Universidade Estadual de Campinas
Instituto de Computação

Rafael Mendonça Soares

Discovering Phase Behavior Through Time-Varying Microarchitecture Independent Characteristics Clustering

Descobrindo o Comportamento de Fases Através do Agrupamento de Características Independentes de Microarquitetura Variantes no Tempo

CAMPINAS
2020

Rafael Mendonça Soares

Discovering Phase Behavior Through Time-Varying Microarchitecture Independent Characteristics Clustering

Descobrindo o Comportamento de Fases Através do Agrupamento de Características Independentes de Microarquitetura Variantes no Tempo

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Dissertation presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Rodolfo Jardim de Azevedo

Este exemplar corresponde à versão final da Dissertação defendida por Rafael Mendonça Soares e orientada pelo Prof. Dr. Rodolfo Jardim de Azevedo.

CAMPINAS
2020

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

So11d

Soares, Rafael Mendonça, 1992-
Discovering phase behavior through time-varying microarchitecture independent characteristics clustering / Rafael Mendonça Soares. – Campinas, SP : [s.n.], 2020.

Orientador: Rodolfo Jardim de Azevedo.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Arquitetura de computador. 2. Avaliação de desempenho. I. Azevedo, Rodolfo Jardim de, 1974-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Descobrindo o comportamento de fases através do agrupamento de características independentes de microarquitetura variantes no tempo
Palavras-chave em inglês:
Computer architecture
Performance evaluation
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Rodolfo Jardim de Azevedo [Orientador]
Edson Borin
Luiz Cláudio Villar dos Santos
Data de defesa: 29-06-2020
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-9498-7405
- Currículo Lattes do autor: http://lattes.cnpq.br/9428121514232352

Universidade Estadual de Campinas
Instituto de Computação

Rafael Mendonça Soares

Discovering Phase Behavior Through Time-Varying Microarchitecture Independent Characteristics Clustering

Descobrindo o Comportamento de Fases Através do Agrupamento de Características Independentes de Microarquitetura Variantes no Tempo

Banca Examinadora:

- Prof. Dr. Rodolfo Jardim de Azevedo
  IC/UNICAMP
- Prof. Dr. Edson Borin
  IC/UNICAMP
- Prof. Dr. Luiz Cláudio Villar dos Santos
  INE/UFSC

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 29 de junho de 2020

Agradecimentos

A Claudia, Paulo e Eduardo. Vocês são a base de tudo em minha vida. Vocês me deram as melhores condições para iniciar o meu entendimento sobre a computação. A minha gratidão e admiração por vocês são sem limites.

Aos meus amigos de Campinas, em especial, ao meu grande amigo Tiago, que dividiu comigo a moradia durante grande parte deste mestrado.

À minha amada namorada Thais, que sempre me apoiou em todos os momentos dessa jornada.

Aos meus colegas de laboratório, pelos debates técnicos e momentos de descontração. Em especial, agradeço àqueles que cuidaram e cuidam voluntariamente da infraestrutura do laboratório.

Ao CNPq, por prover o suporte financeiro que possibilitou este estudo.

Ao meu orientador Rodolfo, por todas as nossas conversas e por contribuir imensamente ao aprimoramento do meu pensamento científico.

Resumo

A análise de fases provou-se uma técnica eficiente para diminuir o tempo necessário para executar simulações detalhadas de microarquitetura. O objetivo deste estudo é solucionar duas dificuldades do estado da arte: (i) a maioria das abordagens feitas na análise de fases adota uma estratégia de granularidade fina, que em alguns casos pode ser interferida por ruídos temporários e não levar em conta um contexto mais amplo; e (ii) a interpretação da assinatura de cada fase de programa é uma tarefa difícil, dado que muitas vezes são empregadas assinaturas de alta dimensão.

Para a problemática (i) adotamos a análise de fases de programas em dois níveis, cada qual com uma granularidade diferente (nível 1 – método de agrupamento de subsequências de séries temporais multivariadas; nível 2 – k-means). No entanto, concluímos que essa abordagem alcançou uma precisão comparável aos trabalhos anteriores. Chegamos então ao estado da arte de forma alternativa, mas com a vantagem de trazer subsídios para uma potencial solução para a problemática (ii), pois com o método empregado, as fases passaram a ter uma assinatura (MRF) muito mais interpretável, além de alinhada ao comportamento dos programas. Demonstramos a eficácia dessa interpretação usando uma medida de centralidade para identificar as principais características de uma fase de programa, contribuindo assim para o uso dessas assinaturas de fases em estudos posteriores.

Abstract

Phase analysis has been shown to be an efficient technique to decrease the time needed to execute detailed micro-architectural simulations. Our study aimed to overcome two limitations of current methods that can be defined as follows: (i) most approaches adopt a fine-grained strategy, which in some cases can be interfered with temporary noises and do not account for a broader context; and (ii) interpreting the resulting program phases is often difficult since it is hard to draw meaningful conclusions from high-dimensional phase signatures.

Regarding (i), we adopted a two-level phase analysis, each with different granularity (level 1 – method of subsequence clustering of multivariate time series; level 2 – k -means). However, we found that, on average, this sampling approach achieved comparable accuracy in phase classification to prior work. Thus, we achieved state-of-the-art precision with a potential solution to the problem (ii), since with the method employed, the phases started to have a much more interpretable signature (MRF), in addition to be closely aligned with the behavior of a program. We demonstrated the effectiveness of such interpretation using a centrality measure to identify the most important characteristics within a program phase.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Performance evaluation in computer architecture research and development is heavily based on simulation. In order to have meaningful insights and conduct research to correct decisions, simulators that model cycle level behavior of processors are often employed, which is extremely time-consuming. As a result, it is often infeasible to navigate the search space with complete benchmark executions.

Sampled simulation is a popular technique for efficient simulation. The key idea of sampled simulation is to infer performance metrics by simulating only a very small fraction (e.g., 1-2%) of a program. The main challenge involved in the use of this technique is to find which portions of code best represent the full execution. A popular choice is based on the fact that programs often exhibit repeating behaviors, which are known as program phases. Figure 1.1 illustrates the basic intuition. It shows the execution of SPEC2006 *bzip2*'s benchmark for a set of performance metrics varying over its execution and two program phases (green and red).

Given such structured behavior, the main idea is to simulate samples of each unique repetitive behavior to arrive at an overall performance metric, instead of running the full execution in a cycle-accurate manner. The set of samples, one or more for each distinct program phase, is known as simulation points and indicate points in the program to start execution at. Only the selected simulation points are simulated in cycle-accurate detail. This can be achieved using checkpoints [54, 66] or fast-forwarding through the remaining execution [67, 64]. Finally, the performance of the complete execution can be predicted by weighting the performance data with the coverage of their phase.

Conventional phase analysis techniques divide a program's execution into fixed-length intervals (e.g., a section of continuous execution), for each interval collect a hardware-independent signature, and then group similar intervals with clustering techniques. The goal is to achieve clusters with intervals that have similar behavior across architecture metrics (e.g., IPC and cache miss), as this will directly impact accuracy (i.e., the error between the predicted value from the partial execution and the actual value).

The time scale at which time-varying program behavior is being observed has a great impact on sampled simulation accuracy. Programs exhibit phase behavior at many different granularities [44]. Some programs exhibit phase behavior at various time scales and, in some cases, they exhibit a hierarchical behavior, i.e., a phase at a time scale that consists of stably distributed fine-grained phases [69], as illustrated by the green phase

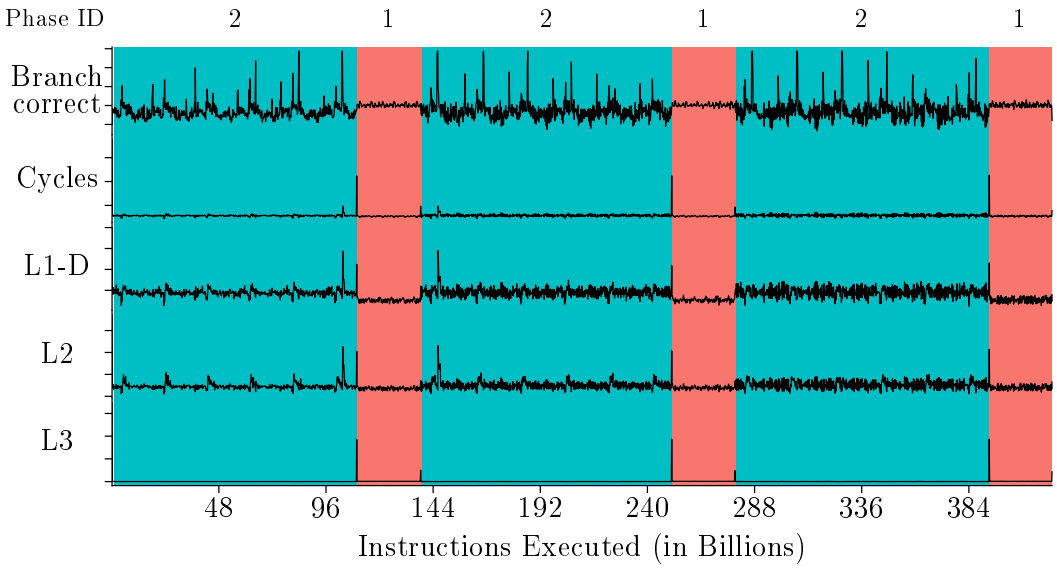


Figure 1.1: Time-varying graph for *bzip2-source*

(Phase ID 2) in Figure 1.1.

Most phase analysis approaches adopt fixed-length intervals, which 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 [44]. Also, most approaches adopt a fine-grained strategy, which in some cases can be interfered with temporary noises and do not account for a broader context [69].

Another limitation of current phase analysis methods is that interpreting the resulting phases is hard, especially when the data is high-dimensional. 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 problems 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 of interpretability 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.

This dissertation has the goal of improving the representativeness of the simulation points and achieving some degree of interpretability of the program phases. We tackle these by treating phase classification as a problem of subsequence clustering of multivariate time series. The metrics we use were originally proposed by Hoste et al. [30] and it is 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. We sample these features per interval of dynamic instructions, which gives us a multivariate time series, where each time-stamped observation consists of readings for a set of microarchitecture-independent metric. This representation is used to discover repeated patterns (or program phases).

We achieve this by segmenting the multivariate data into sequences of states using a recently proposed method by Hallac et al. [27] for multivariate time series clustering called *Toeplitz Inverse Covariance-based Clustering* (TICC). Our goal with this method is to express a program's execution as a timeline of a few hidden states, which correspond to the program behavior pattern of subjects. TICC represents each of these states by a correlation structure, or Markov random field (MRF).

Our solution to achieve well-formed clusters consists 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. We employ k-means clustering applied to the set of intervals of each phase. Each cluster found by k-means is referred to as a fine-grained phase, and it is a set of intervals within a program's execution that have similar MICA, which consequently tends to exhibit similar architectural behavior [15]. Finally, we simulate a single representative from each cluster outputted by k-means to arrive at an estimated metric (weighting the performance by the size of each cluster).

The MICA set of metrics was originally proposed with a focus on finding benchmark similarity [57], later also used for discovering program phases [15]. However, they cluster each observation in isolation, which can still incur in the aforementioned problems of a fine-grained and single granularity strategy, since they are a vector of execution frequencies, and a few elements may have an implicit notion of time within it (i.e., size of stride). Our phase classification also uses MICA as a metric; however, we cluster each observation in the context of a short window of observations in the time series.

Our hypothesis was that the use of TICC as the first clustering strategy (coarse-grained phases) would better capture the overall program behavior and would separate similar MICA signatures that behaved differently into different clusters. However, we found that on average, for an interval size of 160M and a single architecture of study, the phases detected by our approach have a behavior homogeneity comparable to the phases detected by the popular SimPoint clustering algorithm [64] or applying k-means directly to the MICA features [15].

Finally, we achieved phase interpretability using the hidden states inferred from the time series using TICC, as they provide a meaningful interpretation of the raw, high-dimensional MICA data. TICC provides means of having interpretable phases, as an MRF 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 [27].

We demonstrated the effectiveness of such interpretation using network analytics over the MRF representation. More specifically, we used a centrality measure to identify the most important characteristics within a program phase for some SPEC 2006 benchmarks. Another advantage of interpretability is that we can learn previously unknown properties of a workload runtime behavior. We believe that with further research this could unlock

new optimization in both software and hardware design.

1.1 Contributions

The main contributions of this dissertation are:

- A detailed 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 finds a subset of the original features, which may lead to a suboptimal set.

1.2 Organization

This dissertation is organized as follows:

- Chapter 2 describes the basic concepts and related work regarding efficient simulation and phase behavior. It also shows how phase behavior is used for efficient simulation.
- Chapter 3 describes the method of subsequence clustering of multivariate time series we used for segmenting a program's execution profile. It also describes basic concepts on probabilistic graphical models and time series.
- Chapter 4 describes our method of phase analysis for efficient simulation.
- Chapter 5 evaluates the effectiveness of our analysis.
- We draw the dissertation conclusion in Chapter 6.

Chapter 2

Background and Related Work

This chapter is divided into two sections. First, we explain the concept of program phases. Second, we make a brief overview of efficient simulation techniques and show how phase behavior is used for simulation acceleration.

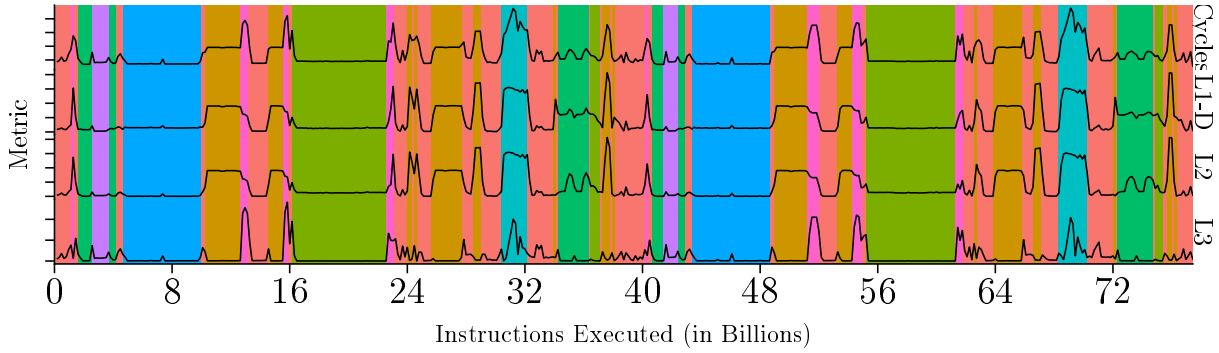
2.1 Phase Behavior Characterization

It is well known that programs exhibit phase behavior [64, 65, 36]. Sheewwood et al. [65] stated that "*the way a program's execution changes over time is not totally random; in fact, it often falls into repeating behaviors, called phases*". Two popular definitions of phases are: (i) a unit of a stable (uniform) behavior [65]; and (ii) a unit of repeating behavior [62]. The goal of phase analysis is to automatically segment a program's execution into a sequence of phases based on some observed features.

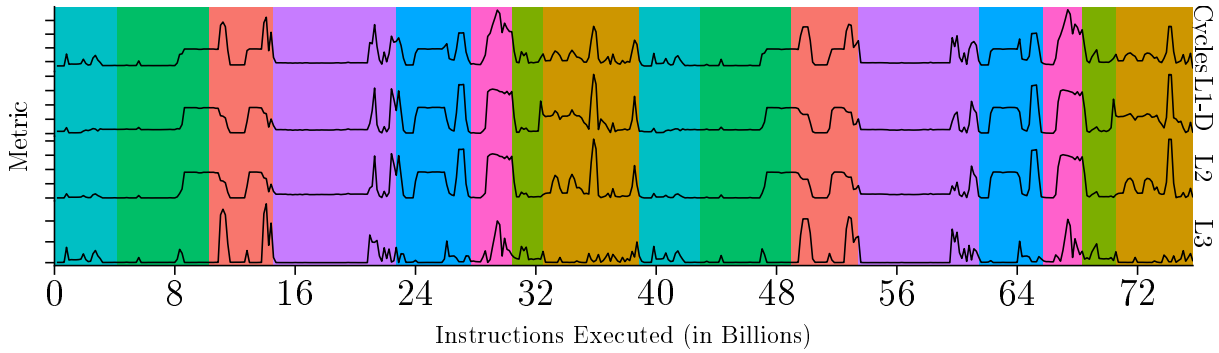
Phase behavior can be observed over many different features of a program. The choice of features can be different for different purposes. For example, it is desirable to have features that are microarchitecture independent for an analysis that will be used across microarchitectures. Phase analysis has been mainly exploited for simulation acceleration [65], power reduction [36, 33, 33], cache optimization [2, 62], and compiler optimization [47].

We use Figure 2.1 to better describe definitions (i) and (ii). It shows the execution of *gcc* with input *166* for a set of three performance metrics varying over the program's execution. Each point on the graph is an average of 160 million instructions. These graphs have been segmented into 10 phases (shown in different colors). In Figure 2.1a the segmentation follows the definition (i), which results in phases with intervals that have similar behavior. In Figure 2.1b each phase represents a recurring behavior, which follows definition (ii). It is possible to notice some intervals that are split into multiple phases in Figure 2.1a and considered a single-phase in Figure 2.1b.

Simpoint [64] is the most well-known technique for phase analysis. Simpoint's general concept of a phase is a period of execution in which the program exhibits a stable behavior, which follows definition (i). To find each period of stable behavior, the dynamic instructions of the program are divided into intervals – instruction sequences of the same length. SimPoint's authors argue that there exists a strong correlation between the paths



(a) Stable behavior phase



(b) Repeating behavior phase

Figure 2.1: Time-varying behavior for *gcc-166*

taken by the program and the expected architectural behavior. Thus, the key idea is to group intervals that execute similar code. This similarity is found by using a structure known as Basic Block Vector (BBV). A BBV shows the frequency of execution of basic blocks, it has one entry for every static basic block in the program. Each set of intervals with similar BBV makes up a phase.

Figure 2.2 gives an overview of the methodology: the upper portion represents the source code that defines an order of N basic blocks, and the lower portion shows the division of the program's execution into intervals with their corresponding phase. Each interval has a BBV signature of size N, such that each position stores the number of times each corresponding basic block was executed.

2.1.1 Related Work

Different researchers have come up with different approaches to partitioning a program's execution into phases. We will discuss related work on phase analysis from three different aspects.

Phase Metrics. Programs exhibit repetitive behavior over different metrics. The representation should be defined according to the purpose of the phase analysis. Phase metrics are usually divided into two main categories: microarchitecture-dependent and microarchitecture-independent characterization.

Balasubramonian et al. [2] uses phase analysis to change cache configurations, in or-

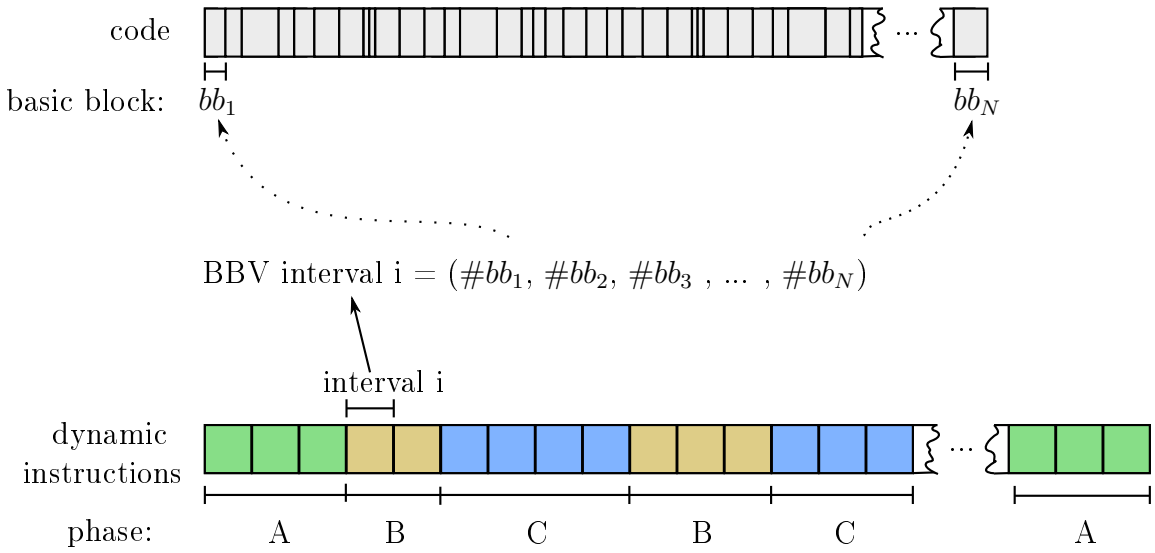


Figure 2.2: Overview of SimPoint profiling

der to improve performance and save power, based on cache miss rates and branch frequencies collected through hardware counters. Also in the context of adaptive systems, Duesterwald et al. [14] uses metrics derived from hardware counter – IPC, data cache miss rates, instruction mix, and branch prediction accuracy. Isci and Martonosi [36] identify program power behavior using the concept of *power vectors*, which represents power values for several processor components of each sampled execution point. Although these microarchitecture-dependent metrics find program phase behavior, they do not allow the resulting phase classification to be used across different hardware designs.

Another approach is to infer phase behavior from microarchitecture-independent metrics. Simpoint [64] views a program's execution as a sequence of Basic Block Vector (BBV). Dhodapkar et al. [12] does the analysis based on the program's working set. Huffmire and Sherwood [34] employ memory addresses. Shen et al. [62] finds patterns on data reuse distances. Eeckhout et al. [15] uses a collection of program characteristics such as instruction mix, register dependency distance, and memory access patterns. Lau et al. [47] represents a program's execution using a hierarchical procedure call and loop graph. Huang et al. [33] examines the procedure calls via a call stack.

Phase Classification. Given a representation of a program's execution, the second aspect concerns how to partitioning this representation into phases. One common way, as previously seen in SimPoint approach, is to break down a program's execution into intervals (usually of fixed length), for each interval collect a signature, and then cluster intervals with similar signature into phases. Common clustering algorithms include k-means [64, 15] and multinomial clustering [59].

Some work finds phase limits that match the procedure call and loop structure of programs. In [47] intervals are aligned with procedure call, return and loop transition boundaries. Each variable-length interval is represented by BBV. In [33] phases are found based on a call stack to apply hardware adaptations at the granularity of subroutines.

Some work relies on signal processing techniques such as Fourier transformation or

wavelet analysis. Shen et al. [62] views a sequence of data reuse as a signal and use wavelet as a filter to expose abrupt changes in the reuse pattern. They partition this filtered trace and find the basic blocks in the code that mark the start of each phase. In [63] a Fourier transform is applied over Basic Block Vectors.

Some works combine clustering algorithms and signal processing techniques. They are based on the aforementioned approach of breaking down a program's execution into intervals, collect a signature for each interval, and cluster these signatures. However, before running a clustering algorithm over each signature, the signature is transformed into the wavelet domain and the wavelet coefficients of each program execution interval are used as the input to the clustering algorithms. Some signature includes a 2D matrix of the memory accesses [34] or vector of performance counters [10].

Dhodapkar and Smith [12] employ a simple threshold rule to indicate a phase change. They define a working set ratio between two intervals of execution. If the ratio is more than some threshold, a working set change (phase change) is registered. If not, the phase is stable.

Profiling Granularity. The last aspect is the granularity at which time-varying program behavior is being observed. Many programs exhibit repeating behavior at different time scales [47]. For example, one can think of a level as an innermost loop body or a coarser level such as an outer-most loop body. Prior work has shown that programs exhibit repeating behavior at a portion of execution of 100K instructions [2, 12], 1M-10M instructions [45, 47], 10-100M instruction [64], or even 100M-1B instructions [53]. Some work observes time-varying program behavior at different time scales at the same time. Zhang et al. [69] observed that coarse-grained intervals belonging to the same phase usually consist of stably distributed fine-grained phases.

2.2 Efficient Simulation

Simulation plays a critical role in computer architecture. Simulators, usually a software tool, are used to estimate processor behaviors. Compared to building hardware prototypes, they allow inexpensively and flexibly future generation hardware design evaluation [16]. For example, one can explore multiple cache organizations (e.g. size, associativity, and block size) by simply changing some of the simulator's parameters. As performance evaluation guides computer architecture research in making decisions in large space design, simulations became a popular evaluation tool.

Simulators are a model of a processor, and as in any model, it is built based on the purpose of its application. Simulators come in different levels of abstraction, each representing a different trade-off in accuracy (when compared to the real hardware), evaluation time, development time, and the fraction of the design space it allows to explore [20]. A popular trade-off in both academia and industry is the cycle-accurate simulation model [6], or detailed simulation, which models the behavior of the microarchitecture and the timing behavior of each instruction.

A commonly used evaluation approach consists of executing a realistic benchmark over cycle-accurate simulators. However, not only do newer computational systems grow more

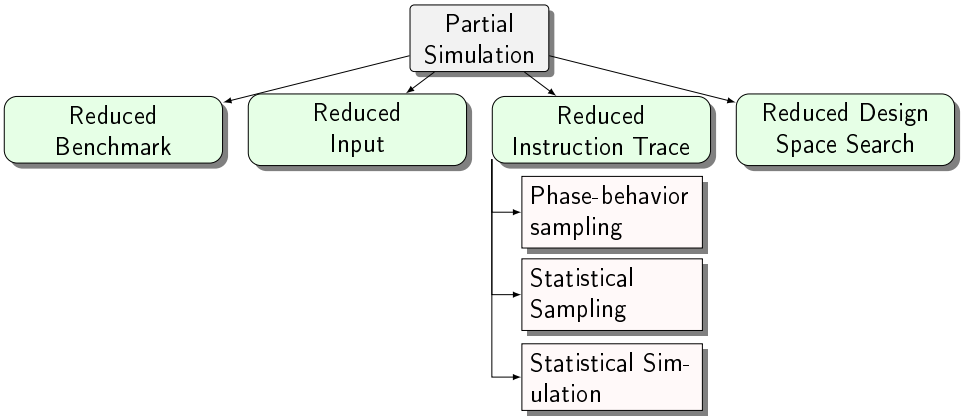


Figure 2.3: Overview of partial simulation approaches

complex each year but also benchmarks grow in complexity and size. These factors have an unsatisfactory effect on designers' productivity when we consider the higher time needed to obtain simulation results. A full cycle-accurate simulation of a modern benchmark suite on some candidate configurations can require months of simulation time. This section gives an overview of several solutions that have been proposed to remedy this problem.

The required simulation time critically depends on the speed of cycle-accurate simulators and the total number of instructions to be simulated (e.g., number of workloads and size of the design space the architect is willing to explore). The former can be mitigated by enhanced simulators and it is out of the scope of this work; the latter by partial simulation.

The next section gives a brief overview of four commonly used techniques for partial simulation, which are summarized in Figure 2.3. The first selects a representative set of benchmarks from a larger set of benchmarks. The second reduces simulation time with reduced input sets size. The third simulates short instruction trace instead of the full instruction trace. The last approach simulates a small number of architectures rather than the whole design space.

2.2.1 Reduced Benchmark

A key aspect to obtain a microprocessor design that is optimal for the target domain of operation is to find a representative workload of that environment of operation. In order to make unbiased decisions, architects pick a broad set of applications. However, due to a limited time for simulation and the aforementioned slow speed of architectural simulations, architects may simulate only a subset of those benchmarks. A bad subset would yield an incorrect conclusion. Ideally, we would like to pick the best representative subset of the target environment that attends time requirements on available simulation time. Several approaches have been proposed to subset a benchmark suite.

Eeckhout et al. [19] uses Principal Components Analysis (PCA) and cluster analysis. They collect a set of p program characteristics of the complete execution for each program-input pair. These characteristics are viewed as a point in a p-dimensional space. Then, they apply PCA to transform the data into a q-dimension space (with $q \ll p$). The transformed space allows better interpretation and has uncorrelated dimensions. Finally,

cluster analysis is used to group similar program-input pairs and a representative program is chosen from each cluster.

The characteristics used in [19] are a collection of microarchitecture-dependent (e.g., instruction mix) and microarchitecture-independent characteristics (e.g., branch prediction accuracy and cache miss rates). However, the microarchitecture-dependent characteristics may incorrect guide decisions of the microprocessor design, as two programs might behave differently on other designs [30]. As an alternative, some works [30, 57] apply the same idea as in [19] but only use a set of important microarchitecture-independent characteristics.

Yi et al. [68] classifies programs based on how they stress the microarchitecture of a design space. Two benchmarks are similar if they stress the same components of the processor to similar degrees. The measurement of how each component is stressed is obtained by Plackett and Burman (PB) design of experiment, which for a design space with N parameters, requires about N simulations, instead of all combinations of configurations. These N microarchitecture configurations are corner cases in the microarchitecture design space [16]. This results in a ranking of performance bottlenecks. The similarity between two benchmarks is measured by the euclidian distance of their ranks.

2.2.2 Reduced Input

Prior section looked into representative benchmarks selection; another approach is to use reduced benchmark inputs. The well known SPEC CPU benchmark is one example of a suite that has multiple inputs for each application. Additionally, each has different sizes of input: *test*, *train*, and *ref*. The *test* is the smallest input set and it is used for testing that an executable is functional, while the *train* gives an intermediate-length run. The *ref* input set is the largest input size and is used to give a complete evaluation of the target environment [41].

MinneSPEC [41] provides reduced input sets for the SPEC CPU 2000 benchmark suite that match the *ref* inputs. The reduced inputs are manually constructed by changing different command-line arguments, truncating the original inputs files or, in few cases, creating new input files. They quantify the difference of the reduced inputs to the SPEC programs when executed with *ref* inputs in terms of function-level execution patterns, instruction mix, and memory behavior [41].

Breughe et al. [4] presents four methods for selecting representative inputs: (i) random selection, (ii) microarchitecture-independent selection, (iii) filtered Selection, and (iv) Min-Median-Max selection. In (i) a number of inputs are selected at random out of the available benchmark inputs; in (ii) BBVs are used to compare similarities among inputs; in (iii) the idea is to explore a very small subspace of a design space and filter out inputs that greatly differed from the optimum design point in terms of Energy-Delay Product (EDP). In (iv), instead of evaluating a complete subspace, they run all inputs on a single design point and pick the inputs that have the minimum, median, and maximum CPI.

2.2.3 Reduced Instruction Trace

This category of approaches concentrates on generating a short instruction trace of a given program-input pair. The first two approaches focus on finding regions within a benchmark's execution that are representative of the entire benchmark execution. They are known as sampled simulation and their core idea is to simulate only small fractions of a program and extrapolate the results for the entire workload execution. Phase-behavior sampling and statistical sampling are the two main ways to select those sampling units. The third approach, known as statistical simulation, finds a short trace by producing a synthetic workload that mimics the behavior of the original program.

Phase-behavior sampling. Phase-behavior sampling uses the techniques described in Section 2.1 to segment a program's execution into phases. In this context, a phase is usually defined as a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency [65]. As a result, it is enough to pick a sampling unit for each unique phase, and then weigh each sampling unit to provide an overall performance number.

SimPoint [65] is the most well-known approach to select those samples. As stated in the previous section, SimPoint divides the dynamic instructions into intervals (e.g., 100M instructions), characterizes each interval by a BBV and cluster these intervals based on its BBV signature. Each cluster, also referred to as a phase, corresponds to a set of intervals with similar behavior. The main idea behind this approach is the strong correlation between code and performance metrics [65]. Simpoint then chooses a representative interval (simulation point) from each phase by finding the interval closest to the cluster's center (centroid). Finally, detailed simulation is performed only at the simulation points and the performance is weighted by the size (number of intervals) in its corresponding cluster. Figure 2.4 gives an overview of the approach.

Zhang et al. [69] proposes a method that combines two levels of granularity for sampling simulation. The first level is coarse-grained and is found by segmenting a program's execution at boundaries of a procedure call and loop structure of programs, with an approach similar to [47]. Each coarse-grained interval is clustered based on its BBV and a single simulation point is chosen for each cluster. Then, they apply a fine-grained sampling to those coarse-grained simulation points. They use 10M instructions as the length for a fine-grained interval and also do a BBV based clustering. The fine-grained simulation points are only used to represent the selected coarse-grained simulation points.

Statistical Sampling. Wunderlich et al. [67] present the SMARTS methodology in which representative intervals are selected based on theories of the statistical sampling field. The advantage of this methodology is to obtain values of performance metrics within a desired confidence interval. A program with N instructions and samples of size M has a total of $\frac{N}{M}$ samples. According to the sampling theory, the confidence interval can be calculated based on the number of samples collected from the total $\frac{N}{M}$ [24]. In addition, if the simulated samples do not meet the confidence interval, more samples can be selected until the appropriate level is reached. In SMARTS, samples are selected periodically during program execution, which is a good approximation for the simple random sampling applied to the microarchitecture simulation field [67].

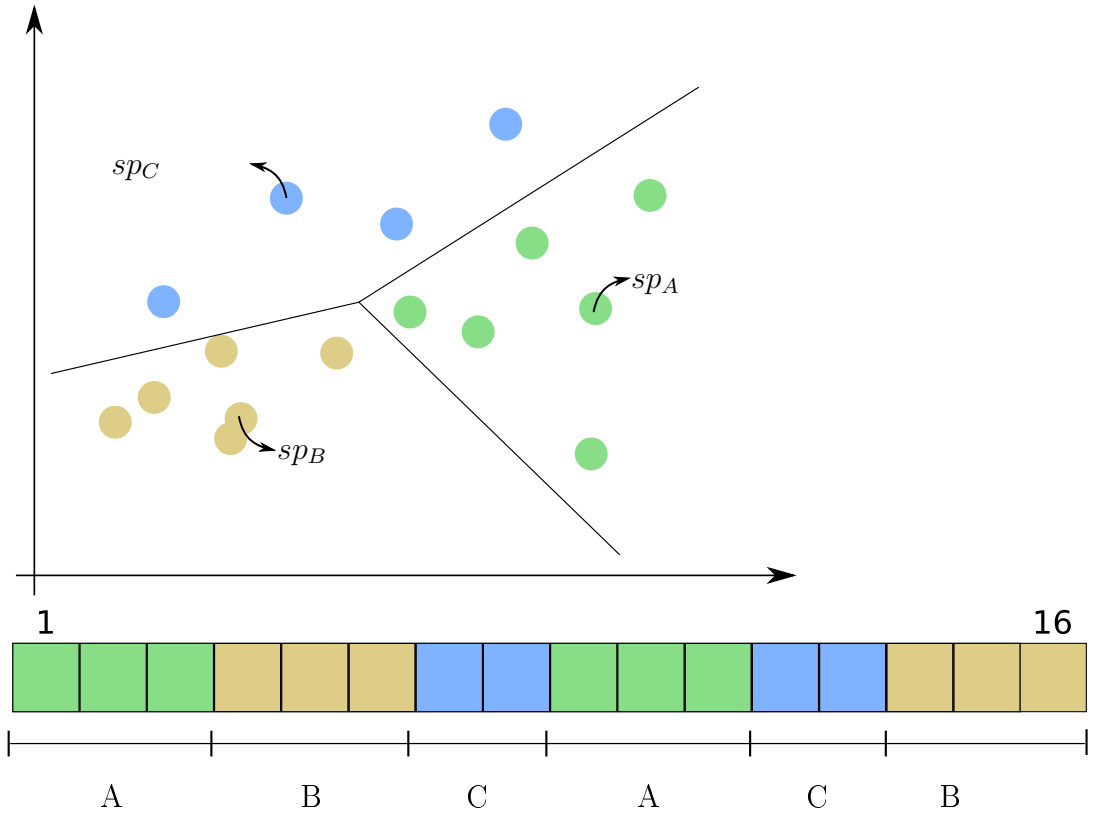


Figure 2.4: Overview of SimPoint clustering

Statistical simulation. Statistical simulation produces a small synthetic workload that is representative for long-running benchmarks based on the statistical profiles of the complete execution [17, 18]. The statistical profile consists of a set of characteristics that are independent of the microarchitecture (e.g., instruction mix, instruction dependencies, control flow behavior) along with a set of microarchitecture-dependent characteristics (typically locality events such as cache and branch prediction miss rates) [16]. With this statistical profile, a synthetic trace with the same statistical properties is generated. Finally, this synthetic trace is simulated on a trace-driven statistical simulator to obtain metric values of interest.

2.2.4 Reduced Design Space Search

Due to the high cost of simulations and a large design space, it is often impractical to do an exhaustive search of the entire design space to find the Pareto curve for a specific computer system. The last category of approaches focuses on reducing the number of simulated architectures with the use of predictive models and heuristic search.

Predictive modeling. This class of approach uses machine learning models to quickly predict the processor response (e.g. performance and power) of unseen architecture configurations. The key idea is to build a model with the simulation results of a small number of configurations in the design space and apply this model, without additional simulations, to all candidate architectures.

Program-based approaches build a predictive model for each program. The relation-

ship between processor configuration and its corresponding response is captured using linear regression [37], radial basis function (RBF) [38], spline functions [48], artificial neural network (ANN) [35], and genetic programming (GP) [11]. Most of the existing DSE methods randomly sample training datasets from the entire design space to train the regression models.

In the aforementioned schemes, whenever a new program is considered, a new predictor must be built. Khan et al. [40] and Dubach et al. [13] proposed cross-program predictive models to reduce the required time whenever a designer wants to consider a new program (e.g. compile with a different optimization level). Dubach et al. [13] argues that the behavior of the architecture space on a new program can be modeled as a linear combination of their behavior on previously seen programs. Khan et al. [40] uses the results of previously seen programs to train ANN models.

Characteristic-based approach [25] incorporates program characteristics to the training data. They find a mapping from a processor configuration and program characteristics to processor response. These characteristics are a subset of the ones described in Section 4.1. The key intuition is that workloads with similar characteristics to the programs used in the training data should have similar reactions to the design space.

All mentioned regression models focus on predicting the response of an architecture configuration to guide decisions in DSE. Chen et al. [8] formulates DSE as a ranking problem to predict which of any two architecture configurations performs best – the model feedback is not an architecture response but the prediction of which of two configurations is the best.

Heuristic searching. The overall goal of heuristic search is to minimize the number of simulations during the DSE. Simulations results can be obtained using the aforementioned predictive models or detailed simulation. Thus, heuristic approaches are exploited in orthogonality to predictive models.

In [23] the design space is divided into clusters, where each cluster is a set of dependent parameters. They use an exhaustive algorithm for calculating the Pareto-optimal configurations for each of the clusters. The global Pareto-optimal configurations are reconstructed from the merged Pareto-optimal configurations from each cluster.

Palermo et al. [68] solves DSE for embedded systems as a multi-objective optimization (e.g. performance, power, delay) and use three searching algorithms: simulated annealing, reactive taboo search, and random search. The idea is to compute an approximated Pareto set of configurations.

In [1] evolutionary algorithms and fuzzy systems are combined to simultaneously reduce the number of system configurations to be simulated and the time required to evaluate each configuration. They employ Multi-objective Evolutionary Algorithms as an optimization technique and Fuzzy Systems for the estimation of the performance of each configuration.

2.3 Summary

In this chapter, we discussed prior work in two main areas related to this dissertation: phase analysis and efficient simulation. We summarize them in Tables 2.1 and 2.2.

Author	Metric	Classification	Granularity	Usage
Balasubramoni et al. [2]	Hardware counters	Threshold	Fine-grained	Dynamic cache reconfiguration
Duesterwald et al. [14]	Hardware counters	Statistical and Table-based history predictors	Fine-grained	Efficient simulation
Isci and Martonosi [36]	Power vectors	First Pivot Clustering and Agglomerative Clustering	Fine-grained	Power reduction
Sherwood et al. [64]	BBV	k-means	Fine-grained	Efficient simulation
Dhodapkar et al. [12]	Working set	Threshold	Fine-grained	Dynamic cache reconfiguration
Huffmire and Sherwood [34]	Memory addresses	Wavelets	Fine-grained	Phase analysis
Shen et al. [62]	Data reuse distances	Wavelets and Sequitur	Fine-grained	Cache resizing and memory remapping
Eeckhout et al. [15]	Collection of program characteristics	k-means	Fine-grained	Efficient simulation
Lau et al. [47]	Hierarchical procedure call and loop graph	Loop or procedure boundaries	Fine-grained	Cache reconfiguration and Efficient simulation
Huang et al. [33]	Call stack		Fine-grained	Power reduction
Sanghai el al. [59]	BBV	Multinomial clustering	Fine-grained	Efficient simulation
Sherwood, et al. [63]	BBV	Fourier transform	Fine-grained	Phase analysis and Efficient simulation
Cho and Li [10]	Performance counters	Wavelets and k-means	Fine-grained	Phase analysis
Zhang et al. [69]	Hierarchical procedure call and loop graph	Loop or procedure boundaries	Both	Cache reconfiguration and Efficient simulation

Table 2.1: Phase analysis summary

Author	Category	Subcategory
Eeckhout et al. [19] Eeckhout and Hoste [30, 57] Yi et al. [68]	Reduced Benchmark	
KleinOsowski and Lilja [41] Breughe et al. [4]	Reduced Input	
Sherwood et al. [65] Zhang et al. [69] Lau et al. [47]	Reduced Instruction Trace	Phase-behavior sampling
Wunderlich et al. [67]		Statistical Sampling
Eeckhout et al. [17, 18].		Statistical simulation
Joseph and Thazhuthaveetil [37] Joseph, et al. [38] Lee and Brooks [48] Ïpek et al. [35] Cook and Skadron [11] Guo et al. [25] Chen et al. [8]	Reduced Design Space Search	Predictive modeling
Givargis et al. [23] Palermo et al. [68] Ascia et al. [1]		Heuristic searching

Table 2.2: Efficient simulation summary

2.4 Relation to This Work

Our work consists of a method for efficient simulation in the category of reduced instruction trace with phase-behavior sampling. We find program phases by inspecting a collection of program characteristics such as instruction mix, ILP, and memory access patterns [15]. Our approach consists of a two-level sampling strategy of those characteristics. The first uses a multivariate time-series clustering method to find coarse-grained phases that follow the definition of "a unit of repeating behavior" [62]. The second level further re-sample each coarse-grained phase into fine-grained phases using k-means. These fine-grained phases are a set of intervals within a coarse-grained phase that have similar behavior. This second-level sampling is heavily based on the work of [15].

The closest work in terms of combing different phase granularity levels is the work proposed Zhang et al. [69]. However, they identify a coarse-grained interval according to the outermost loop or frequently invoked functions. They use BBV to compare intervals and k-means for both levels.

Concerning hierarchical phase analysis, some approaches focus on finding phase transitions that match the procedure call and loop structure of programs [47]. However, as stated before, 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) [62], however, they explore a priori knowledge of memory access patterns of the applications.

Chapter 3

Toeplitz Inverse Covariance-based Clustering

This Chapter describes the method we used to segment a program's execution into a sequence of phases. It is known as *Toeplitz Inverse Covariance-Based Clustering* (TICC) and is heavily based on probabilistic graphical models. For further understanding, we begin in Section 3.1 giving a brief introduction on the subject. It starts by introducing the necessary notation and describing the central concept of conditional independence. Undirected Graphical Models and Gaussian Markov Random Field (GMRF) are then defined and studied in more detail. In Section 3.2 time-series are defined. In Section 3.3, we give a detailed discussion of the TICC method.

3.1 Introduction to Probabilistic Graphical Models

Probabilistic graphical models are a way to represent probability distributions over graphs. One can think of the union of graph theory and probability theory. In this graphical representation, the vertices correspond to random variables and the edges to the conditional dependence among the random variables. Graphical models exploit independencies present in probability distributions to compactly represent distributions in a factorized form [42].

Conditional independence is the fundamental concept that graphical models rely on to compactly represent a probability distribution. Two random variables $\mathbf{X}$ and $\mathbf{Y}$ are said to be conditionally independent given random variable $\mathbf{Z}$, denoted by $\mathbf{X} \perp \mathbf{Y} \mid \mathbf{Z}$, if and only if their joint conditional distribution factorizes according to:

$$P(\mathbf{X}, \mathbf{Y}|\mathbf{Z}) = P(\mathbf{X}|\mathbf{Z})P(\mathbf{Y}|\mathbf{Z})$$

There are essentially two families of graphical representations of distribution. The first is called Bayesian networks and uses a directed graph as its representation; the second, Markov networks, uses an undirected graph. In both families, the graph structure gives a compact representation of a set of independencies that hold in the distribution and show how the corresponding probability distributions factorize. They differ in the independencies they can encode and in the factorization of the distribution that they

induce [42]. In this work, we shall focus on the key aspects of undirected graphical models as needed for our phase analysis.

3.1.1 Undirected Graphical Models

An undirected graphical model, also known as Markov random field (MRF), represents a set of random variables having a Markov property by an undirected graph $G = (V, E)$. Consider a collection of random variables $\mathbf{x} = (x_1, x_2, ..., x_n)$ with associated joint probability distribution $p(\mathbf{x})$. Each vertex in V corresponds to one of the random variables in $\mathbf{x}$, that is, $V = (p \mid x_p \in \mathbf{x})$. Additionally, the edges in E should satisfy the following Markov property:

- **Global Markov property**: Let A, B, C be three disjoint subsets of V, and the notation X_A denotes all random variables corresponding to vertices included in the set A. If C separates A from B then the random variables x_A, x_C are conditionally independent given the variables x_C

  $x_A \perp x_B \,|\, x_C$

The global Markov property is one of the three commonly used Markov properties for undirected graphs. The remaining two properties are:

- **Pairwise Markov property**: two vertices are conditionally independent given the rest if there is no direct edge between them

  $x_i \perp x_j \,|\, x_{ij}$ if $(i, j) \notin E$ and $i \neq j$

- **Local Markov property**: Given its neighbors, each vertex is independent of all other variables

  $x_i \perp x_{V-N(i)} \,|\, x_{N(i)}$

The global Markov implies local Markov which implies pairwise Markov [42]. However, for a positive probability ($p(\mathbf{x}) > 0$), the three properties are equivalent [42]. The importance of this result is that in some situations, as we will see in the next section, it is easier to assess some property than the other.

Theorem 1. For positive distributions,

$$\text{Global Markov} \iff \text{Local Markov} \iff \text{Pairwise Markov}$$

Proof. See Corollary 4.1 from Koller and Friedman [42]. □

3.1.2 Gaussian Markov Random Fields

A popular instance of undirected graphical models is the Gaussian Markov Random Field (GMRF). A GMRF is a Markov random field of a multivariate Gaussian distribution. A multivariate Gaussian distribution over random variables $\mathbf{x} = (x_1, ..., x_n)$ is characterized

by an n-dimensional mean vector $\boldsymbol{\mu}$, and a symmetric $n \times n$ covariance matrix $\boldsymbol{\Sigma}$; the density function is most often defined as:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right] \tag{3.1}$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. We use $\mathbf{x} \sim \mathcal{N}(\mu, \boldsymbol{\Sigma})$ to denote that $\mathbf{x}$ follows a multivariate normal distribution.

The inverse covariance matrix term $\boldsymbol{\Sigma}^{-1}$ is called information matrix (or precision matrix). In the rest of the text we shall denote $\boldsymbol{\Sigma}^{-1} = \boldsymbol{\Theta}$. If $\boldsymbol{\Theta}_{i,j} = 0$, then random variables x_i and x_j in $\boldsymbol{\Theta}$ are conditionally independent (given the values of all other variables) [27].

Theorem 2. Let $\mathbf{x} = (x_1, ..., x_n)$ be normal distributed with mean μ and inverse covariance matrix $\boldsymbol{\Theta} > 0$. Then for $i \neq j$,

$$x_i \perp x_j | x_{\_ij} \iff \boldsymbol{\Theta}_{ij} = 0$$

Proof. See Theorem 2.2 from Rue and Held [58]. □

A Gaussian distribution over $\mathbf{x} = (x_1, ..., x_n)$ can be expressed as a Markov random field with respect to a graph $G = (V, E)$. We use Theorem 2 to build a GRMR by letting the inverse covariance matrix $\boldsymbol{\Theta}$ define the adjacency matrix of G. The missing edges in E correspond to zeros in $\boldsymbol{\Theta}$. This means that G has no edge between vertex i and vertex j if and only if $x_i \perp x_j \,|\, x_{\_ij}$ – the **pairwise Markov property** is satisfied. Because a Gaussian density is a positive distribution then **Global Markov property** is also satisfied (by Theorem 1). Thus, random variables $\mathbf{x}$ make up a valid Markov random field with respect to the graph G.

The formal definition of a GMRF is given by [58].

Definition 1. (GMRF) A random vector $\mathbf{x} = (x_1, ..., x_n)^T \in \mathbb{R}^n$ is called a GMRF with respect to a labelled graph $G = (V, E)$ with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma} > 0$, iff its density has the form

$$\pi(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}|\boldsymbol{\Sigma}|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right] \tag{3.2}$$

and

$$\boldsymbol{\Sigma}^{-1}_{i,j} \neq 0 \iff \{i, j\} \in E, \forall i \neq j \tag{3.3}$$

3.2 Time Series Definitions

Time series is a set of observations collected sequentially in time. Formally

Definition 2. (Time series) A time series S of size T is an ordered sequence of real-value data, $x_i \in \mathbb{R}^n$, where $S = (x_1, x_2, \cdots, x_T)$.

A univariate time series refers to a time series that only one variable is varying over time ($n = 1$). A multivariate time series has multiple variables varying over time ($n > 1$).

Another notation we shall use throughout the text is the concept of time series subsequence.

Definition 3. (Time series subsequence) A subsequence of length m ending at the i-th observation of time series S is defined as $S_{i,m} = (x_{i-m+1}, \cdots, x_{i-1}, x_i)$.

3.3 Toeplitz Inverse Covariance-Based Clustering

Our phase analysis is based on time-series clustering. We view a program's execution as an ordered sequence of inherent program observations, as will be described in Section 4.1. Such data is interpreted as 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. This approach requires simultaneous segmentation and clustering of the time series. To achieve that, we use a recently proposed method of time-series clustering and segmentation called Toeplitz Inverse Covariance-based Clustering (TICC)[27].

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. It can be viewed as a multi-layer network, with one layer for each observation (time step) of the subsequence. The edges inside a layer show an intra time correlation, and the edges between layers show a cross-time correlation. 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 3.1 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 highlighted subsequences of cluster A: both have the same MRF, even though they start at a different position. Intra-layer edges are represented by solid lines and inter-layer edges by dashed lines

Another example can be data from an automobile with three sensors: steering wheel angle, Lateral (Y-)Acceleration, and brake pedal [27]. This data can be interpreted as a sequence of actions: *turning*, *speeding up*, *slowing down*, and *going straight*. In an MRF corresponding to *turning*: the intra time correlation (inside one layer) might show how the steering wheel angle influences the Lateral (Y-)Acceleration; the cross-time edges (across layers) may show how the brake pedal at time t might affect the steering wheel angle at time $t + 1$. The time-invariant property can be thought of as: no matter which subsequence of the cluster we are looking at, in a "car turn" the brake pedal at time t will always affect the steering wheel angle at time $t + 1$ [27].

Recall from Section 3.1.2 that we can learn an MRF representation by estimating a sparse Gaussian inverse covariance matrix $\Theta = \Sigma^{-1}$, which defines a graph structure by

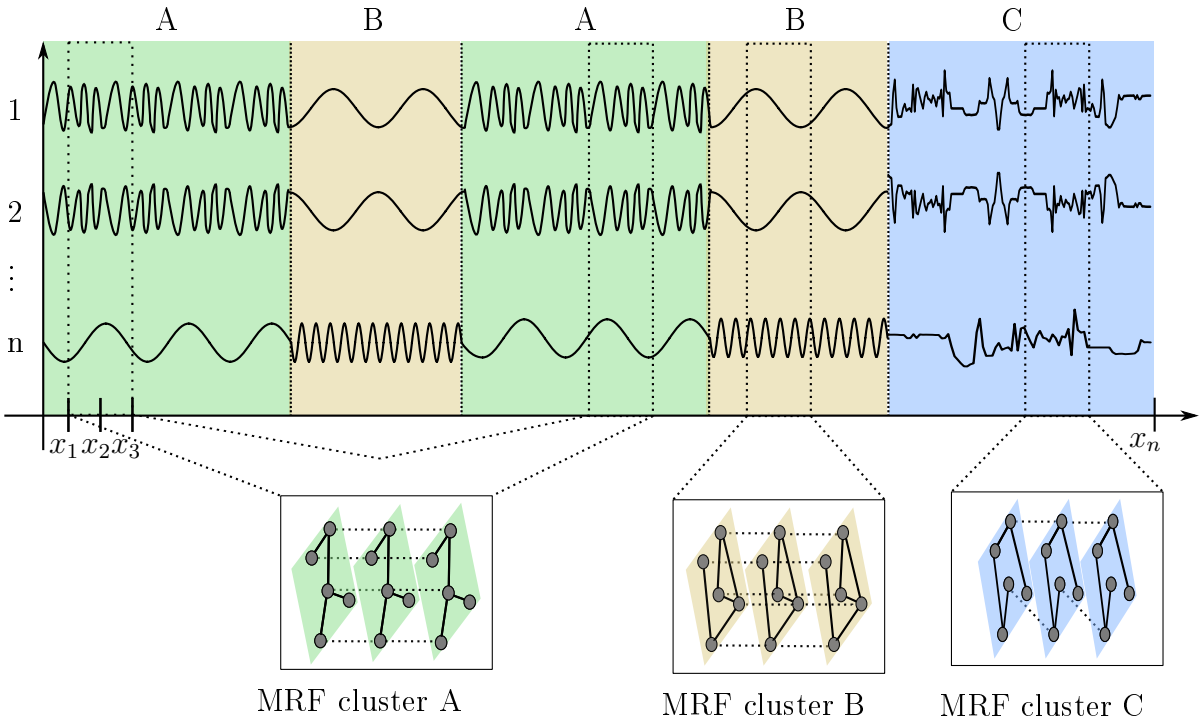


Figure 3.1: Overview of the TICC method segmentation

defining the adjacency matrix of the MRF dependency network. Thus, TICC learns each cluster's MRF by estimating Θ. The time-invariant correlation is learned by imposing a block Toeplitz matrix structure on the inverse covariance matrix.

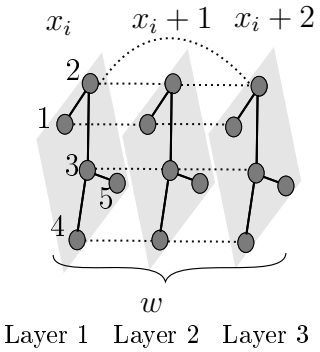
3.3.1 Block Toeplitz Inverse Covariance Matrix

Let w be the size of the subsequence that each cluster MRF is characterized over. To ensure that each cluster's MRF has the aforementioned time-invariant correlation, TICC imposes that Θ has a block Toeplitz matrix structure. A block matrix is a matrix that is interpreted as being partitioned into submatrices. A Toeplitz matrix is a matrix whose entries are constant along the diagonals. Then, a block Toeplitz matrix is a block matrix whose blocks are repeated down the diagonals of the matrix. Given a time series with n variables, the inverse covariance matrix Θ defining each cluster is of the form

$$\Theta = \begin{bmatrix} A_0 & A_1^T & A_2^T & \dots & \dots & A_{w-1}^T \\ A_1 & A_0 & A_1^T & \ddots & & \vdots \\ A_2 & A_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & A_1^T & A_2^T \\ \vdots & & \ddots & A_1 & A_0 & A_1^T \\ A_{w-1} & \cdots & \cdots & A_2 & A_1 & A_0 \end{bmatrix} \in \mathbb{R}^{nw \times nw}$$

with blocks $A_i \in \mathbb{R}^{n \times n}$. Each block $\Theta_{i,j}$ defines the edges between time steps i and j. For $i = j$ (block A_0) we have the adjacency matrix of the edges within each layer. On the other hand, for $i \neq j$ (blocks $A_1, A_2, \cdots, A_{w-1}$), we have "cross-time" edges.

Returning to the example in Figure 3.1, let's suppose it is a time series with $n = 5$ variables. In this case, the inverse covariance matrix Θ over a window of size 3 defining cluster A has the following form:



$$\Theta_A = \begin{bmatrix} A_0 & A_1^T & A_2^T \\ A_1 & A_0 & A_1^T \\ A_2 & A_1 & A_0 \end{bmatrix}$$

Figure 3.2: An example of an MRF with its corresponding block Toeplitz adjacency matrix

All layers have the same intra time correlation $\Theta_A(1,1) = \Theta_A(2,2) = \Theta_A(3,3) = A_0$. The edges between layer 1 and layer 2 must also exist between layers 2 and 3, that is, $\Theta_A(1,2) = \Theta_A(2,3) = A_1$. Each block is of the form

$$A_0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & * & 0 & 0 & 0 \\ * & * & * & 0 & 0 \\ 0 & * & * & * & * \\ 0 & 0 & * & * & 0 \\ 0 & 0 & * & 0 & * \end{pmatrix} \end{array}, A_1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 6 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & * & 0 & 0 \\ 0 & 0 & 0 & * & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}, A_2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

with edges corresponding to the non-zero entries.

3.3.2 Problem formulation

Let $(x_1, x_2, ..., x_T)$ be a time series, where $x_t \in \mathbb{R}^n$ is the t-th multivariate observation. The goal of TICC is to partition these observations into a predefined number of clusters, K. Let $P = \{P_1, ..., P_K\}$ be such a partition. TICC clusters each observation x_t based on the MRF defined over the subsequence $S_{t,w}$, that is, a subquence of size w ending at x_t. Each subsequence $S_{i,m} = (x_{i-m+1}, \cdots, x_{i-1}, x_i)$ forms a nw-dimensional vector and is simply referred to as X_t. Rather than clustering observations $(x_1, x_2, ..., x_T)$, TICC's approach instead consists of clustering the subsequences $(X_1, X_2, ..., X_T)$. Figure 3.3 elucidates the general skeleton of each subsequence.

Each cluster i is represented as an MRF having $\Theta_i \in \mathbb{R}^{nw \times nw}$ as its adjacency matrix. Thus, each subsequence belongs to a cluster i with a Gaussian distribution that has the inverse covariance matrix Θ_i. The objective of TICC is to find a partition $\mathbf{P} = \{P_1, ..., P_K\}$ of $(X_1, X_2, ..., X_T)$ and the corresponding MRF structure of each cluster $\mathbf{\Theta} = \{\Theta_1, ..., \Theta_K\}$ by solving the following optimization problem:

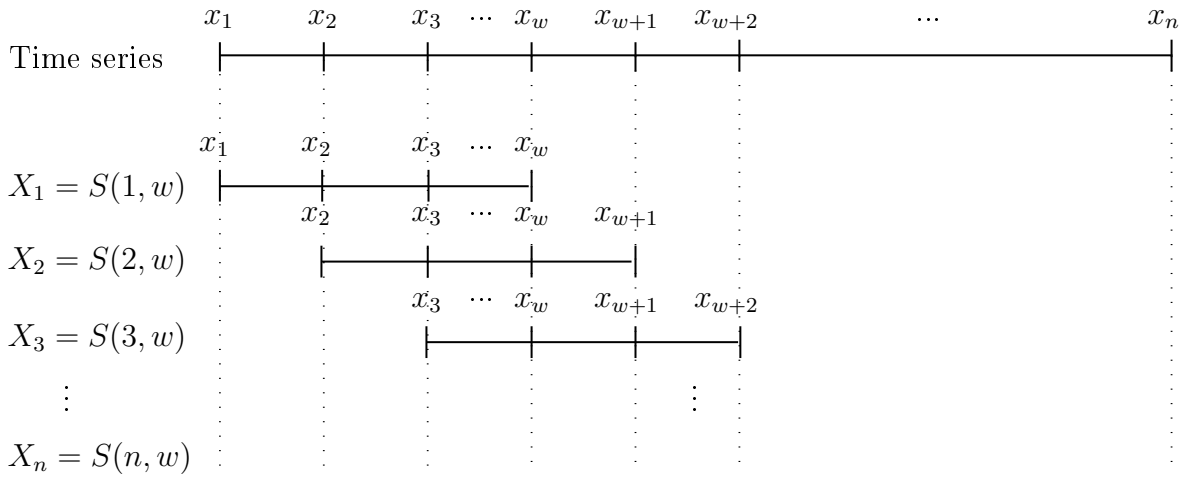


Figure 3.3: Subsequences extracted from a time series

$$\underset{\Theta \in \mathcal{T},\, P}{\operatorname{argmin}} \sum_{i=1}^{K} \left(\underbrace{\|\lambda \circ \Theta_i\|_1}_{\text{(iii) sparsity}} - \underbrace{\log \mathcal{L}(\Theta_i | P_i)}_{\text{(i) log-likelihood}} + \underbrace{\sum_{X_t \in P_i} \beta \mathbb{1}\{X_{t-1} \notin P_i\}}_{\text{(ii) temporal consistency}} \right) \tag{3.4}$$

In the rest of the section, we will go into details about terms (i), (ii), and (iii). We will arrive at Equation (3.4) by adding one term at a time.

(i) Log-likelihood. Given a set of subsequences $X = (X_i, ..., X_T)$, let $\{P_1, ..., P_K\}$ be a partition of X with K clusters. TICC assumes that each subsequence $X_t \in P_i \sim \mathcal{N}(\mu_i, \Theta_i)$. Let $\mathcal{L}(P_i|\Theta_i)$ denote the likelihood that P_i came from cluster i. By definition:

$$\mathcal{L}(\Theta_i | P_i) = \prod_{X_t \in P_i} \mathcal{N}(X_t; \Theta_i) \tag{3.5}$$

As the natural logarithm of the likelihood function is more convenient to work with, we have:

$$\begin{aligned}
\log \mathcal{L}(\Theta_i | P_i) &= \sum_{X_t \in P_i} \log \mathcal{N}(X_t; \Theta_i) \\
&= \sum_{X_t \in P_i} \log \left(\frac{|\Theta_i|^{1/2}}{(2\pi)^{m/2}} \exp \left(-\frac{1}{2} (\mathbf{X_t} - \mu_\mathbf{i})^\mathbf{T} \mathbf{\Theta_i} (\mathbf{X_t} - \mu_\mathbf{i}) \right) \right) \\
&= \sum_{X_t \in P_i} -\frac{m}{2} \log(2\pi) - \frac{1}{2} \log |\Theta_i| - \frac{1}{2} (\mathbf{X_t} - \mu_\mathbf{i})^\mathbf{T} \mathbf{\Theta_i} (\mathbf{X_t} - \mu_\mathbf{i})
\end{aligned}$$

where μ_i is the empirical mean of cluster i and $|\Theta_i|$ is the determinant of Θ_i.

Let $\mathcal{T}$ be the set of symmetric block Toeplitz $nw \times nw$ matrices. We want to maximize the log-likelihood of each cluster in P with constraints that the inverse covariance matrix

is block Toeplitz, that is, $\Theta \in \mathcal{T}$

$$\underset{\Theta \in \mathcal{T},\, P}{\operatorname{argmax}} \sum_{i=1}^{K} \log \mathcal{L}(\Theta_i | P_i)$$

As TICC is a minimization problem, we convert the above objective into an equivalent minimization problem as follows

$$\underset{\Theta \in \mathcal{T},\, P}{\operatorname{argmin}} \sum_{i=1}^{K} -\log \mathcal{L}(\Theta_i | P_i)$$

So far, the problem is to minimize the negative log-likelihood.

(ii) Temporal consistency. When TICC is assigning the data to clusters, it makes an additional goal of temporal consistency. The idea is that neighboring points are encouraged to belong to the same cluster. Let $\beta \in \mathbb{R}$, and $\mathbb{1}\{X_{t-1} \notin P_i\}$ an indicator function as follows:

$$\mathbb{1}\{X_{t-1} \notin P_i\} = \begin{cases} 0, & X_{t-1} \in P_i \\ 1, & X_{t-1} \notin P_i \end{cases}$$

TICC imposes a penalty of summing β every time we find an adjacent point that does not belong to the same cluster. So, our augmented objective is to minimize the negative log-likelihood and enforce temporal consistency.

$$\underset{\Theta \in \mathcal{T},\, P}{\operatorname{argmin}} \sum_{i=1}^{K} \left(-\log \mathcal{L}(\Theta_i | P_i) + \sum_{X_t \in P_i} \beta \mathbb{1}\{X_{t-1} \notin P_i\} \right)$$

(iii) Sparsity penalty. The last penalty TICC adds is what is called a sparsity penalty. Sparsity reduces the tendency of overfitting and increases the interpretability of each cluster's MRF [27]. It is achieved by imposing $\|\lambda \circ \Theta_i\|_1$ to the optimization problem: the ℓ_1-norm of the element-wise product of the inverse covariance matrix Θ_i and the regularization parameter $\lambda \in \mathbb{R}^{nw \times nw}$.

Even though λ is a $nw \times nw$ matrix, the authors of TICC suggest to set all its values to a single constant, reducing the search space to just one parameter. With that, we arrive at Equation (3.4) with the goal of minimizing the negative log-likelihood, enforcing temporal consistency and making sure Θ_i is sparse.

$$\underset{\Theta \in \mathcal{T},\, P}{\operatorname{argmin}} \sum_{i=1}^{K} \left(\|\lambda \circ \Theta_i\|_1 - \log \mathcal{L}(\Theta_i | P_i) + \sum_{X_t \in P_i} \beta \mathbb{1}\{X_{t-1} \notin P_i\} \right)$$

3.3.3 TICC Algorithm

In order to solve problem 3.4 TICC uses a variation of the expectation maximization (EM) algorithm to alternate between (i) assigning points to clusters and then (ii) updating the

cluster parameters.

(i) Assign points to cluster (E-step): assign points to clusters by fixing the value of Θ and solving the following combinatorial optimization problem for $P = \{P_1, ..., P_K\}$:

$$\text{mimize} \sum_{i=1}^{K} \left(-\log \mathcal{L}(\Theta_i | P_i) + \sum_{X_t \in P_i} \beta \mathbb{1}\{X_{t-1} \notin P_i\} \right) \tag{3.6}$$

TICC uses dynamic programming for solving this problem, which allows learning the optimal assignments in time $O(KT)$.

(ii) Update cluster parameters (M-step): Given the point assignments P, the next task is to update the cluster parameters $\Theta_1, ..., \Theta_K$ by solving Equation (3.4) while holding P constant. The goal is to find Θ_i that mostly explains the assignments of points in P. TICC solves for each Θ in parallel. The log-likelihood of Equation (3.5) can be expressed as:

$$\begin{aligned} \log \mathcal{L}(\Theta_i | P_i) &= \sum_{X_t \in P_i} -\frac{m}{2} \log(2\pi) - \frac{1}{2} \log |\Theta_i| - \frac{1}{2} (\mathbf{X_t} - \mu_\mathbf{i})^\mathbf{T} \mathbf{\Theta_i} (\mathbf{X_t} - \mu_\mathbf{i}) \\ &= -|P_i| (\log \det \theta_i + \text{tr}(S_i \theta_i)) + C \end{aligned}$$

where $|P_i|$ is the number of points in cluster i, S_i is the empirical covariance of points in P_i, tr is the trace of a matrix, and C is a constant that does not depend on Θ_i. Using this notation in Equation (3.4), for each Θ_i, the problem becomes:

$$\begin{aligned} &\text{minimize} && -\log \det \theta_i + \text{tr}(S_i \theta_i) + \frac{1}{|P_i|} \|\lambda \circ \Theta_i\|_1 \\ &\text{subject to} && \theta_i \in \tau \end{aligned} \tag{3.7}$$

Problem 3.7 is a variation of the graphical lasso problem [22] with the block Toeplitz constraint on the inverse covariance. TICC solves this problem using an algorithm based on the alternating direction method of multipliers (ADMM).

A Python implementation for TICC's algorithm is available in [26].

3.4 Summary

In this chapter, we described the core method we used in this work to find repeated patterns in the execution of a program. It is built upon probabilistic graphical models and it is a method of subsequence clustering of multivariate time series. Each cluster in TICC is 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 [27]. As we will show in the next chapter, the information we extract from each program's execution makes up a multivariate time-series of inherent program observations, which makes TICC suitable for our problem. We used TICC's segmentation as the first stage of a two-stage sampling strategy and to learn interpretable program phases.

Chapter 4

Phase Classification

In this chapter, we present our method of phase classification for efficient simulation. In Section 4.1 we describe the input necessary to our phase analysis. This information is a set of microarchitecture independent characteristics dynamically extracted from each program's execution, which we refer to as MICA [30]. In Section 4.2 we present our method to find samples for simulation. The core idea is to detect patterns in MICA characteristics using the TICC method described in the previous chapter. This results in a sequence of program phases, each defined by an MRF, which provides an interpretable structure. Then, we use k-means to select representative samples of each phase found by TICC. This results in simulation points that are used to represent the select phases instead of the entire program execution. Our experimental setup is described in Section 4.3.

4.1 Microarchitecture-Independent Characterization of Applications (MICA)

Eeckhout et al. [30] proposed a set of characteristics used to characterize applications in a microarchitecture-independent manner. This set of characteristics has been used for benchmark subsetting [20], find program phases [15], measure similarity among benchmarks [39], performance prediction [32, 13], performance optimization [61], and compiler optimization [9].

In this work, we analyze the time-varying behavior of these characteristics to find unique repetitive patterns. The reason we consider microarchitecture independent characteristics instead of microarchitecture dependent characteristics is that they are not biased towards a specific hardware implementation, and thus needs to be collected once and can be used in any processor configuration. However, these are not instruction set architecture (ISA) or compiler independent.

We now describe the MICA set of characteristics:

Instruction mix Instruction category counters.

Instruction-level Parallelism (ILP) Amount of ILP for different instruction window sizes. This is measured by assuming perfect caches, perfect branch prediction, etc. The only limitations are the instruction window size and the data dependencies.

Register traffic characteristics Several characteristics concerning registers [21]:

- Average number of register input operands per instruction
- Average degree of use (number of times a register is used by other instructions)

$$\text{average degree of use} = \frac{\text{total number of register instance uses}}{\text{total number of register instances created}} \tag{4.1}$$

- Distribution of register dependency distance, that is, the number of instructions between the production and consumption of a register instance.

Working set The number of unique 64-byte blocks and 4KB memory touched for both instruction and data accesses.

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 – profiler needs to keep track of last address for every load and store.
- Global stride is the difference in the memory address of any consecutive memory access – profiler needs to keep track of the last load and store address.

The strides are presented as a histogram and are computed separately for loads and stores, resulting in four histograms: local load stride, local store stride, global load stride, global store stride.

Branch predictability Branch predictability is characterized using the *Prediction by Partial Matching* (PPM) predictor [7].

A Markov predictor of order j predicts the next branch outcome based upon the j preceding branch outcomes. It outputs the most likely branch outcome given the last j outcomes. To do so, it records the number of times a "taken" (T) or "not taken" (N) occurred after every j possible outcomes. Figure 4.1 illustrates how a Markov predictor works. It shows the state of an order 2 Markov predictor given that the sequence seen so far is $N, T, N, T, N, T, T, N, T$. The next branch direction is predicted based on the two immediately preceding bits, that is, N, T. The predictor predicts the next direction to be N with a probability of $2/3$.

A PPM predictor is built upon the combination of multiple Markov predictors. An m-order PPM predictor consists of $(m + 1)$ Markov predictors of orders 0 up to m. The PPM predictor checks if the m-length history appeared in the m-th order Markov predictor (the pattern has a non-zero frequency count). If so, it predicts the next outcome using this mth-order Markov predictor as described previously. If not, it checks if the (m-1)-length history appeared in the (m-1)-th order Markov predictor. If it is not present, then uses the (m-2)-length history to index the (m-2)-th order Markov prediction, and so on. Finally, the PPM updates the Markov predictor that made the prediction and all its higher order Markov predictors.

Branch history:
N,T,N,T,N,T,T,N,T

pattern	next outcome	frequency count
N,N	N	0
	T	0
N,T	N	2
	T	1
T,N	N	0
	T	3
T,T	N	1
	T	0

Prediction: N

Figure 4.1: Example of an order-2 Markov predictor [7]

In this work, we consider four variations of a PPM predictor: GAg, PAg, GAs, and PAs. 'G' means a global branch history for all static branches and 'P' a separate branch history for each static branch. 'g' means a PPM predictor for all static branches and 's' means a PPM predictor for each static branch. Figure 4.2 illustrates these variations.

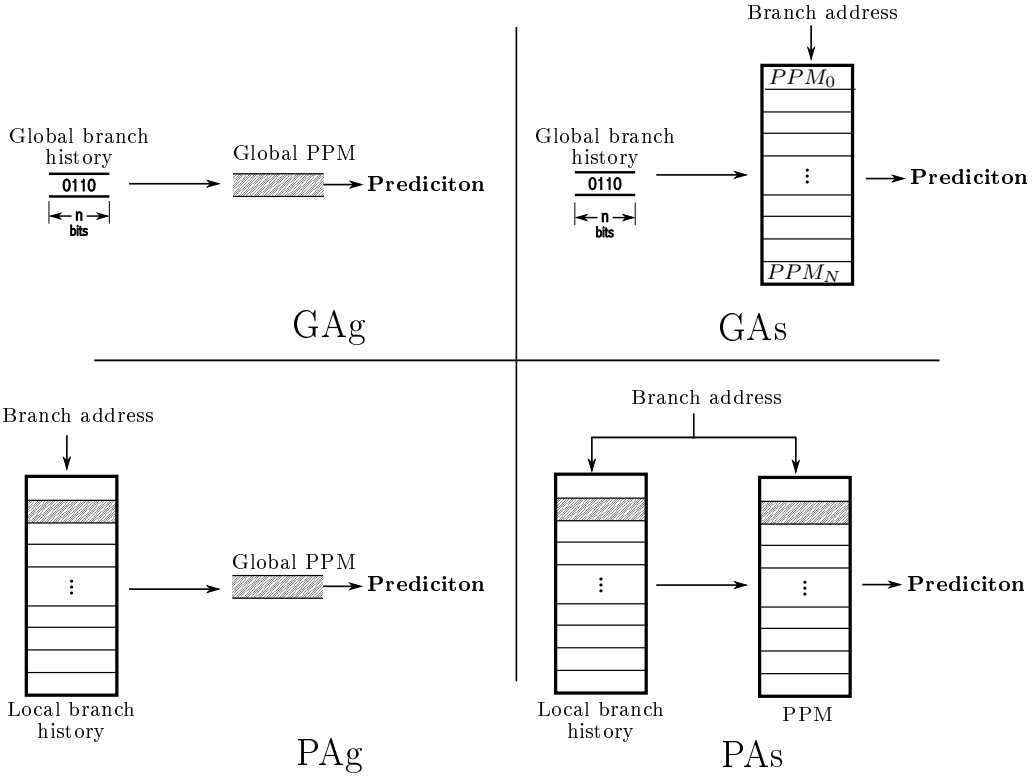


Figure 4.2: Variations of PPM

Additionally, the branch taken rate and branch transition rate are also being measured.

Memory reuse distances The number of distinct memory locations accessed between two memory references to the same location [30]. The reuse distances are reported in a histogram. The i-th bin contains the number of memory reads with reuse distance $[2^i, 2^{i+1})$. There is also a special bin to the number of cold misses.

In this work, we use a total of 97 characteristics (Table 4.1) divided across the 7 categories we described above.

4.2 Phase Classification Formulation

Our phase-analysis approach for accelerating architectural simulation consists of a two-level sampling strategy. The first level follows the phase definition by [62] which states that "a phase is a unit of repeating behavior rather than a unit of uniform behavior". The inherent program characteristics described in Section 4.1 reveal patterns in the program's execution. We automatically found these recurring patterns using TICC applied over the set of characteristics described in the previous section, viewed as multivariate time-series. We refer to the phases found by TICC as **coarse-grained phases**.

We give a simple example of such phases in Figure 4.3, which shows the time-varying behavior of 97 MICA features of one program from the SPEC 2006 benchmark suite, *gcc* with input *166*. Each interval of 160M dynamic instructions is a point in the graph. The first thing we can notice is that although these MICA features vary over time, they show repeated patterns. Different colors indicate different patterns found by TICC. Another important point to notice is the correlation with the performance metrics, which is shown in Figure 4.4 (as discussed in [15]).

The second sampling level consists of running k-means clustering algorithm on each coarse-grained phase. Each cluster found by k-means is referred to as a **fine-grained phase**, and it is a set of intervals within a program's execution that have similar MICA, which consequently tends to exhibit similar behavior similar [15]. We then pick a single representative interval from each fine-grained phase. These intervals are used to represent the selected coarse-grained phase.

The coarse-grained phases found by TICC could be used for accelerating simulation. However, as discussed in Section 5.3.2, regarding the total SPECint 2006 dynamic instructions, that would still require a large proportion of 45% to 55% to be simulated. On the other hand, with the second-level sampling, this number goes down to 1% to 2%, while still getting accurate results.

In summary, our approach consists of a two-level sampling strategy over the MICA set of characteristics. The first uses TICC to find coarse-grained phases – units of repeating behavior. The second level further re-sample each coarse-grained phase into fine-grained phases using k-means. These fine-grained phases are a set of intervals within a coarse-grained phase that have similar behavior. The following steps summarize our phase classification approach, and the following sections explain each step in detail:

Phase classification (needs to be run once for a program/input pair):

1. Profile a set of microarchitecture-independent characteristics (MICA) per interval of dynamic instructions.

2. Reduce the dimension of the MICA data using PCA.

3. Run Toeplitz Inverse Covariance-based Clustering (TICC) on the lower-dimensional data for several number of clusters: from 1 to $cMaxK$, where $cMaxK$ is the maximum

number of coarse-grained phases to be discovered.

4. Use BIC score to choose the clustering with the smallest number of clusters, such that its BIC score is at least *cBIC* percent of the best score.

5. For each cluster outputted by TICC, run k-means clustering on the MICA representation of the intervals belonging to the same cluster for several number of clusters: from 1 to *fMaxK*, where *fMaxK* the maximum number of fine-grained phases to use.

6. Use BIC score to choose the clustering with the smallest number of clusters, such that its BIC score is at least *fBIC* percent of the best score.

7. Pick the center interval of each cluster outputted by k-means as a simulation point.

For each design point of interest:

1. Do detailed simulation on each simulation point.

2. Combine results of each simulation point to get the result of the overall execution.

4.2.1 MICA as Time Series

The representation in which our analysis is based on is the set of MICA characteristics. These are collected in terms of instructions executed. We use SimPoint's [65] model of breaking a program's execution into a set of contiguous non-overlapping intervals. We use interval size at the granularity of 100M. The interval size defines the sampling period of our time-series. Thus, we use the term "interval size" and "sampling period" interchangeable in the rest of the work. The MICA characteristics are collected for each interval of execution, and the state of each MICA feature is reset at the end of each interval. We consider 97 microarchitecture-independent characteristics in this work ($n = 97$), which are fully described in Table 4.1.

The MICA profiler we used outputs absolute values. A further offline conversion is needed to produce the proportional metrics stated in Section 4.1. 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. Similarly, the profiler outputs the total number of register instance uses and register instances created which can be used to derive the register average degree of use (Equation (4.1)).

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.

Formally, let $x_i = (mica_1, ..., mica_n)$ be a n-dimensional vector of readings for n MICA characteristics in the i-th sample over the execution of the program sample. Also let $(x_1, x_2, ..., x_T)$ be a multivariate time series of T sequential MICA observations.

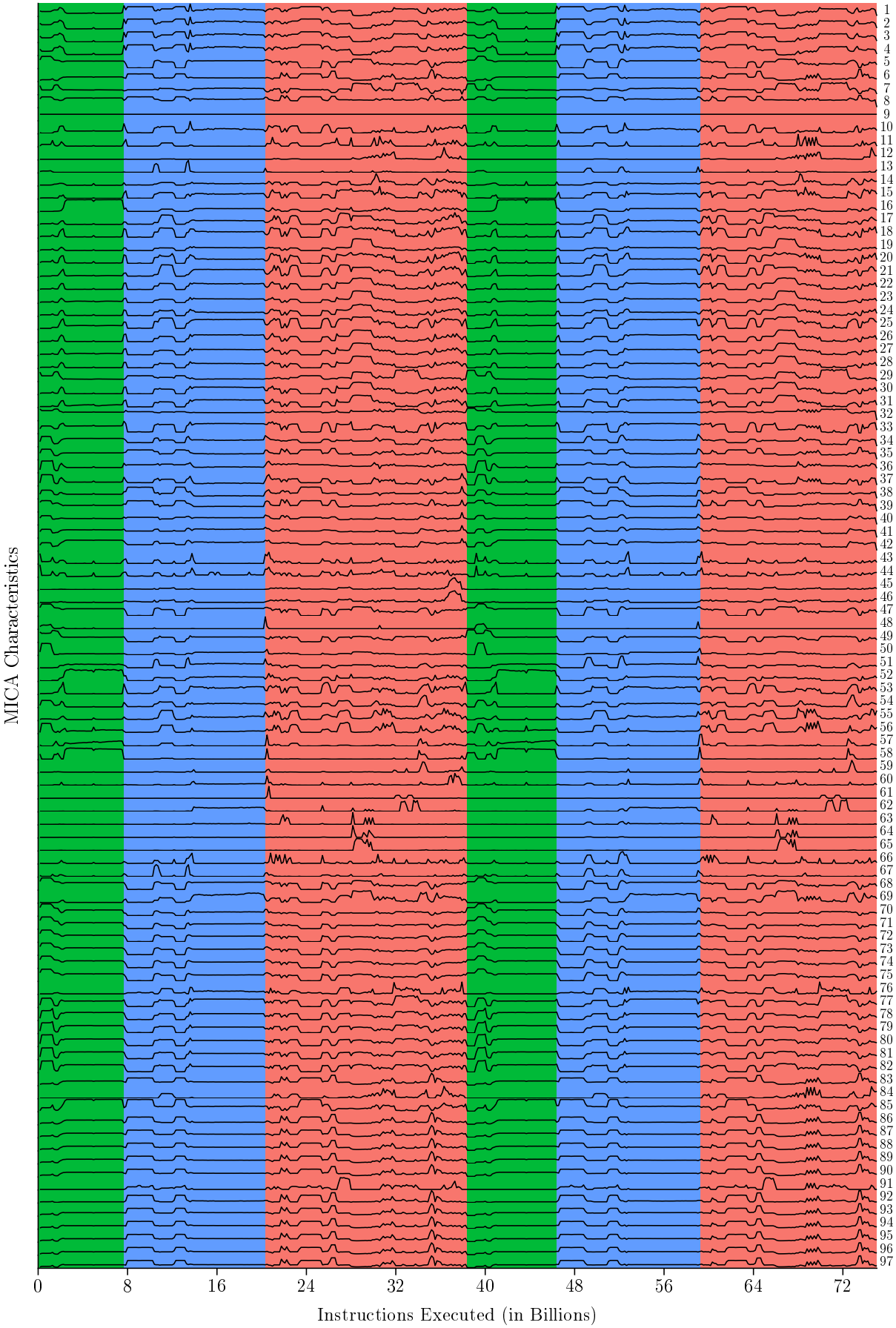


Figure 4.3: MICA profiling for *gcc-166*

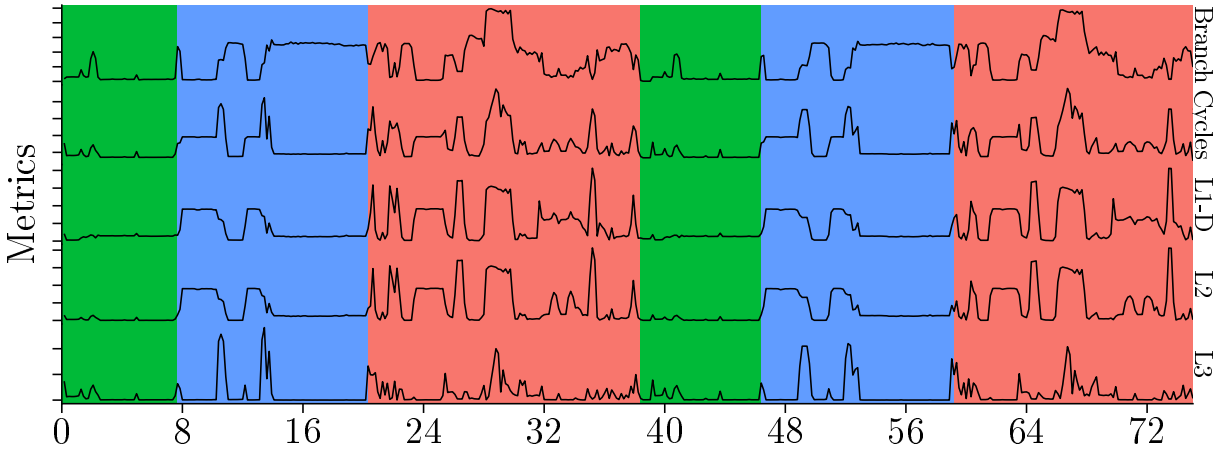


Figure 4.4: Time-varying graph for *gcc-166*

4.2.2 Dimension Reduction

This step and the following are only used when aiming for sampled simulation. When aiming for phase interpretation we consider the raw MICA features. A representation of lower dimensionality of the MICA representation is found in order to speed up the processing time of the clustering algorithms we used (TICC and k-means) and to prevent the features to suffer from the *curse of dimensionality*, which refers to the fact that it becomes extremely hard to cluster data as the number of dimensions increases.

Formally, given an observation $x_i = (mica_1, ..., mica_n)$ of our MICA time-series, this step aims to find a representation of lower dimensionality $z_i = (z_1, .., z_p)$ with $p < n$ which preserves the information content of the original data, as much as possible, according to some criterion.

There are basically two approaches for dimensionality reduction [30]: feature extraction and feature selection. The former transforms the original data into a new representation, which may not have an interpretable solution, thus complicates the understandability of the lower-dimensional workload space; the latter finds a subset of the original features.

We consider principal components analysis (PCA) for dimensionality reduction, which is a feature extraction method. It transforms the original data into a new representation, a set of non-correlated principal components. The input to PCA is a $T \times m$ matrix in which the rows are the T MICA observations and columns are the m microarchitecture-independent characteristics. The output is a $T \times p$ matrix data in which the rows are the T observations the columns are the p retained principal components.

4.2.3 MICA segmentation (Level 1)

We run *Toeplitz Inverse Covariance-based Clustering* (TICC) to discover repeated patterns on the sequential MICA data. These patterns are interpreted as coarse-grained program phases. Besides, each phase is defined by a correlation network, or Markov random field (MRF), characterizing the interdependencies between different observations in a short subsequence of that phase. This gives the ability to have "interpretable program phases", such that we can determine key factors and relationships that characterize each

Category	no.	Description
ILP	1	32-entry window
	2	64-entry window
	3	128-entry window
	4	256-entry window
Instruction mix	5	memory read
	6	memory write
	7	control flow
	8	arithmetic
	9	floating-point
	10	pop/push instructions (stack usage)
	11	shift instructions (bitwise)
	12	string
	13	MMX/SSE instructions
	14	system
	15	nop
	16	other
Branch predictability	17	GAg PPM predictor (4 bits)
	18	PAg PPM predictor (4 bits)
	19	GAs PPM predictor (4 bits)
	20	PAs PPM predictor (4 bits)
	21	GAg PPM predictor (8 bits)
	22	PAg PPM predictor (8 bits)
	23	GAs PPM predictor (8 bits)
	24	PAs PPM predictor (8 bits)
	25	GAg PPM predictor (12 bits)
	26	PAg PPM predictor (12 bits)
	27	GAs PPM predictor (12 bits)
	28	PAs PPM predictor (12 bits)
	29	Total branch count
	30	Total branch transition
	31	Total branch taken
Register traffic	32	Number of reg. operands
	33	Reg. instances created
	34	Reg. instance uses
	35	Total reg. dependency distance
	36	Reg. dependency distance $= 2^0$
	37	Reg. dependency distance $\leq 2^1$
	38	Reg. dependency distance $\leq 2^2$
	⋮	⋮
	42	Reg. dependency distance $\leq 2^6$

Category	no.	Description
Memory footprint	43	Num. of 64-byte blocks data
	44	Num. of 4KB pages data
	45	Num. of 64-byte blocks instr.
	46	Num. of 4KB pages instr.
Memory reuse distances	47	Memory access count
	48	Cold references count
	49	Reuse distance $[0, 2^1)$
	50	Reuse distance $[2^1, 2^2)$
	⋮	⋮
	67	Reuse distance $[2^{18}, 2^{19})$
Data stream strides	68	Memory read count
	69	Local load stride $\leq 8^1$
	⋮	⋮
	75	Local load stride $\leq 8^6$
	76	Global load stride $\leq 8^1$
	⋮	⋮
	82	Global load stride $\leq 8^6$
	83	Memory write count
	84	Local store stride $\leq 8^1$
	⋮	⋮
	90	Local store stride $\leq 8^6$
	91	Global store stride $\leq 8^1$
	⋮	⋮
	97	Global store stride $\leq 8^6$

Table 4.1: Sampled MICA features

phase [27].

Formally, let $(x_1, ..., x_T)$ and $(z_1, .., z_T)$ be T original and lower-dimension sequential MICA observations of a given program, respectively. Also let $(Z_1, ..., Z_T)$ be the sequence where each Z_i consists of observations $z_{t-w+1}, ..., z_t$. The goal of TICC is to cluster observations $(Z_1, ..., Z_T)$ into K phases. Each phase i is defined by a Gaussian inverse covariance $\Theta_i \in \mathbb{R}^{nw \times nw}$, which defines the correlation network the phase.

TICC's overall optimization problem is to find a partition $\mathbf{P} = \{P_1, ..., P_K\}$ of $(Z_1, Z_2, ..., Z_T)$ and the corresponding MRF structure of each cluster $\mathbf{\Theta} = \{\Theta_1, ..., \Theta_K\}$ by solving the following optimization problem:

$$\underset{\theta \in \tau, P}{\operatorname{argmin}} \sum_{i=1}^{K} \left[\|\lambda \circ \Theta_i\|_1 + \sum_{Z_t \in P_i} \left(-\ell\ell(Z_t, \theta_i) + \beta \mathbb{1}\{Z_{t-1} \notin P_i\} \right) \right] \tag{4.2}$$

A detailed description of each symbol from the above equation and TICC method is found in Section 3.3.

In this work, we find the correct choice of K via Bayesian information criterion (BIC) [56]. We run TICC for a range of values and pick the clustering with the smallest K, such that its BIC score is at least X percent of the best score [65]. The importance of this threshold is to avoid often choosing the clustering with the most clusters, as the BIC score often increases as the number of clusters increase [28]. As we also use the BIC score for the next step, we shall refer to the BIC threshold from this step as *cBIC* (coarse-grained BIC). The range of values we run TICC is based on a variable called *cMaxK* (coarse-grained maximum K), which defines a range of values from *1* to *cMaxK*.

4.2.4 Sampling points per phase (Level 2)

The next step in our analysis is to select a set of representative intervals from each coarse-grained phase, which will be the final intervals picked for simulation. These are found by running k-means on each partition (or phase) $P_i \in \{P_1, ..., P_K\}$ outputted by TICC. Running k-means in each coarse-grained phase P_i outputs a new partition $\mathbf{Q}_i = \{Q_{1,i}, ..., Q_{K'_i,i}\}$. We refer to each element of $Q_{i,j}$ as a fine-grained phase. More precisely, it is a fine-grained phase j extracted from the coarse-grained phase i. In summary, we re-partition each coarse-grained phase into multiple fine-grained phases.

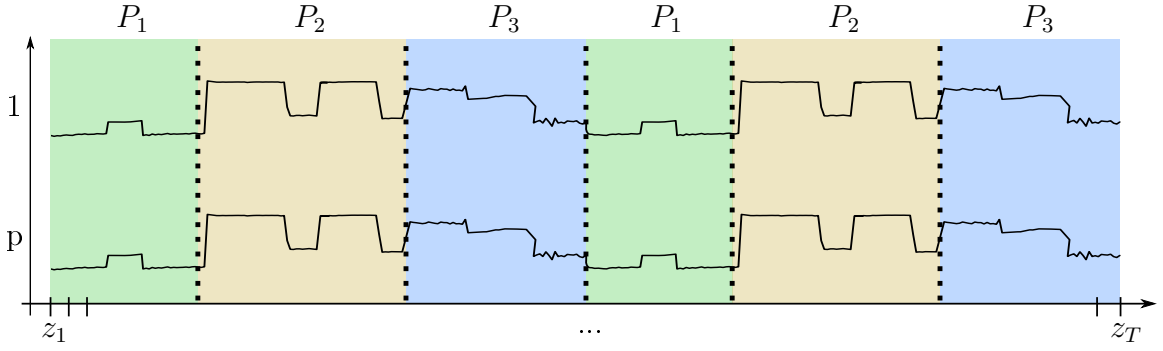
The correct choice on the number of clusters for k-means – the number of fine-grained phases – is also found via BIC. We run k-means for a range from 1 to $fMaxK$ (fine-grained maximum K) and pick the clustering with at least $fBIC$ percent of the best score. We refer to K'_i as the number of phases in which a coarse-grained phase i was partition into, that is, the number of resulting fine-grained phases of i.

We use SimPoint's [64] model of picking the center of each cluster as the representive interval for simulation, which is known as a simulation point, or SimPoint for short. Formally, for each fine-grained phase $Q_{i,j}$ we pick the interval closest to the cluster center (centroid) for simulation, let $q_{i,j}$ be this point. Detailed simulation is performed at the simulation points and the metric of each interval will be the metric value of all in the intervals in the phase. In our context, each interval $q_{i,j}$ is a simulation point. Each result of the execution metric of $q_{i,j}$ is weighted by $w_{i,j}$ according to the phase coverage size, i.e.,

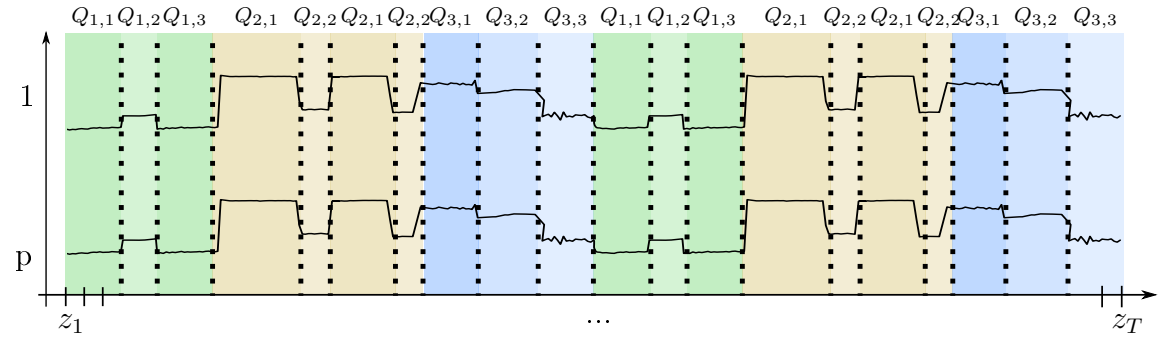
$$w_{i,j} = \frac{|Q_{i,j}|}{T} \tag{4.3}$$

where T is the number of MICA observations (intervals) of a given program.

If we take, for instance, *CPI* as a metric to be evaluated, the *CPI* of the complete execution can be estimated by the following equation:



(a) Coarse-grained phases



(b) Fine-grained phases

Figure 4.5: The relationship between the coarse-grained and fine-grained clustering methods

$$
\begin{aligned}
CPI \approx\ & w_{1,1} \times CPI(q_{1,1}) + w_{1,2} \times CPI(q_{1,2}) + ... + w_{1,k'_1} \times CPI(q_{1,k'_1}) + \\
& w_{2,1} \times CPI(q_{2,1}) + w_{2,2} \times CPI(q_{2,2}) + ... + w_{2,k'_2} \times CPI(q_{2,k'_2}) + \\
& \vdots \\
& w_{K,1} \times CPI(q_{K,1}) + w_{K,2} \times CPI(q_{K,2}) + ... + w_{K,k'_K} \times CPI(q_{K,k'_K})
\end{aligned}
\tag{4.4}
$$

In summary, Figure 4.5a illustrates the second level of sampling with an example of three coarse-grained phases, shown in colors green, yellow, and blue. Figure 4.5b further re-sample each coarse-grained phase into fine-grained, shown by the different tones of the same color.

4.3 Experimental Setup

Our setup is built upon the Pin dynamic binary instrumentation system [49]. It is used to profile the microarchitecture independent characteristics (MICA) and guarantee repeatability. Our analysis requires a program to be run twice. One pass is needed to profile the microarchitecture independent characteristics (MICA). After the profile analysis, another pass is needed to generate the traces of each simulation point. The problem in this approach is that two runs of the same binary may be different as non-determinism should arise from internal sources of the program (e.g., order of shared memory access

and system calls) [54]. This may cause a region description (e.g., instruction count start) to be misaligned from the run used to collect the MICA profile.

PinPlay [54] kit is used to overcome this problem. Pinplay is a tool based on Pin to guarantee deterministic execution of programs. Its main purpose is to overcome non-determinism in parallel programs. Pinplay comprises two Pin tools called *logger* and *replayer*. The *logger* records minimal runtime behavior of a given program execution in a set of files collectively called *pinball*. The *replayer* uses the *pinball* to reproduce the captured execution. A *pinbal* can be created for either the entire execution of a program (a *whole-program pinball*) or for any region of interest (a *region pinball*) [54].

The region *pinballs* allow direct simulation of the regions, instead of fast-forwarding to a given point in execution and then starting the simulation from there. Lastly, the region *pinballs* selected by our multilevel analysis are run in a Pin-based simulator. Figure 4.6 presents an overview of our proposed framework composed of the following five steps:

Trace File Generation – Whole Program *Pinball* We use the PinPlay *logger* to capture the whole program *pinball*.

MICA Profile Collection The MICA features are collected using Pin [50] with the whole-program *pinball* captured in the previous step. A Pin tool for collecting MICA is available in [29]. We modified it to support *pinballs*. The pin tool dumps the MICA state per interval of I dynamic instructions. The states of the interval structures are reset for every interval. The output resulted in this step is a file with a T-by-n matrix, where T is the number of intervals and n is the number microarchitecture-independent characteristic collected per interval of I dynamic instructions. Specifying the parameters of the microarchitecture-independent characteristics is done using a configuration file. Our configuration has $n = 97$ characteristics, as described in Table 4.1.

Discovering Coarse-grained Phases with TICC A Python implementation for the TICC algorithm is available in [26]. It takes as input the T-by-n data matrix generated in the previous step and outputs an array of k cluster assignments for each time point in the form of a dictionary with keys being the cluster assignments (from 0 to k-1) and the values being the cluster MRFs. TICC also returns the BIC score for the resulting clustering. We wrote an application to run TICC for a different number of clusters, from 1 to *cMaxK*, and pick the clustering with the smallest number of clusters, such that its BIC score is at least *cBIC* percent of the best score.

We specify only the three main parameters of TICC: w, which determines the subsequence size, λ, which determines the sparsity level in the MRFs characterizing each cluster, and β, the smoothness penalty that encourages adjacent sub-sequences to be assigned to the same cluster [27]. If necessary, TICC solver also takes as a parameter the maximum number of iterations of the TICC algorithm before convergence and the convergence threshold.

Discovering Fine-grained Phases with k-means We wrote a python application to do the fine-grained analysis. We use the *Sklearn* [55] library to provide k-means. In

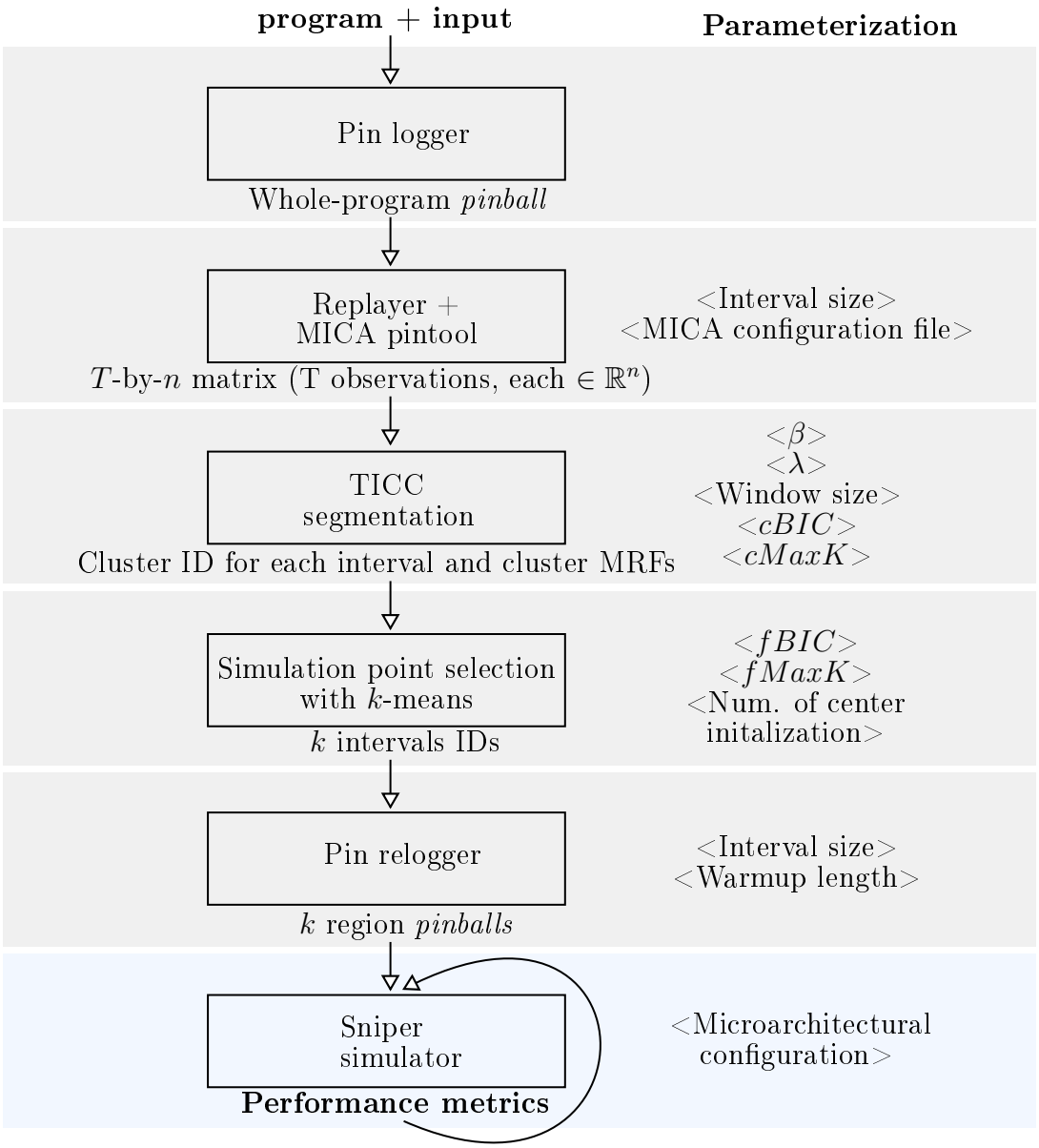


Figure 4.6: Overview of Experimental setup

this work, it takes the MICA profile for all the intervals and the TICC segmentation from previous steps and outputs the interval index for the selected simulation points and weights.

We try a different number of clusters: from 1 to *fMaxK*, where *fMaxK* the maximum number of phases to split each coarse-grained phase. We also use the BIC score to choose the clustering with the smallest number of clusters, such that its BIC score is at least *fBIC* percent of the best score. Last, we pick the center interval of each cluster outputted by k-means as a simulation point. Our BIC algorithm was based on the implementation used by *pyxmeans* [31]. Another parameter we need to specify is the number of random initializations to try for clustering each k. k-means algorithm has a local minima problem, thus different initial centers of clusters may produce different results. One solution is to run k-means repeatedly with different

centroid seeds.

Trace Simulation Points The last step in the phase classification process (required once for a program/input pair) is to generate instructions traces of each simulation point. We first take the interval index from the previous step and generate the instruction counts for each region, which is the region description required by Pinplay. We then use the Pinplay kit to trace the selected regions into region *pinballs*. Pinplay takes the region description and relogges the whole-program *pinball* to selective log just the regions of interest.

As region *pinballs* allow checkpoint-based simulation, it also desirable to specify a number of extra instructions to be included before each region, also known as warmup length.

Simulation and Whole-program Behavior Prediction We used the Sniper simulator [6] in our experiments. It is Pin-based cycle-accurate x86 simulator and allows the execution of *pinballs*. We wrote a python application to compute the weighted average of the simulation points, which gives the metric of interest estimate of the complete execution for a program/input pair.

4.4 Summary

In this chapter, we presented our method of phase program classification with a focus on efficient simulation. We summarize the main steps of our method in Figure 4.7.

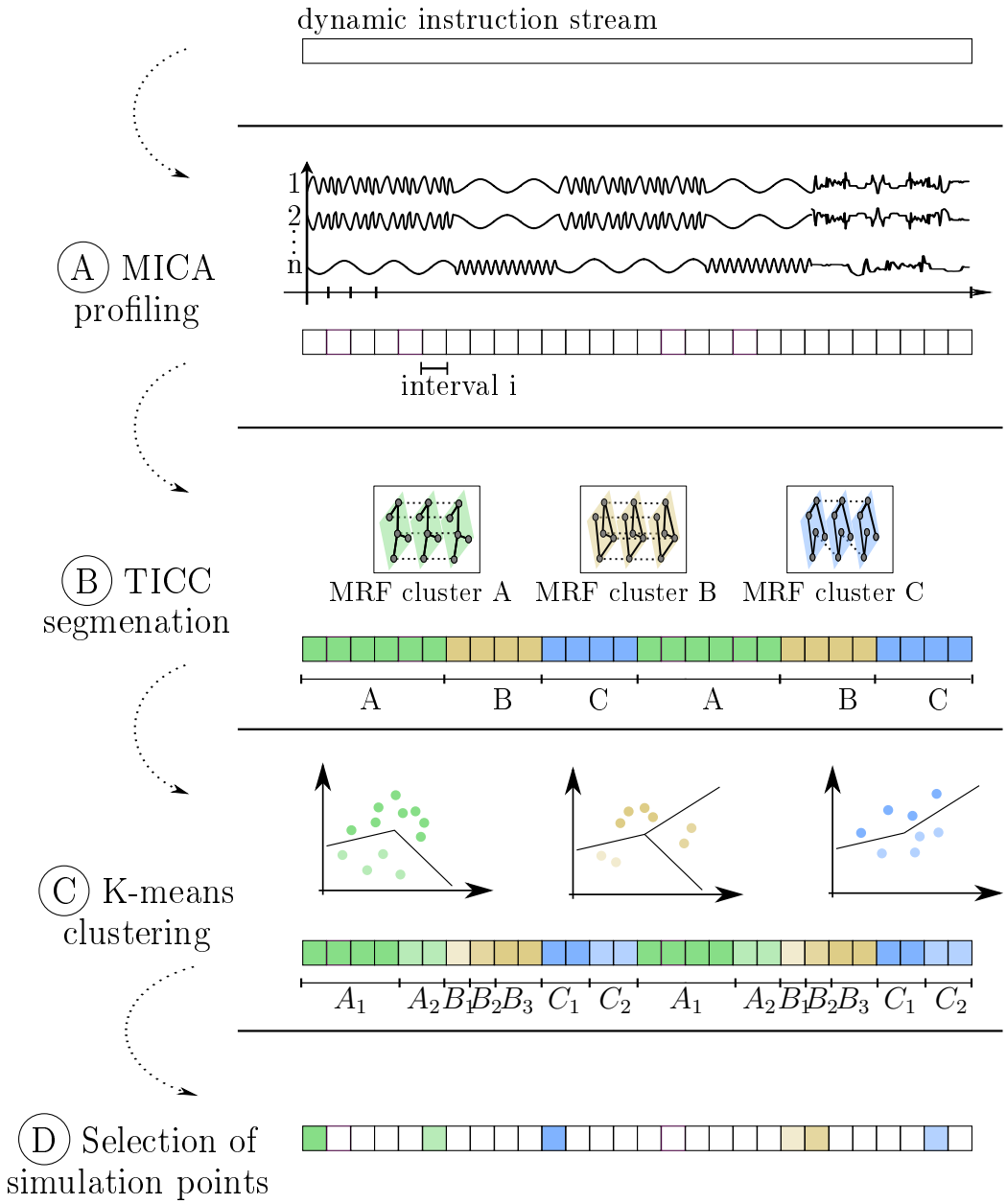


Figure 4.7: Overview of our phase classification approach

Chapter 5

Evaluation

In this chapter, we evaluate our method for automatically characterizing time-varying behavior in programs.

5.1 Evaluation Methodology

We evaluate our proposal using SPEC 2006int benchmarks with reference inputs (34 program-input pairs). The MICA features are 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 retain the principal components which explained 90% of the original data set's total variance.

The performance metrics were extracted assuming a perfect warm-up, as the focus in this chapter is on the error derived from sampling. 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 get the required metric values for every interval outputted by our phase analysis.

Processor	Nehalem-lite
Dispatch Width	4
Window Size	128
L1 I-Cache	32 KB, 4-way set-associative
L1 D-Cache	32 KB, 8-way set-associative
L2 Cache	256 KB, 8-way set-associative
L3 Cache	8 MB, 16-way set-associative
Branch predictor	Pentium M, 8 cycles penalty

Table 5.1: Baseline simulation model

5.2 Metrics for Evaluating Phase Classification

The metrics used in our study are:

Number of coarse-grained phases: The number of phases found by TICC.

Average phase length: The average number of dynamic instructions per phase.

Number of fine-grained phases: Number of fine-grained phases used to characterize each TICC phase. Each fine-grained phase has a corresponding simulation point, which directly relates to simulation time.

Coefficient of Variation (CoV): Coefficient of Variation (CoV) is a commonly used metric for evaluating phase classification techniques [46]. If a phase is a set of intervals within a program's execution with similar behavior, then it is expected that all intervals in a phase behave similarly (e.g., IPC, cache miss rates, branch miss rates). CoV quantifies the variability of interval performance behavior inside a phase – the homogeneity of the phases.

To compute the CoV of a metric of interest, we first calculate, for each phase, the average and the standard deviation in the metric of all intervals in the phase. We then divide the standard deviation by the average to get the per-phase CoV. Finally, the whole program CoV is computed as the weighted average of the per-phase CoVs, which is formally defined in Equation (5.1):

$$CoV = \sum_{p \in phases} \frac{\frac{\sigma_p}{\mu_p} intervals_p}{totalintervals} \tag{5.1}$$

One issue about CoV is that when the mean value is close to zero, it will approach infinity. As we are measuring L3 misses, we often found phases with an average number of misses less than one. This problem is discussed in [34] and they suggested the use of weighted standard deviation, which is the metric we used for L3 misses and it is calculated using the formula in Equation (5.3).

$$\sigma_{weighted} = \sum_{p \in phases} \frac{\sigma_p intervals_p}{totalintervals} \tag{5.2}$$

Mean relative error (MRE): The percent difference between predicting a metric using only simulation points and the true metric value of the complete execution of the program:

$$\text{Approximation error} = \frac{\text{True Metric Value} - \text{Estimated Metric Value}}{\text{True Metric Value}} \tag{5.3}$$

5.3 Coarse-grained (TICC) Program Phases

We first present a graphical visualization of the coarse-grained phases for three different programs from SPEC 2006: *gcc*, *bzip2* and *astar*. The top part of Figures 5.1 to 5.3 shows the time-varying of these programs for several architectural metrics: number of cycles,

Parameter	Value
Number of clusters K	5 and 10
Window size w	1, 2, 4, 8, 16, 32
β regularization	0, 256, 512, 1024, 2048, 4096, 8192
λ regularization	0.1, 0.2, 0.4, 0.8

Table 5.2: Set of TICC parameters investigated (336 configurations)

branch miss rate, L1-D, L2, and L3 cache miss rate; the bottom part shows the values for the principal components retained with explained variance greater than 90% applied to the MICA characteristics. Each interval of 160M dynamic instructions is a point in the graph. The color represents the cluster assignment from TICC algorithm. These graphs clearly show that TICC is able to find phase as a unit of repeating behavior rather than a unit of uniform behavior. We can also notice a correlation between MICA characteristics and overall performance. The parameters used for TICC are window size of 16, $\beta = 3000$, and $\lambda = 0.2$.

We made available online[1] all the plots for all the TICC parameters described in Table 5.5 and SPECint program/input pairs.

5.3.1 TICC Parameter Exploration

Here, we present an investigation into the parameters of TICC for phase classification. As discussed in Chapter 3, TICC method has three main parameters: w, which determines the sub-sequence size, λ, the sparsity level in the MRFs characterizing each cluster, and β, the smoothness penalty that encourages adjacent sub-sequences to be assigned to the same cluster [27]. We performed a grid search for these parameters to find a pseudo-optimal set of configuration points. Our goal is to find parameters that result in a fairly homogeneous set of phases. The values we varied are shown in Table 5.2.

We evaluate each set of parameters with the Coefficient of Variation (CoV) of the resulting TICC segmentation, as the goal of our phase classification is to have segments belonging to the same cluster (program phase) with similar behavior. We measure this similarity by analyzing the overall performance of each segment belonging to a cluster. If all segments in the same phase have precisely the same value of performance, then the CoV will be zero. The formula for computing CoV is shown in Equation (5.4):

$$CoV = \sum_{p \in phases} \frac{\frac{\sigma_p}{\mu_p} segments_p}{phases} \tag{5.4}$$

We examine the CoV of the number of cycles, branch miss rate, L1-D, L2, and L3 cache miss rate for *gcc.166* with 5 and 10 clusters. The output CoV for each TICC configuration is shown in Figure 5.4, with normalized values between 0 and 1 with respect to the "min" and "max" values. A larger value of CoV corresponds to a poorer segmentation result. The gray cells are values where TICC could not converge to a solution. This happens because TICC is a non-convex problem, so there is no way to guarantee to reach the globally

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

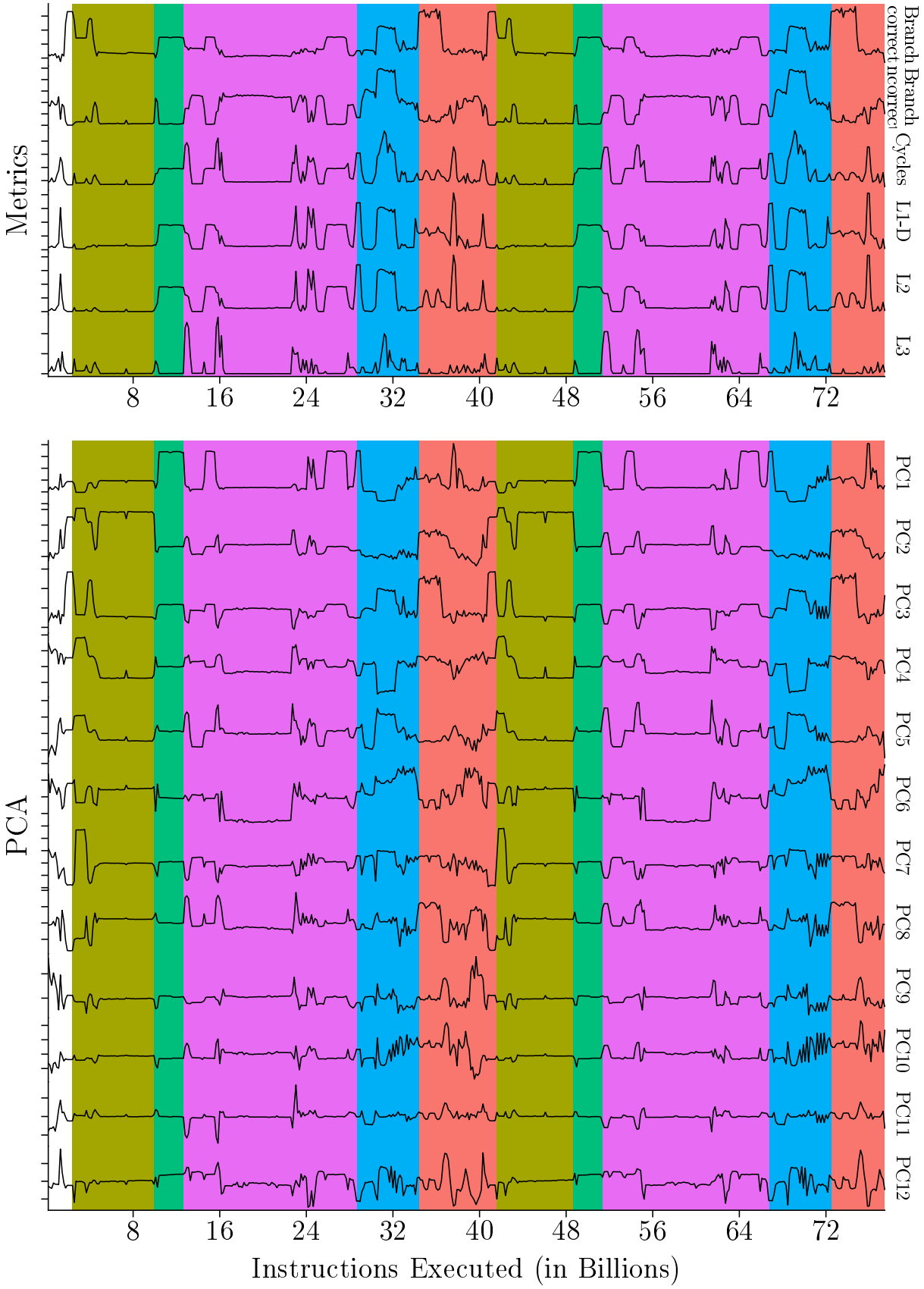


Figure 5.1: Coarse-grained phases for *gcc* with input *166* (403.gcc.1)

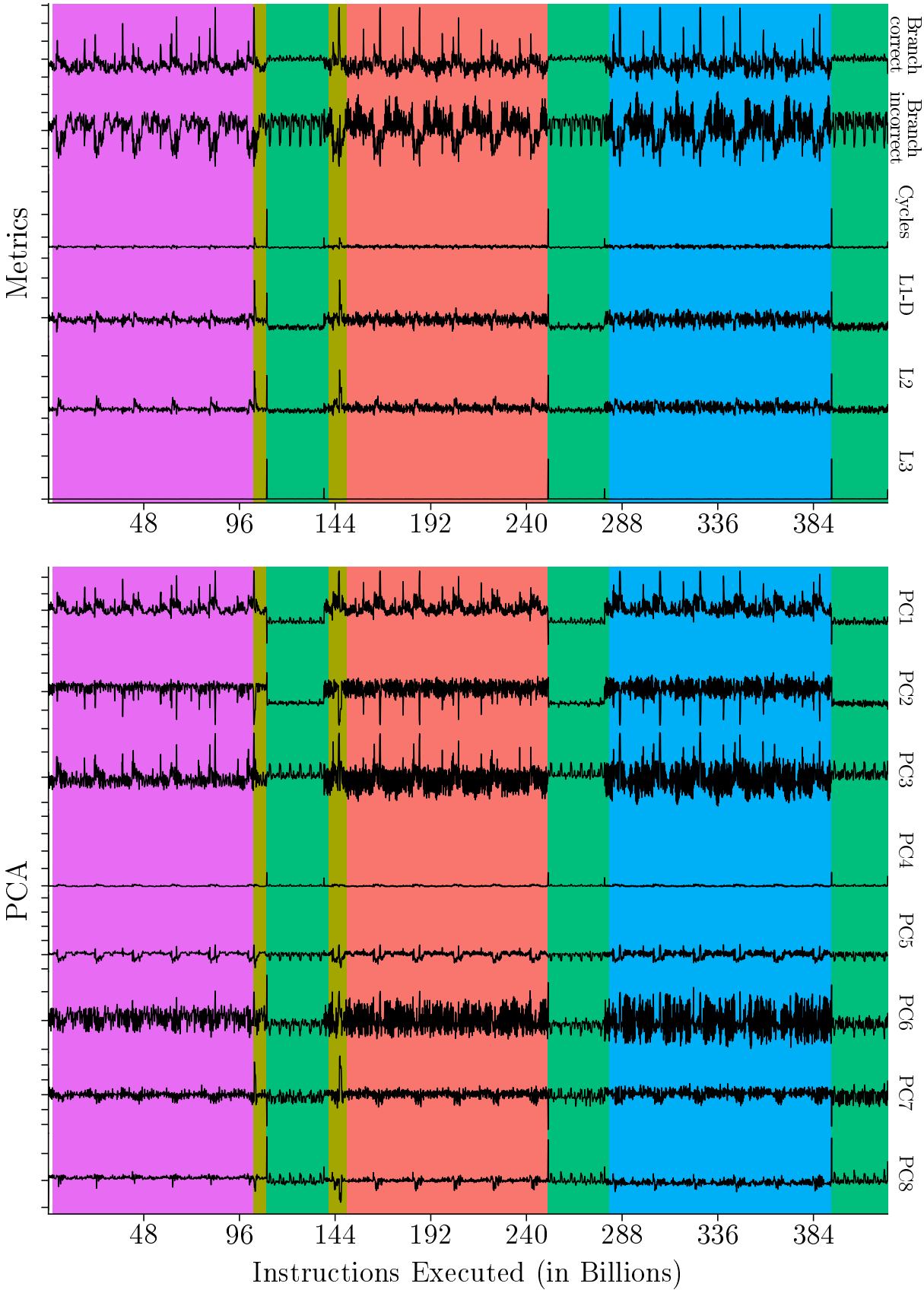


Figure 5.2: Coarse-grained phases for bzip2 with input *source* (401.bzip2.1)

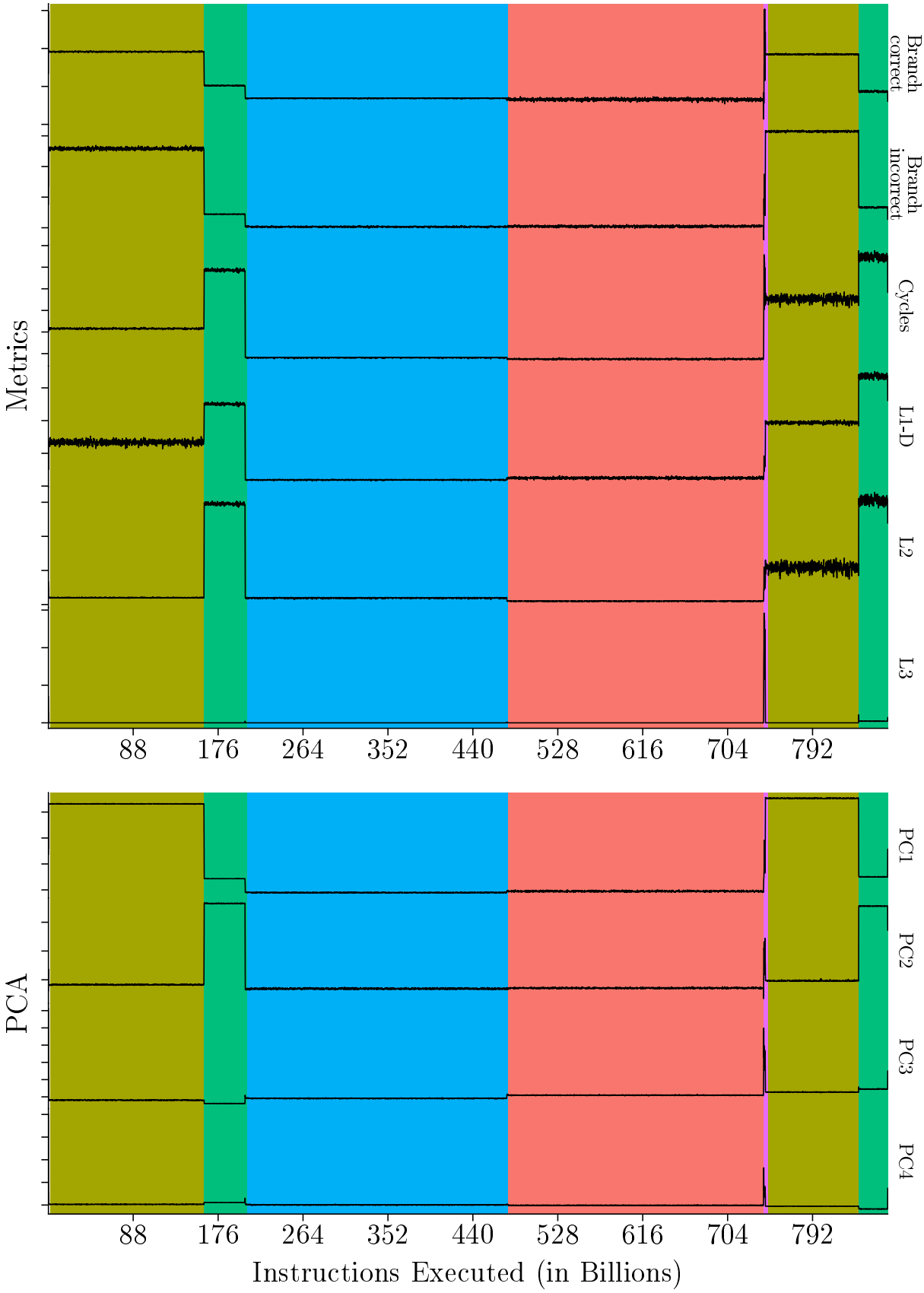


Figure 5.3: Coarse-grained phases for *astar* with input *rivers.cfg* (473.astar.2)

optimal solution, and occasionally it gets stuck in a local minimum that it cannot get out of it [27]. This also suggests that those parameters are not good configuration candidates.

Empirically, we discover that TICC is robust to the selection of λ and the w. Instead, the critical parameters are β and the number of clusters K. The value of CoV decreases as we increase β. However, this increases the probability of not converging to a solution when w is smaller. Similarly, higher values of λ increase the probability of converging to a solution. Based on these results, we found that β between 2048 and 4096, window size between 8 and 16 yield a good set of candidate configurations.

5.3.2 The Ratio of Unique Behavior to Overall Execution

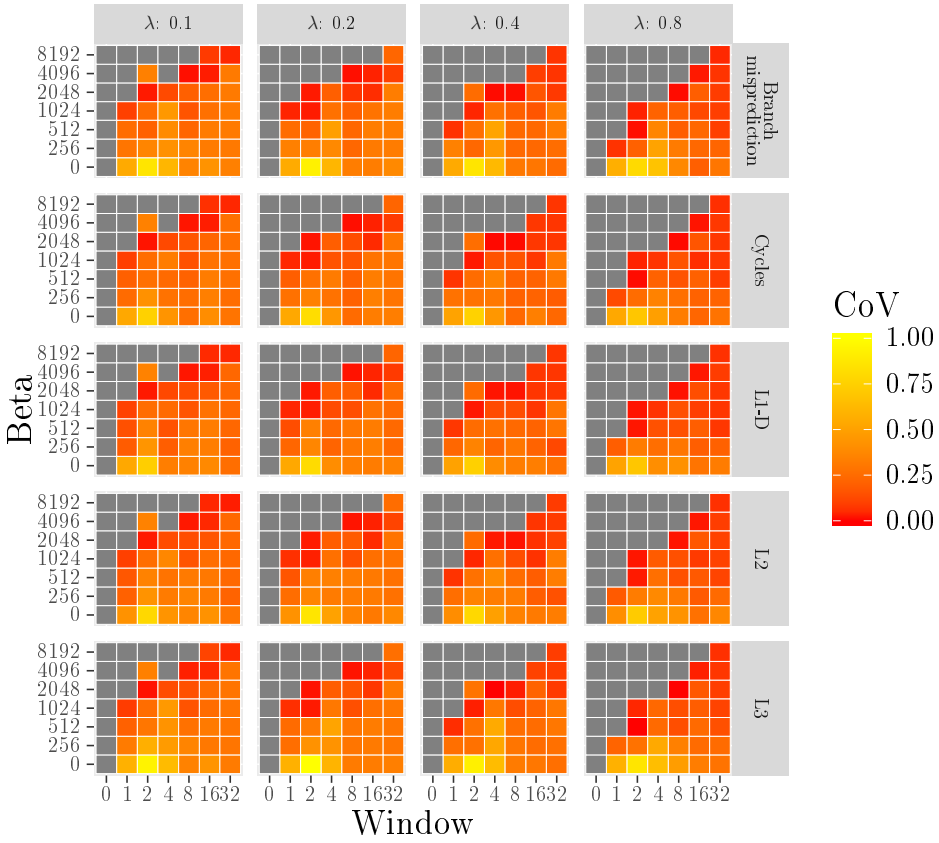
Here we analyze the proportion of unique behavior of a program. The main question we are trying to answer is: if we took a single segment (one occurrence of the phase) from each program phase found by TICC, how much of the complete execution that would represent? As stated earlier, a program's execution is partitioned into a sequence of program phases, where each phase is a unique state (behavior) of the program and may repeat itself across the execution.

We define the size of a coarse-grained program phase as the average number of dynamic instructions of the segments belonging to that phase. We arrive at an average proportion of unique behavior in each program by summing up the program phase sizes, and then dividing by the total number of dynamic instructions of the complete execution. This results in an overall metric that shows the level of repetition of the program phases. Table 5.3 show this analysis for all the SPECint CPU2006 benchmarks. The columns are the range of BIC score from 0.1 to 1. Each cell value shows the average proportion of the whole program execution. The parameters used for TICC are window size of 12, $\beta = 2000$, and $\lambda = 0.2$.

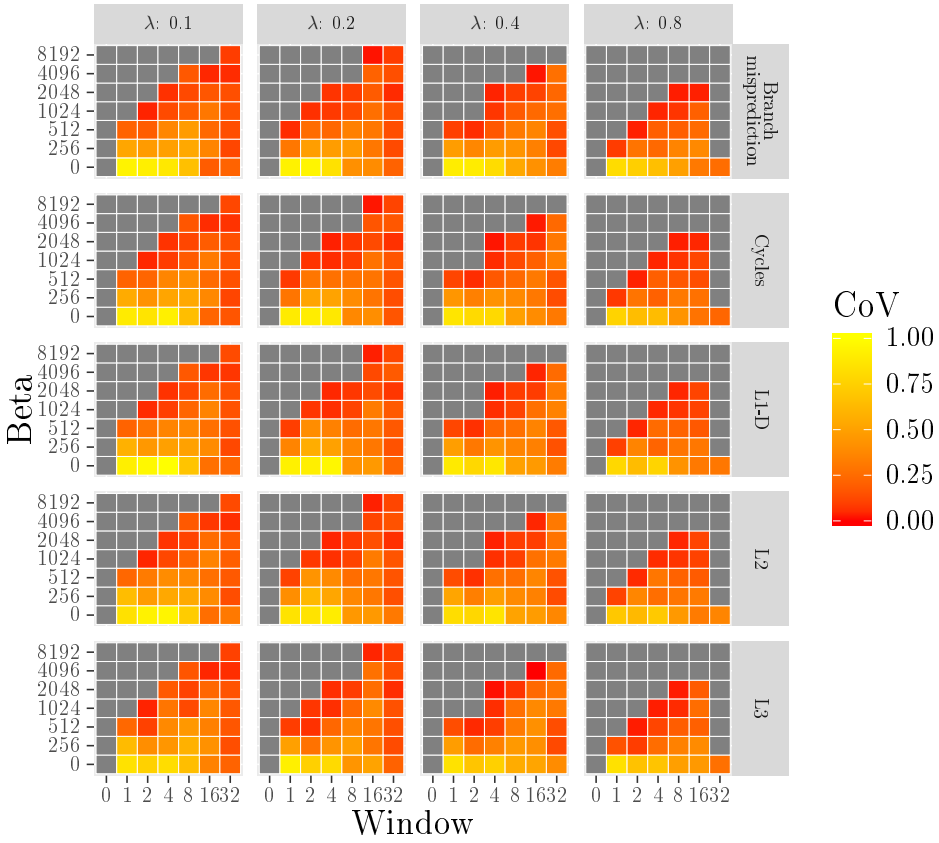
Higher values indicate that the phases of such a program do not reoccur. Returning to Figures 5.1 to 5.3, the segmentation of *astar.rivers* in Figure 5.3 did not result in any reoccurring phase (all phases occur once), which is shown in Table 5.3 by a value close to 100%. On the other hand, each phase in *gcc.166* occur twice, which results in approximately 50% of the whole program execution.

If we were to pick a single representative instance (or segment) of each program phase found by TICC for sampled simulation, those values would represent the proportion of the whole program execution needed for simulation. In other words, the last line of Table 5.3 gives the average proportion of the SPECint 2006 to be simulated if we were to pick a single instance of each coarse-grained phase. Thus, it would be necessary for the simulation of 45% to 55% of the total dynamic instructions, which represent, in absolute values, a high number of 9906 to 12698 trillion instructions. Figure 5.5 shows the dynamic instruction count of each program in the SPECint CPU 2006 suite.

As the focus of this work is on efficient simulation, this also motivates the need for a method to represent each coarse-grained phase with fewer instructions, which is solved by our second-level sampling strategy (Section 4.2.4). We focus again on the analysis of coarse-grained phases in Section 5.5, where we focused on interpreting each program phase based on its MRF structure.



(a) $K = 5$



(b) $K = 10$

Figure 5.4: Heatmap of the CoV produced by candidate TICC parameters

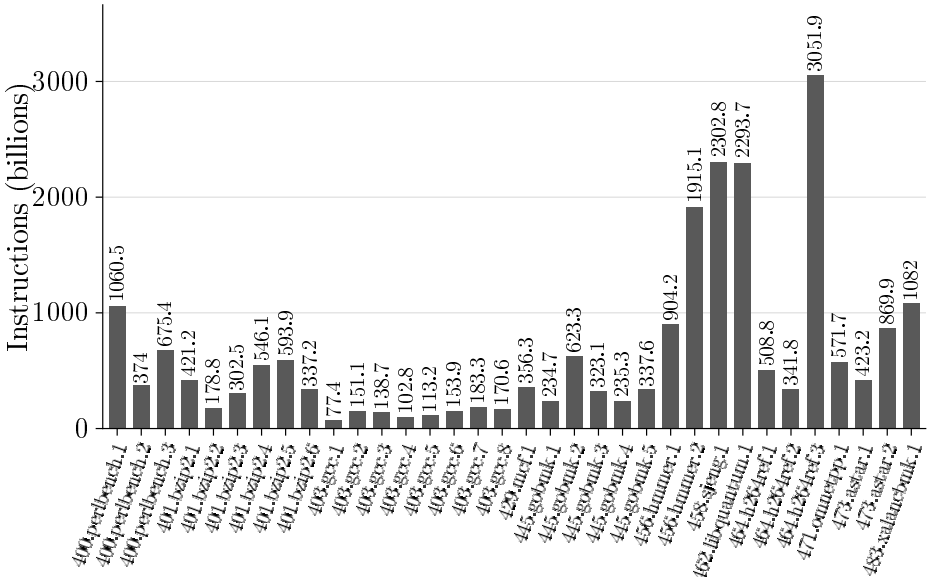


Figure 5.5: Dynamic instruction count of SPEC CPU 2006 integer benchmarks

Program/BIC score	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
400.perlbench.1	10.0	10.0	10.0	10.0	10.0	11.0	11.0	13.0	13.0	13.0
400.perlbench.2	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.9	18.9
400.perlbench.3	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	17.6
401.bzip2.1	59.7	55.8	55.8	55.8	65.1	39.8	39.8	39.8	62.2	62.2
401.bzip2.2	74.5	76.5	76.5	76.5	76.5	86.9	91.1	91.1	83.0	83.0
401.bzip2.3	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0	65.0
401.bzip2.4	33.4	33.4	33.4	33.4	33.4	33.4	33.4	33.4	33.4	83.7
401.bzip2.5	46.7	46.7	46.7	46.7	46.7	46.7	46.7	46.7	46.7	46.7
401.bzip2.6	14.7	14.7	14.7	15.6	14.3	14.3	16.1	16.5	25.0	25.0
403.gcc.1	37.7	37.7	51.4	54.1	49.4	49.4	63.4	63.4	67.3	67.3
403.gcc.2	54.0	55.2	55.2	76.0	51.1	65.6	84.1	84.1	81.1	81.1
403.gcc.3	44.4	44.4	46.7	55.7	55.7	68.1	66.0	67.1	71.2	72.8
403.gcc.4	53.9	49.7	49.7	49.7	74.0	74.0	74.0	67.1	67.1	71.5
403.gcc.5	41.3	48.3	47.9	58.3	61.3	62.7	75.4	76.6	76.5	83.5
403.gcc.6	31.4	42.4	52.3	52.5	52.5	74.1	74.8	76.6	72.0	77.9
403.gcc.7	39.8	39.8	45.7	71.1	78.2	77.9	77.9	80.1	80.4	85.1
403.gcc.8	33.9	33.9	33.9	53.8	53.8	59.1	69.6	70.9	72.3	74.3
429.mcf.1	20.0	20.0	20.0	20.2	20.2	20.2	20.2	20.2	28.3	36.1
445.gobmk.1	29.5	29.5	53.4	55.3	55.3	55.3	50.1	54.1	74.6	74.6
445.gobmk.2	18.3	18.3	31.1	31.1	39.6	43.3	51.9	53.2	60.6	60.6
445.gobmk.3	39.9	43.1	46.6	40.1	40.1	51.5	57.5	57.5	57.6	58.7
445.gobmk.4	27.4	52.1	64.8	63.6	53.3	53.3	57.5	62.4	62.4	62.4
445.gobmk.5	38.8	44.8	60.6	53.6	54.9	69.8	57.6	57.6	74.9	74.9
456.hmmer.1	20.6	20.6	20.6	20.6	20.6	20.6	20.6	33.6	33.6	33.1
456.hmmer.2	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
458.sjeng.1	17.5	17.5	31.7	31.7	20.9	20.9	20.9	20.9	20.9	20.9
462.libquantum.1	99.7	46.6	99.7	99.7	99.7	99.8	100.0	100.0	100.0	100.0
464.h264ref.1	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
464.h264ref.2	50.5	50.5	50.5	50.5	50.5	50.5	50.5	50.5	50.5	50.5
464.h264ref.3	1.4	1.4	1.4	1.4	1.4	1.5	1.5	1.5	1.5	1.9
471.omnetpp.1	99.1	99.1	99.1	99.5	99.5	99.5	99.5	99.5	100.0	100.0
473.astar.1	50.0	90.3	90.3	90.7	90.7	99.4	99.4	99.4	99.4	99.4
473.astar.2	80.8	80.8	80.8	80.8	80.8	99.8	99.6	99.6	99.6	99.6
483.xalancbmk.1	91.2	55.1	55.1	55.1	64.1	64.4	63.4	63.4	63.4	63.4
Coverage (%)	47.0%	40.9%	49.1%	49.6%	49.2%	50.7%	51.2%	52.0%	53.3%	55.2%

Table 5.3: Ratio of unique behavior to overall execution for the SPECint 2006 benchmarks

5.4 Sampled Simulation

This section presents results guided by the sampling error of our method. We begin by exploring the appropriated proportion of coarse-grained and fine-grained phases. Then we present the results we gathered from applying our sampled simulation methodology to the SPECint programs for a set of TICC parameters and BIC configurations. Finally, we compare our work with two other sampled simulation techniques.

5.4.1 Appropriate Number of Coarse-grained and Fine-grained Phases

One of the main parameterization issues in our work is to find the appropriate number of cluster K on both clustering algorithms TICC (first stage of sampling) and k-means (second stage of sampling). As stated in the previous chapter, we use the Bayesian Information Criterion (BIC) score to pick the value of K in both stages. In summary, we look at the BIC score for a range of K's and select the one with a score that achieves some percentage, or BIC threshold, of this range. This results in four parameters to be manually defined:

- *cMaxK*: The maximum number of clusters in TICC.
- *cBIC*: The BIC threshold of TICC.
- *fMaxK*: The maximum number of clusters in k-means.
- *fBIC*: The BIC threshold of k-means.

We show the effect of changing these four parameters on the error and weighed standard deviation for L3, and the resulting number of simulation points. For the coarse-grained phases, we examine values for the maximum number of clusters of 5, 10, 15 and examine BIC thresholds of 10% to 100%. For fined-grained phases, we examine the values for the maximum number of clusters 10, 20, 30, and 40, with BIC threshold of 70%, 80%, and 90%. Table 5.4 summarizes the search space. For this experiment, we used a TICC configuration with $w = 12$, $\beta = 2000$, and $\lambda = 0.2$.

Figures 5.6 to 5.8 plot the results for each value of $cMaxK$. Each point is an average of 20 executions for all the SPECint 2006 programs. In both levels, the number of simulation points increases as the BIC threshold increases, and similarly the average error and weighted deviation decreases. In order to find a subset of parameters in Table 5.4 that result in a good trade-off between accuracy and simulation speed, in Figure 5.9 we aggregate the results across all parameters that resulted in a maximum of 1500 simulation points and reported L3 error of less than 6%.

5.4.2 Evaluation

In Sections 5.3.1 and 5.4.1 we explored, respectively, two main parametrizations in our method: the TICC parameters and the appropriated number of coarse-grained and fine-grained phases. In this section, we present the main evaluation of our phase analysis for the

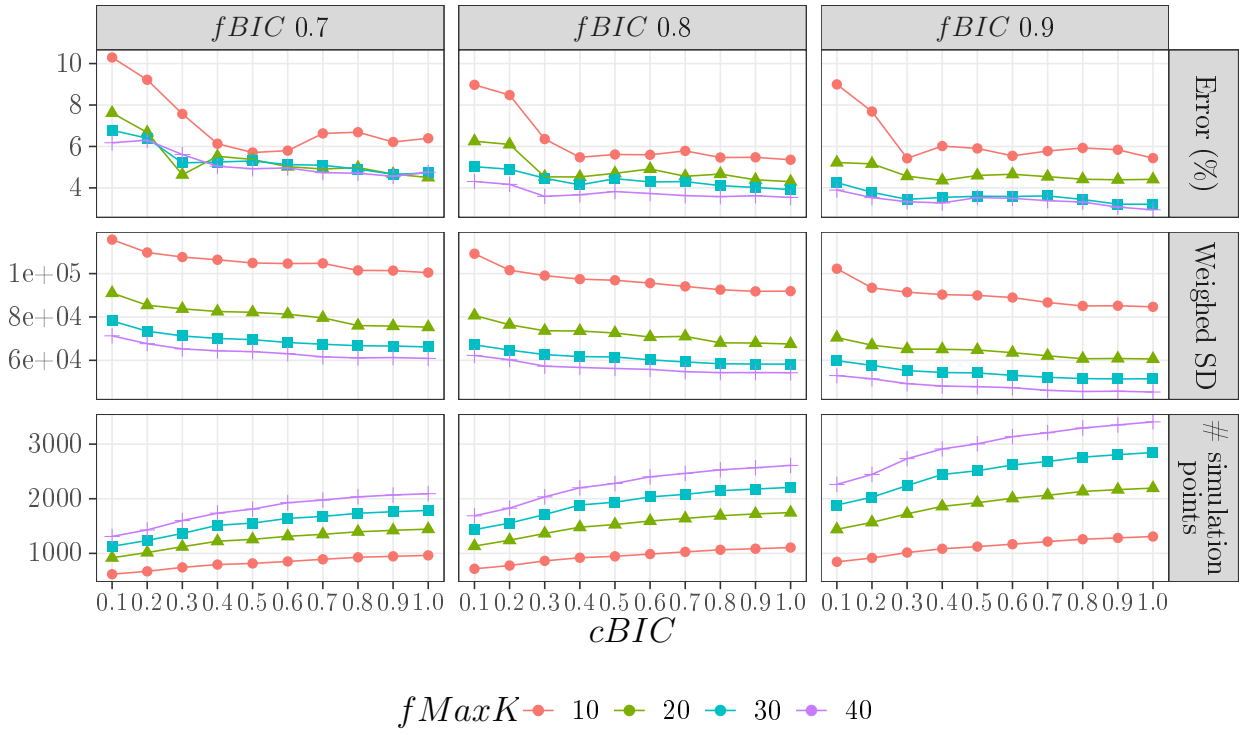


Figure 5.6: Sampled simulation results with $cMaxK = 5$

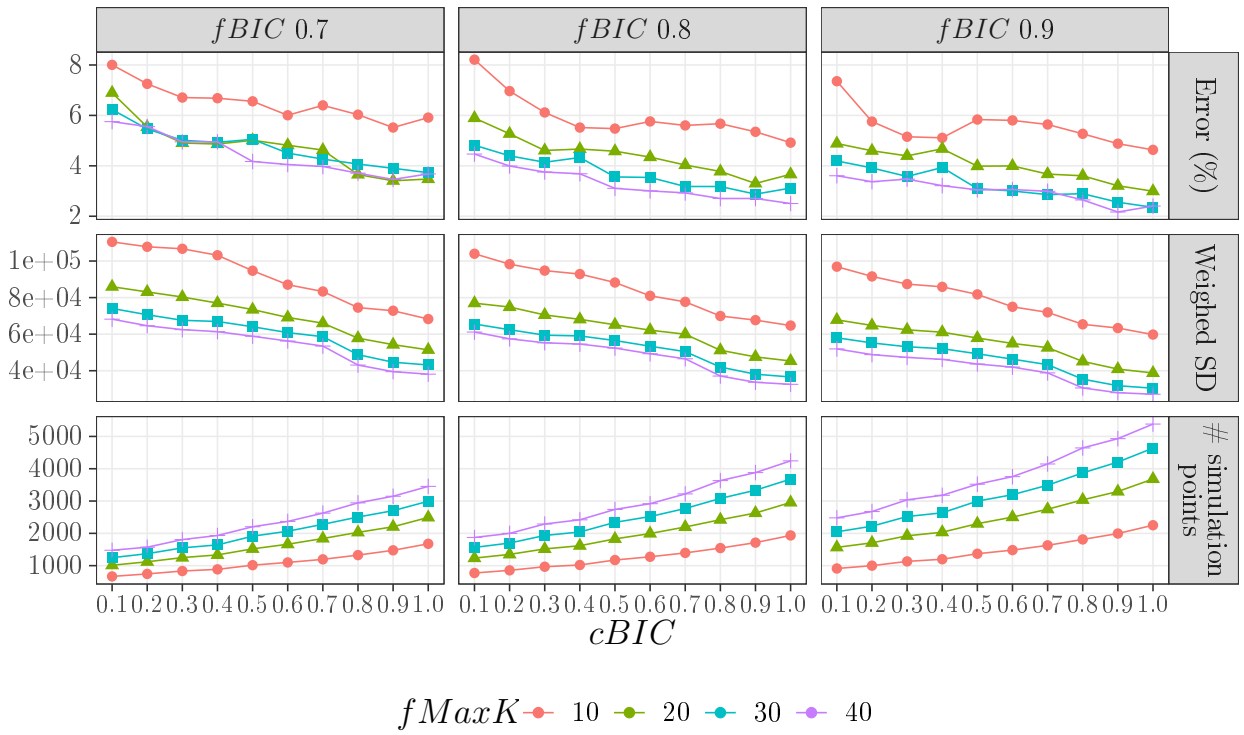


Figure 5.7: Sampled simulation results with $cMaxK = 10$

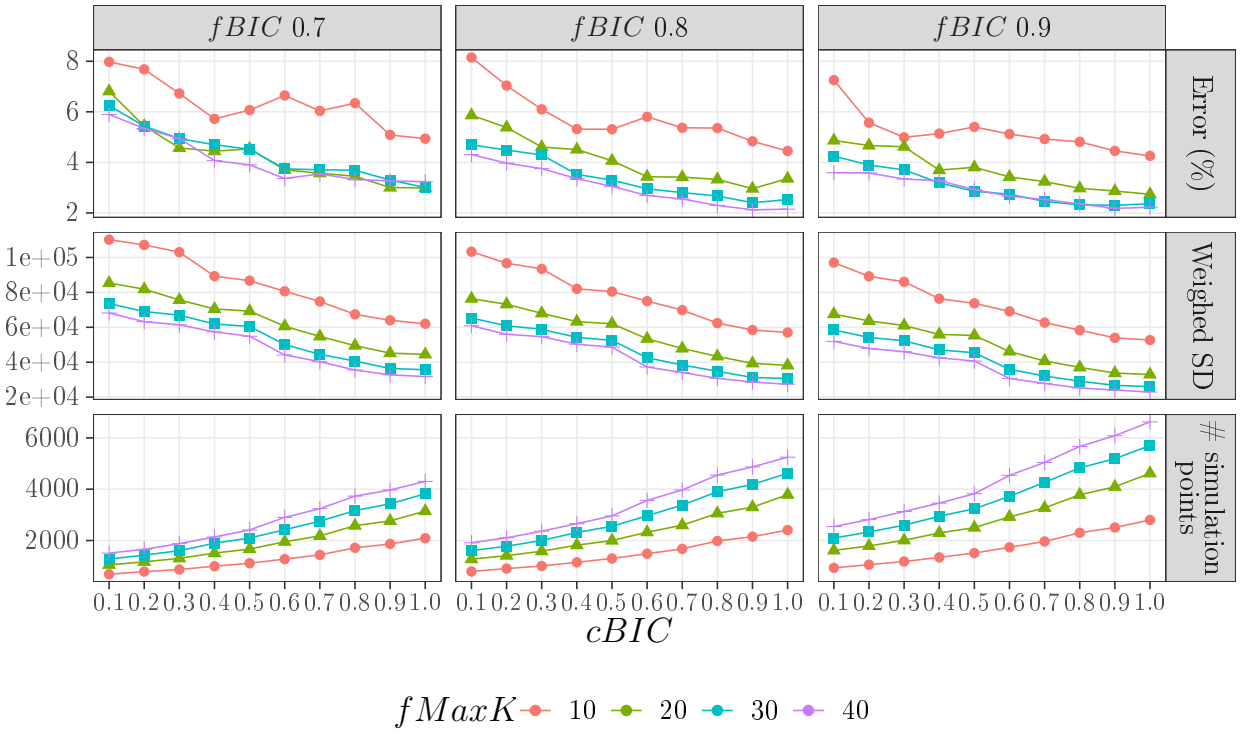


Figure 5.8: Sampled simulation results with $cMaxK = 15$

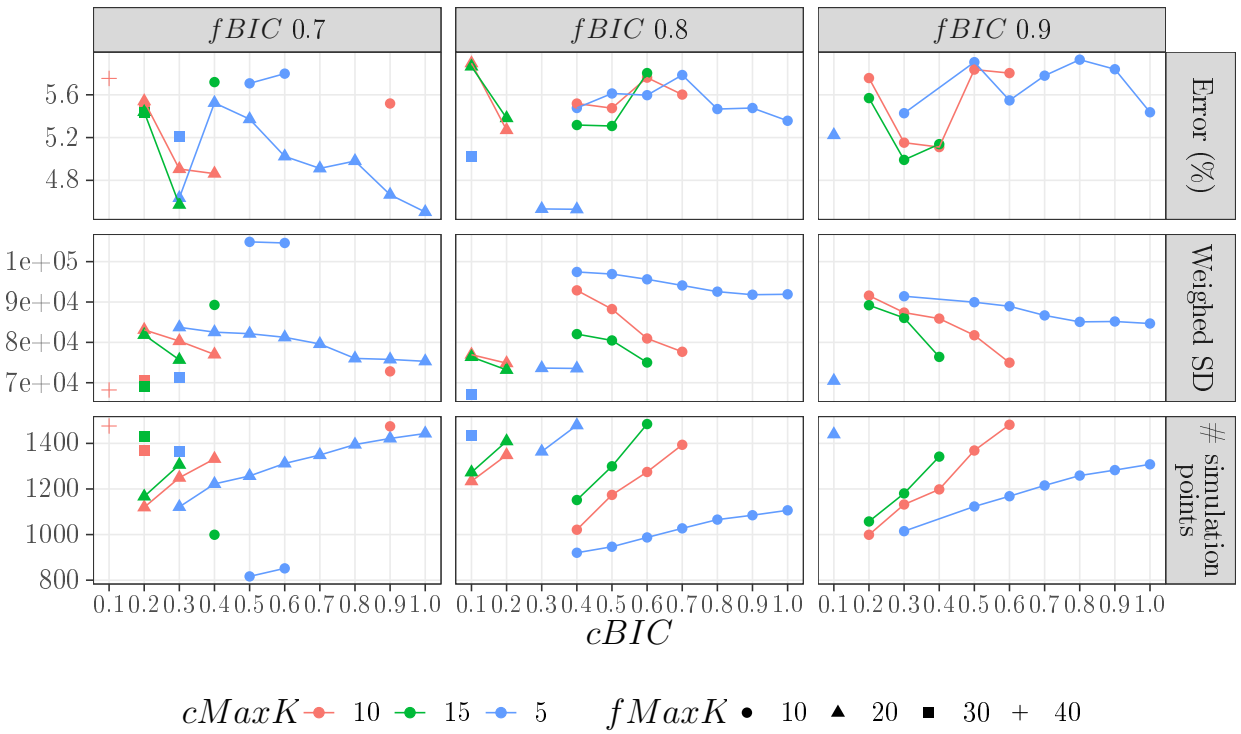


Figure 5.9: Parameters with less than 1500 simulation points and error less than 6%

Parameters	Values
$cMaxK$	5, 10, 15
$cBIC$	0.1 to 1.0
$fMaxK$	10, 20, 30, and 40
$fBIC$	0.7, 0.8, and 0.9

Table 5.4: Maximum number of clusters and BIC score investigated for TICC and k-means

Parameters	Values
$cMaxK$	5
$cBIC$	0.8, 0.9, 1.0
$fMaxK$	10, 20
$fBIC$	0.8, 0.9, 1.0
w	8, 12, 16
β	2000, 3000, 4000
λ	0.1, 0.2, 0.3

Table 5.5: Set of TICC parameters investigated for the sampling simulation results (486 configurations)

parameters found in those sections. In Figures 5.10 to 5.12 we present the mean relative error for cycles, L1-D, L2, and L3, and the number of simulation points chosen. Each plot shows a single value of λ. Inside the figures, each group of bars represents a different configuration of w and β. Each bar (color) represents a value of $cBIC$, $fMaxK$, $fBIC$, and its value is the arithmetic mean of 20 executions of the SPECint 2006 programs. All the parameters we vary in this study are shown in Table 5.5.

Picking K in Second-stage Sampling Proportional to Phase Size and Dispersion

One drawback of the second-stage sampling is that since it uses k-means, it requires the number of clusters K beforehand. So far, we run k-means for several values of K, and then picked values based on a measure of goodness of fit (e.g., BIC). One consequence of this approach is that, for a given TICC segmentation, the greater the BIC score, the more simulation points will be chosen. However, we observed that in some situations, this does not necessarily imply a significant increase in accuracy. This may happen because some times we are giving more simulation points to a coarse-grained that does not have a great impact on the estimation of the whole metric execution.

We now present a preliminary study on an alternative strategy to pick the number of fine-grained clusters. We improved this situation by picking a value of K on a per-phase basis, proportional to the coarse-grained phase homogeneity and size. We measured the homogeneity using the interquartile range of the first principal component of the MICA features. The basic idea is that a homogeneous and small phase should get fewer simulation points than a bigger and more heterogeneous phase.

We used this information to control the threshold of $fBIC$ with respect to the coarse-grained phase i as follows:

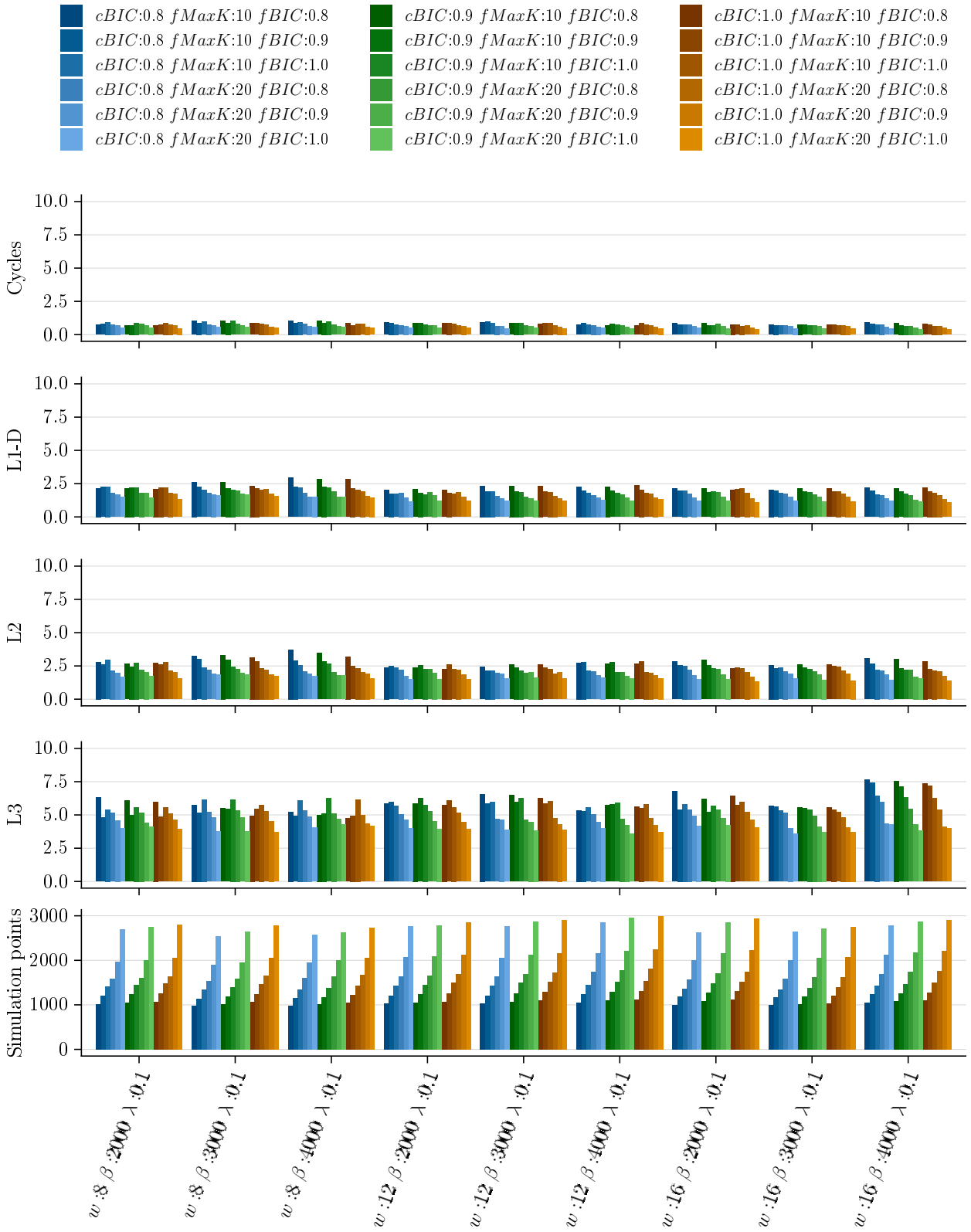


Figure 5.10: Average sampling error and simulation points with fixed λ: 0.1

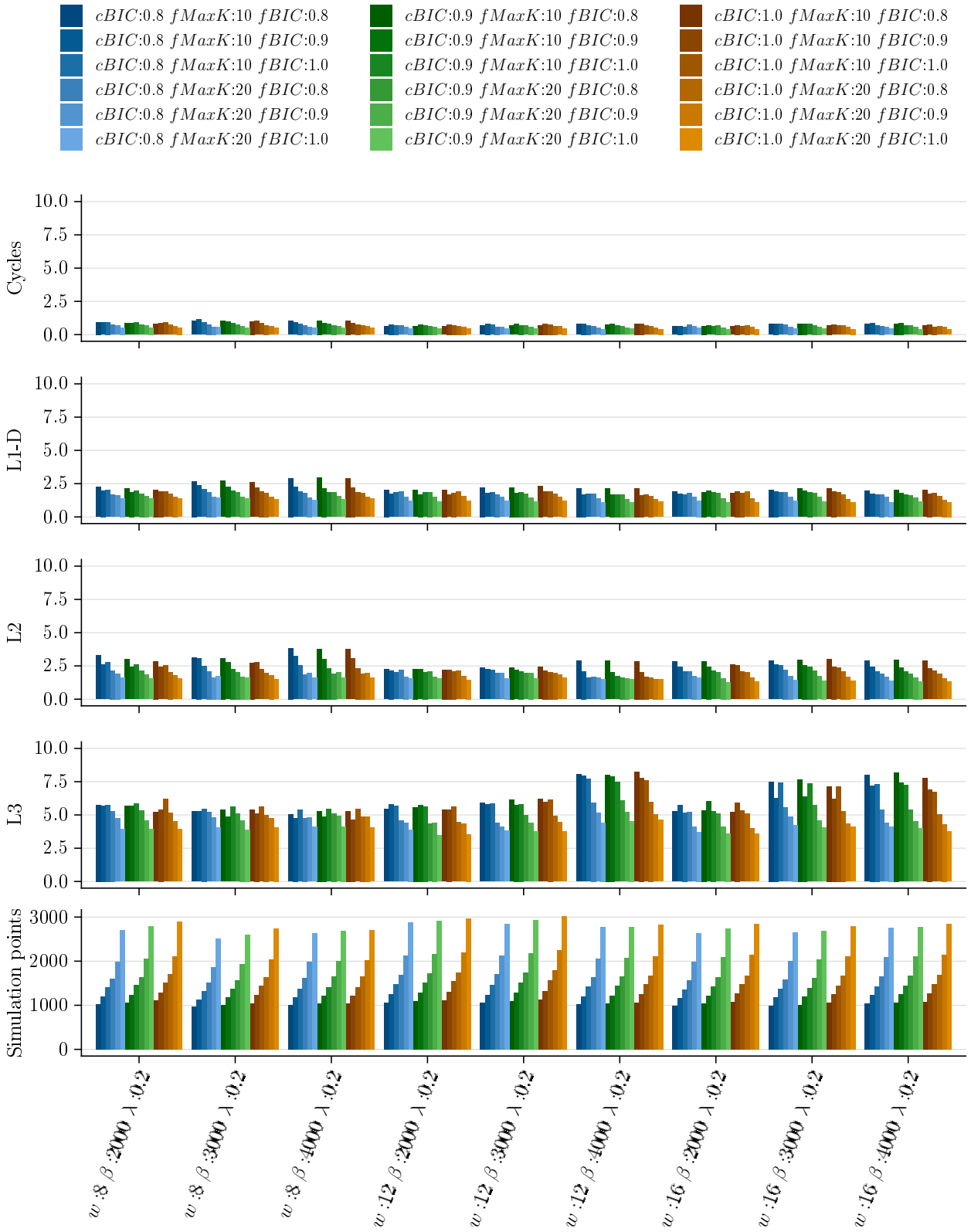


Figure 5.11: Average sampling error and simulation points with fixed λ: 0.2

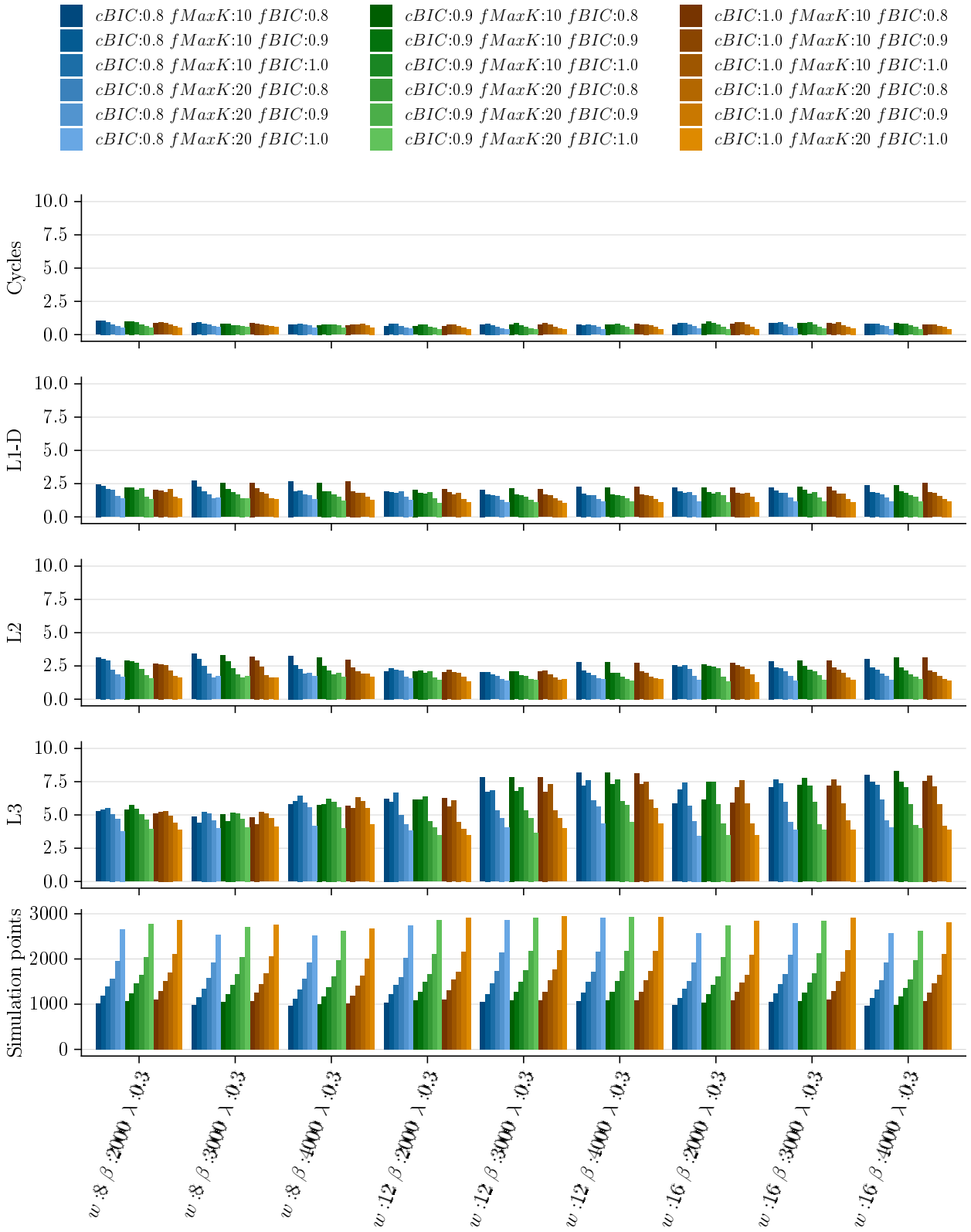


Figure 5.12: Average sampling error and simulation points with fixed λ: 0.3

$$fBIC_i = \min(1, coverage(phase_i) + 2 * IQR(phase_i) + 0.4) \quad (5.5)$$

where $coverage(phase_i)$ is the percentage of execution that the coarse-grained phase i accounts for, and $IQR(phase_i)$ is the interquartile range of the first principal component of phase i. The constant ensures a minimum $fBIC$ threshold of 0.4.

For this preliminary study, we took the best TICC segmentation with a maximum number of cluster of 10 ($cMaxK = 10$ and $cBIC = 1$) and $fMaxK$ of 10. We evaluate this approach on the 8 *gcc* benchmarks. Table 5.6 lists in gray the results for the same set of TICC configurations used in the previous section. For comparison, we also list the CoV for the approach using a fixed value of $fBIC$ that produced simulations points in the same range (between 498 and 545 simulation points). The results are sorted by the CoV. The proportional $fBIC$ approach resulted in the lowest L3 CoV.

5.4.3 Other Clustering Algorithms Comparison

We now compare our work with the simulation points obtained using the popular SimPoint [64] and the work done by Eeckhout et al. [15]. In [15], they build on SimPoint to find simulation points using MICA as an interval signature instead of BBV. This is essentially our second level of sampling aplied to whole program observations. For convenience, we shall refer to the approach in [15] as a single level approach, also denoted as *MICA-SL*, and our approach as a multilevel approach, or *MICA-ML*.

The three methods use k-means, which requires the number of cluster K to be previously defined. We set the value of K for *SimPoint* and *MICA-SL* with the number of simulation points resulted in the *MICA-ML* approach. Thus, the comparison across the three methods is done with the same number of simulation points. For the coarse-grained phases of *MICA-ML*, we set a $cMaxK$ of 10 clusters with $cBIC$ of 1; and for fine-grained phases, we set a $fMaxK$ of 10 and for the $fBic$ we employ the per-phase approach in Section 5.4.2. The BBV and MICA signatures are sampled per interval of 160M dynamic instructions.

We perform the comparison for the SPECint 2006 benchmarks and use the CoV in cycles, L1-D, and L2 estimation for all benchmarks. In the L3 misses, we used two different evaluation metrics, one for each subset of benchmarks. As previously stated, computing the CoV involves dividing by the average, which is a problem in the cases of a long phase with almost no L3 misses. For this reason, we used CoV for a set of programs without mean value is close to zero, and weighted standard deviation for the others. Table 5.7 shows the program used in each metric for the L3 comparison.

Figure 5.13 shows the average of 20 execution across all programs for multiple TICC configuration. There is one bar for each TICC baseline configuration presented in Table 5.5. The CoV of cycles, L1-D, and L2 show that *MICA-SL* achieves a slightly higher accuracy than *SimPoint* and *MICA-ML*. For the L3-CoV results, both *SimPoint* and *MICA-SL* perform the same, with a slightly better result than *MICA-SL*. For L3-Weighted SD, *MICA-SL* and *MICA-ML* significantly outperform SimPoint. This suggested that observing phase behavior via MICA features may be suitable for applications where memory usage is not well aligned to the program code execution.

TICC Configuration	Parameter for choosing # of clusters	L3 CoV	# Simulation Points
w :12 β :2000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.64	542
w :12 β :2000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.65	516
w :16 β :4000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.65	538
w :12 β :3000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.66	529
w :12 β :3000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.66	535
w :12 β :4000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.66	537
w :8 β :2000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.66	502
w :8 β :3000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.66	519
w :12 β :2000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.67	536
w :8 β :3000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.67	528
w :12 β :3000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.68	528
w :12 β :4000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.68	527
w :16 β :2000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.68	523
w :16 β :4000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.68	543
w :16 β :2000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.71	519
w :16 β :2000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.71	527
w :16 β :3000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.72	526
w :16 β :3000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.72	525
w :8 β :2000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.72	501
w :8 β :4000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.74	500
w :16 β :2000 λ :0.2	$cMaxK$:4 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.77	522
w :16 β :2000 λ :0.2	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.77	522
w :8 β :2000 λ :0.2	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.77	538
w :12 β :4000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.78	540
w :8 β :4000 λ :0.3	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.78	521
w :16 β :3000 λ :0.2	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.8	0.79	511
w :16 β :3000 λ :0.2	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.8	0.79	511
w :16 β :4000 λ :0.1	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.79	506
w :16 β :3000 λ :0.2	$cMaxK$:10 $cBIC$:1 $fMaxK$:10 $fBIC$ prop.	0.81	516
w :8 β :2000 λ :0.2	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.81	524
w :12 β :4000 λ :0.1	$cMaxK$:4 $cBIC$:0.8 $fMaxK$:20 $fBIC$:1.0	0.82	540
w :16 β :2000 λ :0.1	$cMaxK$:4 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.82	507
w :16 β :2000 λ :0.2	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.8	0.82	516
w :16 β :2000 λ :0.3	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.82	543
w :8 β :3000 λ :0.2	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.82	526
w :8 β :3000 λ :0.2	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.82	525
w :8 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.82	531
w :12 β :4000 λ :0.1	$cMaxK$:4 $cBIC$:0.9 $fMaxK$:20 $fBIC$:1.0	0.83	540
w :16 β :2000 λ :0.1	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.83	508
w :8 β :3000 λ :0.1	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.83	514
w :8 β :3000 λ :0.1	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.83	524
w :8 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.83	521
w :16 β :3000 λ :0.3	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.85	516
w :16 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.8	0.85	542
w :16 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.8	0.85	543
w :8 β :2000 λ :0.3	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.85	512
w :8 β :4000 λ :0.2	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.85	542
w :16 β :2000 λ :0.3	$cMaxK$:4 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.86	501
w :16 β :3000 λ :0.2	$cMaxK$:4 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.9	0.86	510
w :16 β :3000 λ :0.2	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.86	510
w :16 β :2000 λ :0.2	$cMaxK$:5 $cBIC$:0.9 $fMaxK$:20 $fBIC$:0.8	0.87	505
w :16 β :2000 λ :0.3	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.87	501
w :16 β :3000 λ :0.2	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.8	0.87	527
w :16 β :4000 λ :0.1	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.87	505
w :8 β :2000 λ :0.3	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.87	500
w :16 β :2000 λ :0.2	$cMaxK$:4 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.88	505
w :8 β :2000 λ :0.3	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.88	528
w :12 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.89	537
w :16 β :3000 λ :0.1	$cMaxK$:4 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.89	505
w :16 β :4000 λ :0.3	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.9	0.93	528
w :8 β :3000 λ :0.3	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.94	530
w :16 β :2000 λ :0.1	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.8	0.95	519
w :16 β :4000 λ :0.1	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.8	0.95	510
w :8 β :4000 λ :0.3	$cMaxK$:5 $cBIC$:0.8 $fMaxK$:20 $fBIC$:0.9	0.95	540
w :8 β :3000 λ :0.1	$cMaxK$:4 $cBIC$:0.8 $fMaxK$:20 $fBIC$:1.0	0.98	537
w :16 β :3000 λ :0.3	$cMaxK$:5 $cBIC$:1.0 $fMaxK$:20 $fBIC$:0.8	1.05	507

Table 5.6: Average CoV across *gcc* benchmarks for different TICC settings

Set 1 (CoV)	Set 2 (Weighted Standard Deviation)
400.perlbench.1, 400.perlbench.2, 400.perlbench.3, 403.gcc.1, 403.gcc.2, 403.gcc.3, 403.gcc.4, 403.gcc.5, 403.gcc.6, 403.gcc.7, 403.gcc.8, 429.mcf.1, 445.gobmk.1, 445.gobmk.2, 445.gobmk.3, 445.gobmk.4, 445.gobmk.5, 458.sjeng.1, 462.libquantum.1, 464.h264ref.3, 471.omnetpp.1, 473.astar.1, 483.xalancbmk.1	401.bzip2.1, 401.bzip2.2, 401.bzip2.3, 401.bzip2.4, 401.bzip2.5, 401.bzip2.6, 456.hmmer.1, 456.hmmer.2, 464.h264ref.1, 464.h264ref.2, 473.astar.2

Table 5.7: Two sets of programs used for comparison

In Figure 5.14 we expand the results of a single TICC configuration ($w = 8$, $\beta = 2000$, and $\lambda = 0.2$) to show the results of accuracy for each individual benchmark. We plot the CoV and weighted standard deviations along with the corresponding standard deviations of *MICA-ML* and previous approaches.

On the *gcc* benchmarks, our multilevel approach can reduce CoV by 35% of the single level clustering and 10% on SimPoint. For L3-Weighted SD, *MICA-SL* and *MICA-ML* significantly outperform SimPoint. Particularly, on the *bzip2* benchmarks, our multilevel approach can significantly reduce the weighted SD by 89% of the *SimPoint* method.

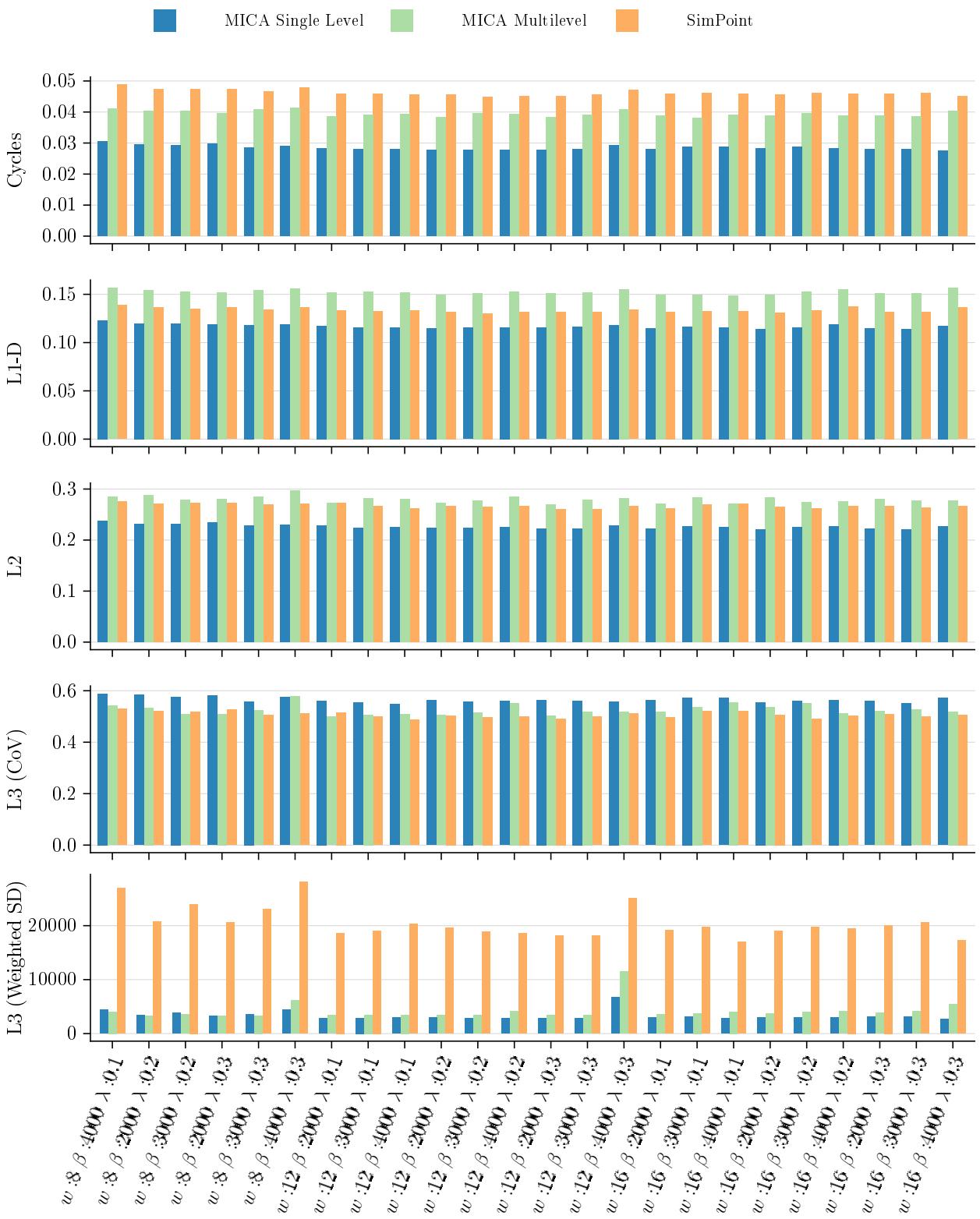


Figure 5.13: CoV and weighted standard deviation (along with standard deviation) for multiple TICC configuration along with prior works with the same number of simulation points

Figure 5.14: CoV and weighted standard deviation (along with standard deviation) of all SPECint 2006 programs. TICC parameters are $w = 12$, $\beta = 2000$, and $\lambda = 0.1$

5.5 Phase Interpretability

In this section, we demonstrate how TICC can be used to learn interpretable clusters in real-world applications. Although PCA reduces the data set's dimensionality effectively, the fact that each principal component is a linear combination of the original workload characteristics complicates the understandability of the lower-dimensional workload space. For this reason, we run this analysis on all the 97 MICA characteristics described in Table 4.1.

We use a centrality measure to show how important each MICA characteristic is, and more specifically 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 [52]. More precisely, the BC of a node v is given by [3]

$$C_B(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{5.6}$$

where σ_{st} is the total number of shortest paths from node s to node t and $\sigma_{st}(v)$ is the number of shortest paths from s to t going through v.

We used the NetworkX [60] Python package to calculate the centrality scores. Tables 5.8 and 5.9 show the betweenness centrality score of each MICA feature of two different programs from SPEC 2006 *gcc.166* and *bzip2.chicken*. 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). Figures 5.15 and 5.16 show the time-varying of some architectural metrics colored accordingly to the resulting TICC segmentation. The numbers on top are the phase identifiers of each column in the betweenness centrality score tables. For example, phase 3 of *gcc.166* in Table 5.8 is the green one in Figure 5.15. 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.

5.5.1 Case study: *bzip2.chicken*

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 [51] with its internal tool Callgrind to produce a call-graph for the complete execution of *bzip2* with input *chicken*, depicted in Figure 5.17a. 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

MICA feature	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	Phase 6
ILP 32-entry window	0	0	0	0	0	0
ILP 64-entry window	0	5	0	0	0	0
ILP 128-entry window	0	3	0	0	0	0
ILP 256-entry window	0	2	0	0	0	0
mem-read	0	0	0	0	0	0
mem-write	0	0	0	0	0	0
control-flow	0	0	0	3	0	0
arithmetic	0	0	0	0	2	0
floating-point	0	0	0	0	0	0
pop/push instructions (stack usage)	0	0	1	0	10	0
shift instructions (bitwise)	0	0	0	0	0	0
string	0	0	0	2	0	0
MMX/SSE instructions	7	0	0	0	0	0
other	0	0	0	0	0	1
nop	0	0	0	0	0	0
other1	0	7	0	0	0	0
GAg PPM predictor (4 bits)	0	0	0	0	0	0
PAg PPM predictor (4 bits)	0	0	0	0	0	0
GAs PPM predictor (4 bits)	0	0	0	0	0	0
PAs PPM predictor (4 bits)	0	0	0	0	0	0
GAg PPM predictor (8 bits)	0	0	0	0	0	0
PAg PPM predictor (8 bits)	0	0	0	0	0	0
GAs PPM predictor (8 bits)	0	0	0	0	0	0
PAs PPM predictor (8 bits)	0	0	0	0	0	0
GAg PPM predictor (12 bits)	0	0	0	0	0	0
PAg PPM predictor (12 bits)	0	0	0	0	0	0
GAs PPM predictor (12 bits)	0	0	0	0	0	0
PAs PPM predictor (12 bits)	0	0	0	0	0	0
Branch count	0	0	0	2	0	0
Branch transition	0	0	0	0	0	0
Branch taken count	0	0	0	0	0	0
Number of reg. operands	0	0	0	0	0	0
Reg. instances created	0	0	0	0	4	0
Reg. instance uses	0	0	1	0	0	0
Total reg. dependency distance	0	0	0	0	0	0
Reg. dependency distance $\leq 2^0$	0	0	7	0	0	0
Reg. dependency distance $\leq 2^1$	0	0	1	0	0	0
Reg. dependency distance $\leq 2^2$	0	0	0	0	1	0
Reg. dependency distance $\leq 2^3$	0	0	0	0	0	0
Reg. dependency distance $\leq 2^4$	0	0	0	0	0	0
Reg. dependency distance $\leq 2^5$	0	0	0	0	0	0
Reg. dependency distance $\leq 2^6$	0	0	0	0	0	0
Instruction Footprint 64-byte	7	0	0	0	0	0
Instruction Footprint 4kB	0	0	0	0	0	0
Data Footprint 64-byte	0	0	0	10	0	0
Data Footprint 4kB	0	0	0	4	0	0
Total number of memory accesses	0	0	0	0	0	0
Reuse distance $[0, 2^1)$	0	0	1	0	0	0
Reuse distance $[2^1, 2^2)$	0	0	0	0	0	0
Reuse distance $[2^2, 2^3)$	0	0	7	0	0	0
Reuse distance $[2^3, 2^4)$	0	0	0	0	0	0
Reuse distance $[2^4, 2^5)$	0	10	0	0	0	0
Reuse distance $[2^5, 2^6)$	0	0	0	0	6	0
Reuse distance $[2^6, 2^7)$	0	0	0	0	0	0
Reuse distance $[2^7, 2^8)$	0	0	0	0	0	0
Reuse distance $[2^8, 2^9)$	0	0	0	0	0	0
Reuse distance $[2^9, 2^{10})$	0	0	0	0	0	0
Reuse distance $[2^{10}, 2^{11})$	0	0	0	0	0	0
Reuse distance $[2^{11}, 2^{12})$	0	0	0	1	0	0
Reuse distance $[2^{12}, 2^{13})$	0	0	0	2	0	0
Reuse distance $[2^{13}, 2^{14})$	0	0	0	0	0	0
Reuse distance $[2^{14}, 2^{15})$	0	0	0	1	0	0
Reuse distance $[2^{15}, 2^{16})$	0	0	0	0	0	0
Reuse distance $[2^{16}, 2^{17})$	0	0	0	0	0	2
Reuse distance $[2^{17}, 2^{18})$	0	0	0	0	0	10
Reuse distance $[2^{18}, 2^{19})$	10	0	0	0	0	0
Reuse distance $[2^{19}, \infty)$	5	0	0	0	0	0
Memory read count	0	0	0	0	0	0
Local load stride $\leq 8^0$	0	0	0	0	0	0
Local load stride $\leq 8^1$	0	0	0	0	0	0
Local load stride $\leq 8^2$	0	0	0	0	0	0
Local load stride $\leq 8^3$	0	0	0	0	0	0
Local load stride $\leq 8^4$	0	0	0	0	0	0
Local load stride $\leq 8^5$	0	0	0	0	0	0
Local load stride $\leq 8^6$	0	0	0	0	0	0
Global load stride $\leq 8^0$	0	0	0	0	0	0
Global load stride $\leq 8^1$	0	0	0	0	0	0
Global load stride $\leq 8^2$	0	0	1	0	0	0
Global load stride $\leq 8^3$	0	0	1	0	0	0
Global load stride $\leq 8^4$	0	0	1	0	0	0
Global load stride $\leq 8^5$	0	0	1	0	0	0
Global load stride $\leq 8^6$	0	0	1	0	0	0
Memory write count	0	0	0	0	0	0
Local store stride $\leq 8^0$	0	0	0	1	0	0
Local store stride $\leq 8^1$	0	0	0	0	0	0
Local store stride $\leq 8^2$	0	0	0	0	0	0
Local store stride $\leq 8^3$	0	0	0	0	0	0
Local store stride $\leq 8^4$	0	0	0	0	0	0
Local store stride $\leq 8^5$	0	0	0	0	0	0
Local store stride $\leq 8^6$	0	0	0	0	0	0
Global store stride $\leq 8^0$	0	0	10	0	0	0
Global store stride $\leq 8^1$	0	0	0	0	7	0
Global store stride $\leq 8^2$	0	0	0	0	0	0
Global store stride $\leq 8^3$	0	0	0	0	0	0
Global store stride $\leq 8^4$	0	0	0	0	0	0
Global store stride $\leq 8^5$	0	0	0	0	0	0

Table 5.8: Betweenness centrality for each MICA feature in *gcc.166*

MICA feature	Phase 1	Phase 2	Phase 3	Phase 4	Phase 5	Phase 6
ILP 32-entry window	0	0	0	0	0	0
ILP 64-entry window	0	0	0	0	0	0
ILP 128-entry window	0	0	0	0	0	0
ILP 256-entry window	0	0	0	0	0	0
mem-read	0	0	0	0	0	0
mem-write	0	0	0	0	0	0
control-flow	0	0	0	0	0	0
arithmetic	0	0	0	0	0	0
floating-point	0	0	0	0	0	0
pop/push instructions (stack usage)	0	0	10	0	1	0
shift instructions (bitwise)	0	1	0	1	0	0
string	0	0	0	0	0	0
MMX/SSE instructions	0	0	0	0	0	0
other	0	2	0	0	0	0
nop	0	0	0	0	0	0
other1	0	0	0	0	0	0
GAg PPM predictor (4 bits)	0	0	0	0	0	0
PAg PPM predictor (4 bits)	0	1	0	1	0	0
GAs PPM predictor (4 bits)	0	0	0	0	0	0
PAs PPM predictor (4 bits)	0	1	0	1	0	0
GAg PPM predictor (8 bits)	0	0	0	0	0	0
PAg PPM predictor (8 bits)	0	1	0	0	0	0
GAs PPM predictor (8 bits)	0	0	0	0	0	0
PAs PPM predictor (8 bits)	0	1	0	1	0	0
GAg PPM predictor (12 bits)	0	0	0	0	0	0
PAg PPM predictor (12 bits)	0	0	0	0	0	0
GAs PPM predictor (12 bits)	0	3	0	0	0	0
PAs PPM predictor (12 bits)	0	0	0	1	0	0
Branch count	0	0	0	0	0	0
Branch transition	0	0	0	0	0	0
Branch taken count	0	7	0	10	0	0
Number of reg. operands	0	0	0	0	0	0
Reg. instances created	0	0	1	0	0	0
Reg. instance uses	0	2	0	3	0	0
Total reg. dependency distance	0	0	0	0	0	0
Reg. dependency distance $\leq 2^0$	0	0	0	0	0	0
Reg. dependency distance $\leq 2^1$	0	0	0	0	0	0
Reg. dependency distance $\leq 2^2$	0	1	0	0	0	0
Reg. dependency distance $\leq 2^3$	0	4	0	1	0	0
Reg. dependency distance $\leq 2^4$	0	2	0	1	0	0
Reg. dependency distance $\leq 2^5$	0	10	0	4	0	0
Reg. dependency distance $\leq 2^6$	0	5	0	4	0	0
Instruction Footprint 64-byte	6	0	0	0	0	0
Instruction Footprint 4kB	0	0	0	0	0	8
Data Footprint 64-byte	0	0	0	0	3	0
Data Footprint 4kB	0	0	0	0	0	0
Total number of memory accesses	0	0	0	0	0	0
Reuse distance $[0, 2^1)$	0	0	0	0	0	0
Reuse distance $[2^1, 2^2)$	0	5	0	0	0	0
Reuse distance $[2^2, 2^3)$	0	3	0	2	0	0
Reuse distance $[2^3, 2^4)$	0	0	7	0	0	0
Reuse distance $[2^4, 2^5)$	0	0	0	0	0	0
Reuse distance $[2^5, 2^6)$	0	0	0	0	0	0
Reuse distance $[2^6, 2^7)$	0	0	0	0	0	0
Reuse distance $[2^7, 2^8)$	0	0	0	0	0	0
Reuse distance $[2^8, 2^9)$	0	0	0	0	0	0
Reuse distance $[2^9, 2^{10})$	0	6	0	1	0	0
Reuse distance $[2^{10}, 2^{11})$	0	1	0	1	0	0
Reuse distance $[2^{11}, 2^{12})$	0	0	0	0	0	0
Reuse distance $[2^{12}, 2^{13})$	0	0	0	0	0	0
Reuse distance $[2^{13}, 2^{14})$	0	0	0	0	3	0
Reuse distance $[2^{14}, 2^{15})$	0	0	0	0	0	0
Reuse distance $[2^{15}, 2^{16})$	0	0	0	0	0	0
Reuse distance $[2^{16}, 2^{17})$	10	0	0	0	0	0
Reuse distance $[2^{17}, 2^{18})$	0	1	0	3	0	0
Reuse distance $[2^{18}, 2^{19})$	0	0	0	0	0	0
Reuse distance $[2^{19}, \infty)$	0	0	0	0	0	0
Memory read count	0	0	0	0	0	0
Local load stride $\leq 8^0$	0	0	0	0	0	0
Local load stride $\leq 8^1$	0	0	1	0	0	0
Local load stride $\leq 8^2$	0	3	0	2	0	0
Local load stride $\leq 8^3$	0	1	0	1	0	0
Local load stride $\leq 8^4$	0	1	0	1	0	0
Local load stride $\leq 8^5$	0	1	0	0	0	0
Local load stride $\leq 8^6$	0	1	0	0	0	0
Global load stride $\leq 8^0$	0	0	0	0	0	0
Global load stride $\leq 8^1$	0	0	0	0	0	0
Global load stride $\leq 8^2$	0	0	0	0	0	0
Global load stride $\leq 8^3$	0	0	0	0	0	0
Global load stride $\leq 8^4$	0	0	0	0	0	0
Global load stride $\leq 8^5$	0	0	0	0	0	0
Global load stride $\leq 8^6$	0	0	0	0	0	0
Memory write count	0	0	0	0	0	0
Local store stride $\leq 8^0$	0	0	0	0	0	0
Local store stride $\leq 8^1$	0	0	0	0	0	0
Local store stride $\leq 8^2$	0	0	0	0	0	0
Local store stride $\leq 8^3$	0	0	0	0	0	0
Local store stride $\leq 8^4$	0	0	0	0	0	0
Local store stride $\leq 8^5$	0	0	0	0	0	0
Local store stride $\leq 8^6$	0	0	0	0	0	0
Global store stride $\leq 8^0$	0	0	0	0	10	0
Global store stride $\leq 8^1$	0	0	0	0	0	0
Global store stride $\leq 8^2$	0	0	0	0	0	0
Global store stride $\leq 8^3$	0	0	0	0	0	0
Global store stride $\leq 8^4$	0	0	0	0	0	0
Global store stride $\leq 8^5$	0	0	0	0	0	0

Table 5.9: Betweenness centrality for each MICA feature in *bzip2.chicken*

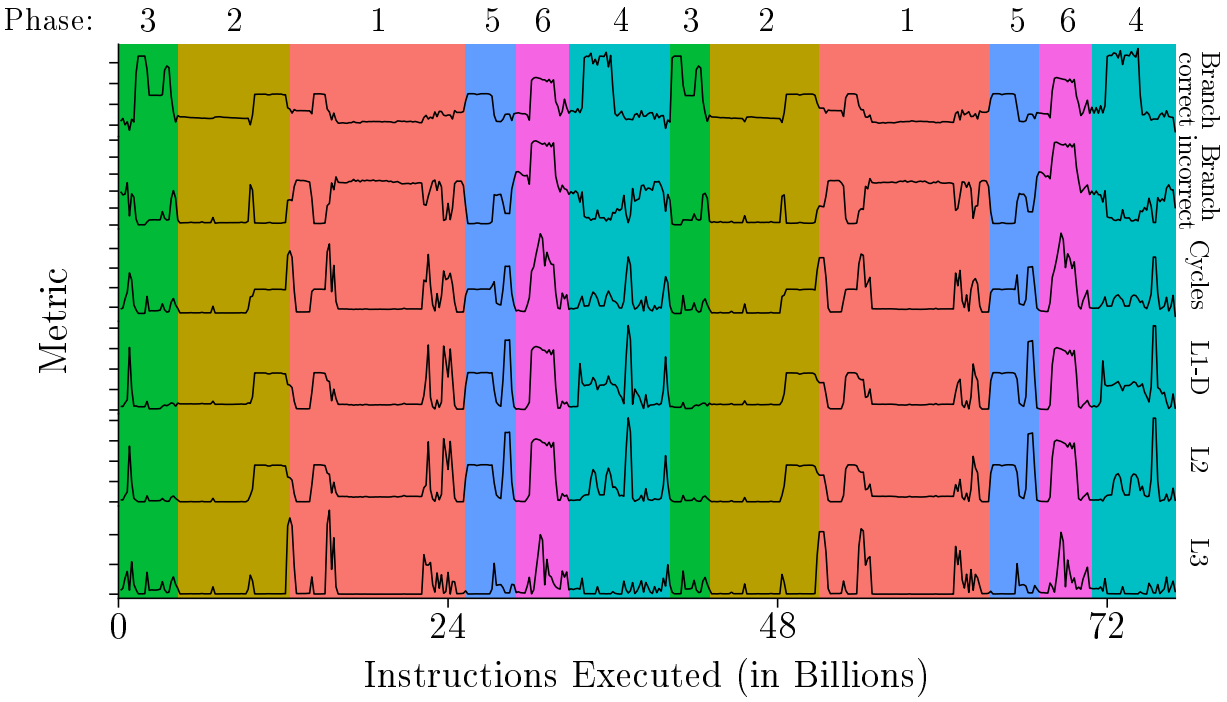


Figure 5.15: Time-varying graph for *gcc.166* with phase identifier on top

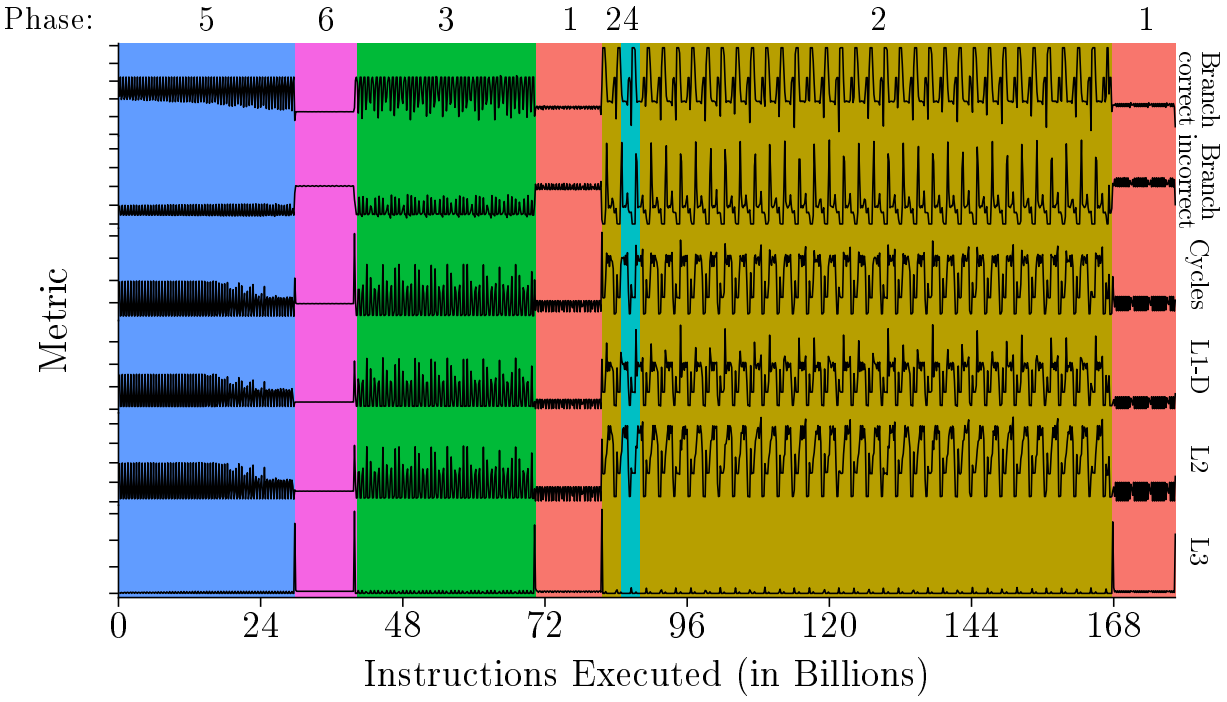
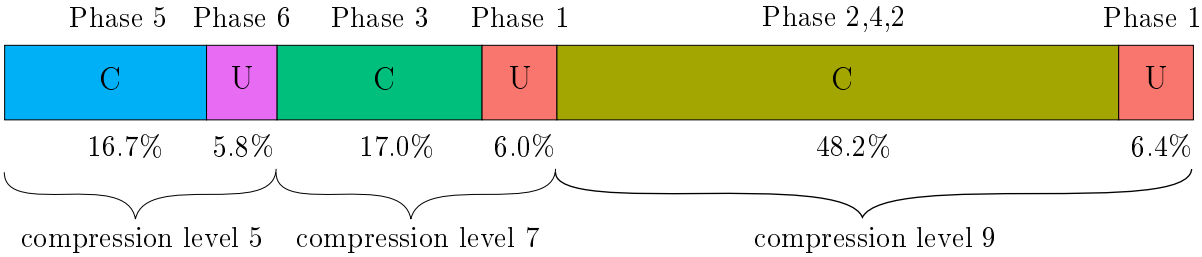


Figure 5.16: Time-varying graph for *bzip2.chicken* with phase identifier on top

specified functions. We used it to dump counters after each compress/uncompress round to produce a seperate call-graph for each iteration, as show in Figures 5.17b to 5.17d.

The instruction counts of each iteration align with the phase segmentation in Figure 5.16 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 and the call graphs follow the same color scheme as in Figure 5.16 (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.

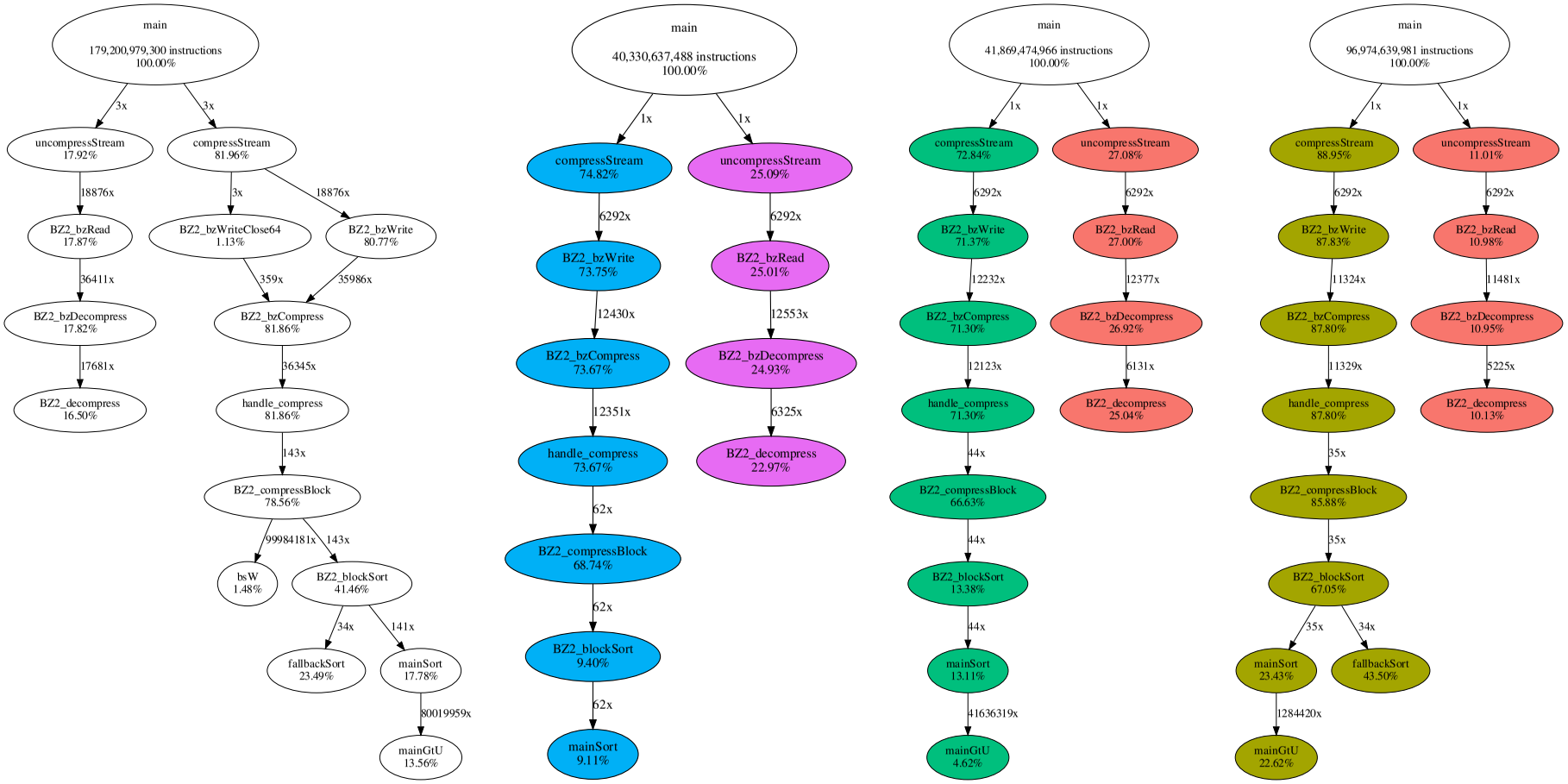
We can also observe that each phase has unique centrality scores representing its behavior of the program. To complete the analysis we make the following observations:

- 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 (unique address accesses that occur between two accesses to the same address) 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).

- Phase 2 and 4 have a pretty 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 (β) that encourages neighboring samples to be assigned to the same cluster [27].

- At compression level 9 (phase 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" [5]. Considering that radix sort tends to exhibit poor cache locality [43], 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).



(a) Complete execution (b) Compression level 5 (c) Compression level 7 (d) Compression level 9

Figure 5.17: Callgraphs for *bzip2.chicken* of the complete execution (a) and for each compresses/decompress round (b) to (d)

5.6 Summary

In this chapter, we explored our method for automatically characterizing time-varying behavior in programs with two focuses: the main parameters that influence our two-level sampled simulation and on the interpretability of an MRF multilayer signature. Our sampled simulation analysis focused on the error derived from sampling. We presented a graphical visualization of the coarse-grained phases found by TICC for some popular benchmarks, in which we graphically observed a proper segmentation. We then presented an investigation into the parameters of TICC along with k-means for phase classification and compared it with two previous approaches in sampled simulation. As the final contribution in this chapter, we showed that TICC can learn interpretable clusters in real-world applications by breaking down the large scale behavior of several complex programs into a clear sequence of states, and abstract to most important features.

Chapter 6

Conclusion and Future Work

This dissertation presented a method for characterizing time-varying program behavior with the main purpose to accelerate microarchitecture simulation. Cycle-accurate simulators typically allow performance evaluation to be done with great flexibility and relatively low cost [16]. However, due to the complexity of modern processors and benchmarks, they are extremely slow, which makes it unfeasible to have full detailed simulation. A popular solution for accelerating detailed simulation is to explore the fact that many applications exhibit a phasic behavior, and by simulating only each unique behavior, it is possible to infer performance metrics with a much shorter time and appropriate accuracy. A second goal of this work was to provide interpretable insights on the key factors and relationships that characterize each program phase.

We observed program phases via a set of important microarchitecture-independent characteristics [30]. 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 approach comprises a two-level sampling strategy of these characteristics. The first uses a multivariate time-series clustering method known as TICC; the second level further re-sample each phase using k-means. To our knowledge, this is the first work to treat phase analysis as a subsequence time series clustering problem.

TICC has four main parameters: β, λ, window size and the number of clusters. Empirically, we discovered that TICC is robust to the selection of λ and w. Instead, the critical parameters are β and the number of clusters K. We also discovered that fewer clusters and a smaller β can decrease the probability of achieving local minima.

Our first step in this work was to visually analyze TICC's segmentation for multiple combinations of parameters (β: 2000, 3000, and 400; window size: 8, 12, and 16; and λ: 0.1, 0.2, and 0.3). We analyzed the segmentation of all the 34 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 K, TICC could not converge to a solution. We made available online[1] the entire segmentation resulted from this exploration.

From the TICC segmentation for features sampled at every 160 million instructions, we

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

observed a large factor of phasic behavior. We found a ratio of unique behavior to overall execution close to 50%. This result was obtained by summing up the program phase sizes and then dividing by the total number of dynamic instructions of the complete execution. Apart from the workload analysis side, this strengthened the use of finer granularity sampling, since our focus was on efficient simulation through sampled simulation.

We analyzed the effect on the number of cluster K and the BIC threshold on both clustering algorithms TICC (first stage of sampling), and k-means (second stage of sampling). We showed the results of accuracy and number simulations points when varying these four parameters altogether. Depending on the architect's objective, these results help to decide on the tradeoff between simulation time and accuracy.

We found that the greater the BIC score, the more simulation points are chosen. However, this does not necessarily result in a great improvement in accuracy. In particular, given a TICC segmentation, our initial approach was to use the same value of $fBIC$ for all phases. Empirically, we discovered that such approach may give unnecessary simulation points to a coarse-grained phase that accounts for a tiny fraction of execution, which does not have a great impact on overall estimation. We proposed an alternative method to pick the value of K for k-means on a per-phase basis proportional to the coarse-grained phase size and homogeneity.

We evaluated our sampled simulation method for 486 configurations of K and BIC on both clustering levels, and the main TICC parameters. Our experiments on the SPECint CPU2006 workloads showed that for configurations resulting in about 1500 ± 50 simulations points (1.1% of the complete execution), the average cycles and L3 (easiest to the most difficult) estimate error is 0.77% (min. 0.60% and max. 0.92%), and 6.23% (min. 5.20% and max. 7.64%) respectively. If willing to trade some speed accuracy for accuracy, those error estimates dropped to 0.63% (min. 0.53% and max. 0.69%) and 4.65% (min. 4.01% and max. 5.54%) for an average of 2000 ± 50 simulations points (1.45% of total execution). It should be noted that all the interval metrics assume a perfect warmup, as we focused on the representativeness of the sampled points.

Our experiments showed that, on average, the phases detected by our approach have behavior homogeneity comparable to the phases detected by SimPoint [64] or clustering MICA signatures over the entire program execution [15] (single-level approach). Thus, for sampled simulation, our multilevel approach did not outperform the single-level approach. It is worth noting that for the 8 *gcc* program-input pairs our method reduced L3 CoV by 32% on average. Lastly, we found a MICA based signature clustering to be more effective on metrics loosely correlated with code (L3 in our experiments).

We also showed that TICC can be used to find meaningful insights into program phases through the analysis of the multilayer 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.

6.1 Publications

The following publication was produced during the development of this dissertation:

- R. Soares, L. Antonioli, E. Francesquini and R. Azevedo. "Phase Detection and Analysis among Multiple Program Inputs". 2018 Symposium on High Performance Computing Systems (WSCAD), São Paulo, Brazil, 2018, pp. 155-161.

6.2 Future Work

Phase analysis has unlocked many optimizations including simulation acceleration, power reduction, cache optimization, and compiler optimization. We hope that TICC phases may unlock many other optimizations. The list of possible future work on TICC-based phase classification includes:

- Explore how phase's MRF can be used to improve or discover compiler optimizations.
- Characterize modern industry-standard programs.
- Evaluate benchmark suites balance using the MRF representation provided by TICC.
- Measure cross-program and cross-input similarities among benchmark suites.
- Explore how TICC can help to improve sampled simulation of multi-threaded applications.
- Use MRF as a signature for predictive modeling.

The list of possible work with the main focus on our multi-level sampled simulation includes:

- Evaluate other clustering algorithms for second-level sampling other than k-means.
- Evaluate the tradeoff between different interval granularities at both clustering levels. Also, evaluate the effectiveness of combining different interval granularities.
- Evaluate the effectiveness of a TICC-segmentation on smaller sampling granularities.
- Explore other heuristics of choosing the appropriate number of clusters on both levels of clustering. Our initial attempt was to pick a value of K on a per-phase basis proportional to its homogeneity and size.
- Evaluate a hybrid approach combing a single-level and two-level – find a way to decide which one performs better for a specific program. In some situations, one may achieve the same accuracy with fewer simulation points.
- Explore if the MRF information can give insights on the appropriate interval size for a particular program. Also, if it can help to determine the warm-up size for an interval extracted from a TICC phase.

Bibliography

[1] Giuseppe Ascia, Vincenzo Catania, Alessandro G. Di Nuovo, Maurizio Palesi, and Davide Patti. Efficient design space exploration for application specific systems-on-a-chip. *J. Syst. Archit.*, 53(10):733–750, October 2007.

[2] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, and Sandhya Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 245–257, New York, NY, USA, 2000. ACM.

[3] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[4] Maximilien B. Breughe and Lieven Eeckhout. Selecting representative benchmark inputs for exploring microprocessor design spaces. *ACM Trans. Archit. Code Optim.*, 10(4):37:1–37:24, December 2013.

[5] bzip2. bzip2. `https://sourceware.org/git/bzip2.git`, 2005. Version 1.0.3.

[6] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 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, 2011. ACM.

[7] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 128–137, New York, NY, USA, 1996. ACM.

[8] Tianshi Chen, Qi Guo, Ke Tang, Olivier Temam, Zhiwei Xu, Zhi-Hua Zhou, and Yunji Chen. Archranker: A ranking approach to design space exploration. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 85–96, Piscataway, NJ, USA, 2014. IEEE Press.

[9] Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. Evaluating iterative optimization across 1000 datasets. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 448–459, New York, NY, USA, 2010. ACM.

[10] C. Cho and T. Li. Using wavelet domain workload execution characteristics to improve accuracy, scalability and robustness in program phase analysis. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 136–145, April 2007.

[11] Henry Cook and Kevin Skadron. Predictive design space exploration using genetically programmed response surfaces. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, pages 960–965, New York, NY, USA, 2008. ACM.

[12] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, ISCA '02, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[13] Christophe Dubach, Timothy Jones, and Michael O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 262–271, Washington, DC, USA, 2007. IEEE Computer Society.

[14] Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 220–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] L. Eeckhout, J. Sampson, and B. Calder. 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, Oct 2005.

[16] Lieven Eeckhout. *Computer Architecture Performance Evaluation Methods*. Morgan & Claypool Publishers, 1st edition, 2010.

[17] Lieven Eeckhout, Robert H. Bell Jr., Bastiaan Stougie, Koen De Bosschere, and Lizy K. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 350–, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Lieven Eeckhout, Sebastien Nussbaum, James E. Smith, and Koen De Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, September 2003.

[19] Lieven Eeckhout, Hans Vandierendonck, and Koen De Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, 5:1–33, 2003.

[20] Lieven Eeckhout, Hans Vandierendonck, and Koenraad De Bosschere. 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, 2002. IEEE Computer Society.

[21] M. Franklin and G. S. Sohi. Register traffic analysis for streamlining inter-operation communication in fine-grain parallel processors. In *[1992] Proceedings the 25th Annual International Symposium on Microarchitecture MICRO 25*, pages 236–245, Dec 1992.

[22] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 12 2007.

[23] Tony Givargis, Frank Vahid, and Jörg Henkel. System-level exploration for pareto-optimal configurations in parameterized systems-on-a-chip. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 25–30, Piscataway, NJ, USA, 2001. IEEE Press.

[24] Qi Guo, Tianshi Chen, Yunji Chen, and Franz Franchetti. Accelerating architectural simulation via statistical techniques: A survey. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 35(3):433–446, March 2016.

[25] Qi Guo, Tianshi Chen, Yunji Chen, Ling Li, and Weiwu Hu. Microarchitectural design space exploration made fast. *Microprocess. Microsyst.*, 37(1):41–51, February 2013.

[26] David Hallac, Sagar Vare, Stephen Boyd, and Jure Leskovec. TICC. `https://github.com/davidhallac/TICC`, 2017.

[27] David Hallac, Sagar Vare, Stephen Boyd, and Jure Leskovec. 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, 2017. ACM.

[28] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[29] Kenneth Hoste and Lieven Eeckhout. MICA. `https://github.com/boegel/MICA`, 2007.

[30] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, May 2007.

[31] Kenneth Hoste and Lieven Eeckhout. pyxmeans. `https://github.com/boegel/MIC://github.com/mynameisfiber/pyxmeans/`, 2015.

[32] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures*

and Compilation Techniques, PACT '06, pages 114–122, New York, NY, USA, 2006. ACM.

[33] Michael C. Huang, Jose Renau, and Josep Torrellas. Positional adaptation of processors: Application to energy reduction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 157–168, New York, NY, USA, 2003. ACM.

[34] Ted Huffmire and Tim Sherwood. Wavelet-based phase classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 95–104, New York, NY, USA, 2006. ACM.

[35] Engin İpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 195–206, New York, NY, USA, 2006. ACM.

[36] C. Isci and M. Martonosi. 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, Feb 2006.

[37] P. J. Joseph and M. J. Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.*, pages 99–108, Feb 2006.

[38] P. J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[39] Ajay Joshi, Aashish Phansalkar, Lieven Eeckhout, and Lizy Kurian John. Measuring benchmark similarity using inherent program characteristics. *IEEE Trans. Comput.*, 55(6):769–782, June 2006.

[40] Salman Khan, Polychronis Xekalakis, John Cavazos, and Marcelo Cintra. Using predictive modeling for cross-program design space exploration in multicore systems. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 327–338, Washington, DC, USA, 2007. IEEE Computer Society.

[41] A J KleinOsowski and David J. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research. *IEEE Comput. Archit. Lett.*, 1(1):7–7, January 2002.

[42] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press, 2009.

[43] Anthony LaMarca and Richard E Ladner. The influence of caches on the performance of sorting. *Journal of Algorithms*, 31(1):66 – 104, 1999.

[44] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. 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, 2005. IEEE Computer Society.

[45] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 57–67, Washington, DC, USA, 2004. IEEE Computer Society.

[46] J. Lau, S. Schoemackers, and B. Calder. Structures for phase classification. In *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, pages 57–67, March 2004.

[47] Jeremy Lau, Erez Perelman, and Brad Calder. Selecting software phase markers with code structure analysis. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 135–146, Washington, DC, USA, 2006. IEEE Computer Society.

[48] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. *SIGPLAN Not.*, 41(11):185–194, October 2006.

[49] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, June 2005.

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[51] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

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

[53] Harish Patil, Robert Cohn, Mark Charney, Rajiv Kapoor, Andrew Sun, and Anand Karunanidhi. Pinpointing representative portions of large intel®itanium®programs with dynamic instrumentation. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 81–92, Washington, DC, USA, 2004. IEEE Computer Society.

[54] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. Pinplay: A framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 2–11, New York, NY, USA, 2010. ACM.

[55] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[56] Dan Pelleg and Andrew W. Moore. X-means: Extending k-means with efficient estimation of the number of clusters. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 727–734, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[57] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John. Measuring program similarity: Experiments with spec cpu benchmark suites. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 10–20, March 2005.

[58] Havard Rue and Leonhard Held. *Gaussian Markov Random Fields: Theory And Applications (Monographs on Statistics and Applied Probability)*. Chapman & Hall/CRC, 2005.

[59] Kaushal Sanghai, Ting Su, Jennifer Dy, and David Kaeli. A multinomial clustering model for fast simulation of computer architecture designs. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, pages 808–813, New York, NY, USA, 2005. ACM.

[60] Daniel A. Schult. Exploring network structure, dynamics, and function using networkx. In *In Proceedings of the 7th Python in Science Conference (SciPy*, pages 11–15, 2008.

[61] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. Hass: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, April 2009.

[62] Xipeng Shen, Yutao Zhong, and Chen Ding. 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, 2004. ACM.

[63] Timothy Sherwood, Erez Perelman, and Brad Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[64] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, October 2002.

[65] Timothy Sherwood, Erez Perelman, Greg Hamerly, Suleyman Sair, and Brad Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6):84–93, November 2003.

[66] Michael Van Biesbrouck, Lieven Eeckhout, and Brad Calder. Efficient sampling startup for sampled processor simulation. In Tom Conte, Nacho Navarro, Wenmei W. Hwu, Mateo Valero, and Theo Ungerer, editors, *High Performance Embedded Architectures and Compilers*, pages 47–67, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[67] Roland E. Wunderlich, Thomas F. Wenisch, Babak Falsafi, and James C. Hoe. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 84–97, New York, NY, USA, 2003. ACM.

[68] Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins. A statistically rigorous approach for improving simulation methodology. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, HPCA '03, pages 281–, Washington, DC, USA, 2003. IEEE Computer Society.

[69] Weihua Zhang, Jiaxin Li, Yi Li, and Haibo Chen. Multilevel phase analysis. *ACM Trans. Embed. Comput. Syst.*, 14(2):31:1–31:29, March 2015.