mix, data and instruction working set sizes, and more), which we refer to as MICA [Hoste and Eeckhout 2007b].

With this method, the execution of a program can be expressed as a timeline of a few hidden states, which correspond to the program behavior pattern of a phase. TICC represents each of these states by a correlation structure, or Markov random field (MRF), which provides information of direct dependencies between variables and, therefore, gives interpretable insights as to precisely what the key factors and relationships are that characterize each cluster [Hallac et al. 2017].

We evaluate the effectiveness of our phase classification by visually analyzing the resulting segmentation for the SPECint 2006 benchmarks. As for the interpretability results, we use network analytics over the MRF representation. More specifically, we used a centrality measure to identify the most important characteristics within a program phase.

## 1.1. Contributions

The main contributions of this paper are:

| Author | Metric | Interval | Classification | Phase Behavior |
|---|---|---|---|---|
| [Sherwood et al. 2003] | Basic Block Vector | Fixed | $k$-means | Uniform |
| [Eeckhout et al. 2005] | Set of program characteristics | Fixed | $k$-means | Uniform |
| [Shen et al. 2004] | Data reuse distance | Fixed | Wavelets & Sequitur | Repeating |
| [Lau et al. 2006] | Basic Block Vector | Loop & Procedure | Call-loop graph traversal | Repeating |
| [Zhang et al. 2015] | Basic Block Vector | Loop & Procedure | $k$-means | Repeating & Uniform |
| Our approach | Set of program characteristics | Fixed | Time series clustering | Repeating |

**Table 1. Summary of works most related to ours**

- A study on the use of a method of clustering multivariate time series subsequences for characterizing the time-varying behavior of programs. To our knowledge, this is the first work that employs such a strategy.
- The use of a correlation structure for program phases that is highly interpretable. Most techniques transform the original data into a new representation, which may not have an interpretable solution, or find a subset of the original features, which may lead to a suboptimal set.

## 2. Related Work

Different researchers have come up with a large set of approaches to partitioning a program's execution into phases. Most phase analysis techniques follow a similar procedure to find phases: they break down a program's execution into intervals, for each interval collect a signature, and then cluster intervals with similar signature into phases. The foundations for many of the techniques used in today's phase classification were proposed before 2006. These older techniques are still the basis of many modern solutions or are used as a baseline for comparison. We only discuss aspects of related work that relate most to ours. We summarize them in Table 1.

**Metrics.** Several metrics have been used to capture phase behavior. These include basic block vectors (BBV) [Sherwood et al. 2002], reuse distance [Shen et al. 2004], and a collection of program characteristics such as instruction mix, register dependency distance, and memory access patterns [Eeckhout et al. 2005].

**Interval**. Most techniques observe phase behavior by looking at fixed-length intervals [Sherwood et al. 2002, Eeckhout et al. 2005]. Some works find phase limits that match the procedure call and loop structure of programs [Lau et al. 2005, Lau et al. 2006].

**Classification.** Given a representation of a program's execution, this aspect concerns partitioning this representation into phases. Common clustering algorithms include $k$-means [Sherwood et al. 2002] and signal processing techniques such as wavelet analysis [Shen et al. 2004] and Sequitur [Shen et al. 2004, Lau et al. 2005].

**Phase Behavior**. Two popular definitions of a program phase are: a unit of a stable (uniform) behavior [Sherwood et al. 2003]; and a unit of repeating behavior [Shen et al. 2004].

## 3. Microarchitecture-independent characterization (MICA)

[Hoste and Eeckhout 2007b] proposed a large set of characteristics to characterize applications in a microarchitecture-independent manner. This work is known as "Microarchitecture-Independent Characterization of Applications" — or MICA, for short. Table 2 summarizes these characteristics. A more detailed description of MICA

| Category | Description | Measurement |
|---|---|---|
| Instruction mix | Categories of instructions | 12 percentages |
| Instruction-level Parallelism (ILP) | IPC achievable for an idealized out-of-order processor assuming perfect caches, perfect branch prediction, etc. Four different instruction window sizes of 32, 64, 128, 256 are measured. | 4 values |
| Register traffic characteristics | Average number of register input operands per instruction, average degree of use (number of times a register is used by other instructions), and distribution of register dependency distance (number of instructions between the production and consumption of a register instance). | 2 values and 7 probabilities |
| Branch predictability | Branch predictability is characterized using the Prediction by Partial Matching (PPM) predictor. Additionally, the branch taken rate and branch transition rate are also being measured. | 2 values and 12 percentages |
| Working set | The number of unique 64-byte blocks and 4KB memory touched for both instruction and data accesses. | 4 numbers |
| Data stream strides | Distribution of global and local strides for loads and stores. Local stride is the difference in the memory address of two consecutive memory access by the same static instruction; Global stride is the difference in the memory address of any consecutive memory access. | 28 probabilities |
| Memory reuse distances | Distribution of reuse distances (the number of distinct memory locations accessed between two memory references to the same location). | 20 probabilities |
| Total | | 91 measures |

**Table 2. Summary of MICA characteristics [Hoste and Eeckhout 2007b, Eeckhout et al. 2005]**

is presented in [Hoste and Eeckhout 2007b]. It is worth noting that MICA is not instruction set architecture (ISA) or compiler independent.

The initial goal of MICA was to study workload balance [Eeckhout et al. 2002], and later also used for other applications. More specifically, [Eeckhout et al. 2005] used these characteristics for discovering phase behavior, however, their work relies on a distance-based classification focusing on matching the raw values of MICA. In this work, we analyze the time-varying behavior of these characteristics to find repetitive patterns.

## 4. Toeplitz Inverse Covariance-based Clustering

Our phase analysis is based on a method of multivariate time series data clustering. We view a program's execution as an ordered sequence of inherent program observations (MICA). Such data is interpreted as a multivariate time series, where each variable is an inherent program feature. We partition this time series into a sequence of program phases, where each phase is a unique state (behavior) of the program and may repeat itself across the execution. To achieve that, we use a method called Toeplitz Inverse Covariance-based Clustering (TICC)[Hallac et al. 2017].

In TICC's terminology, a time series is segmented into a sequence of clusters (which we interpret as a program phase). Each cluster has a signature described by a Markov random field (MRF). This MRF characterizes the conditional dependence structure between different variables of the time series inside a short window (subsequence) of observations, instead of considering each observation in isolation. Such an approach of clustering, based on the graphical dependency structure of each subsequence, makes TICC different from traditional clustering methods, which typically rely on distance-based metrics.

The MRF is viewed as a multi-layer network, with one layer for each observation (time step) of the subsequence. Any window of observations inside a cluster will have the same MRF signature, no matter which subsequence of a cluster we are looking at — it is a time-invariant correlation structure. We use Figure 1 to illustrate the basic intuition of TICC. It shows a time series segmented into clusters $A$, $B$, and $C$. A three-layer MRF characterizes each cluster, each having a different dependency structure. Any subsequence of size 3 inside a segment will have the same MRF, as illustrated by the two

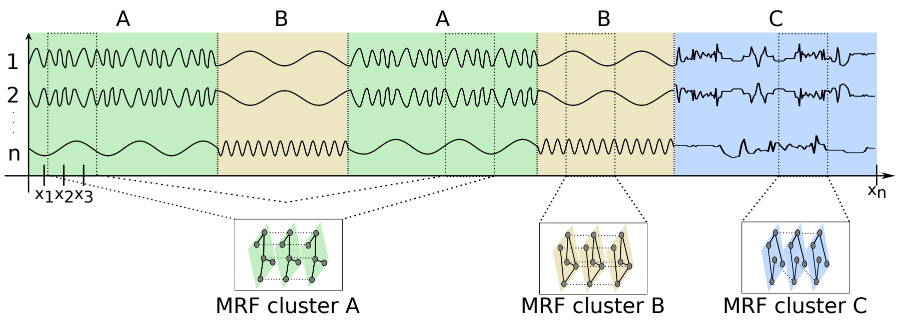


**Figure 1. Overview of the TICC method segmentation**

highlighted subsequences of cluster $A$: both have the same MRF, even though they start at a different position.

TICC learns this MRF representation by estimating a sparse Gaussian inverse covariance matrix $\Theta = \Sigma^{-1}$, which defines a graph structure defining the adjacency matrix of the MRF dependency network. The time-invariant correlation is learned by imposing a block Toeplitz matrix structure on the inverse covariance matrix.

TICC requires the number of clusters to be specified beforehand. The method has three other parameters: $w$, which determines the subsequence size, $\lambda$, the sparsity level in the MRFs characterizing each cluster, and $\beta$, the smoothness penalty that encourages adjacent subsequences to be assigned to the same cluster [Hallac et al. 2017]. A detailed description of TICC's algorithm is presented in [Hallac et al. 2017].

## 5. Phase Classification

In Section 3 we described the input necessary to our phase analysis: a set of microarchitecture independent characteristics dynamically extracted from each program's execution, which we refer to as MICA [Hoste and Eeckhout 2007b]; in Section 4 we presented an overview of the TICC method, which we used to segment a program's execution into phases. We now describe the relationship between MICA and TICC.

[Eeckhout et al. 2005] showed that there exists a strong correlation between MICA and overall performance. We used these inherent program characteristics to reveal patterns in the program's execution. Thus, the core idea of this work is to find phase behavior by automatically finding these recurring patterns using TICC. The resulting classification follows the phase definition by [Shen et al. 2004] which states that "a phase is a unit of repeating behavior rather than a unit of uniform behavior". Additionally, each pattern, or program phase, is defined by an MRF, which provides an interpretable structure.

Figure 2 summarizes the main steps of our method for *gcc* with input *166* from the SPEC 2006 benchmark suite. For visualization purposes, we use PCA to reduce the dimensionality of MICA (explained variance greater than 90%). Different colors indicate different patterns automatically found by TICC.

### 5.1. MICA as Time Series

We profile MICA in terms of instructions executed. We use SimPoint's [Sherwood et al. 2003] model of breaking a program's execution into a set of contiguous non-overlapping intervals. We use interval size at the granularity of 160M instructions. The interval size defines the sampling period of our time series. The MICA

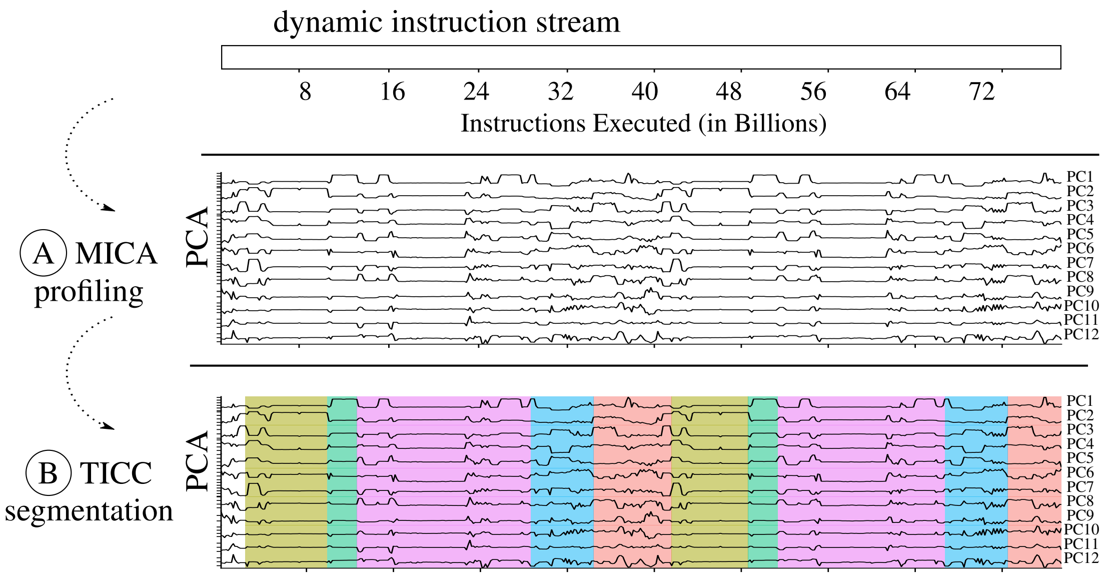


**Figure 2. Coarse-grained phases for *gcc* with input *166* (403.gcc.1)**

characteristics are collected for each interval of execution, and the state of each MICA feature is reset at the end of each interval.

The MICA profiler we used [Hoste and Eeckhout 2007a] outputs absolute values. A further offline conversion is needed to produce the proportional metrics stated in Table 2. For instance, the profiler outputs the total branch count, transition count, and taken count, which are used to produce a branch taken rate and branch transition rate. Empirically, we discovered that running TICC on the raw values outputted by the MICA profiler produced better results when compared to the converted MICA features. The raw MICA features sum up to 97, while the converted metrics, 91.

## 6. Experimental Setup

We evaluated our proposal using the SPEC 2006int benchmarks with reference inputs (34 program-input pairs). The MICA features were sampled at every 160 million instructions. We standardized the features by removing the mean and scaling to unit variance (mean value 0 and standard deviation of 1). For our experiments with PCA, we retained the principal components which explained 90% of the original data set's total variance.

We used the Pin dynamic binary instrumentation system [Luk et al. 2005] to profile the application. A Pin tool for collecting MICA is available in [Hoste and Eeckhout 2007a]. We used the Sniper simulator [Carlson et al. 2011] (Pin-based cycle-accurate x86 simulator) to extract the performance metrics. The baseline micro-architectural model used was *nehalem-lite*, which is included along with the Sniper simulator. This baseline allowed us to sample the metric we needed for every interval of execution so that we could efficiently navigate the search space of TICC parameters.

## 7. Program Phases Plots

Our initial evaluation was to visually inspect the effectiveness of using TICC for discovering program phases. In this study, because we were not interested in the MRF dependency network, but rather the segmentation of the time series, we used PCA to reduce the dimensionality of MICA in order to speed up the processing time of TICC.

We analyzed the resulting segmentation for several TICC configurations for all SPECint program/input pairs. In this paper, we only show the segmentation for a subset of the SPECint programs and a single TICC configuration: window size of 8, $\beta = 2000$, and $\lambda = 0.3$. We made available online[1] the segmentation plots for the multiple TICC parameters (27 configurations for each number of clusters $\in [2, 25]$) and all SPECint program/input pairs.

Due to space constraints, Figure 3 shows the time-varying graphs for only 8 programs and multiple architectural metrics: number of cycles, branch miss rate, L1-D, L2, and L3 cache miss rate. We fix the number of clusters to 5 and look at the resulting segmentation. However, the exact number of clusters will often depend on the application itself. The color represents the cluster assignment from the TICC clustering with the MICA profiling as input.

Overall, these graphs show that TICC is able to find a phase as a unit of repeating behavior rather than a unit of uniform behavior. The method can robustly segment the execution when the phase behavior period length is not stable over time. Finally, we can also notice a correlation between MICA characteristics and overall performance.

### 7.1. TICC Segmentation Applied to Sampled Simulation

We evaluated the use of TICC for sampled simulation. Our solution consisted of a two-level sampling strategy of the MICA characteristics. The first level is the segmentation given by TICC; the second level further breaks down each phase found by TICC into smaller phases using $k$-means. Finally, we simulate a single representative interval from each cluster outputted by $k$-means to arrive at an estimated metric (weighting the performance by the size of each cluster). We found that, on average, this sampling approach achieved comparable accuracy in phase classification to state-of-art SimPoint [Sherwood et al. 2002] or clustering MICA signatures over the entire program execution [Eeckhout et al. 2005] (single-level approach). Thus, we achieved state-of-the-art precision with a much more interpretable signature (MRF), in addition to be closely aligned with the behavior of a program.

## 8. Program Phases Interpretability

In this section, we demonstrate how TICC can be used to learn interpretable program phases in real-world applications. We run this analysis on all the 97 MICA characteristics and use a centrality measure to show how important each MICA characteristic is, and how much it directly affects the other MICA characteristics. This analysis is done over the MRF network defining each program phase. In short, the betweenness centrality (BC) score of a node $v$ is calculated as the fraction of all the shortest paths between all node pairs that pass through $v$ [Newman 2003].

Table 3 shows the betweenness centrality score of each MICA feature of two different programs from SPEC 2006 *401.bzip2.2* and *403.gcc.1*. We normalize each cluster centrality score to [0,10] range. We colored each cell based on the betweenness centrality of each feature from least (white) to greatest (black). Figure 4 shows the time-varying of some architectural metrics colored accordingly to the resulting TICC segmentation. The

[1]`http://students.ic.unicamp.br/~ra191069/mica/`

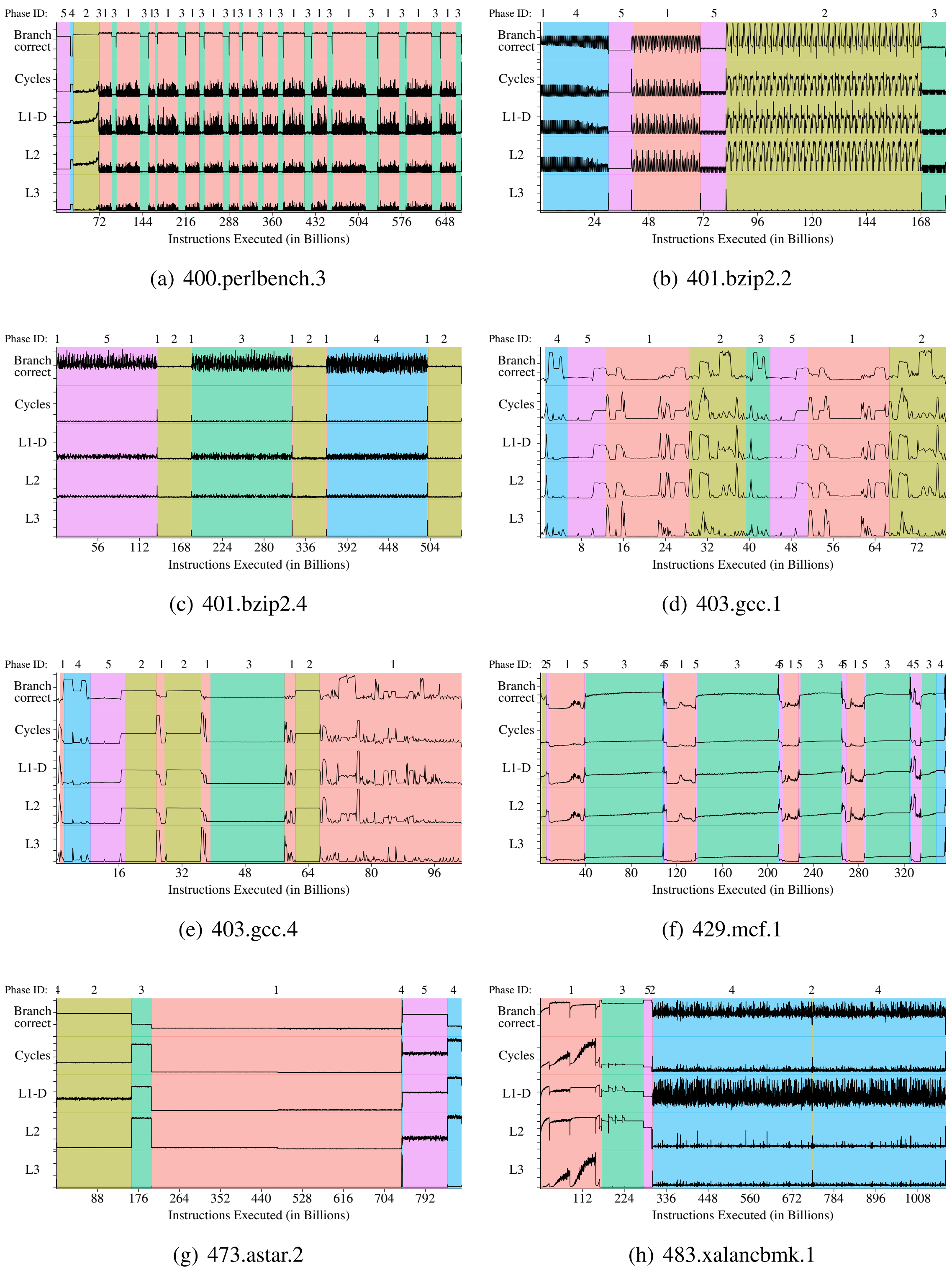


**Figure 3. Plot of metrics segmented according to TICC with MICA as input. The colors represent program phases, each associated with a phase ID.**

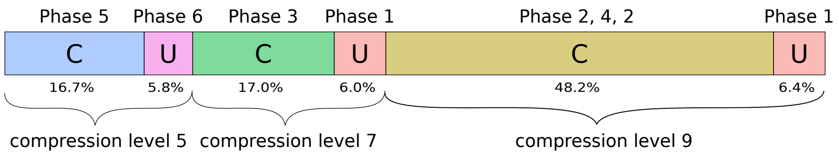
numbers on top (Phase ID) refer to the phase identifiers of each column in the betweenness centrality score tables. For example, Phase 3 of *403.gcc.1* in Table 3 is the green one in Figure 4(b). The parameters used for TICC are window size of 4, $\beta = 4000$, and $\lambda = 1$.

We see that each program phase has a unique characterization (betweenness centrality signature) representing its behavior of the program. We also see that each microarchitecture-independent characteristic influences each phase differently.

## 8.1. Case study: *bzip2*

The *bzip2* file compressor works with data in blocks of size between 100 kB and 900 kB. Block size acts as compression level (1 to 9) with 1 giving the lowest compression and 9 the highest. SPEC2006's *bzip2* compresses and decompresses the data three times, at compression levels 5, 7, and 9, with the result of the process being compared to the original data after each decompression step.

We used Valgrid with its internal tool Callgrind to produce a call-graph for the complete execution of *bzip2* with input *chicken*. As expected, the `main` function has three calls to both `compressStream` and `uncompressStream` (a pair for each compression level). Callgrind also allows dumping counters at enter/leave of specified functions. We used it to dump counters after each compress/uncompress round to produce a separate call-graph for each iteration. The instruction counts of each iteration align with the phase segmentation in a way that each phase represents a compression or decompression operation. Thus we have the following relationship with the source code:



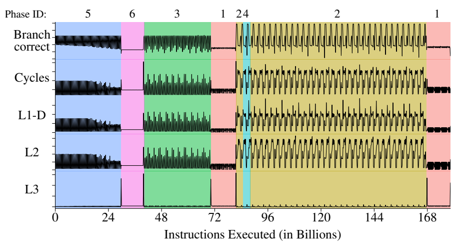
where (C) is a compression step and (U) an uncompress step. The diagram above follows the same color scheme as in Figure 4(a) (i.e., blue for Phase 5, purple for Phase 6, and so forth).

These graphs show that TICC was able to produce regions that align with procedures and loops, each alternating between the compression and decompression steps. We can also observe that each phase has unique centrality scores representing its behavior of the program. To complete the analysis, we make the following observations:

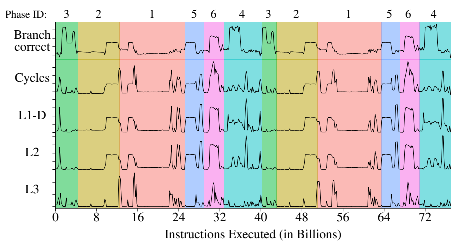
- Both decompress Phases 6 and 1 have a non-zero score on the instruction footprint characteristic. However, for Phase 1, the bin $[2^{16}, 2^{17})$ in the reuse distance distribution has much greater importance. This change in locality might explain a poorer performance to Phase 1 for the particular simulation model.
- Among the compress phases (2, 3, 4, and 5), Phase 3 has the largest score in the stack usage characteristic. When looking at the proportion of execution of the `mainGtU` function, the compression level 7 (Phase 3) has 41M calls to it, against 1M calls at compression level 9 (Phases 2 and 4). Additionally, it is negligible (0.85% relative cost to the complete compress/uncompress step) at compression level 5 (Phase 5). This difference in the number of function calls might be related to the importance given to the push/pop instruction mix feature in Phase 3 (e.g., push/pop into stack-frame).

| | 401.bzip2.2 | | | | | | 403.gcc.1 | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MICA feature | Phase 1 | Phase 2 | Phase 3 | Phase 4 | Phase 5 | Phase 6 | Phase 1 | Phase 2 | Phase 3 | Phase 4 | Phase 5 | Phase 6 |
| ILP 32-entry window | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ILP 64-entry window | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 |
| ILP 128-entry window | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 |
| ILP 256-entry window | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| mem-read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem-write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| control-flow | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 3 | 0 | 0 |
| arithmetic | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 |
| floating-point | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pop/push instructions (stack usage) | 0 | 0 | 10 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 10 | 0 |
| shift instructions (bitwise) | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| string | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| MMX/SSE instructions | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 |
| system | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| nop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| other | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 |
| GAg PPM predictor (4 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAg PPM predictor (4 bits) | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GAs PPM predictor (4 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAs PPM predictor (4 bits) | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GAg PPM predictor (8 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAg PPM predictor (8 bits) | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GAs PPM predictor (8 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAs PPM predictor (8 bits) | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GAg PPM predictor (12 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAg PPM predictor (12 bits) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| GAs PPM predictor (12 bits) | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| PAs PPM predictor (12 bits) | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Branch count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| Branch transition | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Branch taken count | 0 | 7 | 0 | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Number of reg. operands | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reg. instances created | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 |
| Reg. instance uses | 0 | 2 | 0 | 3 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Total reg. dependency distance | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^2$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Reg. dependency distance $\leq 2^3$ | 0 | 4 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^4$ | 0 | 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^5$ | 0 | 10 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reg. dependency distance $\leq 2^6$ | 0 | 5 | 0 | 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Instruction Footprint 64-byte | 6 | 0 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 |
| Instruction Footprint 4kB | 0 | 0 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 0 | 0 | 0 |
| Data Footprint 64-byte | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 10 | 0 | 0 |
| Data Footprint 4kB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 4 | 0 | 0 |
| Total number of memory accesses | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[0, 2^1)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Reuse distance $[2^1, 2^2)$ | 0 | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^2, 2^3)$ | 0 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 7 | 0 | 0 | 0 |
| Reuse distance $[2^3, 2^4)$ | 0 | 0 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^4, 2^5)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^5, 2^6)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 0 |
| Reuse distance $[2^6, 2^7)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^7, 2^8)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^8, 2^9)$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^9, 2^{10})$ | 0 | 6 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^{10}, 2^{11})$ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^{11}, 2^{12})$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Reuse distance $[2^{12}, 2^{13})$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 0 | 0 |
| Reuse distance $[2^{13}, 2^{14})$ | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^{14}, 2^{15})$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Reuse distance $[2^{15}, 2^{16})$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^{16}, 2^{17})$ | 10 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 |
| Reuse distance $[2^{17}, 2^{18})$ | 0 | 1 | 0 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 10 |
| Reuse distance $[2^{18}, 2^{19})$ | 0 | 0 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 |
| Reuse distance $[2^{19}, \infty)$ | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 |
| Memory read count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^1$ | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^2$ | 0 | 3 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^3$ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^4$ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^5$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local load stride $\leq 8^6$ | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global load stride $\leq 8^0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global load stride $\leq 8^1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global load stride $\leq 8^2$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Global load stride $\leq 8^3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Global load stride $\leq 8^4$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Global load stride $\leq 8^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Global load stride $\leq 8^6$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| Memory write count | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^0$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Local store stride $\leq 8^1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^2$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^4$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Local store stride $\leq 8^6$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global store stride $\leq 8^0$ | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 10 | 0 | 0 | 0 |
| Global store stride $\leq 8^1$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 7 | 0 |
| Global store stride $\leq 8^2$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global store stride $\leq 8^3$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global store stride $\leq 8^4$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Global store stride $\leq 8^5$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 3. Betweenness centrality for each MICA feature**



(a) 401.bzip2.2



(b) 403.gcc.1

**Figure 4. Time-varying graphs with phase identifier on top**

- Phase 2 and 4 have a similar signature, as well as the performance metrics. The segmentation into two phases might be related to a poor value given to the TICC's temporal consistency penalty ($\beta$) that encourages neighboring samples to be assigned to the same cluster [Hallac et al. 2017].
- At compression level 9 (Phases 2 and 4), a large amount of time (43.50%) is spent in function `fallbackSort`. *bzip2* uses a fallback sort when the main sort takes too much time. The source code states that it is "kind-of an exponential radix sort". Considering that radix sort tends to exhibit poor cache locality [LaMarca and Ladner 1999], it might explain the relatively poor cache performance to the simulation results. It is interesting to observe the centrality scores between the phases using only the main sort (Phases 5 and 3) and fallback sort (Phases 2 and 4).

## 9. Conclusion

This paper presented a method for characterizing time-varying program behavior by reducing the phase classification problem to a subsequence time series clustering problem. Our goal was to provide interpretable insights on the key factors and relationships that characterize each program phase. Additionally, we aimed at having a classification that is sync with the actual periodic behavior of the program.

We observed program phases via a set of important microarchitecture-independent characteristics [Hoste and Eeckhout 2007b]. We used a set of 97 characteristics, each falling into one of 7 possible categories: ILP, instruction mix, branch predictability, register traffic, memory footprint, memory reuse distance, and data stream.

Our first step in this work was to visually analyze TICC's segmentation for multiple configurations. We analyzed the segmentation of all SPECint 2006 programs for a range of up to 25 clusters for each program. Overall, the plots indicate a strong correlation between MICA and performance metrics, and also a good segmentation. In most cases, for larger values of clusters, TICC could not converge to a solution.

We also showed that TICC can be used to find meaningful insights into program phases through the analysis of the MRF network it produces. In particular, we used a measure of centrality in a graph to show, for each program phase, how significant each MICA characteristic is for a program phase, and how much it affects the other characteristics. We were able to have a much cleaner interpretation of high-dimensional data.

## 10. Acknowledgments

This work is supported by the Sao Paulo Research Foundation (FAPESP) (2013/08293-7), CAPES (2013/08293-7), and CNPq (438445/2018-0, 309794/2017-0).

## References

Carlson, T. E., Heirman, W., and Eeckhout, L. (2011). Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 52:1–52:12, New York, NY, USA. ACM.

Eeckhout, L., Sampson, J., and Calder, B. (2005). Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 2–12.

Eeckhout, L., Vandierendonck, H., and Bosschere, K. D. (2002). Workload design: Selecting representative program-input pairs. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, PACT '02, pages 83–94, Washington, DC, USA. IEEE Computer Society.

Hallac, D., Vare, S., Boyd, S., and Leskovec, J. (2017). Toeplitz inverse covariance-based clustering of multivariate time series data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 215–223, New York, NY, USA. ACM.

Hoste, K. and Eeckhout, L. (2007a). MICA. `https://github.com/boegel/MICA`.

Hoste, K. and Eeckhout, L. (2007b). Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72.

Isci, C. and Martonosi, M. (2006). Phase characterization for power: evaluating control-flow-based and event-counter-based techniques. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 121–132.

LaMarca, A. and Ladner, R. E. (1999). The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66 – 104.

Lau, J., Perelman, E., and Calder, B. (2006). Selecting software phase markers with code structure analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, Washington, DC, USA. IEEE Computer Society.

Lau, J., Perelman, E., Hamerly, G., Sherwood, T., and Calder, B. (2005). Motivation for variable length intervals and hierarchical phase behavior. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ISPASS '05, pages 135–146, Washington, DC, USA. IEEE Computer Society.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200.

Newman, M. E. J. (2003). A measure of betweenness centrality based on random walks. *arXiv e-prints*, pages cond–mat/0309045.

Shen, X., Zhong, Y., and Ding, C. (2004). Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 165–176, New York, NY, USA. ACM.

Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. (2002). Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57.

Sherwood, T., Perelman, E., Hamerly, G., Sair, S., and Calder, B. (2003). Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93.

Zhang, W., Li, J., Li, Y., and Chen, H. (2015). Multilevel phase analysis. *ACM Trans. Embed. Comput. Syst.*, 14(2):31:1–31:29.